Computing Science

INTELLECTUAL PROPERTY PROTECTION USING OBFUSCATION

Stephen Drape

Collaboration between Siemens, Munich and the University of Oxford

Research Project sponsored by Siemens AG, Munich

CS-RR-10-02

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

In this technical report, we discuss the use of code obfuscation as means of protecting the intellectual property of software. An obfuscation is a behaviour preserving program transformation which aims to make a program harder to understand (which can mean it becomes unintelligible to automated program comprehension tools or that the result of program analyses become less useful to a human adversary).

This report consists of three main parts. The first part of the report discusses some of the possible definitions of obfuscation and gives an extensive survey of some of the current obfuscation techniques. The second part considers a report written by University of Applied Sciences of Upper Austria, Hagenberg which discuss a variety of different protection techniques (including obfuscation). The last part reviews the techniques that we have seen so far. We provide an analysis of some of the reasons why current obfuscators are generally weak and why some of the better obfuscation techniques have not been implemented. Finally, we give recommendations for how better obfuscators can be created using obfuscation techniques which have not yet been implemented.

Acknowledgements I would like to thank to Siemens AG, Munich for sponsoring the work and, in particular, Barbara Fichtinger, my contact in Siemens, who gave constructive comments during the writing of this report.

Contents

Part I
Survey

1 Definitions of Code Obfuscation

The goal of software protection through code obfuscation is to transform the source code of an application to the point that it becomes "harder to understand" — which can mean it becomes unintelligible to automated program comprehension tools or the result of program analyses become less useful to a human adversary.

The motivation for protecting software through obfuscation arises from the problem of software piracy, which can be summarised as a reverse engineering process undertaken by a software pirate when stealing intellectual artefacts (such as a patented algorithm) from commercial software. Commercially popular software such as the Skype VoIP client [3], the SDC Java DRM (according to Santos *et al* [29]), and most license-control systems rely, at least in part, on obfuscation for their security. Collberg *et al* [6, 7] were the first to formally define obfuscation in terms of a semantic-preserving transformation function $\mathcal{O}$ which maps a program P to a program $\mathcal{O}(P)$. However, after the landmark result of Barak *et al* [2], which proved that every obfuscator will fail to completely obfuscate some programs, 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.

A variety of different definitions for obfuscation, and thus what "harder to understand" means, has been proposed in the literature. We will briefly look at four definitions and highlight some of the problems with each of the definitions.

1.1 Collberg *et al*

The definition of an *obfuscating transformation* from [6, Page 6] is:

> Let $P \xrightarrow{\mathcal{T}} P'$ be a transformation of a source program P into a target program P'. The transformation $P \xrightarrow{\mathcal{T}} P'$ is an *obfuscating transformation*, if P and P' have the same *observable behaviour*. More precisely, in order for $P \xrightarrow{\mathcal{T}} P'$ to be a legal obfuscating transformation the following conditions must hold:
>
> - If P fails to terminate or terminates with an error condition, then P' may or may not terminate.
> - Otherwise, P' must terminate and produce the same output as P.

The quality of an obfuscation transformation is measured in terms of various metrics which measure the complexity of code such as:

- *Cyclomatic Complexity* [22] — the complexity of a function increases with the number of predicates in the function.

- *Nesting Complexity* [15] — the complexity of a function increases with the nesting level of conditionals in the function.

- *Data-Structure Complexity* [24] — the complexity increases with the complexity of the static data structures declared in a program. For example, the complexity of an array increases with the number of dimensions and with the complexity of the element type.

Using such metrics Collberg *et al* [6] measure the *potency* of an obfuscation as follows. Let $\mathcal{T}$ be a transformation which maps a program P to a program P'. The potency of a transformation $\mathcal{T}$ with respect to the program P is defined to be:

$$\mathcal{T}_{pot}(P) = \frac{E(P')}{E(P)} - 1$$

where $E(P)$ is the complexity of P (using an appropriate metric). $\mathcal{T}$ is said to be a *potent obfuscating transformation* if $\mathcal{T}_{pot}(P) > 0$ (*i.e.* if $E(P') > E(P)$). In Collberg *et al* [6], P and P' are not required to be equally efficient — it is stated that many of the transformations given will result in P' being slower or using more memory than P.

Other properties Collberg *et al* [6] measure are:

- *Resilience* — this measures how well a transformation survives an attack from a deobfuscator. Resilience takes into account the amount of time required to construct a deobfuscator and the execution time and space actually required by the deobfuscator.

- *Execution Cost* — this measures the extra execution time and space of an obfuscated program P' compared with the original program P.

- *Quality* — this combines potency, resilience and execution cost to give an overall measure.

These three properties are measured informally on a non-numerical scale (*e.g.* for resilience, the scale is *trivial*, *weak*, *strong*, *full*, *one-way*).

Another useful measure of obfuscation is the *stealth* of an obfuscation [7]. An obfuscation is stealthy if it does not "stand out" from the rest of the program, *i.e.* it resembles the original code as much as possible. Stealth is context-sensitive — what is stealthy in one program may not be in another one and so it is difficult to quantify (as it depends on the whole program and also the experience of the reader).

The metrics mentioned above are not always suitable to measure the degree of obfuscation. Consider these two code fragments, which use predicates p and q and statements A, B and C:

$$\textsf{\textbf{if}}\ (p)\ \{A\}\ \textsf{\textbf{else}}\ \{\ \textsf{\textbf{if}}\ (q)\ \{B\}\ \textsf{\textbf{else}}\ \{C\}\} \tag{1}$$

$$\begin{aligned}&\textbf{if } (p)\ \{A\};\\&\textbf{if } (\neg p \wedge q)\ \{B\};\\&\textbf{if } (\neg p \wedge \neg q)\ \{C\}\end{aligned} \tag{2}$$

These two fragments are equivalent if the statements do not change the values of the predicates. If we transform (1) to (2) then the cyclomatic complexity is increased but the nesting complexity is decreased. Which fragment is more obfuscated?

1.2 Barak *et al*

Barak *et al* [2] takes a more formal approach to obfuscation — their notion of obfuscation is as follows. An obfuscator $\mathcal{O}$ is a "compiler" which takes as input a program P and produces a new program $\mathcal{O}(P)$ such that for every P:

- *Functionality* — $\mathcal{O}(P)$ computes the same function as P.
- *Polynomial Slowdown* — the description length and running time of $\mathcal{O}(P)$ are at most polynomially larger than that of P.
- *"Virtual black box" property* — "Anything that can be efficiently computed from $\mathcal{O}(P)$ can be efficiently computed given oracle access to P" [2, Page 2].

With this definition, Barak *et al* construct a family of functions which is unobfuscatable in the sense that there is no way of obfuscating programs that compute these functions. Thus their notion of obfuscation is impossible to achieve.

This definition of obfuscation, in particular the "Virtual Black Box" property, is too strong — obfuscators will still be of practical use even if they do not provide perfect black boxes. This definition does not give an indication of the quality of a proposed obfuscation technique (the measure says whether the transformation completely obfuscates or not). After the publication of Barak *et al* [2], the focus of obfuscation research has changed to designing obfuscations that are *difficult*, but not necessarily *impossible*, for an attacker to undo.

1.3 Assertion Obfuscations

The obfuscations discussed in the DPhil thesis of Drape [9] were defined for abstract data-types. The data-types contained a declaration of the type (including invariants), a list of operations and a list of assertions that the data-type operations satisfied. Drape observed that when considering what an operation computes, we often see what properties that operation has, *i.e.* the assertions that it satisfies. If an operation is obfuscated then it should be harder to find out what properties this operation has. Since a list of assertions is given when defining a data-type, then each the definition of each operation will have to satisfy the assertions and so a proof for each assertion will have to be constructed. If we have an obfuscated operation then the assertion proofs should be harder to construct than for unobfuscated operation. This observation lead to the following definition for obfuscation:

> Let g be an operation and $\mathcal{A}$ be an assertion that g satisfies. We obfuscate g to obtain g^O and let $\mathcal{A}^O$ be the assertion corresponding to $\mathcal{A}$ which g^O satisfies. The obfuscation O is said to be an assertion obfuscation if the proof that g^O satisfies $\mathcal{A}^O$ is more complicated than the proof that g satisfies $\mathcal{A}$.

An obfuscation metric, which tried to measure whether one proof is more complicated than another, was given. The metric used proof trees and it was validated by considering various data obfuscations. This definition only measures how well an operation is obfuscated with respect to a certain set of assertions — there is no guarantee that the operation will be obfuscated for other assertions. Also, as the metric is dependent on the way assertions are proved so if the assertions for an operation and its obfuscation are proved in different ways then the proofs cannot be compared.

1.4 Slicing Obfuscations

Program slicing as a software engineering technique was first introduced by Weiser [35] in 1980. A program slice consists of the parts of a program that potentially affect the values of some variables computed at some point of interest — the program point and the set of variables are called the slicing criterion [32]. Collberg *et al* [6] discussed how a program slicing could be used as a deobfuscation tool and potentially identify parts of an obfuscated program which are bogus.

Drape *et al* [13] proposed a way of obfuscating by first slicing a program P for a particular variable, v (say), to create a slice S_v. Rather than just considering the size of the slice, Drape *et al* instead looked at those statements of P which were not contained in S_v — which were called *orphan* statements. Obfuscations would then be added to P with the aim of reducing the number of orphaned statements when the slice S_v was recomputed. This lead to the definition of a *slicing obfuscation* — that is an obfuscation which decreases the number of orphan statements for a particular variable. Obfuscations which are particularly good slicing obfuscations are those which add dependencies between the variables in the slicing criterion.

This definition is very specific and only deals with obfuscations which impede slicing. There is no guarantee that a slicing obfuscation will be a suitable obfuscation in other contexts.

1.5 Using Metrics

We have seen that definitions for obfuscation often refer to metrics to measure the effectiveness of an obfuscation. In Section 1.1 we saw that to measure the potency of an obfuscation Collberg *et al* [6] used various software complexity measures such as Cyclomatic Complexity [22]. However these metrics by themselves cannot give a definite measure of how good an obfuscation is. In Section 1.1 we saw that a program could be obfuscated according to one metric but not another. One of the obfuscation measures mentioned by Collberg *et al* [7] is the concept of stealth. However, it is very hard to produce a metric which would directly measure the stealth of an obfuscation as it is very context-sensitive and so the metric would have to take into the whole program (or at the very least a localised portion of the program).

Nakamura *et al* [25] performed experiments to estimate the cost of mentally simulating programs. The experiments consisted of giving eight Java programmers one of two obfuscations of a particular Java program. The programmers then had to simulate the program mentally using a model (called the Virtual Mental Simulation Model) which used a queue representing short term memory. The results were then used to assess how comprehensible the programs were. As with the metrics used by Collberg *et al* it would be easy to add bogus obfuscations, to make programs appear more obfuscated. Since the metric of Nakamura *et al* relied on a queue of recently seen variable definitions we could decrease the comprehension metric by interleaving bogus variable assignments. It was also not clear whether the experience of the programmer would make a difference to the results — for example, would similar results have been obtained if experienced functional programmers had taken part?

For the assertion definition from the thesis of Drape [9] (Section 1.3) a metric for proof trees was developed. This metric measured the cost of a proof (essentially how many steps a proof takes) and the height of a proof tree. So the definition tries to measure the degree of obfuscation by considering the complexity of proofs rather than the complexity of operations. In the thesis, it was discussed what made a good assertion obfuscation by using the results of these metrics. However for a fair comparison using these measurements, all proofs have to be constructed in a consistent way (and some ways to help with the consistency were discussed). Also, we can only be sure of the quality of an obfuscation with respect with a certain of assertions — the obfuscation may not be suitable for other assertions.

