UNIVERSITY OF OSLO
Department of informatics

Analysis of Obfuscated CIL code

Master Thesis
60 credits

Linn Marie Frydenberg

1st August 2006

Preface

This master thesis is the result of one year's work in order to fulfill the requirements for the Master degree at University of Oslo, Faculty of Informatics.

Acknowledgments

First of all, I would like to thank my supervisors, Torgeir Broen at Norwegian Defence Research Establishment (FFI) and Birger Møller-Pedersen at Faculty of Informatics, University of Oslo, for their excellent guidance, various thoughts and good comments during my thesis work.
I also want to thank Norwegian Defence Research Establishment (FFI) for making laboratory space available.

Last but not least, I would like to thank my boyfriend Alex for his love and support, and also my family and friends for their support and encouragement during my work.

Oslo, August 2007
Linn Marie Frydenberg

Abstract

This thesis focuses on a technique known as obfuscation. The area has been given much attention in recent years as a low cost technique for software protection. There are already numerous papers concerned with techniques for both obfuscation and deobfuscation, but there are still many untouched issues, and little have been done in practice.

The use of obfuscators in practice is one of the issues explored in this thesis, on the .NET platform. Most of today’s obfuscation research is designed and tested on the Java platform. It is therefore interesting to see which techniques that are implemented in the .NET obfuscators. The used .NET language in this thesis is C#.

Obfuscation is especially used on high level languages to increase the resilient towards reverse engineering. Reverse engineering is a big problem for software written in Java or .NET languages, because of their bytecode nature. Another issue in this thesis is therefore to look at the techniques used by the obfuscators and investigate if there are techniques which are vulnerable to reverse engineering. A classification of the different obfuscation techniques is therefore given to sort out which techniques that can be viewed as reversible and which are one-way.

One thing that is lacking in numerous papers about obfuscation is the actual process of reversing obfuscation techniques. Many papers propose techniques that can be used in reverse engineering, but none show the actual methods. This thesis will therefore try to reverse engineer the obfuscation techniques that are defined as reversible.

Contents

List of figures

List of tables

1 Introduction

In recent years software manufactures have been struggling against a growing problem – software piracy. The illegal copying and reselling of software applications and games has become a huge concern for all software vendors. Business Software Alliance (BSA) claimed that in 2006, 35 % of all software installed on personal computers worldwide was obtained illegally [6]. This corresponds to a worldwide loss of 40 billion dollars in 2006, and the amount seems to be rising every year.

Software developers (and manufactures) are therefore looking for methods which can protect their applications, both from illegal copying and reselling, but also against those who wants to steal algorithms (like secret company algorithms). The game industry has for instance developed numerous advanced protection algorithms to prevent illegal copying of games. However, most of these protection algorithms are broken after hours with reverse engineering techniques [51].

Protection against reverse engineering has therefore become an important strategy to prevent illegal copying, reselling and stealing of code. The need for protection against reverse engineering has also increased with the introduction of bytecode languages, which is much easier to reverse engineer than native, binary code. More and more software is distributed in bytecode formats rather than binary code., since there are several popular high level languages used in software development today, such as Java and C#.

Code obfuscation has in recent years attracted attention as a low cost approach for improving the resilience of software towards reverse engineering [8, 12-14, 43, 45, 46]. The research in this area has been, and still is very extensive. It is, however, a race between the research of obfuscation and deobfuscation since there are interests in both these areas. Those primarily interested in obfuscation wants to protect their software, while those who are interested in deobfuscation wants to break this protection.

Obfuscation is a technique that is meant to protect software by changing a program in such way that it is difficult to both reverse engineer and understand the decompiled code. It is though not possible to make it impossible. Obfuscation plays with the fact that humans are fickle and quick to tire. The most effective tactic against humans is therefore to make it too expensive in terms of either resources or time to reverse engineer a piece of software [44, 45].

Since obfuscation is merely a technique, it can also be used for negative purposes. Just as it can protect software, it can also hide malicious content. It is known that viruses that have been discovered use obfuscation in newer versions to fool the virus scanners and to make the job for the malware analysts harder. Although the virus scanners already has the original virus in their databases, they are not able to recognize the new obfuscated version [9]. In this context it is also interesting to find techniques to reverse engineering the effects of the obfuscation techniques.

The challenge in code obfuscation is therefore both to come up with obfuscation techniques that serves their purpose, but also in finding methods to reverse engineer these techniques when they are used in unwanted ways.

1.1 Scope of this thesis

This thesis will look into both obfuscation and deobfuscation. The research is extensive in both areas, but there are still many unanswered topics left to investigate.

In terms of obfuscation, much of the research has been performed on the Java platform, and the majority of the obfuscators that have been developed are also intended for the Java language. This thesis will therefore examine obfuscators on the .NET platform and investigate which techniques they use. It is interesting to see if these tools have included the same features that has been described in research and implemented in Java obfuscators.

Very little research presents results towards resilience of obfuscated .NET assemblies. After discovering the techniques that are used in the obfuscation process, some information about how resilient these techniques are towards reverse engineering should be obtained. There are some references in today's research on which methods that can be used for reversing obfuscation techniques, but no paper presents a practical approach. The techniques that are discovered will be evaluated using current knowledge to analyze if they are reversible or one-way. If any reversible techniques are found, we will try to reverse engineer them using the methods described in research. We will also use other techniques if needed.

Many obfuscators include features such as encryption, watermarking and tamper-proofing. In current research all these techniques are considered as software protection, but they are not regarded as obfuscation techniques [12, 13, 15]. Throughout this thesis we will use the same view of encryption, watermarking and tamper-proofing. If such options are present in the obfuscators, they are not included in the simulation settings. There is also already much literature available on these techniques [12, 42]. If there are other options in the form of optimization or techniques that eases the distribution of software, these are also not included in our classification, since they are not regarded as obfuscation techniques.

1.2 Problem statement

This thesis will focus on obfuscation and reverse engineering of .NET assemblies. It will look into the techniques used in today's obfuscation tools and try to answer the following questions:

- Which obfuscation techniques are one-way/not reversible?
- Which obfuscation techniques are vulnerable to reverse engineer?
- Is it possible to reverse engineer these obfuscation techniques in practice?

1.3 Research methods

In order to ensure an adequate analysis, while answering the problem statements above, three research methods will be used:

- Simulations will be used to identify obfuscation techniques that are used in today's obfuscators. The simulation environment is made up of several tools (covered in Chapter 3), including Visual Studio .NET, ILDASM, Salamander .NET, Spices .NET and DotFuscator.

- Analysis will be performed on the discovered obfuscation techniques to decide if they are one-way and reversible.

- Algorithms will be given for obfuscation techniques that have shown to be reversible. An extensive explanation together with pseudo code, will explain the algorithm used to reverse an obfuscation method.

1.4 Outline of this thesis

The next chapter provides background information on CIL code and common obfuscation techniques. The chapter also explains which techniques that is reversible or not, along with known methods for reverse engineering obfuscation techniques. Chapter 3 presents the simulation environment used in this thesis, both tools and approach that are used. In Chapter 4, the results from the simulations are explained, before they are evaluated and discussed in Chapter 5. The algorithms developed for the reversible obfuscation techniques are presented and evaluated in Chapter 6, before the conclusion is drawn in Chapter 7. The main contributions in this thesis are summarized in Chapter 8, before future work is outlined in Chapter 9. The last chapter holds the references.

Source code for all the code examples is included on the CD. The original CIL code and the obfuscated CIL code from the simulations are also included on the CD.

2 Background

This chapter gives a background of the various components used in this thesis. Most of the analysis in this thesis is performed on CIL (Common Intermediate Language) code, and therefore a basic understanding of this language is needed. CIL code is briefly described in section 2.1. Some analysis is also performed on C# code, and the reader is expected to have a basic understanding of high-level languages (like C# or Java).

The most common obfuscation techniques are described in section 2.2. The reader must have a basic understanding of the different obfuscation techniques that exists and how they operate.

Section 2.3 separates the obfuscation techniques into one-way techniques and reversible techniques. In section 2.4, the main techniques presented in current research for reverse engineering obfuscation techniques are described.

2.1 CIL code

Common Intermediate Language is an assembly like language, and a component of the Common Language Infrastructure (CLI) used on Microsoft .NET Framework [17, 24]. It is also known as Microsoft Intermediate Language (MSIL), which was the former name of CIL, before the standardization of the C# language. The ECMA standard and the ISO standard contains more information about both C#, CLI and CIL [16, 18, 19].

The CIL is a set of CPU-and platform independent instructions which is compiled with the Common Language Runtime (CLR), converting the CIL instructions to native code. In Java the corresponding compilation is performed, and the Java code is compiled into Java bytecode before it is compiled to native code [37]. The CIL is completely stack-based, meaning that all information is stored on an evaluation stack [17]. On the Microsoft .NET Framework this compilation is known as just-in-time (JIT) compilation [22]. Figure 2.1 (from [48]) illustrates the transformation of C# (and other .NET languages) to native code.

