Obfuscating a typed intermediate language

Stephen Drape

May 11, 2002

Transfer Dissertation

1 Protection

Consider a situation where a programmer develops a new piece of software that contains a unique innovative algorithm. If the programmer wants to sell this software, it is in the programmer's interest that the algorithm should remain secret from purchasers. What can the programmer do to stop the algorithm from being seen or modified by a client? A similar issue arises when a piece of software contains a date stamp that allows it to be used for a certain length of time. How can you stop people from identifying (and changing) the date stamp? So, the issue of Software Protection is very important for software developers. There are many ways of protecting software – here are some that are identified in [6] and [7]:

- Server-side execution – this involves clients remotely accessing the software from the programmer's site. The downside of this approach is that the application may become slow and hard to use due to limits of network capacity.
- Encryption – an encryption algorithm could be used to encrypt the entire code. This is really only viable if the encryption and decryption process takes place in hardware. If encrypted code was given to a client, then the client could intercept the code after decryption has taken place.
- Watermarking – this is only useful for "stamping" a product. Stamps are used for copyright protection - but they still do not prevent people from stealing parts of the program, although there are methods for embedding a watermark in a program [23].

The protection that we want must work for the situation of a client obtaining a fully (or partially) compiled program from a developer. The techniques outlined above will not be sufficient for this situation. So we need another approach.

1.1 Obfuscation

The approach that the rest of this dissertation will be concerned with is *Obfuscation*.

Collberg's main obfuscation paper [6] does not really give a concrete definition of what obfuscation is. The basic idea of obfuscation is to take a program and transform it in such a way as to make it "hard to understand". This is quite a vague definition. Collberg tries to qualify this definition by introducing various metrics which measure the complexity of code. For example:

- *Program Length* – the complexity increases with the number of operators and operands in the program.
- *Cyclomatic Complexity* – the complexity of a function increases with the number of predicates in a function.
- *Nesting Complexity* – the complexity of a function increases with the nesting level of conditionals in a function.

From this we can then measure the *Potency* of an obfuscation. Let $\mathcal{T}$ be a transformation which maps program P to 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 $\mathcal{T}_{pot}(P) > 0$.

Other properties we can measure are:

- *Resilience* – this measures how well a transformation survives an attack from a deobfuscator. Resilience should take 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 to the original program P.
- Quality – this combines potency, resilience and cost to give an overall measure.

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

Barak[4] takes a more formal approach to Obfuscation. There, for a program P, an obfuscator $\mathcal{O}$ has two conditions:

- Functionality – $\mathcal{O}(P)$ computes the same function as P.

- "Virtual black box" property – "Anything that can be efficiently computed from $\mathcal{O}(P)$ can be efficiently computed given oracle access to P".

Barak then goes on to show how, under this definition, obfuscation is impossible.

This definition of obfuscation is too strong for our purposes. An obfuscation will be deemed effective it makes a program harder to read or if it foils a decompiler. The effectiveness of a obfuscation is an area that needs further research (see Section 4).

1.2 Examples of obfuscations

This section will highlight some of the many obfuscations stated in [6] and [8].

One of the simplest obfuscations is variable renaming. This simply consists of taking any variables that have a "helpful" name (such as "total" or "counter") and renaming them to a less useful name. This is not a very resilient transformation - but many obfuscators simply employ little more than this method – Section 1.3

1.2.1 Opaque predicates

One of the most useful techniques that Collberg[6] describes is *opaque predicates*. An opaque predicate P is a predicate whose value is known at compile time – P^T denotes a predicate which is always True (similarly for P^F) and $P^?$ denotes a predicate which could be either True or False.

Some example predicates which are always True (supposing that x and y are integers):

$$\begin{aligned} x^2 &\geq 0 \\ x^2(x+1)^2 &\equiv 0 \ (\mathrm{mod} 4) \\ x^2 &\neq 7y^2 - 1 \end{aligned}$$

Opaque predicates can be used to transform a program block `B`:

- `if (`P^T`) B`
  This hides the fact that `B` will always be executed.

- `if (`P^F`) B'; else B;`
  The block `B'` can be a version of `B` which may contain errors. Again, because the predicate is always false, `B` will be always be executed.

- `if (`$P^?$`) B; else B';`
  In this case, we can have two versions of `B` which have the same functionality.

The uses of opaque predicates described above can be used to insert a branch into a block of statements. Opaque predicates can also be used to create an irreducible flow graph (see Section 2.2.1 for more details).