The definition of slicing obfuscations of Drape *et al* [13] also used metrics to validate their definition [18]. The metrics considered, such as Maximum and Minimum Coverage [26], were ones that have used to assess the quality of program slicers. The obfuscations considered in Majumdar *et al* [18] all increased the metrics and so decreased the effectiveness of slicing.

The slicing obfuscation definition (Section 1.4) is the only definition of obfuscation that we have discussed that has an explicit attack model — namely, obfuscations were created with the knowledge that a potential adversary would use a program slicer to try to deobfuscate the program. The metrics that were used to assess the quality of the obfuscations also took into account the attack model. Other obfuscation definitions could be considered to have attack models. For instance, the assertion definition (Section 1.3) could have an attack model where an adversary is armed with a theorem prover. Given the impossibility result of Barak *et al* [2] it is unlikely to create obfuscations that can be defended against all possible attackers. Therefore when designing obfuscations, and metrics to measure the effectiveness of the obfuscations, we should be aware of what attacks they could defend against and we should use a wide range of techniques where possible to defend against different attackers.

2 Example of Obfuscating Transformations

This section gives a brief overview of some the obfuscation techniques mentioned in the taxonomy of Collberg *et al* [6] and we will also consider some techniques that have developed since the taxonomy paper was written. We will start by discussing obfuscations that fall in the three main categories given in Collberg *et al* [6], namely: **layout**, **control-flow** and **data** obfuscations. The obfuscations discussed in these categories will be applicable to a wide variety of programming languages — more language specific obfuscations will be discussed in a later section.

2.1 Layout Obfuscations

Layout Obfuscations are concerned with changing the look of a program rather than with changing the semantics. One simple obfuscation is to remove any comments that a programmer may insert as part of the documentation of the program. A more malicious approach is to insert bogus comments so that a portion of the program appears to be doing something different to what was claimed. Comments are often removed anyway when compiling/decompiling a program.

Another layout obfuscation is to change the formatting of a program, for instance, removing all of the unnecessary whitespace and indenting. There are famous examples of formatting obfuscations such as in the Obfuscated C contest [16].

A popular layout obfuscations is to change identifier names — not only for variables but also for classes, methods, *etc.* This transformation aims to change meaningful names such as "total" or "output" into names such as "a" or "ghe251c". A more malicious renaming would aim to name variables in a confusing manner — for example a variable which works out the total could be named "average" instead. Tools that use renaming also try to have as many items (including methods, fields and classes) as possible sharing the same name either by identifying the scope of a variable or by using overloading. Again this type of transformation may be undone by compiling/decompiling depending on how the compiler names its variables.

Layout obfuscations are generally not considered to be very strong as they can often be easily undone and, as mentioned earlier, they are not concerned with the semantics of a program. Despite this, layout obfuscations are often used in automatic obfuscation tools.

2.2 Control-Flow Obfuscations

The layout obfuscations discussed in the last section attempt to change the syntax of the program rather than the semantics. One set of semantic transformations are those which are targeted at the control-flow and so these transformations try to change conditional statements, jumps and loops. We can change the control-flow in a program by, for example, adding bogus conditional jumps or changing **while** loops. Majumdar *et al* [21] surveyed control-flow obfuscations and highlighted two techniques which we will discuss in Sections 2.2.1 and 2.2.5.

2.2.1 Opaque Predicates

A predicate P is *opaque* [7] at a program point s if the value of P at s is known at compile time. The notation P^T_s (P^F_s) denotes a predicate which always evaluates to True (False) at s (the program point s can be omitted if it is clear from context) and $P^?$ to denote a predicate which sometimes evaluates to True and sometimes to False. Opaque predicates can be used to create bogus code in programs, for example

$$\begin{array}{lcl} S & \Rightarrow & \textsf{\textbf{if}}\ (P^T)\ \{S\} \\ S & \Rightarrow & \textsf{\textbf{if}}\ (P^F)\ \{S_{bug}\}\ \textsf{\textbf{else}}\ \{S\} \\ S & \Rightarrow & \textsf{\textbf{if}}\ (P^?)\ \{S\}\ \textsf{\textbf{else}}\ \{S_{copy}\} \end{array}$$

This first transformation tries to disguise the fact that S will always be executed; the second uses a copy of S which contains bugs and the third uses a functionally equivalent copy of S. Opaque predicates can also be used to add bogus jumps and are particularly useful for creating irreducible flow graphs (see Section 2.2.3).

It is an open problem to generate opaque predicates which are suitable for obfuscations. Predicates can be based on known mathematical identities, for example $n^2 + 1 \neq 0 \pmod 7$ for all integral n. These identities are either quite easy prove or are so complicated that they stand out from the rest of the program (*i.e.* they are not stealthy). Also these mathematical identities tend to be true for all values of a variable and so we could test the predicate with a large set of random numbers to see that the predicate has a constant Boolean value. It would be more effective if we could have a predicate that was based on some program invariant at a particular program point. This could mean that such a predicate would be true (say) at a particular point, which is known to the obfuscator, but could be false elsewhere. We could also develop linked predicates such as the following simple block which obfuscates the assignment $x = 2$:

```
x = integer_value;
...
if  (x mod 3 = 1) {x = x + 1; }  else {x = x + 3; }
      < x is not used or modified >
if  (x mod 3 = 1) {x = 3 * x + 2; }  else {x = 2 + ((x mod 3) mod 2); }
```

The first predicate is not opaque but the second one is as it is always false. It would be harder to spot that $x = 2$ if we could separate the **if** statements by a block of code that does not use or modify x.

When creating slicing obfuscations, Majumdar *et al* [19] used opaque predicates to extend the scope of variables. Suppose that we have the following code fragment:

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

where S is a block of statements in which x may be used but not defined and F is an expression which does not depend (directly or indirectly) on x. This means that the new assignment $x = F$ kills the previous definition of x and so the backwards slice for x may not contain the block S. If we have an opaquely true predicate p then we can transform the code fragment to:

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

So, now the backwards slice for x should contain the whole code fragment.

It is important that the opaque predicates we choose are stealthy. This is so that the predicates do not "stand out" from the rest of the program and signal to an attacker that something important is been obfuscated. However we can use predicates that are not stealthy with pieces of code that are bogus (*i.e.* not executed) or unimportant. The aim of using deliberately unstealthy opaque predicates is to try to fool an attacker into believing the predicates are protecting important pieces of code.

2.2.2 Extensions of Opaque Predicates

As mentioned in the previous section, one of the limitations with using opaque predicates is that often the value of an opaque predicate is true for all possible inputs. This lead Palsberg *et al* [27] to developing *dynamically* opaque predicates. These are a set of predicates which all evaluate to the same result in any given run, but in different runs they may evaluate to different values. For example, if we had a block of three statements $S_1;\ S_2;\ S_3$ and two linked predicate p and q (which evaluate to the same value in the same run) then we can obfuscate the block as follows:

$$\begin{array}{c}\textsf{\textbf{if}}\ (p)\ S_1;\ \textsf{\textbf{else}}\ \{S_1;\ S_2;\}\\ \textsf{\textbf{if}}\ (q)\ \{S_2;\ S_3;\}\ \textsf{\textbf{else}}\ S_3;\end{array}$$

This transformation is valid under certain conditions such as ensuring that the statements S_1 and S_2 do not change the value of q. We can easily extend this to obfuscate larger blocks with a bigger set of dynamically opaque predicates but the conditions for ensuring that the transformation is valid become more complex.

Palsberg *et al* [27] mention that it is advantageous to use both traditional opaque predicate and dynamically opaque predicate when obfuscating. This is because it will be harder for an attacker to know whether a certain predicate is opaque, part of a set of dynamically opaque predicate or one that comes from the original, unobfuscated program.

Majumdar and Thomborson [20] proposed creating *temporally unstable* opaque predicates for distributed environments using values from an aliased data structure. A temporally unstable opaque predicate can be evaluated at different program points during an execution run and the values from this predicate are not identical. This means that the same predicate can be used to obfuscate different portions of control-flow.

2.2.3 Irreducible flow graphs

In Collberg *et al* [6] there is a discussion about how opaque predicates can be used to create bogus statements which appear to jump into the middle of a loop — thus creating an *irreducible* flow graph. For example:

```
if ($P^F$) {goto $L$;}
...
while ($C$)
{   ...
    $L$ : ...   }
```

Languages such as C# and Java do not permit such jumps to be written. However this type of jump would be allowed in the intermediate languages (CIL or Bytecode) — see Section 2.4.4. Thus we could write a program in the high-level language, convert to intermediate language where we add the bogus jump and then recompile. If a decompiler tried to re-construct the high-level code then it would have to produce more complicated code as it tries to turn the irreducible jump into a reducible one. This type of transformation is an example of a *language breaking* transformation. A possible specification of this transformation at the intermediate language level was described in the thesis of Drape [9].

2.2.4 Loop Transformations

There are many transformations we can do to a loop which has the form:

```
while (condition)
{   body   }
```

One such transformation is to change one of the induction variables using an encoding (see Section 2.3.2). For example, suppose we had the loop:

```
$i = 0$;
while ($i < N$)
{   ...$i$...
    $i = i + 1$;   }
```

and using the variable encoding $\lambda\, x.(2x + 1)$ we can transform the loop to:

```
$i = 1$;
while ($i < 2 * N + 1$)
{   ...($(i - 1)/2$)...
    $i = i + 2$;      }
```

We can add a bogus induction variable into the loop which can make the loop conditions more complicated. For example, if we had the loop:

```
$i = 0$;
while ($i < 10$)
```

```
{   ...
    $i = i + 1;$    }
```

and assuming that the value of i is only changed at the end of the loop then we can add a variable j into the loop as follows:

```
$i = 0;$
$j = 1;$
while $((i < 10)$ && $(j < 120)$
{   ...
    $i = i + 1;$
    $j = j + 2 * i;$    }
```

If we assume that i and j are only modified at the end of the loop then the loop terminates when $i = 10$ as before but the proof of the termination value is harder now than it was before the transformation. The condition for j (*i.e.* $j < 120$) is an opaque predicate for loop as we know that it is true throughout the whole of the execution of the loop (but maybe not true outside the loop). We can also use bogus induction variables to create bogus dependencies for other variables — this type of transformation is useful when creating slicing obfuscations [19].

A program can contains many loops and often the same variable is used for an induction variable. The scope of an induction variable is usually just the loop itself as the variable is initialised before the start of the loop and not used outside the loop. If we have two loops that use the same variable as an induction variable (and the variable is not used outside the loops):

```
$i = 0;$
while $(i < M)$
{   ...
    $i = i + 1;$    }
    < $i$ is not modified here >
$i = 0;$
while $(i < N)$
{   ...
    $i = i + 2;$    }
```

We can extend the scope of the induction variable by changing the initialisation of the second loop:

```
$i = 0;$
while $(i < M)$
{   ...
```

```
        $i = i + 1;$    }
        $< i$ is not modified here $>$
    $i = i - M;$
    while ($i < N$)
    {   ...
        $i = i + 2;$    }
```

The value of i in the second initialisation is still 0 since we know at the end of the first loop that $i = M$. We could also change the first loop so that it decreased, making the value of i at the end of loop be 0. Thus we would not need an initialisation of i for the second loop. If the first loop used a different induction variable j (say) then providing it was safe to do so (for example, ensuring that j is not used after the first loop) we could make both loops use the same induction variable i. Again, we should try to remove the initialisation to i in the second loop. Extending the scope of the induction variables tries to ensure that the first loop would be included in a backwards slice for i (as used in Majumdar *et al* [18]).