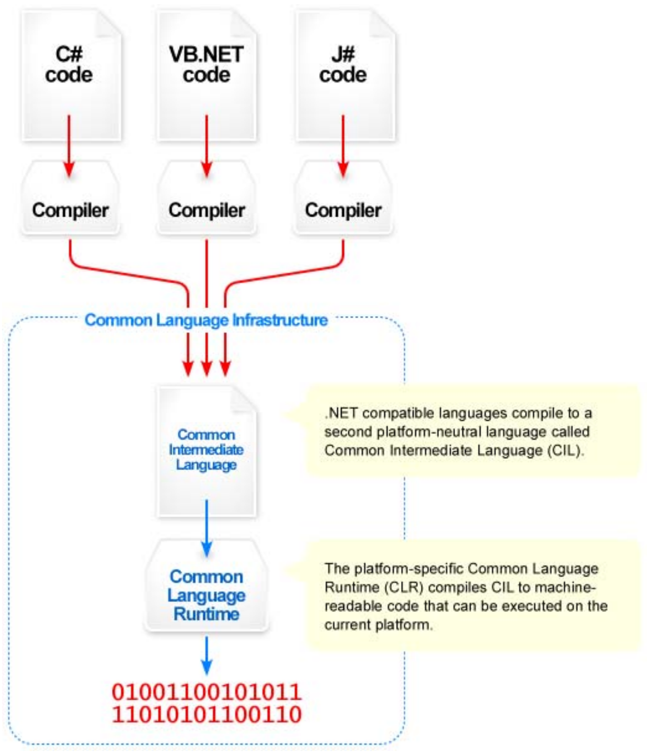

C# code
VB.NET code
J# code
Compiler
Compiler
Compiler
Common Language Infrastructure
Common Intermediate Language
.NET compatible languages compile to a second platform-neutral language called Common Intermediate Language (CIL).
Common Language Runtime
The platform-specific Common Language Runtime (CLR) compiles CIL to machine-readable code that can be executed on the current platform.
01001100101011
11010101100110

Figure 2.1: Transforming .NET languages to native code

The CIL instruction set contains over 200 instructions, and therefore only the most important instructions are covered here. For detailed information about the full instruction set, refer to the CLI specification document, partition III in [16]. A good overview of the CIL instructions can also be found in [29].

Table 2.1 contains a summary of the most common CIL instructions, together with some instructions that will be used in the later chapters.

Instrucion	Description	Comment
ldloc <uint16 (indx)>	Load local variable of index indx onto stack.	
ldloc.0	Load local variable 0 onto stack.	
stloc <uint16 (indx)>	Pop a value from stack into local variable indx.	
stloc.0	Pop a value from stack into local variable 0.	
ldarg <uint16 (num)>	Load argument numbered num onto the stack.	
ldarg.0	Load argument 0 onto the stack.	
starg <uint16 (num)>	Store value to the argument numbered num.	Pop value of the stack
ldfld <field>	Push the value of field of object (or value type) obj, onto the stack.	
stfld <field>	Replace the value of field of the object obj with value.	
ldc.i4 <int32 (num)>	Push num of type int32 onto the stack as int32.	
ldc.i4.0	Push 0 onto the stack as int32.	
call <method>	Call method described by method.	
callvirt <method>	Call a method associated with an object.	
ret	Return from method, possibly with a value.	
br <int32 (target)>	Branch to target.	Jump offset is 4 bytes
br.s <int8 (target)>	Branch to target, short form.	Jump offset is 1 bytes
brfalse <int32 (target)>	Branch to target if value is zero (false).	Decision made with
brtrue <int32 (target)>	Branch to target if value is non-zero (true).	The top item on stack
Beq <int32 (target)>	Branch to target if equal.	
bne.un <int32 (target)>	Branch to target if unequal or unordered.	
Bge <int32 (target)>	Branch to target if greater than or equal to.	Decision made with the
bgt <int32 (target)>	Branch to target if greater than.	two top items on stack
ble <int32 (target)>	Branch to target if less than or equal to.	
blt <int32 (target)>	Branch to target if less than.	
add	Add two values, returning a new value.	
sub	Subtract value2 from value1, returning a new value.	
mul	Multiply values.	Pop their arguments
div	Divide two values to return a quotient or floating-point result.	from the stack and
ceq	Push 1 (of type int32) if value1 equals value2, else push 0.	push the result in
cgt	Push 1 (of type int32) if value1 > value2, else push 0.	
clt	Push 1 (of type int32) if value1 < value2, else push 0.	
nop	Do nothing.	
ldstr <string>	Push a string object for the literal string.	
dup	Duplicate the value on the top of the stack.	pushes the duplicate on top of the stack

TABLE 2.1: SUMMARY OF CIL INSTRUCTIONS

2.2 Obfuscation techniques

The research of obfuscation techniques is a very young research field, even though there are many papers covering the area [5, 10, 11, 13, 15, 17, 45]. The definition of what obfuscation is has not been approved or standardized yet. The common perception of obfuscation is to transform a program, thus making it *harder* to understand than the original program if it is reversed engineered [11, 15, 17, 45].

An attempt to standardize a definition was attempted in [11] and also described in [15], by introducing various metrics for measuring the complexity of the code. These new metrics were later used in [5] to prove that obfuscation is impossible under these definitions. This definition will therefore not be used in this thesis.

With no accurate definition of what obfuscation is, the most common perception of obfuscation is used in this thesis – “Obfuscation is a technique which transforms a given program into a new program, which is harder to understand when decompiled”.

There is a vast amount of obfuscation techniques that has been proposed, and the most common methods used are presented in the following sections.

2.2.1 Renaming symbols

The technique of renaming symbols is described in most papers although under different names such as layout transformations [11], lexical transformations [13], variable renaming [15, 45]. Regardless of the name, the idea behind stays the same. When the original source code is compiled into CIL code, all the names of classes, methods, variable names, method parameters and so on are identical to their names in the source code, which is the opposite of native (binary) assemblies. This makes it easy to read and understand the code if it should be decompiled, since the various variable names are often given a name that represent its action or function. By changing all of these names into others, such as one letter or maybe a string like d8934fakc984fjoira3, it will make the code more difficult to understand when decompiled. This is especially the case for names that are “helpful” such as a variable called *index*, *counter* or *total*. A completely random name will make the understanding of the function worse.

Renaming a lot of methods, variables and such into the same name can also confuse the reader even more. Referring to several different types (ie. boolean, int and string) using the same variable name, like *a*, can confuse the reader even further. This is not allowed in the C# language because of the scope rules of the language, but it is legal in the CIL code. It is, however, legal to give different variable types the same name as long as they are in different scopes (true in both C# and Java). This will make it difficult to see which of the values that is being referenced at a specific point in the CIL code.

2.2.2 Control flow obfuscation

Control flow obfuscation is a general term for techniques that are concerned with altering the actual control flow in a program [10, 11], *Collberg et.al.* present several different methods of altering the control flow. In later papers [4, 45, 46] new techniques are presented, but these are often enhancements or a combination of the techniques in [10, 11]. The next sections will present some of the techniques that are most common in the current research literature.

2.2.2.1 Opaque predicates

Opaque predicates are extensively used in the work of *Collberg et.al.* in [10, 11, 13]. An opaque predicate P is a predicate (a boolean expression) whose value is constant and known at compile time. It is worth noticing that in [11] it is stated that also variables can be viewed as opaque if they hold the same properties as an opaque predicate.

The basic functionality of the opaque predicates are also described in [14, 15, 17, 52]. Opaque predicates are used for transforming a program block by introducing branching based on a predicate.

There are three different notations that are normally used for opaque predicates.
$\mathbf{P}^T$ denotes that the predicate will always be true. $\mathbf{P}^F$ similar denotes that the predicate always will be false and $\mathbf{P}^?$ denotes a predicate that can be either true or false.

Some examples of predicates which are always true (supposing that x and y are integers) are shown in Figure 2.2.

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

Figure 2.2: Examples of opaque predicates that are always true

Because the properties of the opaque predicates, they can be used to transform a program block B, like:

- **`if (P`^{T}`) B`**
  This will hide the fact that B will always be executed
- **`if (P`^{F}`) B'; else B`**
  This hides the fact that B always will be executed. B' can be a version of B which may contain errors
- **`if (P`$^?$`) B; else B';`**
  Here one can have to versions of B which can have the same functionality.

The opaque predicates described above can be used in to insert a branch into a block of statements. For more a more detailed study of opaque constructs and various implementation ideas see [11, 52].

2.2.2.2 Basic control flow flattening

Control flow flattening was first presented in [4, 46] and later in [45]. The technique aims at flattening the control flow graph in such way that all the program blocks appears to have the same predecessors and successors [45].