1.2.2 Variable transformations

In this section, we shall show how to transform an integer variable i within a method. To do this, we need to define two functions f and g:

$$\begin{aligned} f : X &\rightarrow Y \\ g : Y &\rightarrow X \end{aligned}$$

where $X \subseteq \mathbb{Z}$ – this represents the set of values that i takes. We require that f and g are bijective so that we can make g to be the inverse of f. To replace the variable i with a new variable, j say, we need to perform two types of replacements:

- Any assignments of i of the form $i := E$ are transformed to $j := f(E)$.
- Any uses of i (i.e. not a definition of i) are transformed to $g(j)$.

These replacements can transform a simple algorithm into a much more complicated one (see the next section for an example with a `while` loop).

1.2.3 Loops

In this section, we shall show how to transform a simple `while` loop:

```
i=1;
while(i<100)
{
   ...
   i++;
}
```

The first technique will be to transform the loop counter i by using the methods of the last section. To do this, we have to define functions f and g. We shall take:

$$\begin{aligned} f(i) &= 2i + 3 \\ g(i) &= \frac{i-3}{2} \end{aligned}$$

Let us now transform the loop. Note that the statement `i++` (i.e. `i=i+1`) corresponds to a use and an assignment. The loop becomes:

```
i=5;
while((i-3)/2<100)
{
   ...
   i=(2*(((i-3)/2)+1))+3;
}
```

With some simplifications, we obtain:

```
i=5;
while(i<203)
{
   ...
   i=(2*i)-1;
}
```

Another method, we could use is to introduce a new variable, k say, into the loop and put an opaque predicate (depending on k) into the guard. This means that k performs no function in the loop, so we can make any assignment to k. The original loop then could become:

```
i=1;
k=20;
while(i<100 && (k*k*(k+1)*(k+1) % 4 == 0))
{
   ...
   i++;
   k=k*(i+3);
}
```

We could also put a false predicate into the middle of the loop that jumps out of the loop. Here is a simple example:

```
i=1;
while(i<100)
{
   ...
   if (i*i<0) break;
   i++;
}
```

The next example changes the original single loop into two loops:

```
j=0;
k=1;
while(j<10)
{
```

```
  while(k<10)
  {
   <replace uses of i by (10*j)+k >
   k++;
  }
 j++;
}
```

The final example uses a *try/catch* block. This will create an error deliberately so that we can jump out of the loop:

```
try
{
  int d=1000;
  int i=1;
  while (true)
  {
     d=d/(100-i);
     ...
     i++;
  }
}

catch(Exception)
{
   return ...
}
```

For this to work, we have to make sure that no other exceptions (in particular, division by zero) will be thrown by the loop.

1.2.4 Array transformations

There are many ways in which an array can be obfuscated. The simplest, perhaps, is a reordering of the array indices. This could be either be achieved by a variable transformation (such as in Section 1.2.2) or by defining a permutation. An example permutation for an array of size n:

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

Other array obfuscations will change the structure of the array.

The first structural change we will consider is an array split. Suppose we have an array A of size n. Then we need to define a split, so that A is broken up into two other arrays. To do this, we need three functions c, f_1 and f_2 and two new arrays B_1 and B_2 of sizes m_1 and m_2 respectively (where $m_1 + m_2 \geqslant n$).

The types of the functions are as follows:

$$\begin{aligned} c : [0..n) &\rightarrow \{True, False\} \\ f_1 : [0..n) &\rightarrow [0..m1) \\ f_2 : [0..n) &\rightarrow [0..m2) \end{aligned}$$

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

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

To ensure that there are no index clashes in each array, we require that f_1 is injective for the values for which $c(i)$ is true, (similarly for f_2). We should also make sure that the elements of A are fairly distributed between B_1 and B_2. This means that c should partition $[0..n)$ into (approximately) equal pieces.

This relationship can be generalised so that A could be split between more than two arrays. For this, c should be regarded as a choice function - this will determine which array each element should be transformed to. Again we should require that c partitions $[0..n)$ into fairly equal pieces.

An example is:

$$A[i] = \begin{cases} B_1[i \text{ div } 2] & \text{if } i \text{ is even} \\ B_2[(i-1) \text{ div } 2] & \text{if } i \text{ is odd} \end{cases}$$

Now, we are going to look at the reverse the process of splitting an array by merging two (or more) arrays into one large array. As with a split, we will need to define functions which we determine the order of the elements in the new array. If we perform a split first and then follow it by a merge in which has functions which are inverses of the split functions, we should get the original array back. However, if we do not use inverses then performing a split followed by a merge means that we should get a single array in which the elements are in different order from the original array. Here is 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 $m_1 + m_2$. The relationship between the arrays is defined to be:

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

We need to generalise a merge of arrays B_1 and B_2 into a new array A, which has size $n \geq m_1 + m_2$. We could insist on the stronger condition that $n = m_1 + m_2$, however this imposes extra restrictions on the transformation. To perform the merge, we shall need two functions, g_1 and g_2:

$$
\begin{aligned}
g_1 : [0..m1) &\rightarrow [0..n) \\
g_2 : [0..m2) &\rightarrow [0..n)
\end{aligned}
$$

with the property that

$$\mathrm{ran}(g_1) \cap \mathrm{ran}(g_2) = \emptyset$$

To avoid index clashes, we also require that g_1 and g_2 are both injective. This, of course, could be easily generalised to cope with more than two arrays merging.