Other loop transformations mentioned in Collberg *et al* [6] include loop fission, which attempts to split up one loop which has more one than one statement in the loop body into several separate loops, blocking, which tries to repeatedly split the iteration of the loop into an inner and outer loop, and loop unrolling, which repeats a loop body one or more times. These loop transformations can be used as optimisations by a compilers — for example, loop blocking can be used to improve cache behaviour. So we may find that some of these loop transformations can be undone in the compilation process.

When creating obfuscations to restrict the usefulness of program slicing, Majumdar *et al* [19] found that loop transformations were helpful in decreasing the effectiveness of slicing but not every program uses a loop. However, Majumdar *et al* [19] discuss how it is possible to "fake" a loop. Suppose that we had a block of code B and state before B is σ. Then we need to find a predicate p for which $p(\sigma)$ is true but $p(B(\sigma))$ is false. If we have such a predicate then B can be transformed to

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

2.2.5 Control-Flow flattening

In Wang *et al* [34], a technique for flattening control-flow is described. The idea of this transformation is to remove control-flow constructs such as **while** loops so that all basic blocks have the same predecessor and successor in the control-flow graph. So, for example, suppose we have this piece of code:

```
init;
while (cond)
{   loop_body;   }
```

Then we could remove the **while** loop and replace it with a **switch** block which loops until an **end** statement is reached:

```
$var = 1$;
switch ($var$)
{   case 1 :
            init; $var = 2$; break;
    case 2 :
            if ($cond$) $var = 3$; else $var = 4$; break;
    case 3 :
            loop_body; $var = 2$; break;
    case 4 :
            $var = 1$; end;      }
```

The variable *var* acts as a dispatcher and effectively controls the execution of the blocks. Each case contains an assignment to *var* (and in the last case the assignment is a dummy one).

Here is a more complicated example which has conditional statements:

```
init;
while ($cond$)
{   loop_body;
    if ($test$) stat1; else stat2;   }
```

Again, we can use a **switch** block to convert the program to:

```
$var = 1$;
switch ($var$)
{   case 1 :
            init; $var = 2$; break;
    case 2 :
            if ($cond$) $var = 3$; else $var = 7$; break;
    case 3 :
            loop_body; $var = 4$; break;
  case 4 :
          if ($test$) $var = 5$; else $var = 6$; break;
  case 5 :
            stat1; $var = 2$; break;
    case 6 :
            stat2; $var = 2$; break;
    case 7 :
            $var = 1$;  end;        }
```

It is quite easy to reconstruct the conditional statement and the **while** loop from the **switch** block. So, Wang *et al* [34] describe some extra levels of obfuscation. First, some dummy cases can be added to the **switch** to obscure the control-flow — for instance, irreducible jumps could be added (Section 2.2.3). Also, a global variable, such as an array, can be used to dynamically determine the values of the dispatcher variable. Finally, pointers can be used so that static analysis of the control-flow becomes difficult to perform.

Madous *et al* [17] created a rewriting framework called DIABLO which contains a control-flow flattening transformation which is applied to x86 code. Another technique is described in Chow *et al* [4]. The transformation in Chow *et al* consists of a number of different steps including splitting basic blocks into pieces, introducing dummy pieces, renaming variables and forming “lumps” (essentially sequences of instructions with particular properties). As with the flattening technique of Wang *et al* , the method of Chow *et al* uses a dispatcher to control the progression between the various lumps. In this case, the dispatcher can be modelled as a deterministic finite automata in which the flow is specified using a specific state space. To ensure that the is hard to determine, the dispatcher state space is expanded by including dummy states.

2.2.6 Method Transformations

Most of the transformations that we have seen so far have been quite localised and usually affect just one method. In Collberg *et al* [6] there are many transformations we can apply to methods themselves:

- **Inline Methods** We can replace a method call with the body of the method. If a method name is overloaded then we may have to include code which checks the type of the method and the parameters.

- **Outline Methods** We can make a sequence of statements into a method by replacing the statements with a method call.

- **Clone Methods** We can create many copies of the same methods by applying different obfuscations (and other transformations). At the method call, we can choose which clone to call from a set of clones. If we have different calls to the original method in the program then we can replace each call with calls to a different set of clone methods.

- **Interleave Methods** We can merge two separate methods into one method. At the call sites for each method we replace the call to the original method with a parameter, which is used to distinguish between the two original methods, and a call to the new method.

As well as using these transformations by themselves, we can combine these transformations to give more complicated obfuscations. For example, we could inline a method m

into the calling method c and then we could outline a different block from c into a new method m'.

2.3 Data Obfuscations

As well as obfuscating the control-flow, we can also obfuscate the data and the data structures that a program may use. We may transform a variable individually by using variable encodings (Section 2.3.2) or we may change the whole structure of a data-type (see, for example, the array transformations in Section 2.3.4).

2.3.1 Specifying Data Obfuscations

In the thesis of Drape [9], the idea of obfuscating abstract data-types was introduced. The abstract data-types consist of a module containing a declaration of the data-type and the operations that are implemented in the data-type. This abstraction matches the object-oriented view of classes, constructors and methods. By obfuscating abstract data-types we can consider the operations more generally without worrying about implementation issues. The abstract data-types in Drape's thesis [9] were specified using a functional language and obfuscation was described in terms of a data refinement [8].

In Drape *et al* [14], this work was extended to develop a framework for specifying imperative data obfuscations. A data obfuscation $\mathcal{O}$ can be specified by defining two functions af, the abstraction function, and cf, the conversion function, which satisfy $cf;\ af \equiv skip$. The conversion function is a statement (or a block of statements) which performs the obfuscation and the abstraction function undoes the obfuscation. Drape *et al* then showed how to obfuscate assignments, conditional statements and loops. To obfuscate a block of statements we can obfuscate each statement in the block and then compose the results. If a block B is obfuscated, using the functions cf and af, to obtain $\mathcal{O}(B)$ then

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

This equation gives us a way of proving the correctness of the obfuscation of B.

Another benefit of using this alternative formulation is that we can precisely specify where we want to apply a data obfuscation. We can localise our transformation to a specific block. For example, if our program had three code blocks $A;\ B;\ C$ then we can apply our obfuscation $\mathcal{O}$ to just block B and so

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

2.3.2 Variable Encoding

The idea of a variable *encoding* (or transformation) is to change a variable into an expression. For example,

$$i \Rightarrow a * i + b$$

where a and b are constants. The encoding needs to be invertible so that the "correct" value of the variable can be obtained if needed (for example, if the value of the variable is output or is needed for the computation of another variable). Under an encoding we need to transform both uses and definitions of i separately. For example, using the transformation above,

$$i = 2 \quad \Longrightarrow \quad i = a * 2 \qquad\qquad j = i + 1 \quad \Longrightarrow \quad j = \frac{i-b}{a} + 1$$

An expression such as $i++$ is both a definition and a use and so

$$i++ \quad \Longrightarrow \quad a * \left(\frac{i-b}{a} + 1\right) + b$$

which (using exact arithmetic) can be simplified to

$$i = i + a$$

In general (from the thesis of Drape [9]), we can encode a variable i by using an encoding function f and we also require a function g such that $g \cdot f = id$. We have two different rewrite rules:

- a use of i is replaced by $g(i)$
- an assignment to i of the form $i = E$ is replaced by $i = f(E')$ where $E' = E[g(i) \diagup i]$

where $\diagup$ denotes variable substitution. The rewrite rule for assignment covers the case where the variable i appears in the right-hand side of the assignment. An implementation of this transformation for intermediate language was described in Drape *et al* [11].

Using the specification discussed in Section 2.3.1 the encoding above can be specified as:

$$cf \equiv i = a * i + b \qquad\qquad af \equiv i = \frac{i-b}{a}$$

In terms of the functions f and g mentioned above, we have:

$$cf \equiv i = f(i) \qquad\qquad af \equiv i = g(i)$$

and the condition $g \cdot f = id$ is equivalent to $cf;\ af \equiv skip$.

A stronger variant of this transformation (given in Drape *et al* [12]) uses two variables so that a variable seems to depend on the value of another variable. For example

$$i \quad \Longrightarrow \quad a * i + b * j$$

where j is another variable and a and b are constants. Using the specification of Drape *et al* [14], the conversion and abstraction functions are:

$$cf \equiv i = a * i + b * j \qquad\qquad af \equiv i = \frac{i - b * j}{a}$$

The rewrite rules for this encoding are more complicated because whenever we have a definition for j then we must also have a definition of i. The rewrite rules are as follows:

- (use of i) $U(i) \implies U(\frac{i-b*j}{a})$
- (def of i) $i = E \implies i = a*E' + b*j$ where $E' = E[\frac{i-b*j}{a} \,/\, i]$
- (def of j) $j = f(i) \implies \begin{Bmatrix} t = i - b*j; \\ j = f(t/a); \\ i = t + b*j; \end{Bmatrix}$

This transformation is useful for creating false dependencies and can be used to reduce the effectiveness of program slicing [19].

As can be seen above, this obfuscation relies on exact arithmetic to obtain the true value of the variable when needed. When using this obfuscation we must ensure that the value of the variable does not overflow. A problem for automation is deciding on criteria to pick a suitable variable for encoding. If, for example, the most frequently occurring variable is picked then there will be a corresponding slow-down of the execution time as there will be many more arithmetic operations.

2.3.3 Merging and Splitting

As well as obfuscating individual variables, we can also work with obfuscating a number of variables together. Collberg *et al* [6] discuss *merging variables* where we can merge two (or more) variables into one. Suppose that we want to merge two variables x and y. If we know that $0 \leqslant x < N$ and $y \geqslant 0$ then we can define z as follows:

$$z = N*y + x$$

As well as merging two or more variables, we can split up one variable in two (or more) other variables. Collberg *et al* [6] shows how Boolean variables can be split. In Drape *et al* [14] an integer variable x is split into two variables a and b such that

$$a = x \text{ div } 10 \quad \text{and} \quad b = x \text{ mod } 10$$

For example, under this transformation, the statement $x{+}{+}$ is transformed to:

$$\begin{array}{l} a = (10*a + b + 1) \text{ div } 10; \\ b = (b+1) \text{ mod } 10; \end{array}$$

These assignments are equivalent to:

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

The proof of correctness, using the specification discussed in Section 2.3.1, for this equivalence is given in Drape *et al* [14].

2.3.4 Array Transformations

As well as obfuscating a single variable (or a small set of variables) we can obfuscate arrays. Before performing array transformations, we must ensure that the arrays are safe to transform. For example, we may require that a whole array is not passed to another method or that elements of the array do not throw exceptions.

One of the simplest ways to obfuscate arrays is by changing the array indices. Such a change could be achieved either by a variable encoding (such as in Section 2.3.2) or by defining a permutation. Two other transformations which obfuscates array indices are called *folding* and *flattening*. So, for example, we could fold a 1-dimensional array of size $m \times n$ into a 2-dimensional array of size $[m, n]$. Similarly we could flatten an n-dimensional array into a 1-dimensional array. These transformations can be defined by using the variable merging (for array flattening) and variable splitting (for array folding) techniques mentioned in Section 2.3.3.

The array transformation that we have seen so far effectively just encoded the array index. Next we will consider some transformations that change the structure of one or more arrays.