During execution the actual control flow is controlled by a dispatch variable. This variable is used by a switch block, in order to jump indirectly by a jump table to the indented successor. The dispatchvariable is given a value in each program block at runtime, which will identify which code block that should be executed next. Figure 2.3 (Figure 1 in [45]) shows an example program and its corresponding control flow graph.

```
int f(int i, int j)
{
  int a = 1;
  if (i < j) {
    a = j;
  }
  else
    do {
      a *= i--;
    } while (i > 0);

  return a;
}
```

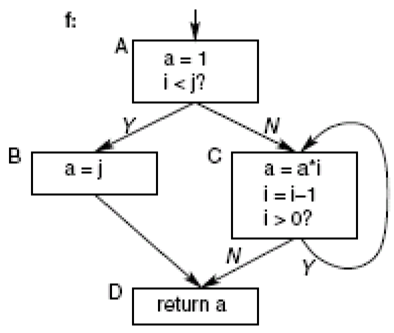


Figure 2.3: Example program and its control flow graph

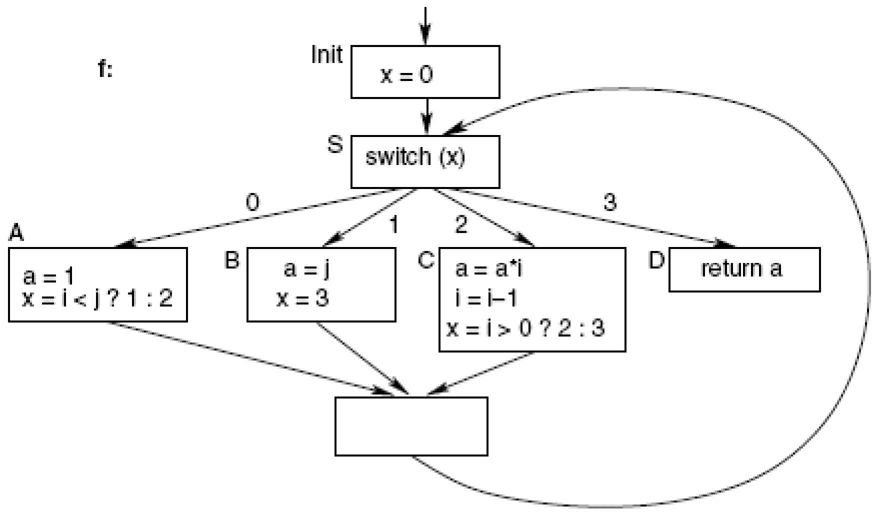


Figure 2.4: The control flow graph after emitting basic control flow flattening

Figure 2.4 (Figure 2 in [45]) shows how the control flow graph of the example program in Figure 2.3 looks like after basic control flow flattening has been applied. As seen in Figure 2.4, S is the switch block and the dispatch variable is x. The initiation of x shows that the program block A is the first block to be executed. Program block A contains the same functionality as it does in Figure 2.3 where it also is the entry block of f(). The flow of the program in Figure 2.4 is controlled by the various assignments to x in the program blocks.

The basic control flow flattening technique presented in this section does have some similarities to the Branch Insertion transformation described by *Collberg et.al.* in [10]. The main difference is that in [10] opaque predicates are inserted in the various program blocks to change the flow of the program as opposed to the switch statement and the dispatch variable introduced in the flattening technique. The basic control flow flattening technique does, however, seem to be more difficult to break than the use of opaque predicates [45].

2.2.2.3 Interprocedural Data Flow

In [45] *Udupa et.al.* present two enhancements to the basic control flow flattening technique. The first is called Interprocedural Data Flow and is presented in this section. The second enhancement is presented in section 2.2.2.4 and is based upon artificial blocks and pointers.

As seen from Figure 2.5, the values that are assigned to the dispatch value x in the program blocks are available within the program itself. Although the control flow of the program in Figure 2.4 is not obvious and more difficult to understand compared to the control flow in Figure 2.3, it can still be extracted rather easily by examining the constants that are assigned to x in the program blocks. This only requires intraprocedural analysis. (In terms of static analysis, intraprocedural is concerned with the flow from the program blocks entries to its exits – i.e. within the program block [30])

The enhancement purposed in [45] is interprocedural information passing, which can make the control flow flattening technique more resilient towards reverse engineering. (Interprocedural is the opposite of intraprocedural – it is concerned with the flow between the program blocks [30]). The idea is to use a global array for the dispatch variable values, so the values are no longer represented within the program blocks. At each call to the function, the values are written into a global array at some random offset within the array (the offset needs to be appropriately adjusted to avoid buffer overflows). The offset that is chosen may be different at the each call to the function, and is passed on as a global variable or an argument to the function.

Within the obfuscated code, the dispatch variable is assigned values from the global array. In this way neither the actual locations accessed, nor the contents of these locations, are constant values and are therefore not evident by examining the obfuscated code of the method. Figure 2.5 (Figure 3 in [45]) shows how the program example in Figure 2.3 would look after applying this obfuscation.

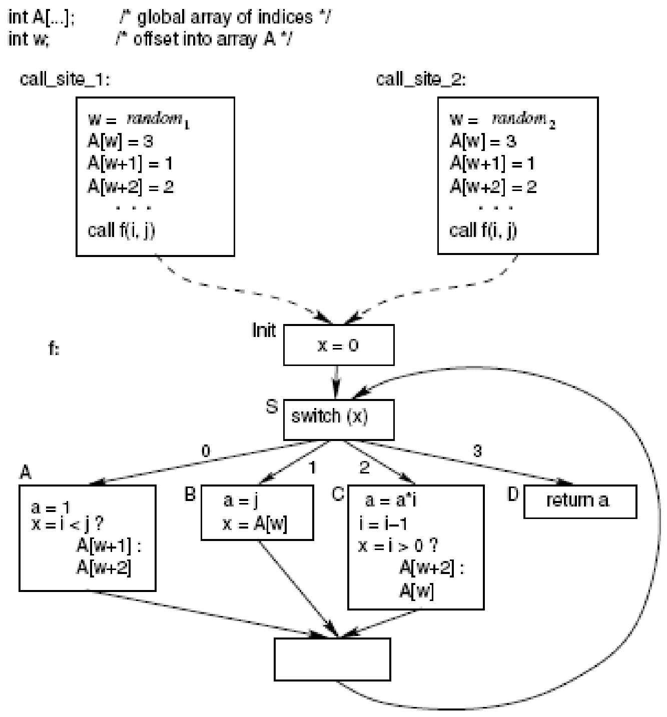


Figure 2.5: Enhancements of the basic flattening with Interprocedural Data Flow

2.2.2.4 *Artificial blocks and pointers*

The second enhancement presented in [45] is introducing artificial blocks and pointers. This method can be combined with the enhancement in section 2.2.2.3 to increase the resilience of the obfuscation technique even more. The idea is to add artificial blocks to the control flow graph. Some of these blocks will never be executed, but this is very difficult to determine by a static examination of the program, because of the dynamically computed indirect branch targets in the obfuscated code. It is although possible to retrieve information about which code blocks that are considered to be dead by dynamic analysis if it is possible to generate all the relevant input [47].

By adding indirect loads and stores through pointers in these unreachable blocks, it confuses static analysis about the possible value taken of the dispatch variable. Figure 2.6 (Figure 4 in [45]) shows the control flow graph of the program in Figure 2.3 after applying this technique.

Collberg et.al. presents similar ideas when they purpose the insertion of dead or irrelevant code [11]. They use the technique in combination with opaque predicates.

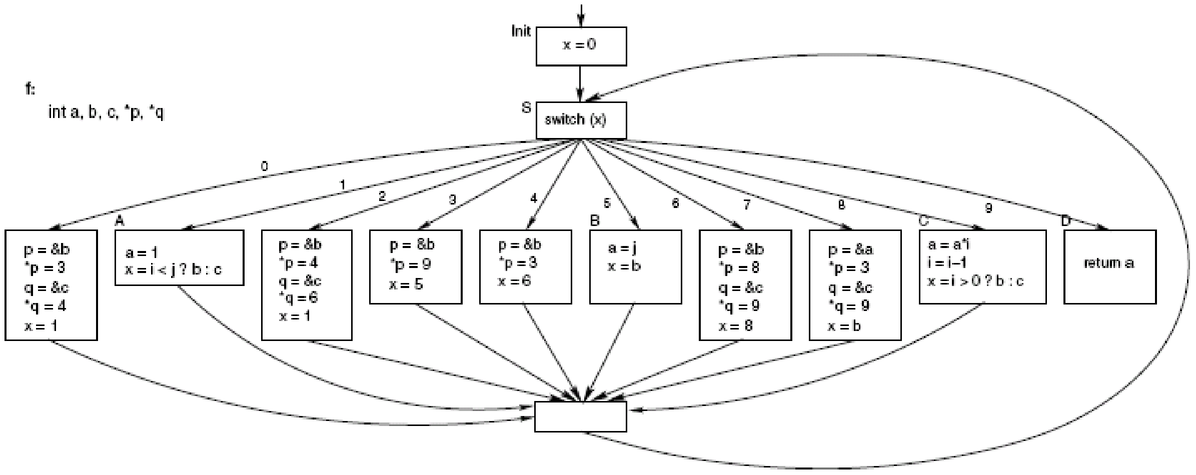


Figure 2.6: Enhanced flattening with artificial blocks and pointers

2.2.2.5 Transformations

This section will briefly explain the basic of transformations of variables and loops. In [15] both are described, but [17] only mentions variable transformations and [11] describes loop transformations. Since all the proposals are connected, they are presented together in this section.