Another type of array restructuring deals with the dimensions of the array. We could fold a 1-D array of size $m \times n$ into a 2-D array of size $[m, n]$. Similarly we could flatten an n-D array into a 1-D array. The requirements for folding and flattening are very similar to those for splitting and merging. 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 an element of the array does not throw an exception.

1.2.5 Other obfuscations

Some of the other transformations deal with methods – for example:

- Inlining – this replaces a method call with the body of the method.
- Outlining – a sequence of statements are turned into a method call.
- Interleaving – this merges two methods into one. To distinguish between the methods, an extra parameter could be passed.
- Cloning – this creates many copies of the same method by applying different transformations.

Other obfuscations that are mentioned are: parallelising code, merging variables and using opaque variables.

Collberg[6] also mentions it is important how obfuscations are performed. For example, obfuscations should take place throughout as much of the program as possible and the order in which obfuscations are performed is also important.

1.3 Current obfuscators

There are many commercially available obfuscators. The following list section outlines some of the obfuscators that are available for Java:

- *JCloak* [15] – This works on all classes in a program. It looks at the symbolic references in the class file and it generates a new unique and unintelligible name for each symbol name.

- *DashO* [10] – This uses a renaming system called "Overload-Induction" which renames classes, methods and fields. Overload-Induction induces method overloading as much as possible and it tries to rename methods to a small name (e.g. a single character).

- *Zelix* [33] – This obfuscator uses three different methods. The first method is the usual name obfuscation. The second is flow obfuscation. This tries to changes to the structure of loops by using `goto`s. The last method is string encryption. This encrypts strings such as error messages and the decryption is performed at runtime.

- *Smokescreen* [26] – Again, this obfuscator uses a number of different methods. The standard name change is used. Then, some control flow obfuscations are used. The first shuffles stack operations so that some operations that pop a value into a local variable are delayed. The aim of this transformation is to make it more difficult for decompilers to determine where the value on the stack came from. Another method is to add fake exceptions so that the exception block partly overlap with an existing block of code. This should make control flow analysis more difficult to determine. Another change to the control flow is made by changing some switch statements. The obfuscator attempts to make the control flow appear to bypass the switch statement and go straight to a case statement.

2 C# and IL

C#[21] is a new language that Microsoft have developed to work with the .NET platform[22]. It is an object-oriented language which has been developed from C and C++. One of the features of .NET is that applications can be written in a variety of languages which are compiled to same target language. This is the basis of the *Common Language Runtime* [17]. When .NET compiles a program, it creates a portable executable (PE) file. This PE can be disassemble to an intermediate language (IL) [9] – also, an IL program can be assembled to create a PE. [Note: the intermediate language used in .NET is should be referred to as CIL (Common IL) or MS-IL – for sake of brevity, it will be simplify called IL] IL is a typed stack-based language - a stack-based typed assembly language is discussed in[20]. To make sure that IL works with a variety of languages, including functional languages, some extensions will be necessary [28]. Also, care needs to taken with how the language is typed to cope with many languages [14]. It is also possible to go from IL to C# and there are decompilers available – such as *Anakrino* [2]. This means that a PE file can be converted into C#. Therefore, there is a need for obfuscation.

Intermediate languages are not a new idea. Java has an intermediate language called *Java Bytecode* which runs on the Java Virtual Machine[27]. Java bytecode is also a stack-based language for which we could infer static types[13]. It is possible to convert between Java and bytecode – Krakatoa is one system which decompiles Java bytecode into Java source[24]. So the need for obfuscation of Java bytecode also arises – examples of obfuscators were given in Section 1.3.

With IL, we have two opportunities for obfuscation - at the C# and the IL levels. We will concentrate on obfuscations at the IL level.

2.1 Converting to IL

If we are to perform obfuscations, it is important to understand IL, particularly the relationship between C# and IL.