Array Splitting Collberg *et al* [6] give an example of a structural transformation called an *array split*:

$$\begin{array}{lcl}
\textbf{int}\ [\,]\ A = \textbf{new int}\ [10]; & & \textbf{int}\ [\,]\ A1 = \textbf{new int}\ [5]; \\
 & & \textbf{int}\ [\,]\ A2 = \textbf{new int}\ [5]; \\
\ldots & \Rightarrow & \ldots \\
A[i] = \ldots; & & \textbf{if}\ ((i\,\%\,2) == 0)\ A1[i/2] = \ldots; \\
 & & \qquad\qquad\quad \textbf{else}\ A2[i/2] = \ldots;
\end{array}$$

This transformation was generalised in Drape [10]. An array split in which an array A of size n is broken up into two other arrays B_1 and B_2 of sizes m_1 and m_2 respectively (where $m_1 + m_2 \geqslant n$) can be specified by defining three functions. The types of the functions are as follows:

$$\begin{array}{rcl}
ch :: [0..n) & \rightarrow & \mathbb{B} \\
f_1 :: [0..n) & \rightarrow & [0..m_1) \\
f_2 :: [0..n) & \rightarrow & [0..m_2)
\end{array}$$

Then the relationship between A and B_1 and B_2 is given by the following rule:

$$A[i] = \begin{cases} B_1[f_1(i)] & \text{if } ch(i) \text{ is true} \\ B_2[f_2(i)] & \text{otherwise} \end{cases}$$

To ensure that there are no index clashes we require that f_1 is injective for the values for which ch is true (similarly for f_2). A specification of this transformation for intermediate language was described in Drape *et al* [11].

This relationship between the original array and the two split arrays can be generalised so that A could be split between more than two arrays. For this, ch should be regarded as a choice function which will determine which array each element should be transformed to. The transformation is further generalised in Drape [10] by defining splits for more general data-types such as lists and matrices. Using these generalisations gives us more scope for obfuscation. For example, we could fold an array into a matrix, then use a matrix split and finally flatten the individual split matrices — this would give us a new way of splitting an array.

Array Merging We can the reverse the process of splitting an array by merging two (or more) arrays into one larger array. As with a split, we will need to determine the order of the elements in the new array.

Let us give a simple example of a merge. Suppose we have arrays B_1 of size m_1 and B_2 of size m_2 and a new array A of size $m_1 + m_2$. We can define a relationship between the arrays as follows:

$$A[i] = \begin{cases} B_1[i] & \text{if } i < m_1 \\ B_2[i - m_1] & \text{if } i \geq m_1 \end{cases}$$

This transformation is analogous to the concatenation of two lists.

2.3.5 Other data obfuscations

Some other data obfuscations mentioned in Collberg *et al* [6] are:

- **Variable Promotion** In an object-oriented language, we can promote a local integer variable to an object. Then it would be possible to reuse the object in a variety of unrelated methods (assuming the value of the variable is not needed between different methods) and so make it appear that there is a relationship between the methods.

- **Variable Scope** We can change the scope of variable so that, for example, we make a local variable global. As with variable promotion we can try to link up unrelated methods by using this new global variable. Extending the scope of variables is important when trying to create slicing obfuscations [18] — for example, in Section 2.2.4 we saw that we can try to re-use the same induction variable.

2.4 Language dependent transformations

Most of the obfuscations mentioned so far can be applied to a variety of language paradigms. We will briefly look at some obfuscations that require specific language features.

2.4.1 Exceptions

Many programming languages have features which allow the user to add in code which can deal with any exceptions that may be thrown. We can use exceptions to change the control-flow of programs. For example, suppose we had the following loop:

$$i = 0;$$

```
while ($i < N$)
{   loop code
    $i++$;   }
after loop
```

Then using a fresh variable s we can change it to:

```
try
{   $i = 0$;
    $s = 1$;
    while ($true$)
    {   $s = s + s/(N - i)$;
        loop code
        $i++$;   }
}
catch ($DivideByZero$)
{   $s = dummy$;   }
finally
{   after loop   }
```

We can replace the predicate *true* with some other predicate that is true for the loop but maybe false otherwise — this is to make it less obvious that the loop will only terminate by throwing an exception.

We can also use opaque predicates (Section 2.2.1) with **try**/**catch** blocks. For instance, suppose we had the following code fragment:

```
stat$_1$;
stat$_2$;
```

Then using the false predicate p we can transform it to:

```
try
{   if ($p^F$) {throw $error$} else stat$_1$;   }
catch ($error$)
{   bogus code   }
finally
{   stat$_2$;   }
```

When using **try**/**catch** blocks we must make sure that no other uncaught exceptions are thrown. However we could choose to throw a number of different exceptions which can all be handled by different **catch** blocks — thus we could obfuscate a **switch** block by a series of **try**/**catch** blocks.

2.4.2 Object-oriented Transformations

Collberg *et al* [6] discuss some transformations which are suitable to obfuscate object-oriented programs. Many Java programs rely on calls to standard libraries but we cannot obfuscate these calls. Instead some of these libraries calls could be implemented in the program itself which we can then obfuscate separately.

One series of transformations mentioned in Collberg *et al* [6] aim to modify the inheritance relations between different classes by inserting a bogus class or by refactoring. Refactoring is a technique for finding elements which are common to classes and then moving these features to a new class. For obfuscation, false refactoring could be performed on classes which have no common behaviour.

We could also use variations of the method transformations described in Section 2.2.6 to obfuscate classes and in Section 2.3.5 we described some transformations that could be used to obfuscate fields.

2.4.3 Pointers

We can add pointers to a program to help to obfuscate a program since performing accurate alias analyses is known to be a hard problem. We saw in Section 2.2.5 that the control-flow flattening technique of Wang *et al* [34] is strengthened by the presence of pointers. Collberg *et al* [5] and Palsberg *el al* [27] discuss a number of ways in which pointer structures can be used to create watermarks, for example, by encoding a number as a linked list or as a tree. Once these structures are created then they could be used to generate opaque predicates as the pointer is known at obfuscation time and should be unchanged throughout the execution of the program.

2.4.4 Intermediate Language

All of the example obfuscations we have seen have been targeted at high-level programming languages. However many of the obfuscations can also be applied to intermediate languages such as Java Bytecode and .NET CIL (Common Intermediate Language). Both of these languages are stack based and are the compilation targets for different source languages such as Java and C#.

An advantage of using intermediate languages is that we can add obfuscations that may not allowed in the source language. For instance, in Section 2.2.3 we saw that we can add irreducible jumps to confuse the control-flow. In Java and C#, jumps into loops are not allowed but they can be added at the intermediate language level.

There may also be instructions in the intermediate language that do not have a direct translation back into the higher-level language. For example, CIL has a type called a *typed reference* which creates an object which contains a pointer and a type — the command `mkrefany` creates a typed reference and the commands `refanyval` and `refanytype` unpacks the value and type from the typed reference. These types cannot be easily converted back into legal C# code and so the use of typed references may break a C# decompiler. So, for example, if we want to obfuscate the local variable `V_0` with a typed reference then we need a variable `V_1` (say) which has type `typedref`. We replace a store instruction `stloc.0` with

```
stloc.0
ldloca.s   V_0
mkrefany   int32
stloc.1
```

This code block stores the value into `V_0` and then puts the address of `V_0` and the type `int32` onto the stack which are stored into the variable `V_1`. For a use of `V_0` we replace `ldloc.0` with

```
ldloc.1
refanyval int32
ldind.i4
```

The command `ldind.i4` loads the value of type `int32` (`i4` means 4-byte integers) from an address onto the stack.

It is hard to manually add obfuscations to intermediate languages, particularly those that aim to obfuscated loops as loops are written using jumps and conditionals. However there are tools that automatically add obfuscations to intermediate language such as DashO [30] and Dotfuscator [31].

2.5 Different Classifications

In the previous sections we have discussed various obfuscations according to one of the categories used by Collberg *et al* [6] — namely *layout*, *control-flow* and *data* obfuscations. It has been quite easy to split up our obfuscations using this classification. As some obfuscations can belong to more than one category — for example, most obfuscations change the program layout and an array split changes the control-flow as well as changing the program data — we have classified each obfuscation according to its primary effect. So an array split is a data obfuscation as it primarily obfuscates the program data by changing a data-type. This classification also determines what metrics we should use to measure the effectiveness of an obfuscation. For instance, if we have a data obfuscation then we should assess its effectiveness using a metric which somehow measures the data in a program (such as Data-Structure Complexity [24]) rather than one which measures the control-flow. Thus as well as partitioning the obfuscations, this classification also instructs us on which metrics we should use to assess the quality of an obfuscation.

Collberg *et al* [6] subdivided the control-flow and data obfuscations into subcategories such as *aggregation* transforms (those which break up or merge computations) and *ordering* transformations (those which change the order of computations). These subcategories does not appear to as useful as the overall classification as it is harder to completely classify some obfuscations using these subcategories. For example, an array split can be seen to be an aggregation transformation and an ordering transformation as the data stored in the array is broken up into different pieces and in a different order.

Another category of obfuscations discussed in Collberg *et al* [6] are *Preventive Transformations*. These obfuscations essentially show us how to defend against a specific attack model — in particular against automatic deobfuscation techniques such as slicing and statistical analysis. The slicing obfuscations in Drape *et al* [13] use both data and control-flow obfuscations to defend against attacks from program slicers. Thus we could classify our obfuscations according to whether they defended against various attacks.

Another obfuscation classification could be obtaining by grouping the obfuscations according to how they affected various software metrics. For example, if we consider Data-Structure Complexity [24] from the Collberg *et al* definition (Section 1.1) then array transformations such as splits and folds would increase this measure. This type of classification would not split up obfuscations into disjoint groups as, for example, the array split will also increase the cyclomatic complexity [22]. Five slice-based metrics were proposed by Meyers and Binkley [23] as measures of the quality of software. Majumdar *et al* [18] used these metrics to assess the quality of obfuscations, such as inserting bogus predicates, which were designed to reduce the effectiveness of program slicing. Thus the obfuscations used by Majumdar *et al* could grouped together using a metrics classification. We can also classify obfuscations according to the assertion definition of Drape [9] (Section 1.3). A metrics classification would be similar to an attacks classification as we often use metrics to measure the effectiveness of attacks.

Part II
Evaluation

3 *Intellectual Property Protection* report

We now discuss the project report "Intellectual Property Protection" written by Stefan Vogl, Emanuel Mathis, Martin Ortner and Georg Schönberger from the University of Applied Sciences of Upper Austria, Hagenberg [33]. This report was written as part of a semester project on behalf of Siemens and, henceforth, will be referred to as the *IPP report*.

The report considers various protection methods, including obfuscations, which aim to protect the Intellectual Property of programs. Three different programming languages (and associated lower level languages) were considered: ANSI-C (with MS-DOS), Java (with Bytecode) and C# (with .NET CIL).

We will summarise the main findings of the report and discuss the protection techniques consider with a particular focus on obfuscation techniques. We will concentrate on the methods rather than the specific protection tools used — further details of the tools can be found in the report [33].