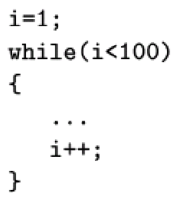
Transformations of variables in a program will make the code more difficult to understand when decompiled. This is because the intuitive meaning of variable values will not be immediately clear. There are many ways of changing the variables; such as bit shifting and transformations by functions [15, 17]. The latter technique is described here.

These techniques show how one can change an integer variable *i* inside a method. To make these transformations, two functions *f* and *g* are needed. They need to be bijective so that *g* can be the inverse of *f*: [15]

$f\colon\ X \rightarrow Y$
$g\colon\ Y \rightarrow X$

where $X \subseteq Z$. To replace the variable *i* with the new variable *j*, two types of replacements is needed:

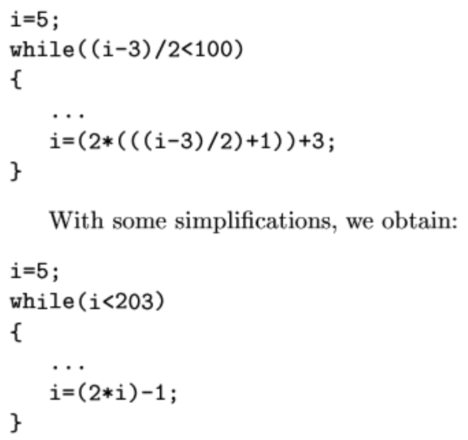
1. Any assignment of *i* of the form $i := E$
   is transformed to $j := f(E)$
2. Any uses of *i* (not a definition of *i*)
   is transformed to $g(j)$



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

Figure 2.7: The original while loop

This method can be used to transform a loop. By changing the variables it is possible to transform the while loop in Figure 2.7 into the while loop in Figure 2.8 (Both from section 1.2.3 in [15])



```
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;
}
```

Figure 2.8: While loop after transforming the variables

The transformation is performed with the functions defined as:

$$f : 2i + 3$$

$$g : \frac{i-3}{2}$$

It is also important to note that the statement **`i++`** (i.e. **`i=i+1`**) corresponds to a statement and an assignment (see the two types of replacements above).

There are also other known methods for changing a loop by introducing new variables and opaque predicates, and also by techniques known as loop blocking, loop unrolling and loop fission [11, 15]. These techniques are variants of the transformations described earlier in this section.

2.2.2.6 Inlining and Outlining

Inlining is a well-known compiler optimization technique that can be used in obfuscation [11, 17]. Inlining duplicates a method to any place in the program that calls the method. Thus instead of having all the calls directed to one copy of the real method, the calls are instead replaced with an actual copy of the body of the called method. Inlining is a powerful tool in obfuscation since it eliminates the internal abstractions created by the software developer.

Outlining is the opposite of inlining. It takes a certain code sequence that is originally in the content of one method and creates a new method that just contains this code sequence. It has been suggested to use outlining together with inlining [11]. It is important to notice that the outlining is an efficient technique if random pieces of code are created in the new methods. However, this can affect the efficiency of the program with all the extra calls made. Inlining is used in compilers to optimize efficiency.

2.2.3 Breaking Decompilation

Since byte-code based languages, like the CIL code, are highly detailed, they are not so difficult to decompile from bytecode executables into CIL code. There are also numerous reflectors which can disassembly executables into higher level languages such as C#, like the .NET Reflector (see section 3.4.3).

To prevent decompilation of executables, many obfuscators are trying to prevent disassembly of the obfuscated executable to higher level languages. One way of achieving this is by modify the binary or bytecode, so it contains statements that can not be translated into a high level language, because they are not “legal” instructions.

In [11] there is a description of how this could be achieved in the Java language. The same technique is described in [15] on the C# level. In C# (and in Java) there is a jump instruction called goto which allows the program to jump to a statement marked by the appropriate label. However, there are some restrictions; the goto instruction needs to be within the scope of the labeled statement. The consequence of this is that it is possible to jump out of a loop but not the other way around. It is also not possible to jump into a conditional statement. Figure 2.9 (from section 2.2.1 in [15]) shows an illegal C# program. The problem is that this jump is legal in CIL code, and will therefore compile without errors. It will break a decompillation, however, since it is not a valid C# statement.

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

Figure 2.9: Illegal goto condition in C#

There are two other general methods for preventing decompilation and disassembly on .NET assemblies [17]. ILDASM, which is the most common disassembler tool for the .NET Framework (see section 3.4.2), can be disrupted by changing some metadata entries. The metadata can be altered in a way that will produce an error from ILDASM when trying to disassemble the executable.

Another way to crash decompilers is to corrupt the metadata by inserting bogus references to nonexistent strings, fields or methods [17]. Some programs do not deal with such broken links in a proper way and simply crashes when loading corrupted metadata. But this is very program specific technique, and there is an ongoing race between the developers of the obfuscators and decompilers.

2.2.4 Other obfuscation techniques

Beside the obfuscation techniques described in the previous sections, there are still a lot of other techniques that can be applied. Some of those that are most mentioned is:

- Array transformations – reordering arrays [15]
- Interleaving – merges two methods into one [11]
- Cloning – creates several copies of the same method [11]
- Table interpretation - break code sequences into multiple chunks to hide the structure of the program [11, 17]

2.3 Classification of the Obfuscation techniques

One of the goals in this thesis is to look at which obfuscation techniques that are used in practice and which of these that are possible to reverse engineer. I order to do that, a classification of which obfuscation techniques that are considered reversible and one-way is needed. The following section will briefly classify the obfuscation techniques described in section 2.2.

2.3.1 One-way obfuscation techniques

In this context an obfuscation technique is considered to be one-way if, after the obfuscation technique is performed on the program, there is nothing present in the program that makes it possible to undo the changes that were performed by the obfuscation technique.

The obfuscation techniques from the previous chapter that are viewed as one way are therefore:

- Renaming of fields. After the renaming has taken place there will be no information about what the name originally was.
- Inlining. By replacing a call with the function that is called it is impossible to say whether or not this was added or not.

Even though it will never be possible to obtain the actual code after these alterations, it is some techniques that can make the obfuscated code easier to understand. For example by using a parser to map out the different fields, using the scope of the language, and rename them in a proper way to reverse the different symbols [45].

2.3.2 Reversible obfuscation techniques

An obfuscation technique is considered to be reversible if, after the obfuscation technique is performed on the program, it is possible to undo the changes in the program. Since the goal of obfuscation techniques are to make the code more difficult to understand, the opposite goal is to make the code easier to understand for the reader. It is impossible, however, to ensure that the reversing of the obfuscated code does not alter the original source code too.

In this context the obfuscation techniques from the previous chapter that are reversible are:

- All the control flow obfuscation techniques minus the inlining technique. They all change the flow of the program, but the original flow is still presented in the program, though hidden beneath the modifications.
- Breaking decompilation. The various techniques are based on adding bogus code to some extent. This code is therefore possible to remove.

2.4 Known Methods for Reverse Engineering Obfuscated Code

As described in section 2.2 there has been and still is extensive research in the obfuscation field, with the purpose of finding better obfuscation techniques. There is, however, also research in a field with the opposite goal; deobfuscation, or reverse engineering of the obfuscation techniques. However, there are not many papers out today that describes the actual method that are used in the deobfuscation process.

The techniques that are explained in this section is collected from four different papers [7, 11, 45, 52].

In Gregory Wroblewski's [52], a list of the most common ways of deobfuscation techniques that is based upon his own and fellow research authors experiences are presented. The techniques that are mentioned here are:

- Identifying and evaluating opaque constructs – allows to detect and remove inserted opaque predicates [11].
- Identification by pattern matching – comparison of fragment with database of patterns to detect inserted patterns. The database is built with strategies employed by known obfuscators [11].
- Identification by program slicing – is one of the classic methods of decompilation, can detect unimportant fragments of code [11].
- Statistical analysis – analysis of partial results in code extracted during program execution. Used for analyze the outcome of predicates present in the code [11].
- Evaluation by data flow analysis – classic method of optimization, allows to bind separated fragments of original code and remove inserted code (data flow analysis is one kind of static analysis [30]) [11] .
- Evaluation by theorem proving – allows to obtain result of program without its execution, useful only for simple constructions [7, 11] .

In [45], two other methods that can be used in reverse engineering of obfuscated code is presented:

- Program cloning (which is also considered an obfuscation technique) – clone portions of the program such that the spurious paths no longer join the original execution path. Has to be applied judiciously.
- Static Path Feasibility Analysis – static analysis, determine an (acyclic) execution path is feasible.
- The two techniques can and should be combined.

3 Simulation Environment

This chapter will describe the different methods, code example programs and tools which are being used throughout this thesis. Section 3.1 describes the methods that are used for mapping out the different obfuscation techniques used in obfuscation tools. The code examples that are used to expose obfuscation techniques are explained in section 3.2, before the obfuscation tools are discussed in section 3.3. Other tools of relevance for this thesis are given a short explanation in section 3.4

3.1 Methods

To find the obfuscation techniques used by today's obfuscation tools, a simple and efficient method was created which in turn can be broken down into five parts.