Let us look at an example. Here is a C# method that computes the GCD of two integers:

```
public static int gcd(int a, int b)
{
int x=a;
int y=b;
while(x!=y)
if (x<y) y=y-x; else x=x-y;
return x;
}
```

After compilation, this disassembles to:

```
.method public hidebysig static int32
           gcd(int32 a, int32 b) cil managed
  {
     .maxstack  2
    .locals (int32 V_0,
             int32 V_1,
             int32 V_2)
    IL_0000:  ldarg.0
    IL_0001:  stloc.0
    IL_0002:  ldarg.1
    IL_0003:  stloc.1
    IL_0004:  br.s       IL_0014

    IL_0006:  ldloc.0
```

```
    IL_0007:  ldloc.1
    IL_0008:  bge.s      IL_0010

    IL_000a:  ldloc.1
    IL_000b:  ldloc.0
    IL_000c:  sub
    IL_000d:  stloc.1
    IL_000e:  br.s       IL_0014

    IL_0010:  ldloc.0
    IL_0011:  ldloc.1
    IL_0012:  sub
    IL_0013:  stloc.0
    IL_0014:  ldloc.0
    IL_0015:  ldloc.1
    IL_0016:  bne.un.s   IL_0006

    IL_0018:  ldloc.0
    IL_0019:  stloc.2
    IL_001a:  br.s       IL_001c

    IL_001c:  ldloc.2
    IL_001d:  ret
}
```

Each instruction is preceded by a line number (which do not have to be in order). At the start of the method, local variables are declared - each variable has a type. For example. `int32` is a the type for 32-bit integers. Values can be stored in variables by using `stloc` and the values are retrieved with `ldloc`. Values passed to the method are accessed using `ldarg` and `starg`. Constants can be loaded onto the stack by the command `ldc`. Arithmetic operations, such as `add` and `sub`, take values off the stack and then put the result back onto the stack. Branches can either be unconditional (of the form `br`) or conditional (such as `bne` – "jump if not equal to"), which compares values from the stack. Branches then require a target to jump to.

When producing a PE, we require that the code is *verifiable*. In terms of IL, here are some conditions that must be met:

- The stack must have the same height and contain the same types when control flow paths meet.
- The stack must be empty when leaving a method.
- Operations must have the correct number of items on the stack (e.g. a binary operation must have at least two elements on the stack).

- Operations must receive the correct type off the stack.

If the code is verifiable, then this means that some code hiding will not be possible. One method of hiding code is to have a block of dead code within a program and the program itself writes code into this dead area.

It is quite hard to perform obfuscations on IL. Few of the methods outlined in Section 1.2 will work in IL as we do not have instructions such as assignments – this problem will be dealt with later. However, some obfuscations can done by hand.

2.1.1 An IL obfuscation

We will show how to do a variable transformation (as outlined in Section 1.2.2) in IL. The (simple) functions we will use to perform the transformations are:

$$\begin{aligned} f(i) &= 2i - 1 \\ g(j) &= (j+1)/2 \end{aligned}$$

Assignment of a variable corresponds to `stloc` and use corresponds to `ldloc`. Looking at the GCD example above, we aim to transform the local variable `V_0`. So, in IL, any occurrences of `stloc.0` will need to be replaced by:

```
ldc.i4.2
mul
ldc.i4.1
sub
stloc.0
```

Occurrences of `ldloc.0` need to be replaced by:

```
ldloc.0
ldc.i4.1
add
ldc.i4.2
div
```

These changes can be made by editing the IL directly and then assembling the file to create a PE. Also, we need to change `.maxstack 4` as there are more items stored on the stack.

2.2 Breaking a decompiler

The aim of obfuscating IL is to the restrict the usefulness of C# decompilers, such as Anakrino [2]. This section outlines some ways in which Anakrino can be broken.

2.2.1 Jumping into while loops

C# contains a jump instruction `GOTO`. The `GOTO` statement allows the program to jump to a statement marked by the appropriate label. However, there are restrictions on the use of `GOTO`. The `GOTO` statement must be within the scope of the labelled statement. This means that it is possible to jump out of a loop but we cannot jump into a loop. Also, we cannot jump into a conditional statement. So, the following program is illegal in C#:

```
if (P) then goto S1;
...
 while (G)
  {
   ....
   S1:
   ....
  }
```

So, if we insert this kind of jump in IL (which is allowed), a decompiler would produce incorrect C# code. We do not actually want this jump to happen, so we use an opaque predicate that is always false.

Let us look at the GCD example in Section 2.1 again. Suppose that we want to insert the jump:

```
if ((x*x)<0) goto L;
```

outside the `while` loop where `L` is a statement in the loop.