3.1 Measurements

For each of the protection methods, the IPP report uses a variety of measures to judge the effectiveness of each of the protections. The report uses the following measures to answer different questions on a five point scale:

- **Ease of Integration** How easy is it to protect an application? Is the protection process automatable or scriptable? (1 means "very easy")
- **Complexity** What is the level of protection? How hard is it to read and understand the protected code? How hard is it to automate a process which will reverse the protection? (1 means "low complexity")
- **Maintainability** Is it possible to apply patches and hotfixes? (1 means "hard to maintain")
- **Mutation of code** How much code has been modified or appended? (1 means "little change")
- **Error Tracing** How easy is it to trace errors/exceptions? (1 means "hard to trace")
- **Antivirus behaviour** Does the protected file appear suspicious to virus scanners? (1 means "very suspicious")

These properties were measured on a five point scale and we summarise the results for each of the protection methods that we will discuss.

As well as these measures, for each protection method, the IPP report also discussed *Functionality Impact* — measured on a three point scale (None, Partial and Full) — *File Size* and *Known attacks* (in particular, whether the protection has been defeated).

3.2 Protection Methods for C and DOS

The IPP report first considers programs written in ANSI-C and the associated PE (portable executable) files that are created when compiling the programs. The PEs were then disassembled to an MS-DOS assembler language and the program protections were applied at this level. This means that the protections are used at quite a low-level and thus most of the obfuscations described in Part I would not be suitable as protections for the disassembled code.

3.2.1 Assembler Obfuscation

The IPP report discusses some obfuscations which can be applied to assembler code. It is possible to interleave different instructions, if the machine architecture supports pipelining, so that the execution of multiple instructions can be overlapped. Since this process is often performed by compilers it would be hard to add obfuscations that would deliberately interleave instructions as the compiler may remove these transformations in the compilation process. Similarly, the use of "non-intuitive" instructions was discussed

— for example using a left shift instead of multiplying by a power of two — but the compiler may perform such transformations if they speed up the execution of the code.

An example of a control-flow transformation was discussed next. Suppose that we have that following piece of code:

```
main()
{   FunctionA();
    FunctionB();   }

FunctionA()
{   FuncAPart1();
    FuncAPart2();
    FuncAPart3();   }

FunctionB()
{   FuncBPart1();
    FuncBPart2();
    FuncBPart3();   }
```

This can be transformed to

```
main:
         jmp FAP1
  FBP3:  call FunBPart3
         jmp end
  FBP1:  call FuncBPart1
         jmp FBP2
  FAP2:  call FuncAPart2
         jmp FAP3
  FBP2:  call FuncBPart2
         jmp FBP3
  FAP1:  call FuncAPart1
         jmp FAP2
  FAP3:  call FunAPart3
         jmp FBP1
end:
```

and we could rename the labels and the function calls to make the order of execution less obvious. The kind of transformation is similar to the Method Interleaving transformation that was discussed in Section 2.2.6. A further transformation was be made to the code by adding opaque predicates (see Section 2.2.1). So, for example, the instruction `jmp FAP1` above was transformed to

```
cmp 1,2
jnz FAP1
```

which compares 1 and 2 and then performs a jumps if the two values are not equal — the other predicates were of a similar form. The predicates given where very trivial and it is possible that an optimising compiler may remove such predicates as the jumps can be statically determined. Adding more resilient predicates into assembly language is hard to do since it is a low-level language.

Another assembly language obfuscation discussed was token renaming, which was discussed in Section 2.1. The report shows an example using a renaming transformation. In C, the code looks very different after the transformation and is much harder to read. But the new code requires an extra header and so many of the renaming for functions, such as `printf`, can be seen. However, at assembly language level, there are some similarities between the code before and after the transformations. Some of the standard function calls are replaced by calls to renamed methods but as some of the output strings are visible then it is possible to determine which calls represents `printf`. There is also no guarantee that these renamings would survive the compilation/decompilation process.

The obfuscations which were used to try to protect assembly code were fairly trivial and they do not provide sufficient protection as the obfuscations could be removed by an optimising compiler. As stated earlier, it is hard to add strong obfuscations to such a low-level language.

3.3 Other protection methods

We now briefly describe some of the other (non-obfuscation) protection methods for assembly code that were discussed in the IPP report.

- **Code Virtualisation** changes commands that are understood by (say) x86 processors into commands that can only be read by custom virtual machines. The report discuss how code virtualisation makes the job of a reverse engineer more difficult but sometimes it is possible to figure out some details of the code as strings and constants are not changed. Code virtualisation can slow down the performance of code as the code has to be transformed from the virtual machine to the real processor.

- An application called a **PE Protector** uses a variety of protection techniques such as optimisations and compressions as well as tricks to hinder debugging and tracing. Although the application uses a variety of methods, the report mentions that tutorials exist showing how to reverse the protection.

Measures (described in Section 3.1) for the three protection methods we have discussed can be seen in Table 1.

The methods used to protect C programs were used on a specific assembly language in MS-DOS. The report does not really consider whether different operating systems or compilers would make a difference to the protection methods.

Metric	**Assembler**	**Code Virtualisation**	**PE Protector**
Ease of Integration	2	3	3
Complexity	1	3	4
Maintainability	5	1	1
Mutation of Code	2	5	4
Error Tracing	4	4	1
Anti-Virus Behaviour	5	5	2

Table 1: Metric measures for the C protection methods

4 Protection Methods for C# and Java

Programs written in Java and C# (and other languages for the .NET framework) are compiled into an intermediate language, for Java the intermediate language is called *Java Bytecode* and for .NET it is called the *Common Intermediate Language* (CIL). Compiled programs can be distributed to a variety of platforms and architecture as it is only necessary to have a virtual machine (such as the JVM) which can Just-In-Time compile the intermediate language to the native machine code.

The intermediate languages are typed, stack-based languages and are human readable (although CIL is more complicated than Bytecode as CIL is designed to be the compilation target for a variety of languages). It is possible to decompile the intermediate language back into a higher-level language, for example, using a tool such as the Salamander .NET decompiler [28]. Thus if we want to prevent reverse engineering then we have to protect our code. The IPP report examines various protection methods which were applied at the intermediate language level. The attack model for these protection methods is that it is assumed that an attacker is armed with an automatic decompilation tool and so it is the aim of the protection methods to make the decompilation process harder.

4.1 Obfuscation tools for C# and Java

First, we will briefly look at the some of the obfuscations for Java and .NET which were discussed in the IPP report. Measures (described in Section 3.1) for the methods in Java can be seen in Table 2 and for those in .NET can be seen in Table 3.

4.1.1 Shrinking

For Java, a technique, called *shrinking*, was considered as a protection method. The aim of shrinking is to remove information, such as debugging information, from bytecode which is not needed to execute the program. This transformation can be considered as a layout obfuscation (see Section 2.1) as the main aim of the transformation is to change the “look” of the program rather its flow or data structures. The IPP report show an example of a `main` method which had the shrinking transformation applied to it. After

Metric	**Shrinking**	**Name**	**String**	**Flow**
Ease of Integration	5	4	2	2
Complexity	4	2	3	3
Maintainability	3	3	5	5
Mutation of Code	4	3	3	3
Error Tracing	2	3	5	5
Anti-Virus Behaviour	5	5	4	5

Table 2: Metric measures for the Java protection methods

Metric	**Name**	**String**	**Flow**
Ease of Integration	3	5	5
Complexity	2	3	3
Maintainability	5	4	1
Mutation of Code	1	1	3
Error Tracing	3	1	2
Anti-Virus Behaviour	5	5	5

Table 3: Metric measures for the .NET protection methods

shrinking, some of the methods in the original class do not seem to appear in the new class (because they have been removed from the variable table), some names are renamed and it was not possible to fully decompile the bytecode into Java. However, analysing the bytecode instructions for the original and shrunken `main` method shows that there is little difference between the two methods. This means that an attacker can still see what the program does by inspecting the bytecode even if the program cannot be decompiled.

4.1.2 Name Obfuscation

A protection method that was applied to both Java and .NET programs was called *Name Obfuscation* — which we mentioned as a layout obfuscation in Section 2.1. Name obfuscations aim to change meaningful identifier names into ones which do not help an attacker understand what a program does. This type of transformation can be used to try to protect "sensitive" areas of a program such as those that deal with license checks or password verifications.

The IPP report applied renaming tools to example programs at the intermediate language level. After renaming, the example programs could be completely decompiled. Comparing the before and after programs, it could be seen that the structure of the programs was the same. The only significant difference was that the some methods were renamed to the same method name — this is possible if the methods have different type signatures.

The IPP report noted that it is not possible to rename standard library calls such as routines which print text to the screen. It is also not possible to rename methods or variables that may be used outside the obfuscation scope and so we may have problems renaming public methods.

4.1.3 String Encryption

Identifier names can often useful information to an attacker wanting to understand what a program does. In the previous section, we mentioned a transformation that aimed to change the names of identifiers as names can often give hints to attackers. An attacker may also consider the text strings of a program to try to understand what a program does. As mentioned earlier, variable renaming is useful to protect "sensitive" program areas but we should also transform strings (such as "Please enter your password") which give information to an attacker.

The IPP report discussed the use of *string encryption* as a protection method and showed some example transformations. A string encryption transformation aims to replace all the strings (and the associated calls to a **print** method) with a call to a new method. Each time this new method is called a parameter is used so that the appropriate string can be outputted. This transformation can be considered as a data obfuscation (see Section 2.3).

Since all the strings need to be printed out correctly the obfuscated program will contain a method which handles the decryption and formatting of the strings. If an attacker can find this method then it would be possible for the attacker to change the code and force all of the strings to be decrypted and printed out. This means that an attacker could use these strings to find any areas of interest (such as passwords checks). Thus string encryption alone is not sufficient to protect a program.

4.1.4 Flow Obfuscations

None of the transformations considered so far for Java and .NET have significantly changed the structure of the programs. The IPP report applied flow transformations to some **for** loops written in Java and C#.

A **for** loop such as

```
for (int i = 0; i < N; i++)
{ [body] }
[after loop]
```

is transformed into something like this in .NET CIL:

```
.method example()
{
  .locals init(
      [0] int32 i,
      [1] int32 N,
      ...      )
  L_11: ldc.i4.0
  L_12: stloc.0
  L_13: br_s L_09
  L_21: [body]
        ...
  L_31: ldloc.0
  L_32: ldc.i4.1
  L_33: add
  L_34: stloc.0
  L_41: ldloc.0
  L_42: ldloc.1
  L_43: ble.s L_04
  L_51: [after loop]
```

A loop in intermediate language consists of four parts. The first part (labels L_11 to L_12) consists of the initialisation of the loop which then jumps to the test. The second part (L_11) is the main loop body and the third part is the increment of the induction variable (in labels L_31 to L_34). The final part of the loop (from L_41 to L_43) contains the loop test — if the test is true then there is a jump back to the loop body; otherwise the loop finishes.

For Java, a **for** loop was transformed at the bytecode level using an obfuscation tool. The tool mainly added two new items to the bytecode. Firstly, an instruction athrow, which throws an exception, was added in between the loop initialisation and the body. Since there is a branch immediately after the initialisation and the code will eventually jump back to the top of the loop body, it is possible to add bogus in between these parts as it will not be executed. Note that this does not create an irreducible jumps as the bogus code does not jump into the middle of the loop. Similarly, a bogus block of code, which contained some goto statements, was added after the main method. Both of these additions meant that the bytecode could not be fully decompiled.

For .NET (and CIL), the transformation to obfuscate loops tried to complicated the control-flow by adding a **switch** statement. The new **switch** block appear near the beginning of the transformed code and contains a list of branch targets — each of the targets correspond to a particular part of the loop. The **switch** statement uses a new local variable which it set to the appropriate value (corresponding to the list of branch targets) at various points in the program. The transformation is similar to the control-flow flattening transformation described in Section 2.2.5. The IPP report shows the IL code before and after the transformation but does not give the output (or error message!) of any attempt to decompile the transformed code. However it is possible to manually trace the program to work out the control-flow.