1. Compile a code example without any form of obfuscation
2. Compile the code example with one (if possible) obfuscation technique applied
3. Reverse engineer the compiled EXE file from both compilations, so the CIL code is obtained
4. Compare and analyze the non obfuscated CIL code with the obfuscated CIL code
5. If there are remaining obfuscation techniques, start again from part two

It is important to make the process of mapping obfuscation techniques as simple as possible so many obfuscation techniques can be found. By using the approach described above, the obfuscation techniques will be possible to analyze in smaller parts. However, much of this is determined by the obfuscation tool used and whether it is possible to enable only specific kinds of obfuscation technique (step two in the method above). For example, if it is possible to obfuscate code using only variable renaming obfuscation or only control flow obfuscation.

Step two in the method above was performed 32 times in order to determine if an obfuscation technique was random or not. This is needed to determine if an obfuscation technique is random or static. For example if a variable renaming obfuscation technique is using random variable names or uses pre-generated variable names. In simulations, 32 is considered a statistical good foundation [20]. By obfuscating a code example 32 times, a random behavior of an obfuscation tool will probably also be revealed. For example if the obfuscation tool will only use a control flow obfuscation techniques 50% of the time.

In addition to the method used for mapping the obfuscation techniques, a more sophisticated approach is used when trying to reverse engineer obfuscated CIL code. Obfuscation techniques that are found (using the method above) will be classified according to section 2.3 in section 6.1. Techniques that are classified as reversible will then be further analyzed in order to develop a general reversible algorithm for that obfuscation technique.

If it is possible to find an algorithm, both pseudo code and a general description of the algorithm will be used to explain the reversible algorithm for the obfuscation technique.

3.2 Code examples

To analyze the obfuscation techniques used in today's obfuscator, four different code examples were written. All code, both C# and CIL code, can be found on the CD. The different code examples were developed with Microsoft Visual Studio (see section 3.3.2 for more detailed information).

The code examples were designed to trigger different obfuscation techniques, as will be explained in the following sections.

3.2.1 Code example 1

The first code example is a small, simple program where all the functionality is in the main method of the program. The main method is the only method in the example. The code was inspired by Figure 2.3 which is a figure of an example program used to test the basic control flow flattening technique. The program either writes a random large number or zero to the screen in a loop.

In the code example there are many interesting features, regarding how obfuscation is done. There are several system calls, two if statements, a while loop and multiple variables. All in all, not a very complex program, which is also the intension. The main motivation behind this code example is to trigger control flow obfuscation and renaming of symbols. If the obfuscation tool includes protection against decompilation, this should also be triggered. It should be easy to analyze which control flow obfuscation techniques that are used, because of the small program size.

3.2.2 Code example 2

In the second code example, the program size is also small given the same reason as with example one. There is only one method in this example, the main method. The program writes today's date into a text file before reading this information and writing the information on the screen.

This code example contains many interesting features regarding how obfuscation is applied. There are several system calls and some variables involved, but there are no conditions in form of loops and if statements. The complexity of this program is even smaller than the first code example, which is intentional. The main motivation is to trigger control flow obfuscation and see what alteration the obfuscation will perform since there are no conditions that the control flow is usually applied to. Renaming of symbols should also be present here and protection against decompilation. It should be easy to analyze the obfuscation in this example, because of the small size.

3.2.3 Code example 3

In the third code example the code from examples one and two is combined. The first code example is copied into the method called numbers and the second code example into the method called writeDate. The Main method calls both these method respectively.

There are many reasons for choosing this as the third example. First of all, it is interesting to see whether or not the obfuscation tools apply the same obfuscation techniques on this code, as used on code example one and two. If not, this will indicate dynamic obfuscation techniques.

Secondly, the code example is also larger and contains more methods than the previous code examples. This can trigger obfuscation techniques that are only applied if the program's size is over a specific threshold. This could be an internal defense mechanism against obfuscation analysis. If such a mechanism exists, we have to assume that the size of example four is considered normal.

3.2.4 Code example 4

The largest code example is number code example four. The program is a solution to the known problem in programming algorithms called the "Eight queen puzzle" [49]. The program creates a solution of how n queens can be placed on an n x n chess board without setting each other in check. One can choose to have the actual chessboard printed out with placement of the queens shown, or to only print the number of distinct solutions. The time it takes to solve the puzzle is also printed out.

The code in example four contains four different classes and a lot of methods. The classes also include sub-classes. This should demonstrate if there are differences in how classes are renamed or altered in the obfuscation process.

The content of the code should also trigger control flow obfuscation, because the control flow in the program is very complex with many methods containing loops, branch statements and conditions. The size of the code example should also be more than large enough if there are obfuscation techniques that need a certain threshold before being initiated. As the other code examples, protection against decompilation should also be present in the obfuscated result of this example.

3.3 Obfuscators

Three different obfuscators were chosen so the results from the simulations should reflect a trustworthy and realistic situation of the obfuscation techniques used in today's obfuscators. The use of different obfuscation tools means that there will be differences in the obfuscation techniques used on the code examples. This will point out which techniques that are more common and which is more vendor specific. The focus should be on the commonly applied obfuscating techniques. It is also interesting to see whether or not the results show the use of obfuscation techniques which are not presented in literature at writing date.

3.3.1 Salamander .NET Obfuscator by Remotesoft

The Salamander .NET Obfuscator is referenced in current literature about reversing .NET [17] and is also listed on Microsoft's list over obfuscators which can be used on C# [25]. On the homepage of the product [32] the implemented features of the obfuscator includes renaming of symbols such as variables, classes, methods and interfaces among others. Control flow obfuscation and string encryption are also implemented.

Because of the features in Salamander .NET Obfuscator, it seems like a tool suitable for the simulation environment in this thesis. It also includes protection against the ILDASM disassembler. As discussed in section 2.4, control flow obfuscation techniques and also protection against decompilation is viewed as reversible. Hopefully these features will provide some interesting results that can be used for reverse engineering these types of obfuscation techniques.

In the simulations a trial version of the Salamander .NET Obfuscator was downloaded. The information from the vendor specifically states that the trial version contains full functionality; hence buying the software was not necessary [33].

When an evaluation copy of the obfuscator is downloaded, a whole package is distributed called Remotesoft .NET Explorer. This explorer is a generic object browser and MSIL disassembler and also has the Salamander .NET Protector installed. The protector is a software tool for code protection through encrypting and replacing CIL code with native source code.

To get the obfuscated code needed for the analysis, all four test code examples were run through the Salamander .NET Obfuscator with different degrees of options enabled. All functionality of the Salamander .NET Protector was turned off, since this is not within the scope of this thesis. The version of Remotesoft .NET Explorer used for simulations was the 2.0.0 version.

Settings

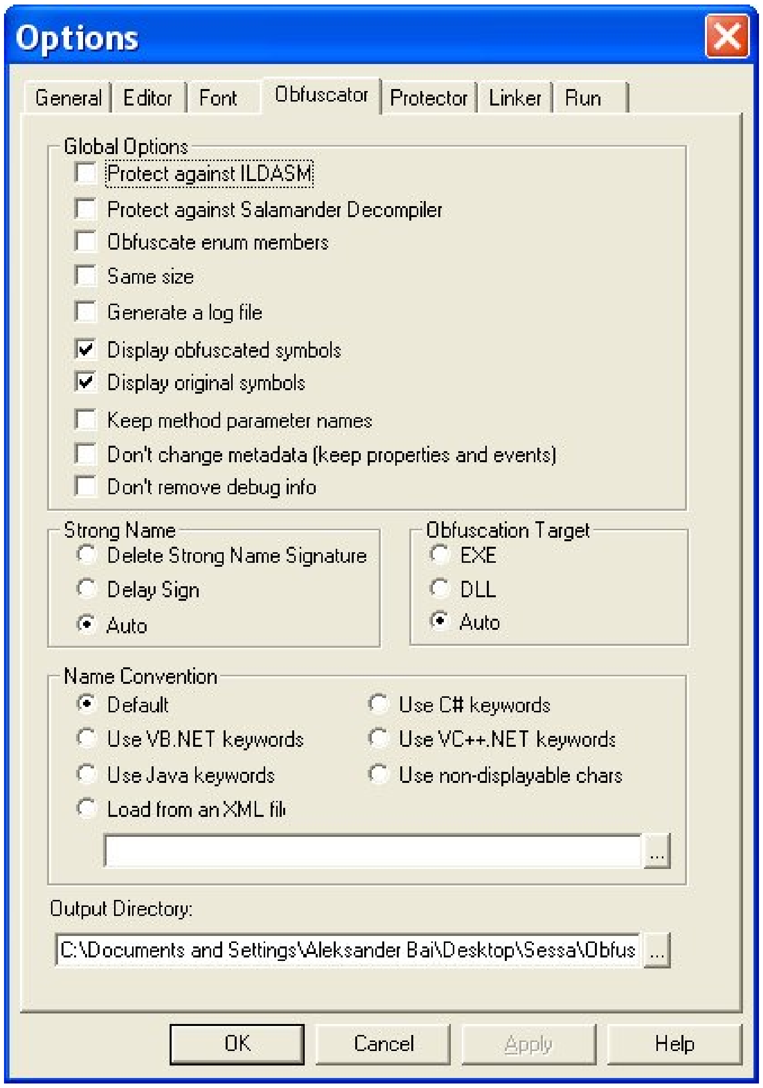
Figure 3.1 shows the different options provided in the Remotesoft .NET Explorer. The most interesting options in this figure, is the ILDASM option. It is difficult to know how this option is implemented, if it changes the actual CIL code or if extra code is added. Either way it is interesting to see if this protection method increases the resilience of the CIL code towards reverse engineering.