We need to create the IL for this:

```
IL_0003:  stloc.1
IL_0100:  ldloc.0
IL_0101:  ldloc.0
IL_0102:  mul
IL_0103:  ldc.i4.0
IL_0104:  ble.s     IL_0010
IL_0004:  br.s      IL_0014
```

The place that we jump to in the IL needs to be chosen carefully. We must (obviously) ensure that it does actually jump to a place inside the `while` loop (in fact, in this case, it jumps into an `if` statement which is inside the loop). Also, we must ensure that we do not interfere with the depth of the stack (so that we can still verify the program).

After assembling the file, Anakrino recovers the illegal jump. Here is the output that we obtain:

```
public static int gcd(int a, int b) {
int local0;
```

```
int local1;
int local2;

local0 = a;
local1 = b;
if (local0 * local0 <= 0)
goto i1;
while (local0 != local1) {
if (local0 < local1) {
local1 = local1 - local0;
continue;
}
i1: local0 = local0 - local1;
}
local2 = local0;
return local2;
}
```

This cannot be compiled in C# as it has an illegal jump. In the case, though, we can easily see that the jump serves no purpose, so we will need to be more stealthy in our approach.

2.2.2 Typed References

Since IL has been developed to be used with a variety of languages, there will be IL commands that are not used in C#. One such set of commands refer to *Typed References*. Using typed references, we can create a variable that contains a pointer and a type – the IL command to push a pointer to a variable (say `V_0`) on the stack is `ldloca.s V_0`. There are three typed reference instructions:

- `mkrefany` – if we have a pointer to a variable with type T on the stack, then we can use `mkrefany`, with type T, to place a typed reference variable on the stack.
- `refanyval` – this unpacks the address of a value on the top of the stack that has been stored as a typed reference.
- `refanytype` – this unpacks the type from a typed references.

If we want to use the commands to obfuscate the use of a local variable, `V_0` say, then first we need to create a new local variable, `V_1` say, which as type `typedref`. Then we need to do two kinds of replacement:-

We replace `stloc.0` with:

```
stloc.0
```

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

and we replace `ldloc.0` with

```
ldloc.2
refanyval  int32
ldind.i4
```

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

This type of change in a method is enough to prevent Anakrino from decompiling the method. However, the obfuscation is not very resilient as these commands can be easily changed.

2.2.3 Method calls

A slightly more resilient use of typed references is with method calls. In IL, the format for a method call of`'myMethod'`(say) in `Class1`, which takes an integer as a value parameter and returns an integer, is:

```
call int32 Class1::myMethod(int32)
```

When an instruction such as this is reached, the top of the stack will have type `int32`.

To transform this method call using typed references, we first need a new local variable, say `int32 V_1`. The method call is replaced by:

```
stloc.1
ldloca.s   V_1
mkrefany   int32
call       int Class1::myMethod(typedref)
```

In the IL code, the method header for `myMethod` needs to be changed from `myMethod(int32 a)` to `myMethod(typedref)`. In the method body, we need to declare a new variable, e.g. `int32 V_1` and then we have to make two types of replacements.

Any uses of `ldarg.0` should be replaced by:

```
ldarg.0
refanyval  int32
ldind.i4
```

and `starg.s` should be replaced by:

```
stloc.1
ldloca.s    V_1
mkrefany    int32
starg.s
```

Now, let us suppose that we have a method `myMethod2` that expects an `int32` value passed by reference and does not return anything. When the method is called, a pointer to an `int32` should be on the top of the stack. The call should be changed to:

```
mkrefany      int32
call          void class1::myMethod2(typedref)
```

In the method body, the changes are similar to those outlined for passing by value. Again, the method signature has to be changed to expect a `typedref`. In the body of the method, `ldarg.0` should be followed by `refanyval int32`. As this method uses a reference, there should not be any uses of `starg`.

Can this process be adapted in general? If a method requires more than one parameter, then each of the parameters are pushed on to the stack individually. This means that using typed references should not cause a problem, although more local variables will be needed.

What about overloading? If two methods with the same name expect a different number of parameters, then there should not be a problem. However, if two methods expect the same number of parameters but with different types then this method would cause problems as the methods would both expect `typedref` parameters. A way around would be to enclose the method bodies in a try/catch block.

As an example, suppose we had the methods:

```
int myMethod(int a)
int myMethod(float a)
```

Then we could construct a new method block which incorporates the code for both of these methods. The outline for this new method would be of the form:

```
.try
{
 ldarg.0
 refanyval   int32
 <code for myMethod(int a)>
 leave.s     IL_aaaa
}

catch [mscorlib] System.InvalidCastException
{
 pop
 ldarg.0
 refany      float32
 <code for myMethod(float a)>
```

```
 leave.s     IL_bbbb
}

IL_aaaa:  <return for the first method>

IL_bbbb:  <return for the second method>
```

The idea of this method is that we first try to see if the typed reference had type `int32` or not. If it did, we proceed to the code for the first method. If not, an exception will be thrown and so we can execute the code for the second method.