Both of the loop obfuscations considered aimed to change the control-flow so that it becomes difficult for an automatic tool to decompile the IL code (we can consider this to be the attack model). While the transformations did change the control-flow structure of the loop (and indeed made the decompilation harder) the main body of the loop was left unchanged and it was also fairly easy to manually identify the various parts of the loop. The IPP report also mentions that (in common with many obfuscations) these transformations increase the code size and could impact on the efficiency of transformed programs.

5 Summary

The IPP report considered a variety of protection methods (including some obfuscating transforms) for some different situations.

We can classify the obfuscating transforms using the classification given in Section 2:

- **Layout Obfuscations**: Name obfuscations for Java and C# (discussed in 4.1.2), token renaming (Section 3.2.1) and shrinking for Java (Section 4.1.1) are layout obfuscations.
- **Control-Flow Obfuscations**: The flow obfuscations considered by the IPP report were the use of opaque predicates to confuse assembler control-flow (Section 3.2.1), adding bogus code between flow blocks in bytecode (Section 4.1.4) and using **switch** blocks in CIL.
- **Data Obfucations**: The only obfuscation discussed by the IPP report that could be considered to be a data obfuscation was string encryption (Section 4.1.3).

As mentioned in Section 2.1, layout obfuscations are commonly used in automatic tools but flow and data obfuscations are less so. There was only one obfuscation, namely string encryption, that could be considered as a data obfuscation and as discussed in Section 4.1.3 it is not a strong protection method. One of the example programs considered by the IPP report used arrays (a Java password program) but the obfuscations considered did not obfuscate these arrays and none of the obfuscations attempted to change the variables (other than by changing the names). It was not clear whether the lack of good data obfuscations was specific to the specific obfuscation tools considered or if data obfuscations are not generally implemented in commercial obfuscators.

The assembler obfuscation used trivial predicates to obfuscate control-flow. Since the assembler language is a very low-level language then it is hard to write more complicated predicates. It would be interesting to see if such predicates could be written in the high-level language (in this case using C) but there would be no guarantee that these transformations would survive the compilation/decompilation process. Since the compilation of Java and C# (and other languages in the .NET framework) programs use an intermediate language then it should be possible to add in obfuscations that are specific to the intermediate language that try to make the decompilation process harder. In Sections 2.2.3 and 2.4.4 we discussed some transformations that would be suitable as obfuscations to prevent decompilation. However, none of these decompilation preventative transformation have been implemented in the tools considered by the IPP report. As with the lack of data obfuscations, it is not clear whether this is due to the particular tools considered or indicative of commercial obfuscators.

In Section 1.5 we considered some metrics that have been used to try to measure the effectiveness of obfuscations. We discussed the problems with defining obfuscations and when designing obfuscations and metrics we should aim to defend against particular attacks (*i.e.* the attack model). In Section 3.1 we discussed the measurements that the IPP report used to evaluate the different protection methods. Only the *Complexity* measure gives an indication of the quality of an obfuscation — the others mainly measure the impact of the obfuscations on the code such as whether the transformed code is easy to maintain or whether the transformation is easy to apply to the code. It is not clear from the IPP report how the complexity measure was calculated. The most complex obfuscation according to the complexity measure was the shrinking transformation for Java (Section 4.1.1) with a score of 4. This high score seems to be due to the fact that the code after applying shrinking could not be fully decompiled although the bytecode looks very similar before and after the transformation. However, the flow obfuscations (Section 4.1.4) had lower scores but the programs could not be fully decompiled and the programs look different (and in the case of the flow obfuscation for .NET, the IL code has been restructured). It is unclear why the shrinking obfuscations scores higher despite having a similar effect to the flow obfuscations.

In Section 2 we discussed a number of different obfuscation techniques including a number from Collberg *et al* [6]. We have seen that the IPP report has considered a range of different protection tools but these tools did not use many of the obfuscation that we discussed in Section 2. Our aim now will be to consider what obfuscation methods might be contained in the next generation of obfuscation tools. Thus we will consider some of the obfuscations from Section 2 that were not surveyed in the IPP report and discuss whether they can be implemented easily.

Part III
Overview

6 Review of different techniques

As we stated in Section 2.1, layout transformations are often implemented by obfuscation tools. This statement is supported in the IPP report [33] which discussed a number of different layout obfuscations: namely assembler language renamings (Section 3.2.1), Java and C# renamings (Section 4.1.2) and shrinking for Java (Section 4.1.1). Layout transformations affect the syntax of programs and are generally concerned with how the program “looks” not with the semantics of the program. As discussed in Section 4.1.1, some layout transformations do not have any effect on a program at intermediate language level (because compilers and decompilers will perform their own renamings) and so layout transformations are considered to be weak obfuscations.

In the rest of this section we will consider some of the obfuscations discussed in Section 2 to see whether they were considered in the IPP report. We will also discuss some of the problems associated with trying to automate these obfuscations.

6.1 Problems with Creating Opaque Predicates

We considered opaque predicates in Section 2.2.1 and showed how they can be used with a variety of transformations such as creating irreducible flow graphs (Section 2.2.3), loop transformations (Section 2.2.4) and exceptions (Section 2.4.1). However the only tool investigated in the IPP report [33] that used opaque predicates was the Assembler Obfuscation (Section 3.2.1) and as we discussed earlier the predicates were simple and could be removed by an optimising compiler. As discussed in Section 2.2.1 it is difficult to generate suitable opaque predicates which are hard to break (*i.e.* they are not trivially true) and stealthy (see Section 1.1). Collberg *et al* [7] proposed some predicates such as $x^2(x+1)^2 \equiv 0 \pmod 4$ and $7y^2 - 1 \neq x^2$ which are based on mathematical identities. These predicates may not be stealthy and so attackers may concentrate on trying to remove them. Also, the predicates are true for all integers and not just at the particular program points that they are used. In Section 2.4.3 we discussed how Collberg *et al* [5] and Palsberg *et al* [27] created data structures using pointers which could be used to generate opaque predicates. However these predicates suffer similar problems as they are not stealthy (due to the fact that a generated data structure is lurking in the program) and the code for generating the structure would be contained within the program itself which would make it reasonably straightforward to compute the values of the predicates. Palsberg *et al* also proposed dynamically opaque predicates which are a set of predicates which all evaluate to the same value in each run. It appears that no obfuscation tool has implemented such a set of predicates since it is hard to find a group of statements which could be obfuscated using these predicates. We saw in Section 2.2.2 that using just two linked predicates require us to have a set of conditions for the predicates and statements — a large set of predicates would require more complex conditions.

We could instead try to use program invariants as predicates. Using invariants could address the problems with the mathematical identities proposed by Collberg *et al* as the invariants should be stealthy as they relate specifically to the program and the invariants may not be true throughout the whole program.

An example of using invariants to create opaque predicates was given in Majumdar *et al* [19]. The paper used various obfuscation techniques, aimed to decrease the usefulness of program slicing, on an example word count method. The unobfuscated method, written in C, was as follows:

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

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

Figure 1: Obfuscated word count method

In the method, nc represents the number of characters, nw the number of words and nl the number of lines. As nc is always incremented for every character, in the loop we have the following invariant

$$nc \geq nw \wedge nc \geq nl \tag{4}$$

Note that if a normal piece of prose was the input then we would expect that $nw \geq nl$ but this is not always the case (suppose that we had a file that just consisted of newline characters) and so we cannot use this as a predicate.

If the method above is backwards sliced from the **out** statement (which represents some output from the method) with nl as the slicing variable then the only statement in the body of the loop that is in the slice is

$$\textbf{if}\ (c ==' \backslash n')\ \ nl\,{+}{+};$$

To make the rest of the body of the loop appear in the slice we can use the invariant (4) to make bogus conditional statements — the obfuscated method can be seen in Figure 1. These conditionals create false dependencies between the three variables and so the whole loop body will be included in the backwards slice for any of the three variables from the end of the method.

To remove these predicates, an attacker would need to understand the program itself rather than just spot mathematical identities (as with the predicates mentioned earlier). However, as with all opaque predicates, it is possible to run a tracer over the program which would show that the bogus predicates evaluate to the same value in every run.

Having chosen a suitable opaque predicate, the obfuscator has now to decide where to place the predicate. Using an invariant based predicate restricts the scope of the

Top Level

```
block
⇒
  switch (var)
  { case A : block; var = B; break;
    case B : end; }
```

Sequence

```
case A : {p; q}; var = B; break;
⇒
  case A : p; var = C; break;
  case C : q; var = B; break;
```

Conditional

```
case A : if (c) {p;} else {q;}; var = B; break;
⇒
  case A : if (c) var = C; else var = D; break;
  case C : p; var = B; break;
  case D : q; var = B; break;
```

Loop

```
case A : while (c) q; var = B; break;
⇒
  case A : if (c) var = C; else var = B; break;
  case C : q; var = A; break;
```

Figure 2: Rewrite rules for control-flow flattening

placement to the places in the program where the invariant holds. When creating slicing obfuscations, Majumdar *et al* usually place obfuscations just before the slicing point so that false dependencies could be created. Also, as we discussed in Section 2.2.1, opaque predicates can be used in the middle of a method to extend the scope of a variable.

6.2 Flattening

In Section 2.2.5 we discussed the control-flow flattening described in Wang *et al* [34] and how a tool for x86 language [17] had been developed. We also saw in Section 4.1.4 that a variance of the flattening tool was implemented for CIL. So there are tools that implement control-flow flattening for low-level languages but it should be possible to apply this transformation to source code.

Figure 2 describes a possible set of rewrite rules for flattening source code control-flow. The **Top Level** rule creates a **switch** statement for an entire block of code and should only be applied once. Each of the other three rules are used to deal with different cases in a block of code. Note that in the rules, the labels C and D denote fresh labels which have to be created. The **Sequence** rule can be used to break up a block of code into different statements — this transformation can be used to separate contiguous pieces of code by using labels which are not consecutive. The **Conditional** rule is should not be used on conditional statements which are already of the form

$$\textsf{\textbf{if}}\ (\mathit{condition})\ \mathit{var} = v_1;\ \textsf{\textbf{else}}\ \mathit{var} = v_2;$$

(these statements actually do not match our rewriting rule). Finally, the **Loop** rule is used to remove **while** loops. Note that this rule can deal with nested loops and will include such loops within the same **switch** statements. So we can use these rules to perform flattening on source level code — in fact, we can easily produce the flattening examples we gave in Section 2.2.5 (with different label names) using these rules. The only situation in which we would have difficulty using these rules is if we already had a **switch** statement in our block of code. In that case, we would have to "inline" the inner **switch** statements by using the same switch variable and renaming the labels as necessary.

6.3 Creating Loop Transformations

Creating loop transformations at the intermediate language is difficult as we have to be able to identify loops in the code before we can perform transformations. We would need to identify the header, the body and the exits of the loop and the presence of nested loops makes the identification much harder [1]. So it would be easier to apply loop transformations at the source code level.