As shown in Figure 3.1, the list of options does not contain any references to control flow or renaming techniques . Based on the reported features of the tool, this is a bit surprising. It is possible that this is a part of the default implemented obfuscation and can therefore not be excluded.

The four different test code examples were tested with the following settings:

1. without any extra options turned on other than the default as shown in the picture
2. with the protection against ILDASM in additional to the first setting

TABLE 3.1: SETTING FOR THE SALAMANDER .NET OBFUSCATOR



Global Options
- [ ] Protect against ILDASM
- [ ] Protect against Salamander Decompiler
- [ ] Obfuscate enum members
- [ ] Same size
- [ ] Generate a log file
- [✓] Display obfuscated symbols
- [✓] Display original symbols
- [ ] Keep method parameter names
- [ ] Don't change metadata (keep properties and events)
- [ ] Don't remove debug info

Strong Name: Delete Strong Name Signature, Delay Sign, [✓ Auto]

Obfuscation Target: EXE, DLL, [✓ Auto]

Name Convention: [✓ Default], Use C# keywords, Use VB.NET keywords, Use VC++.NET keywords, Use Java keywords, Use non-displayable chars, Load from an XML fil

Output Directory: C:\Documents and Settings\Aleksander Bai\Desktop\Sessa\Obfus

Figure 3.1: Options for the Salamander .NET Obfuscator

The options for obfuscating enumerated members is a option that is used if a distributed software needs changes, and with this option it is possible to obfuscate the extra patch and distribute only the new patch instead of the software as a whole. This is beyond the scope for this thesis and was therefore not tested. The same applies to the other options such as the protection against Salamander Decompiler and same size option.

3.3.2 Spices .NET Obfuscator by 9Rays.NET

The tool from 9Rays.Net called Spices .NET Obfuscator [3] was the second product chosen as an obfuscation tool for this thesis. The obfuscator from 9Rays.NET was also referred in [17, 25]. The information provided by the manufacturer describes a well designed obfuscation tool. Some of the features include entity renaming, control flow obfuscation, method call anonymization and ILDASM protection.

The Spices .NET Obfuscator is included in a package called Spices .NET when downloaded. On the manufacturers homepage several highly reputable companies are listed as selected customers using the Spices .NET Obfuscator [2]. Some of the companies mentioned are Microsoft, Nokia and CSC. This supports the impression of a professional obfuscation tool that should deliver the results and features as described and is therefore a representative tool for the simulations in this thesis.

To create a simulation environment with the Spices .NET Obfuscator, a trial version of the software was downloaded. The FAQ on the homepage [1] emphasized that the only difference between the full version and the trial is that the trial version marks the obfuscated assembly in such way that distributing the obfuscated code would be useless. The documentation of the Spices .NET Obfuscator does not describe how this marking is performed.
However, since this thesis is interested in locating and analyzing the different obfuscation techniques used by the different obfuscation tools, this clear marking of the obfuscated CIL code might make the analysis easier.

The Spices .NET version used is 5.2.0.2

Settings

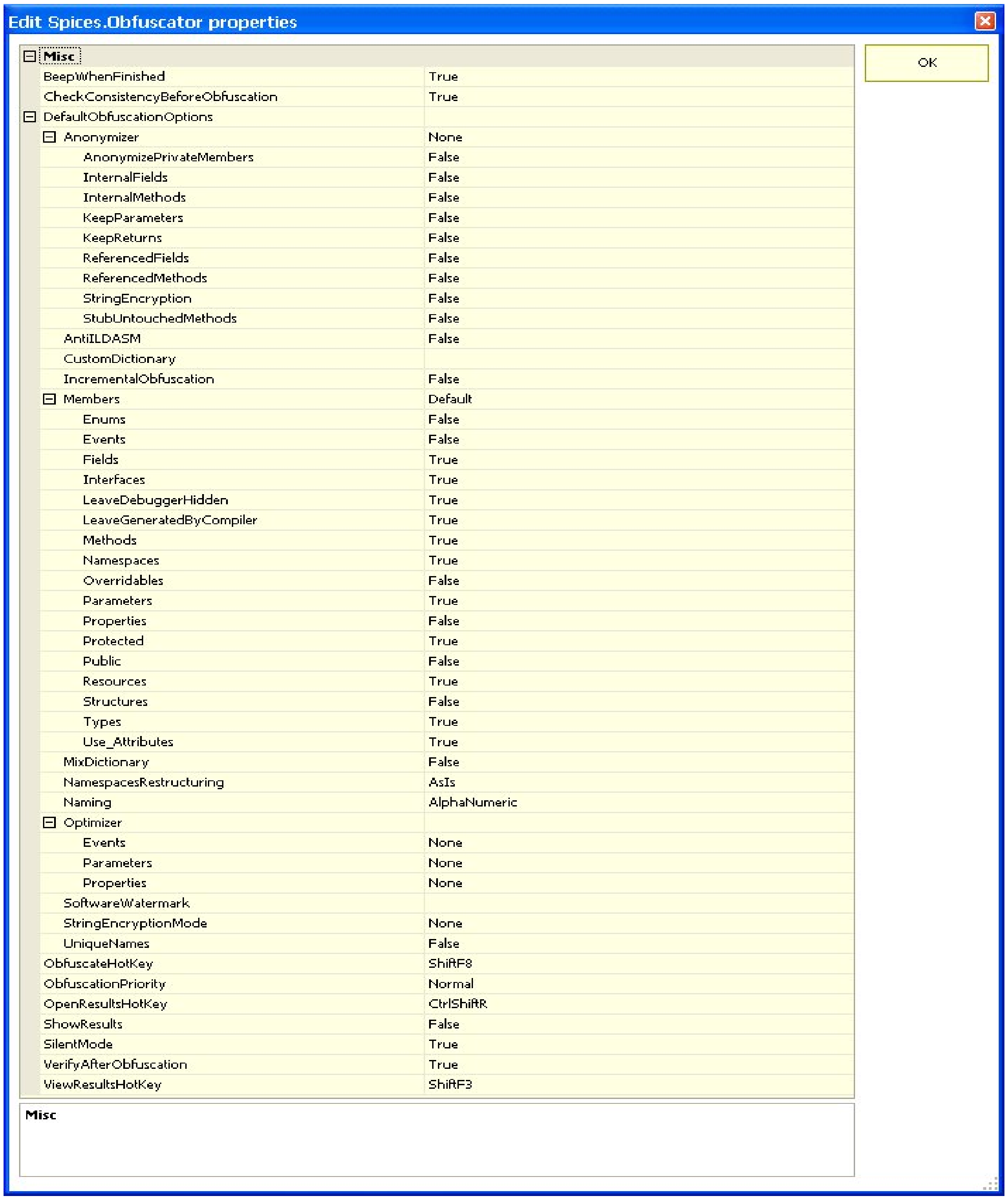
The different options of the Spices .NET Obfuscator is shown in Figure 3.2.

The code examples were tested with these different settings:

1.	with no extra options as the figure shows (default), which means the members option set to default
2.	with ILDASM protection, the option set to “true”
3.	with ILDASM protection, the option set to “complete”
4.	with members and anonymizer options set to default, but anonymizer is without the two string encryption options
5.	with members on full and anonymizer on high, but without the string encryption options

TABLE 3.2: SETTINGS FOR THE SPICES .NET OBFUSCATOR

As shown in Table 3.2 the string encryption option was not tested in the simulations, since this is outside the scope of the thesis. It is also worth noticing that Spices .NET Obfuscator does not have any options regarding control flow obfuscation, just like the Salamander .NET Obfuscator. Beforehand it is not possible to say if this functionality is a default obfuscation operation that can not be left out.



Misc	
BeepWhenFinished	True
CheckConsistencyBeforeObfuscation	True
DefaultObfuscationOptions	
Anonymizer	None
AnonymizePrivateMembers	False
InternalFields	False
InternalMethods	False
KeepParameters	False
KeepReturns	False
ReferencedFields	False
ReferencedMethods	False
StringEncryption	False
StubUntouchedMethods	False
AntiILDASM	False
CustomDictionary	
IncrementalObfuscation	False
Members	Default
Enums	False
Events	False
Fields	True
Interfaces	True
LeaveDebuggerHidden	True
LeaveGeneratedByCompiler	True
Methods	True
Namespaces	True
Overridables	False
Parameters	True
Properties	False
Protected	True
Public	False
Resources	True
Structures	False
Types	True
Use_Attributes	True
MixDictionary	False
NamespacesRestructuring	AsIs
Naming	AlphaNumeric
Optimizer	
Events	None
Parameters	None
Properties	None
SoftwareWatermark	
StringEncryptionMode	None
UniqueNames	False
ObfuscateHotKey	ShiftF8
ObfuscationPriority	Normal
OpenResultsHotKey	CtrlShiftR
ShowResults	False
SilentMode	True
VerifyAfterObfuscation	True
ViewResultsHotKey	ShiftF3

Misc

Figure 3.2: Options for the Spices .NET Obfuscator

3.3.3 DotFuscator by PreEmptive Solutions

The DotFuscator obfuscation tool by PreEmptive Solutions [38] seems to be the most commonly used obfuscator, and it is the most widely mentioned obfuscator in the research field [17, 25, 45]. One of the reasons for this is probably because of their involvement with Microsoft and especially with the Visual Studio tool. The community edition of DotFuscator is a part of the Visual Studio 2005 [28].