This technique is not applicable if the methods returned values of different types.

Typed references, in general, are a way of breaking Anakrino. However, it is obvious where typed references in IL and so they could be removed (even though it may be time consuming if they were used across a method call). It would probably be quite hard to automate this process that have outlined as they are not straightforward rewritings.

3 Path Logic Programming

We will need to find a way to express transformations, in particular obfuscations, on IL - we have used *Path Logic Programming* [11]. In Path Logic Programming, transformations are expressed as logic predicates using regular expressions. The transformations are applied to a representation of IL called EIL (Expression IL) which replaces stack locations with expressions.

3.1 Representations of IL

It is quite hard to perform many obfuscations on IL. The transformations outlined in Section 1.2 rely on the use of variables, loops and assignments. Unfortunately, as IL is a stack based language, many of the obfuscations cannot be readily applied directly.

3.1.1 EIL

Consider the following IL instructions:

```
ldc.i4.2
stloc.1
ldc.i4.3
ldloc.1
sub
stloc.2
```

This performs the same function as:

```
local(V_1) := 2
local(V_2) := sub(ldc.i4.3,local(V_1))
```

The second version uses assignments, so more obfuscations should be applicable. The latter version of IL is form of EIL – Expression IL. The aim of EIL is to change from a stack-based language to one which has expressions instead. In practice, the conversion from IL to EIL does not result in such compact code as in the example. For ease of conversion, extra local variables are created to represent stack locations. If we directly convert the example at the start of the section into EIL, it would look something like:

```
local(ib4_104) := ldc.i4.2
local(V_1)     := local(ib4_104)
local(ib4_103) := ldc.i4.3
local(ib4_102) := local(V_1)
local(ib4_101) := add(local(ib4_103), local(ib4_102))
local(V_2)     := local(ib4_101)
```

To make EIL more useful, then we will want to try to eliminate the extra variables used in the conversion. The control flow structure of EIL is essentially the same as that of IL.

3.1.2 LIL

As well as expressions, for obfuscations, we will need some form of loops and a more sophisticated method of branches.

For example, conditional branches in EIL are of the form:

```
brif <cond> <target if true>
```

such as:

```
brif clt(local(ib4_200),local(ib4_201)) IL_0010
```

This jumps to the instruction labelled by `IL_0010` if the local variable `ib4_200` is less than `ib4_201`. We would really would like to have if-then-else constructs.

We also want to recover loops and switch statements. Baker[3] shows us an algorithm for transforming a flow graph into a program which contains constructs such as if/then/else and repeat. Ramshaw[25] shows how many `GOTO` statements can be eliminated. Another way to remove `GOTO`s is outlined in [12]. An algorithm is outlined which eliminates `GOTO`s by first applying a sequence of transformations that move a `GOTO`, followed by a goto-elimination transformation. This is achieved by introducing extra variables and predicates.

3.2 Specifying transformations

Many transformation systems already exist – we require a system that can work on IL and be suitable for obfuscations. JOIE[5] is a system which works with Java classes. TyRuBa[30] is based on representing programs as logic propositions.The system is not really used for program transformations, instead it is used with implementing classes and interfaces. Gospel[32] is a specification language to express program improving transformations. This needs a declaration of variables (which might be statements), a precondition consisting of a code pattern and dependencies and an action. Unfortunately, these methods involve quite a lot of work. We would like a system in which an obfuscation could be expressed clearly and precisely. One way of expressing transformations is by using rewrite rules. A specification using logic in described in [16]. This uses constructors such as E (there exist a path) and A (for all paths) which express paths in a control flow graph. Perhaps, an easier way to express paths in by using regular expressions (see, for example, [1]).

Our system was implemented in Moscow ML[19] and uses the idea of *Universal Regular Path Queries*[18] – the transformations are expressed as logic predicates using regular expressions.

We will also need a way of applying transformations. For example, consider a variable transformation (Section 1.2.2). We only need to find a variable once, but replacement of this variable needs to be applied exhaustively. So, we shall need some strategies – Stratego[29] is such a language which performs rewriting strategies. At the moment, we have a system which will perform some optimisations and obfuscations on EIL.