Loop obfuscations are particularly helpful in creating dependencies between different variables — adding these dependencies is the key to producing obfuscations which impede the usefulness of program slicing [13]. For example, suppose we want to find the backwards slice for a variable x. To obfuscate the slice we want to make x depend on different variables and so increase the size of the slice. Consider the code fragment:

```
while (i < N)
{   ...
    S : x = ... y ...
    ...   }
```

The variable x is dependent on the variable y but, as we are in a loop, the statements in the loop that occur after S (in addition to those that come before S) will be included in the backwards slice. The variable x is also dependent on the induction variable i. As mentioned in Section 2.2.4 we can introduce a bogus induction variable j (say) by adding a condition for j into the loop guard and we can then use j to create dependencies between the variables in the loop. We have to ensure that the condition for j is invariant during the execution of the loop but could false elsewhere in the program. Full details of this transformation can be found in Majumdar *et al* [19]. Another loop transformation discussed in Section 2.2.4 that is useful for obfuscating slices is to extend the scope of an induction variable. For this transformation we need a number of safety conditions to ensure that we can extend the scope safely.

In Section 2.2.4 we mentioned some of the loop obfuscations discussed by Collberg *et al* [6] such as unrolling, fission and blocking. As we stated earlier, these transformation may be performed by an optimising compiler and so any obfuscations that we add using these loop transformation may be altered in the compilation phase. Collberg *et al* [6] mention that these transformations could be removed fairly easily but that a stronger obfuscation could be achieved by combining these transformations. The stealthiness of these transformations could also be an issue — for example, if an attacker sees a number of nested loops then the attacker may suspect that loop blocking has taken place.

6.4 Data Obfuscations

In our summary of the IPP report (Section 5) we noted that string encryption was the only obfuscation discussed by the IPP report that could be considered to be a data obfuscation. However, in Section 2.3, we discussed a number of different data obfuscations — why were none of these obfuscations implemented by the tools discussed in the IPP report?

One of the main reasons that obfuscation tools rarely implement data obfuscations is that they are generally hard to automate. String encryption (discussed in Section 4.1.3) generally encrypts all strings at every point in the program (usually by replacing strings by a method call) and so this means that string encryptions are easier than most data obfuscations to automate. Moat other data obfuscations require the obfuscator to select a particular variable to obfuscate. Also, in Section 2.3.1 we saw that it is possible to localise where we apply a data obfuscation which gives an obfuscator freedom to choose where to place a data obfuscation.

No array obfuscations were amongst the techniques considered in IPP report. We are unsure whether the lack of array obfuscations is due to specific obfuscation tools or specific programs considered or if array obfuscations are just not implemented in commercial obfuscators. Array obfuscations are generally harder to implement due to the conditions needed to ensure that we can safely transform an array (for instance, we may have arrays that are globally used or are entirely passed between methods). Some conditions for finding arrays which are suitable for splitting and a specification of array splitting for intermediate language were given in Drape *et al* [11]. However, we can specify all of the array obfuscations discussed in Section 2.3.4 using the framework described in Section 2.3.1 (see Drape *et al* [12, 14] for more details). Thus we can localise any array obfuscation so that it can be applied where it is safe to do so. This means that we can obfuscate an array in a particular method and then remove the obfuscation before it is passed to another method.

6.5 Review of Language Dependent Transformations

In Section 2.2.3 and 2.4.4 we discussed some transformations that could be used to obfuscate intermediate languages. As we mentioned earlier, higher level languages such as C# and Java do not allow jumps into the middle of loops but we can write these bogus irreducible jumps in the relevant intermediate language. This would mean that a decompiler would have problems trying to re-construct the source code since the irreducible jump would have to be turned in a reducible one. Thus the code produced may be incorrect or the loop would have to be removed. However, none of the transformations considered by the IPP report tried to create irreducible jumps. This might be due to the fact that when creating a bogus jump we have to use an opaque predicate and, as we discussed in Section 6.1, good opaque predicates are harder to create. Also, as we mentioned in Section 6.3, it is hard to create loop obfuscations at the intermediate language level and it is also hard to create suitable opaque predicate at this level too.

In Section 2.4.4, we mentioned how we can create obfuscations using intermediate language instructions which do not have a direct translation to source code. We showed how to use typed reference which can be used to obfuscate an variable and we gave rewrite rules for assignments and uses of the variable. The rewrite rules need to be applied to all occurrences of the variable and we may encounter problems if the variable is passed to another method (however the simplest thing to do would be to localise the obfuscation to a single method and unobfuscate the variable before it is passed to another method). Also, this obfuscation is not stealthy since this transformation uses instructions which are not commonly used — the use of these instructions may appear suspicious to an attacker.

In Section 2.4.1 we discussed ways in which exceptions could be used to create obfuscations. One transformation showed how we could exit from a loop by throwing an exception. Another used an opaque predicate to fake an exception to obscure the control-flow but, as with all transformations that use opaque predicates, this obfuscation may be hard to implement as it requires finding a suitable predicate to use. When adding exception obfuscations we must ensure that the new exception handling does not interfere with the exceptions which may be thrown by the original program. One of the Java bytecode obfuscations considered by the IPP report [33] (mentioned in Section 4.1.4) inserted an exception (using `athrow`) into a bogus part of the code. The exception was placed between the initialisation and body of a loop in a part of the code that is never executed. As intermediate languages generally have less structure than source code, it is often harder to identify parts of the intermediate code which are not executed and thus make it easier to insert bogus code at the intermediate language level.

In Section 2.2.6 we mentioned a number of method transformations that we given in Collberg *et al* [6]. Two method obfuscations were considered in the IPP report. The first (described in Section 3.2.1) was similar to the method interleaving described in Section 2.2.6 — control-flow between the various fragments was handled by jumps. The second (discussed in Section 4.1.2) was the standard renaming transformation but applied to method as well as variable names. As we mentioned earlier this transformation is safe if the methods have different type signatures. Method transformations are much harder to implement than many of the other transformations that we have discussed. The conditions for ensuring that a method transformation is safe to perform need to take into account the program (rather than just a specific portion of a program) as a method could be called from any place in the program. Also, if a method is part of the interface of a program then we may not be able to transform the method.

6.6 Automation Problems

In the last few sections, we discussed a number of different techniques and identified some of the problems that may occur when trying to implement and automate these obfuscations.

6.6.1 Placement

One of the main concerns with creating obfuscations is deciding where the “best” places are to put obfuscations. For example, if we had a program with some loops, which loop (or set of loops) should we choose to obfuscate? An obfuscation may slightly slow down the execution of a program so choosing a loop that is executed many times or one that has a large loop body may adversely slow down a program. Conversely choosing a small loop or one that is executed rarely may not result in an obfuscation that is sufficiently complex to deter an attacker. Even when we have decided on which loop to obfuscate, should we add in a bogus variable, try one of the optimising compiler transformations or try to extend the scope of the induction variable to a neighbouring loop? In Section 2.2.4 we mentioned that we can fake a loop but which block of code should we apply this transformation to?

A similar problem arises when applying data obfuscations, namely which variable should we choose to obfuscate? If we choose a variable that is used extensively throughout a program and choose an obfuscation that slows down execution then we risk affecting program performance. We also have to decide whether we want a variable to be obfuscated for the entire scope of the variable or whether we want to localise the effect.

When creating slicing obfuscations, Majumdar *et al* [19] first used a program slicer to analyse a program. The output variables (for example, those that are part of a `printf` statement) were used to determine the slicing criterion for backwards slices of the program. Obfuscations were then added to the program essentially to try to increase the size of the slice for each output variables by trying to include parts of the program that were not contained in the original slices. The approach determined which variables to obfuscate and where to place obfuscations. Thus we should try to use attack tools to aid us in deciding the places were our programs are “vulnerable” which will help us decide where to place obfuscations. Note that the obfuscations in Majumdar *et al* [19] were added manually and it would be interesting to see if the creation of slicing obfuscations could be implemented in an obfuscation tool.

6.6.2 Stealth

As we mentioned in Section 1.1 the notion of the stealth of an obfuscation is context sensitive. This means that a transformation that is stealthy (*i.e.* it does not “stand out”) when applied to one program is not necessarily stealthy when used with another program. Thus when using an automatic obfuscation we cannot be sure whether an obfuscation will be stealthy as we cannot easily measure the “context” of a program. Thus we will be restricted to using more general obfuscations that can be applied to a variety of situations and so we may not be able to use more complex obfuscations.

6.6.3 Applying different obfuscations

If we build a tool that contains a number of different obfuscations then it is possible that many different obfuscations could be applied to the same piece of code or variable. Thus we need to be aware of the effect of applying two obfuscations to the same piece of code and so we will need to consider the resulting obfuscation obtained by composing two different obfuscations together. For example, using the array transformations discussed in Section 2.3.4, suppose that we split an array A into two arrays B_1 and B_2 using the odd/even split (so elements of A with even indices go into B_1 and the rest into B_2). If we now use the array merge described in Section 2.3.4 which effectively concatenates two arrays then we will obtain an array which has the same elements as A but in a different order. However, if we merge the arrays by interleaving, *i.e.* taking one element from B_1 and then one from B_2 and so on, then we will get A back again. This process is illustrated below:

$$\begin{tikzcd}
& {[0,1,2,3,4,5,6]} \arrow[d, "\text{split}"] & \\
{[0,1,2,3,4,5,6]} & {[0,2,4,6]\ [1,3,5]} \arrow[l, "\text{interleave}"'] \arrow[r, "\text{concatenate}"] & {[0,2,4,6,1,3,5]}
\end{tikzcd}$$

So, by applying two different obfuscations we could in fact remove the original obfuscation. But, on the hand, we may find that applying two different obfuscations may yield a more complicated transformation such when we applied a concatenate merge after a split. Using the specification framework of Drape *et al* [14] we will be able to work out the resulting obfuscation obtained by composing two data obfuscations. Another issue that we need to consider when applying two obfuscations is that the order in which we apply the obfuscation. For example, if we have a single array then we not be able to apply an array merge first but if we split the array first then we could perform a merge.

6.6.4 Intermediate Language vs. Source Code

The IPP report [33] considered applying obfuscation at the intermediate language level. Obfuscations such as language breaking transformations (Section 6.5) and adding irreducible jumps are easier to add at the intermediate level than at source code level. However, there are some obfuscations such as loop transformations (see Section 6.3) that are harder to add at the lower level. Thus when creating an obfuscation tool we should have the flexibility to add transformations at both the intermediate and source code levels as appropriate.

6.6.5 Other issues

We now briefly summarise some of the other issues we need to consider when trying to implement obfuscations.

- We need to decide on how many obfuscations we apply to a program and how many times we apply one particular obfuscation. A limit of the number of obfuscations that we apply to a program is important to ensure that we do not adversely affect the speed of the program.

- We have seen that it is hard for an automatic tool to decide where to place obfuscations and to choose suitable candidates for variable obfuscations. Often it is easier for a programmer to choose the suitable variables for obfuscation and so it would be useful if a programmer could annotate the program to give hints to an automatic obfuscator. Annotations could also be used to indicate invariants which could be used to create opaque predicates.
- We have not considered how to apply obfuscations across methods. For example, if we select a variable for obfuscation should we limit the scope of the obfuscation to a single method or to all the places where the variable is used?
- Some obfuscations have conditions that describe when they can be applied (such as when it is "safe" to apply an obfuscation). An automatic obfuscator needs to be aware of such conditions and has to have a mechanism for checking the conditions.

7 Looking to the Future

So far we have surveyed various obfuscation techniques and the IPP report [33] evaluated some protection tools— however we found that the obfuscations contained in these tools were not very potent. We would like to develop more successful obfuscations than the ones that are commonly found in protection tools and we have discussed a number of issues that may hinder the successful implementation of more productive obfuscation techniques.