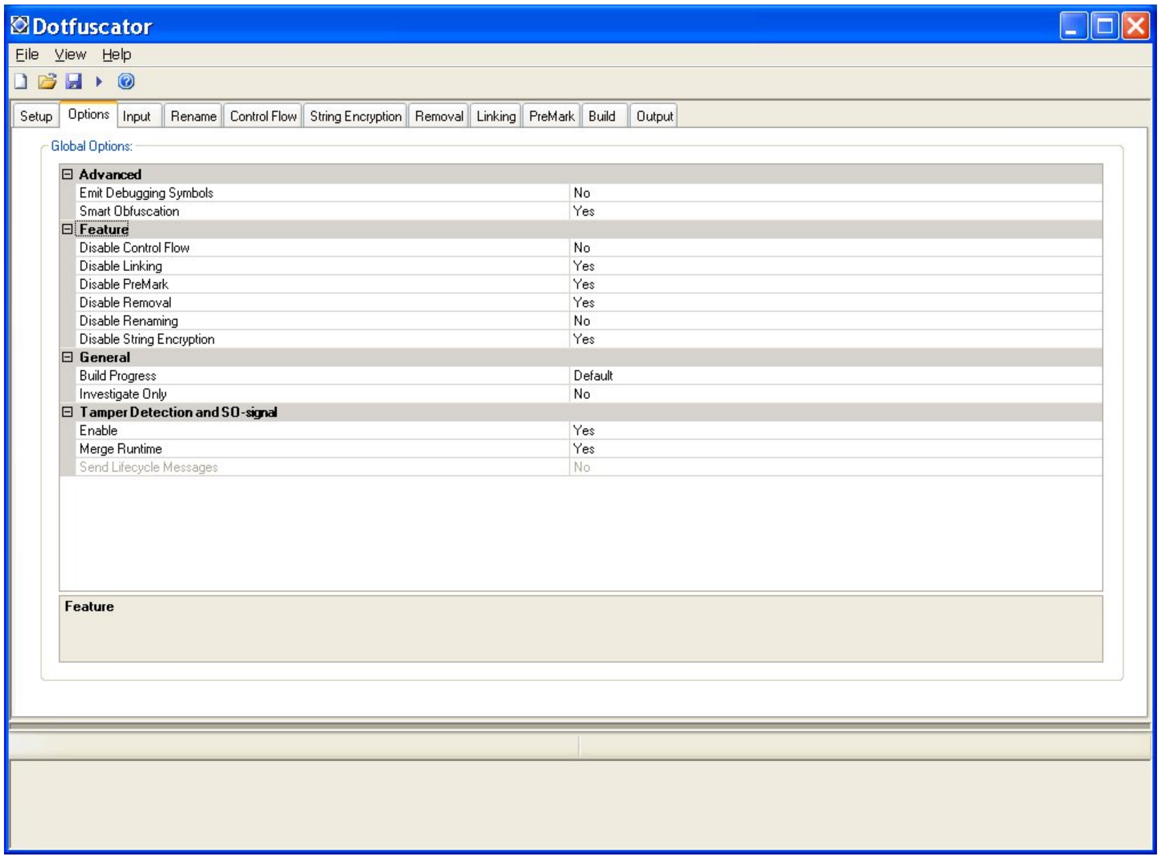
The DotFuscator states that it has implemented a lot of obfuscation features, like control flow obfuscation, renaming, string encryption and also protection against various decompilers [41]. Hopefully the features will produce obfuscated CIL code which can be used in an approach to reverse engineer the applied obfuscation techniques.

A trial version of the DotFuscator was obtained from the manufacturers by a request from their home page. The evaluation is a full copy of the Professional Edition of the tool and there are no limitations compared to the original product. The version of DotFuscator used for simulations in this thesis is the 4.1.2743.30058 version. More detailed information about the DotFuscator can be found in the user manual [40].

Settings

As a simulation environment the DotFuscator provided several possible options, but not all of these were interesting for this thesis. Figure 3.3 shows the different options included in the obfuscator.

The Linking option in DotFuscator is only interesting for distributing a patch for upgrading software, but this is out of the scope for this thesis as already explained. The watermarking and string encryption options provide more security for the obfuscated code, but also they are outside the scope of this thesis. The removal option is optimization of code in terms of sorting out and removing unused parts of the program. This is an interesting feature, however, not in the context of this thesis.


Dotfuscator
File View Help
Setup Options Input Rename Control Flow String Encryption Removal Linking PreMark Build Output
Global Options:

Advanced	
Emit Debugging Symbols	No
Smart Obfuscation	Yes
Feature	
Disable Control Flow	No
Disable Linking	Yes
Disable PreMark	Yes
Disable Removal	Yes
Disable Renaming	No
Disable String Encryption	Yes
General	
Build Progress	Default
Investigate Only	No
Tamper Detection and SO-signal	
Enable	Yes
Merge Runtime	Yes
Send Lifecycle Messages	No

Feature

Figure 3.3: Options for DotFuscator

1.	Default setting which means default renaming and high control flow obfuscation
2.	Only default renaming
3.	Renaming with the Enhanced overload option on
4.	Only control flow on level low
5.	Only control flow on level medium
6.	Only control flow on level high

TABLE 3.3: SETTINGS FOR DOTFUSCATOR

3.4 Other tools used

3.4.1 Microsoft Visual Studio 2005

Microsoft Visual Studio is a set of development tools which can be used to develop programs on the.NET platforms. The C# language is the language used in this thesis. For further information about Visual Studio, see [27].

In this thesis Visual Studio was used for developing the code examples discussed in section 3.2. The version used was version 8.0.50727.42.

3.4.2 MSIL Disassembler - ILDASM

The MSIL Disassembler, popular called ILDASM, is a basic part of the Microsoft .NET Framework package. The ILDASM used in this thesis was the 2.0.50727.42 version from the SDK 2.0 version of the .NET framework [23].

ILDASM was used in this thesis to extract the CIL code from the executables (both from the code examples and the obfuscated versions), although some of the obfuscated executables were not possible to decompile in ILDASM. Most of the figures showing CIL code in this thesis are text files that the ILDASM dumps the complete CIL code into.

Throughout this thesis the MSIL Disassembler is referenced by both its full name and also as ILDASM.

3.4.3 Lutz Roeder's .NET Reflector

The Lutz Roeder's .NET Reflector is a freeware program developed by Lutz Roeder, who is an employee in Microsoft Research department. His homepage is found on [35] and the .NET Reflector can be downloaded from this site [36].

The .NET Reflector was used in this thesis to decompile CIL code into C# code when possible. It was also used for navigating through both C# and CIL code. The .NET Reflector is referred to as Lutz Roeder's .NET Reflector and .NET Reflector throughout this thesis. The version of the tool used was version 5.0.25.0.

3.4.4 ILASM

ILASM is another tool included in the Microsoft .NET Framework, and it is the opposite of the ILDASM tool described in section 3.3.2.2. ILASM can generate executables from CIL code; hence the use in this thesis was mostly when tracking the changes made by the different obfuscators. For some obfuscated CIL code the .NET Reflector would not be able to show the actual C# source code of the program. The solution to this was to alter the CIL code (by removing the lines that made the decompiler crash) and then use ILASM to create a new executable that could be tested for decompilation in .NET Reflector.

More details about the ILASM tool can be found on [21]. The version of ILASM used in this thesis was the 2.0.50727.832

3.4.5 Diff method from Linux

diff is a method which finds the differences between two files. The method is explained in more detailed in [31]. *diff* was used in this thesis to compare the CIL code extracted from two different simulations in the obfuscators. The diff method was thus a method to ensure that results that looked identical were in fact identical, or to show the differences between two results.