3.3 Writing transformations

My main contribution to the work on using PLP for transformations was in the specification of obfuscations. The first task was to check that the existing optimisations that we had – dead assignment elimination, propagation and elimination of unique uses – were correct. Then, I wrote a transformation to evaluate constant expressions and helped to write dead branch elimination. After this, I could start to write some obfuscations. Using the work I had developed on generalising array splitting from Collberg [6], I wrote the queries for performing a split after checking it is safe to do so (for this check, I required that the array is not used without an index after it has been created). Once the strategy language was developed, we could write a complete transformation that would pick a suitable array and then perform a split. It was then straightforward for us to write an array merging transformation. Finally, I wrote a simple variable transformation.

3.4 Future work in this area

We would like to perform some loop obfuscations – for this, we will need to use LIL. Also, some of the syntax for the expressions in EIL is quite cumbersome and so we will need a more succinct approach. Running the transformations at the moment is quite time-consuming, but we hope to develop a faster method.

4 Research directions

4.1 Metrics

We must make sure that any obfuscations we perform are resilient and stealthy. We will, therefore, need a way of measuring the quality of an obfuscation. Collberg[6] gave some metrics that could be used as measures. These metrics do not also give an accurate measure of how much a piece of code is obfuscated. Sometimes, a transformation can be considered an obfuscation according one metric but it is clearly not sufficient as an obfuscation. For example, we could introduce new variables (that served no real function) to increase the program length. However, this variable could probably be eliminated by an optimisation, so this obfuscation is not very resilient.

An area for research could be to take these metrics and formalise them and also to devise new metrics more suited for particular obfuscation applications. Tools could be developed to automated the measuring process. The execution cost also needs to be considered. An obfuscation is worthless if it adversely affects the speed and efficiency of an algorithm. We, therefore, need to develop ways of finding the effect an obfuscation has on the running time of a method. It is possible for an obfuscation to affect different algorithms in different ways, so we will also have to a way of deciding how much an effect an obfuscation has in different situations. Can we devise a metric we measure how obfuscated a program is to a human reader?

4.2 Tighten definitions

Collberg[6] and Barak[4] give very different definitions of obfuscation. Neither of them were really suitable for our purpose. We need to tighten up the definitions and make them more suitable for our applications. We can then measure how effective an obfuscation is with respect to our definitions. We will need to take account of the metrics described in Collberg and also other factors such as stealth and resilience. A particular area to discover is to find out how resilient a particular obfuscation is and how it would stand up to optimisations.

4.3 Proving the quality of obfuscations

In cryptography, proofs can be developed to show how effective a cryptographic system is (see, for example [31]). We could look at any obfuscations we use and prove how well they function. This can be done in conjunction with some metrics. How can we decide what a machine or a human deem is obfuscated? In particular, we need to see how well an obfuscation holds up against a deobfuscator. In practice, it will impossible to completely foil a deobfuscator. We could also try applying optimisations to the obfuscations to see how well they stand up to attack. It will be difficult to decide how obfuscated a program looks to a human reader. The level of obfuscation of a program will be different for different people – this depends on the experience of the reader. Can we, therefore, measure how effective an obfuscation is with respect a human reader?

4.4 Developing more obfuscations

Collberg[6] discusses many different obfuscations. We shall need to develop obfuscations particular to an application (e.g. IL). This could be done by exploiting known weaknesses and trying to keep ahead of new deobfuscators. An important area for investigation is devising opaque predicates. It is important that a predicate is fairly easy to compute. We do not want to slow down a program simply by trying to compute a complicated predicate. On the other hand, we need a non-trivial predicate. We need to ensure that it is hard for a theorem prover (or a human) to determine whether the predicate is always true (or false). We could specialise the a program block according to the value of a variable. For example:

```
if (i==0) {a=b*c;}
    else  {a=((b+i)*c)+(i*(b-a));}
```

In this example, there is an extra branch as the value of a would be the same if $i = 0$ was substituted in the right-hand equation.

Many of the obfuscations outlined are localised to a particular area of a program. We will need obfuscations that can work on more than one method. In particular, the array transformations from Section 1.2.4, only works within one method. A more powerful obfuscation would be obtained if the transformations could be applied (given certain conditions) across various methods and classes.

4.5 Specifying and Implementing Obfuscations

One of the main developments of the current system was to find a way to specify the requirements of an obfuscation. This was achieved by path logic programming. Can all obfuscations be specified in this way? What improvements can be made to the current process? To perform obfuscations, first of all, we needed to find the conditions in which we could apply the transformation – this had to be written as a logic path query. Is this a suitable way to specify the conditions? Next, we need to specify the obfuscation, again using path queries. How can we be sure that the queries that we have constructed are correct? Finally, to apply the transformation we have to find a suitable "place". Which are the best classes/methods/variables to apply the transformations?