In the rest of this section we will try to predict which obfuscation techniques we think can be implemented quickly, which techniques will have a small latency before they can be successfully implemented and finally those which we feel will require more research and analysis before implementation can be achieved.

7.1 Immediate

Now, we propose some obfuscation techniques which we believe can be created fairly quickly.

- In Section 1.4 we discussed how Drape *et al* [13] proposed creating obfuscations by first attack a program with a program slicer. We could extend this method to cover other attack tools such as pointer analysis and theorem provers. The results from these tools could help us in creating obfuscations that would guard against future attacks using these tools.
- In Section 2.2.3 we saw that we could create irreducible jumps in intermediate language. To create irreducible jumps, a loop needs to be identified (in Section 4.1.4 we gave an outline for what a **for** loop looks like in IL) and then a jump can

be created using an opaquely false predicate. For the short term, as the aim of the transformation is to make decompilation harder, a simple predicate could be chosen to use for the creation of the jump.

- Another intermediate language transformation that we could implement is to look instructions at the intermediate level which cannot be easily translated into higher level code. The aim of adding code which use such instructions is to impede the decompilation process. In Section 2.4.4 we gave an example of a possible set of instructions for CIL.

- In Section 2.1 we saw a number of layout obfuscations and we mentioned that this type of transformation is popular in protection tools (as witnessed by the IPP report [33]). Instead of performing the routine layout obfuscations, deliberately malicious layout obfuscations, such as renaming variables to misleading names or documenting comments which do not describe the true intent of the program, could be added.

7.2 Short Term

We now describe some techniques which require some further study but should be reasonably straightforward to implement.

- In Section 2.2.1 we discussed opaque predicates and in subsequent sections we showed how such predicates could be used in the creation of obfuscations. It is hard to create suitable opaque predicates but in Section 6.1 we discussed how Majumdar *et al* [19] used program invariants to manufacture opaque predicates. So when code is developed a programmer should try to document suitable invariants at various program points. However if the code is modified then it may be necessary to modify the invariants (but if the code is modified in a “functional” way then it should be possible to recompute the invariants). The invariants can then be used to manually add obfuscations.

  To automate this process it will be necessary to have some way of converting the documentation of predicates into creating predicates and also some way of documenting suitable invariants — this might be a longer term goal.

- In Section 2.2.5 we saw a technique for flattening control-flow and we discussed how this transformation had been implemented for x86 code. In Section 6.2 we gave some rewrite rules which should enable flattening to be implemented at source code level. Wang *et al* [34] also discussed some extra obfuscations which can be added and so any source code implementation should also include extra levels of protection.

- In Section 2.3.4 we described a number of array obfuscation from Collberg *et al* [6] and, as we mentioned in Sections 5 and 6.4, none of techniques considered in the IPP report [33] used array obfuscations. Drape *et al* [11] gave a specification for an array split including safety conditions. As we mentioned in Section 6.4, the array obfuscations discussed by Collberg *et al* [6] can be specified using the framework described in Section 2.3.1 (from the work of Drape *et al* [14]). Thus it should be possible to implement array obfuscations using the techniques described above.

- In Section 6.6.1 we discussed the problem of choosing suitable places to add obfuscations. We mentioned that when creating slicing obfuscation Majumdar *et al* [19] used an attack tool (namely a program slicer) to aid in the choice of suitable places. So other attack tools could be used to identify program portions (or variables) which are vulnerable to attack and this information can be used to create obfuscations which could defend against future attacks.

- For various obfuscations, such as array transformations (Section 2.3.4) and loop transformation (Section 2.2.4), we need conditions which describe when it is safe to apply these obfuscations. Thus before implementation it will be necessary to consider each obfuscation and think about what safety conditions the obfuscation needs to satisfy.

- In Section 1.1 we mentioned the concept of the stealth of an obfuscation (discussed further in Collberg *et al* [7]). Obfuscations that are not stealthy flag important areas of a program or give indications about how obfuscations may be removed. In Section 2.2.1 we remarked on the fact that deliberately unstealthy opaque predicates can be created which protect unimportant areas of a program. Other obfuscations can be faked too — for instance, bogus program areas can be created (by using an opaque predicate) and then many obfuscations can be added to these areas. By concentrating on areas of bogus code, obfuscations can be added generally without affecting the efficiency with the aim of focussing an attacker's interest on these parts of the program.

7.3 Long Term

Finally we outline some obfuscation techniques which we believe require extensive research and effort before they can be successfully created.

- When describing obfuscations we have often mentioned the concept of stealth (described in Collberg *et al* [7]) To effectively measure the quality of an obfuscation, a way of measuring stealth is needed but currently no such measure exists. One of the main problems with measuring stealth is that it is context dependant. An area for future work is to develop a metric for stealth — to do this it would be necessary to have some way of measuring the "context" of a program point.

- Most of the obfuscations that we have described are applicable within a single method or procedure. In Section 2.2.6 we described a number of transformations that could be used to obfuscate methods. Two potential areas for investigation are to implement the method transformations and to make the existing transformation be applicable across different methods.

- In Section 2.2.4 we discussed a number of transformations that could be used to obfuscate loops. However, as we mentioned in Sections 6.3 and 6.5, it is difficult to add loop obfuscations to intermediate code. This is because it can be difficult to identify loops in the intermediate code. In Section 4.1.4 we showed what a `for` loops could look like in CIL but not all `for` loops may be constructed in this way. More general loops may be constructed in different ways and the code may also contain nested loops. So, before loop obfuscations can be implemented at the intermediate level, it is necessary have patterns for identifying potential loops, conditions for the loop transformations and have suitable rewrite rules for the transformations.

- In Section 1.4 we described how Drape *et al* [13] and Majumdar *et al* [19] used a slicer when creating obfuscations (which were added manually to a program). In Section 6.6.1 we discussed how this approach aided in deciding which program places and variables should be obfuscated. Further research is needed to see how these slicing obfuscations could be implemented in an automatic tool. To able to decide where to place obfuscations, the tool needs to be able to interpret the results of a slicer and to choose an appropriate obfuscation. When creating slicing obfuscations, Majumdar *et al* [19] used a slicer to test the quality of the obfuscations. It would also be useful for an automatic obfuscator to be able to check potential obfuscations to decide which obfuscation is the most suitable.

- We discussed how we could use exceptions to create obfuscations (in Section 2.4.1). To implement exception obfuscations, it is necessary to decide on suitable places to add such obfuscations and ensure that the placement of any new exception handling does not interfere with any exceptions that the program may legitimately throw. As intermediate language has less structure than source code, it may be harder to implement exception obfuscations at the intermediate level.

- In Section 6.6.3 we discussed that if we add two successive obfuscations to the same data object then we may find that the resulting effect removes the first obfuscation or is more complicated than the either of the two original obfuscations. It is important to study the effects of repeatedly applying different data obfuscations to a data object. Since data obfuscations can be specified functionally (using the framework of Drape *et al* [14]) it is possible to compose obfuscations to discover the resultant transformation. The order in which obfuscations are applied may also play a role in the strength of the resulting obfuscation.

7.4 Concluding remarks

In the previous sections, we have highlighted some of the important implementation issues that need to be addressed before we can expect to build successful obfuscation tools — a summary of the various techniques can be found in Table 4. We expect that with further research into the area of obfuscation, we should be able to successfully implement many of the techniques that we surveyed in Part I.

	Immediate	**Short Term**	**Long Term**
Data		Arrays	Ordering
Control-Flow	Irreducible Jumps	Invariant predicates Flattening	IL loops Methods
Other Techniques	Attack tools IL commands Malicious layout	Placement Conditions Bogus	Stealth Exceptions Automate slicing

Table 4: Summary of potential future work in obfuscation implementation

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

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

[3] 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`.

[4] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *ISC '01: Proceedings of the 4th International Conference on Information Security*, pages 144–155, London, UK, 2001. Springer-Verlag.

[5] Christian Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation — tools for software protection. *IEEE Trans. Software Eng.*, 28(8):735–746, 2002.

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

[7] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient and stealthy opaque constructs. In *ACM SIGACT Symposium on Principles of Programming Languages*, pages 184–196, 1998.

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

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

[10] Stephen Drape. Generalising the array split obfuscation. *Information Sciences*, 177(1):202–219, January 2007.

[11] 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.

[12] Stephen Drape and Anirban Majumdar. Design and evaluation of slicing obfuscations. Technical Report 311, CDMTCS, The University of Auckland, June 2007.

[13] Stephen Drape, Anirban Majumdar, and Clark Thomborson. Slicing aided design of obfuscating transforms. In *Proceedings of the 6th Annual IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007)*, pages 1019–1024, Melbourne, Australia, July 2007. IEEE Computer Society.

[14] Stephen Drape, Clark Thomborson, and Anirban Majumdar. Specifying imperative data obfuscations. In *Proceedings of the 10th Information Security Conference (ISC '07)*, volume 4779 of *Lecture Notes in Computer Science*, pages 299–314. Springer, October 2007.

[15] Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *SIGPLAN Notices*, 16(3):63–74, 1981.

[16] IOCCC. The International Obfuscated C Code Contest.
URL: www.ioccc.org.

[17] Matias Madou, Ludo Van Put, and Koen De Bosschere. Understanding obfuscated code. In *Proc. 14th IEEE International Conference on Program Comprehension (ICPC06)*, pages 268–271, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[18] Anirban Majumdar, Stephen Drape, and Clark Thomborson. Metrics-based evaluation of slicing obfuscations. In *Proceedings of the Third International Symposium on Information Assurance and Security*, pages 472–477, August 2007.

[19] Anirban Majumdar, Stephen J. Drape, and Clark D. Thomborson. Slicing obfuscations: design, correctness, and evaluation. In *DRM '07: Proceedings of the 2007 ACM workshop on Digital Rights Management*, pages 70–81, New York, NY, USA, 2007. ACM.

[20] Anirban Majumdar and Clark Thomborson. Manufacturing opaque predicates in distributed systems for code obfuscation. In *ACSC '06: Proceedings of the 29th Australasian Computer Science Conference*, pages 187–196, Darlinghurst, Australia, 2006. Australian Computer Society, Inc.

[21] 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.

[22] Thomas McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[23] 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.

[24] John C. Munson and Taghi M. Khoshgoftaar. Measurement of data structure complexity. *Journal of Systems Software*, 20(3):217–225, 1993.

[25] Masahide Nakamura, Akito Monden, Tomoaki Itoh, Ken ichi Matsumoto, Yuichiro Kanzaki, and Hirotsugu Satoh. Queue-based cost evaluation of mental simulation process in program comprehension. In *METRICS '03: Proceedings of the 9th International Symposium on Software Metrics*, page 351, Washington, DC, USA, 2003. IEEE Computer Society.

[26] 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.

[27] J. Palsberg, S. Krishnaswamy, K. Minseok, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *Proceedings of the 16th Annual Computer Security Applications Conference, ACSAC '00*, pages 308–316. IEEE, 2000.

[28] RemoteSoft. Salamander .NET obfuscator. Available from URL: `http://www.remotesoft.com/salamander/obfuscator.html`.

[29] 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.

[30] PreEmptive Solutions. Dasho.
Available from URL: `www.preemptive.com/products/dasho`.

[31] PreEmptive Solutions. Dotfuscator.
Available from URL: `www.preemptive.com/products/dotfuscator`.

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

[33] Stefan Vogl, Emanuel Mathis, Martin Ortner, and Georg Schönberger. Intellectual property protection. Technical report, The University of Applied Sciences of Upper Austria, Hagenberg, 2008.

[34] 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.

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