5 Plan

The main aim of the thesis will be to consider how to obfuscate IL. To do this, I will look at obfuscation from two different perspectives.

The first approach is to consider how to make the obfuscations work. For this, I try to develop a small set of obfuscations such as variable and loop transformations. Implementing these may involve changing the way that path logic programming works. When applying obfuscations, it is necessary to decide which objects should be obfuscated in each method. The decision could be left up to the person applying the obfuscations but it would be more helpful if it could be suggested which objects would be the best to obfuscate - this could be the objects which are used the most or to avoid objects which would, when transformed, slow down the program. New obfuscations will have to be developed - the obfuscations that have implemented so far only work on variables and are not suitable for loops.

The opposite view is to consider how to reverse the obfuscations. The aim will be to recover the program before the obfuscation was applied. This will require some form of program analysis. The quality of the obfuscation can be determined by how much effort it takes to recover the unobfuscated program. By looking at how to undo obfuscations, it should make the obfuscations stronger by highlighting the weaknesses. A good obfuscation should be one that takes much time and effort to remove.

In considering these views, I will need to develop a suitable definition of obfuscation and try to give a measure of how effective each obfuscation is. The current definitions in Barak and Collberg are not suitable for our purposes. Some measures of the quality of the obfuscations will also be needed. The measure should take into account things such as the readability of the code and the increase in execution time.

I would also like to give a more substantial obfuscation that will work across methods. This will require looking at how the control flow graphs for methods are represented or indeed whether graphs are suitable. This obfuscation should draw on the techniques and results used with the smaller examples.

References

[1] Aho, Sethi, Ullman *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1985.

[2] Anakrino: available from
`http://www.saurik.com/net/exemplar/`

[3] Brenda Baker *An Algorithm for Structuring Flowgraphs*, 1977

[4] Boaz Barak et al *On the (Im)possibility of Obfuscating Programs*, 2001

[5] Cohen and Chase *Automatic Program Transformation with JOIE*

[6] Christian Collberg, Clark Thomborson, Douglas Low *A Taxonomy of Obfuscating Transformations*

[7] Christian Collberg and Clark Thomborson *Watermarking, Tamper Proofing and Obfuscation - Tools for Software Protection*, 2000

[8] Christian Collberg, Clark Thomborson, Douglas Low Manufacturing Cheap, Resilient and Stealthy Opaque Constructs

[9] *Common Language Infrastructure* Partitions I - V, ECMA TG3, 2001

[10] DashO – `http://www.preemptive.com/tools/index.html`

[11] S. Drape, O. de Moor, G. Sittampalam *Transforming the .NET Intermediate Language Using Path Logic Programming*, submitted to PPDP 2002

[12] Erosa and Hendren *Taming Control Flow: A Structured Approach to Eliminating Goto Statements*

[13] Gagnon, Hendren and Marceau *Efficient Inference of Static Types for Java Bytecode*

[14] Andrew Gordon and Don Syme *Typing a Multi-Language Intermediate Code*, 2000

[15] JCloak – `http://www.force5.com/JCloak/jcloak_overview.html`

[16] David Lacey and Oege de Moor, *Imperative Program Transformation by Rewriting*, 2000

[17] Erik Meijer and John Gough *Technical Overview of the Common Language Runtime*, 2001

[18] Oege de Moor, David Lacey and Eric Van Wyk *Universal Regular Path Queries*, 2001

[19] Moscow ML is available from:
`http://www.dina.dk/~sestoft/mosml.html`

[20] Greg Morrisett et al *Stack-Based Typed Assembly Language*

[21] MSDN *Microsoft C# Language Specifications* Microsoft Press, 2001

[22] .NET framework – details available at:
`http://msdn.microsoft.com/netframework/`

[23] Jens Palsberg et al *Experience with Software Watermarking*, 2000

[24] Todd A. Proebsting and Scott A. Watterson, *Krakatoa: Decompilation in Java*

[25] Lyle Ramshaw *Eliminating go to's while Preserving Program Structure*, 1988

[26] Smokescreen – `http://www.leesw.com/`

[27] Stärk, Schmid and Börger *Java and the Java Virtual Machine*, Springer, 2001

[28] Don Syme *ILX: Extending the .NET Common IL for Functional Language Interoperability*

[29] Visser, Benaissa, Tolmach *Building program optimizers with rewriting strategies*, 1998

[30] Kris De Volder *Type Oriented Logic Meta Programming for Java* 1998

[31] Dominic Welsh *Codes and Cryptography* OUP 1988

[32] Whitfield and Soffa *An Approach for Exploring Code Improving Transformations*

[33] Zelix – `http://www.zelix.com/`