4 Simulation Results

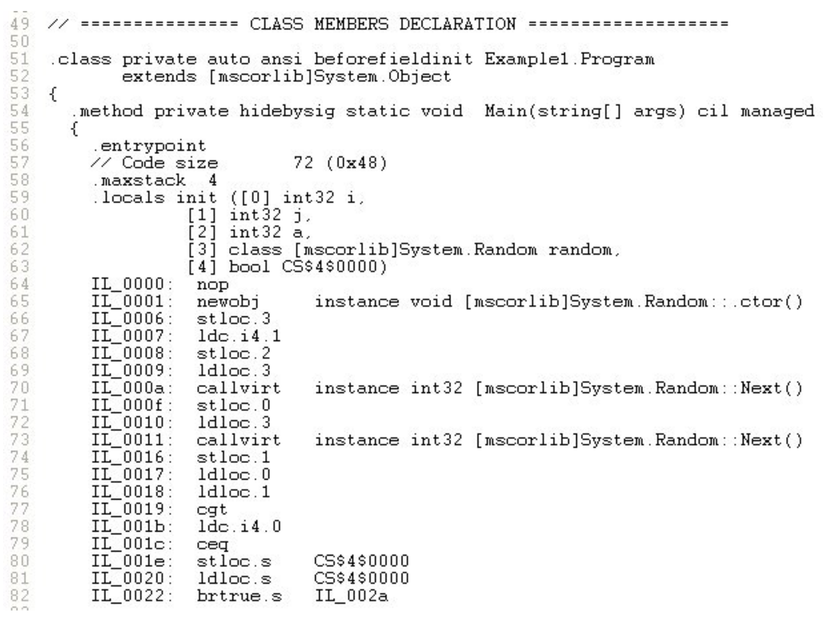
This chapter presents the results from all the simulations. The results are presented under the obfuscator tools used, and are explained briefly to point put which techniques that were used. The techniques that are found in this chapter will be evaluated further in chapter 5. Chapter 6 will thereafter classify them and try to reverse the techniques that are considered to be reversible.

A general notice is that the obfuscated executables in all the following section has been tested. If no comment regarding the execution of these is present in the sections, then the output of the obfuscated executables are similar to the original executables.

4.1 Salamander .NET Obfuscator

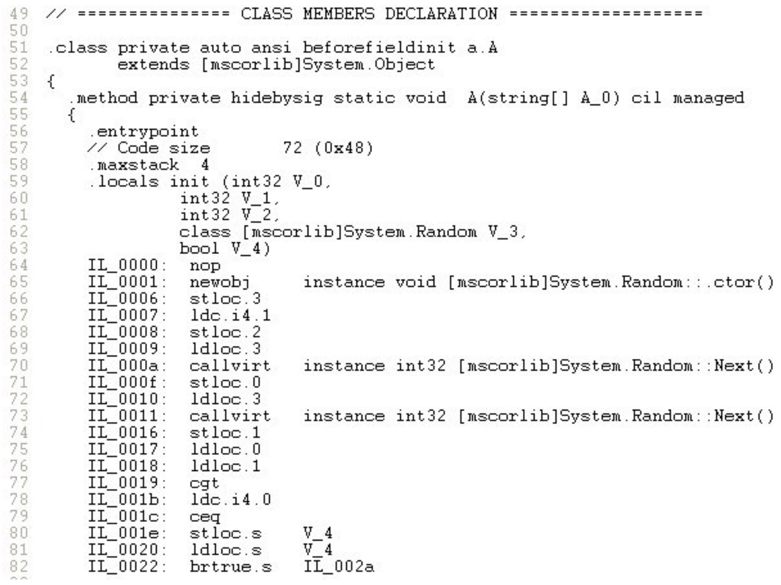
4.1.1 CIL code of example 1

The first code example is a small, simple program where all the functionality is in the main method of the program. The program writes a random large number or zero to the screen. Hopefully this program will initiate the control flow obfuscation on the loops in the program. The results from renaming of fields and also protection against disassembling should also be present.



```
// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit Example1.Program
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  Main(string[] args) cil managed
  {
    .entrypoint
    // Code size       72 (0x48)
    .maxstack  4
    .locals init ([0] int32 i,
             [1] int32 j,
             [2] int32 a,
             [3] class [mscorlib]System.Random random,
             [4] bool CS$4$0000)
    IL_0000:  nop
    IL_0001:  newobj     instance void [mscorlib]System.Random::.ctor()
    IL_0006:  stloc.3
    IL_0007:  ldc.i4.1
    IL_0008:  stloc.2
    IL_0009:  ldloc.3
    IL_000a:  callvirt   instance int32 [mscorlib]System.Random::Next()
    IL_000f:  stloc.0
    IL_0010:  ldloc.3
    IL_0011:  callvirt   instance int32 [mscorlib]System.Random::Next()
    IL_0016:  stloc.1
    IL_0017:  ldloc.0
    IL_0018:  ldloc.1
    IL_0019:  cgt
    IL_001b:  ldc.i4.0
    IL_001c:  ceq
    IL_001e:  stloc.s    CS$4$0000
    IL_0020:  ldloc.s    CS$4$0000
    IL_0022:  brtrue.s   IL_002a
```

Figure 4.1: Original CIL code from example one

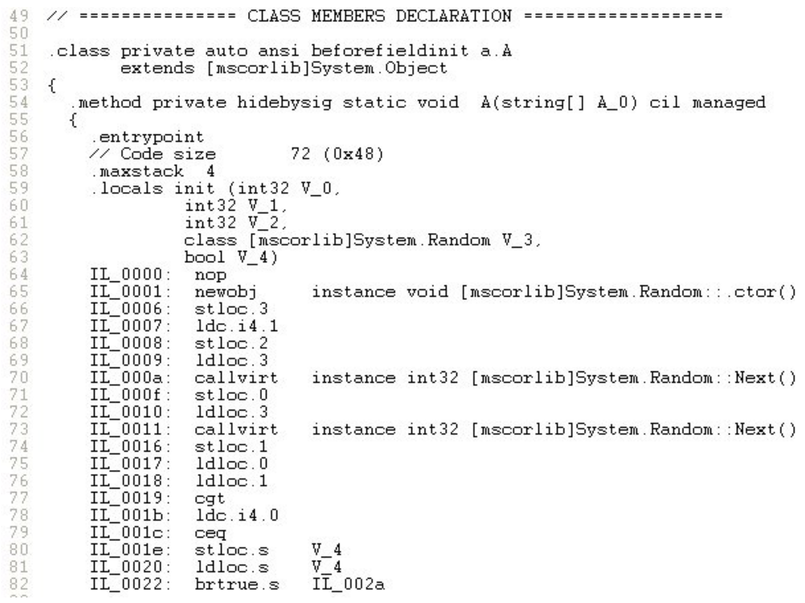


```
49 // =============== CLASS MEMBERS DECLARATION ===================
50
51 .class private auto ansi beforefieldinit a.A
52        extends [mscorlib]System.Object
53 {
54   .method private hidebysig static void  A(string[] A_0) cil managed
55   {
56     .entrypoint
57     // Code size       72 (0x48)
58     .maxstack  4
59     .locals init (int32 V_0,
60              int32 V_1,
61              int32 V_2,
62              class [mscorlib]System.Random V_3,
63              bool V_4)
64     IL_0000:  nop
65     IL_0001:  newobj     instance void [mscorlib]System.Random::.ctor()
66     IL_0006:  stloc.3
67     IL_0007:  ldc.i4.1
68     IL_0008:  stloc.2
69     IL_0009:  ldloc.3
70     IL_000a:  callvirt   instance int32 [mscorlib]System.Random::Next()
71     IL_000f:  stloc.0
72     IL_0010:  ldloc.3
73     IL_0011:  callvirt   instance int32 [mscorlib]System.Random::Next()
74     IL_0016:  stloc.1
75     IL_0017:  ldloc.0
76     IL_0018:  ldloc.1
77     IL_0019:  cgt
78     IL_001b:  ldc.i4.0
79     IL_001c:  ceq
80     IL_001e:  stloc.s    V_4
81     IL_0020:  ldloc.s    V_4
82     IL_0022:  brtrue.s   IL_002a
```

Figure 4.2: Default obfuscation of example one

Figure 4.1 shows the a fragment of the original CIL code without any obfuscating techniques applied, while Figure 4.2 shows the same example after obfuscation with Salamander .NET Obfuscator. Setting 1 from Table 3.1 was used in this simulation.

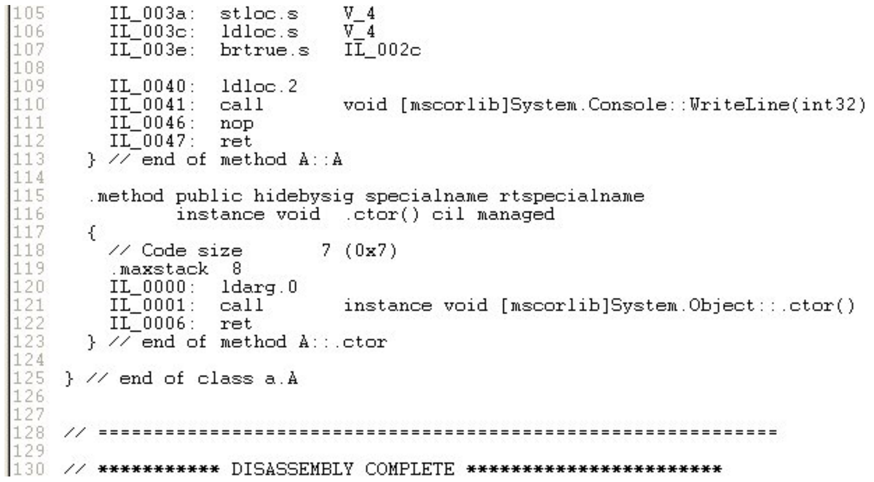
The only differences from Figure 4.2 compared to Figure 4.1, is the change of variable names, the class and the method. This can be seen at line 51, 54 and 59-63. On line 80 and 81 these new names are used instead of the original ones. After the 32 simulations as described in 3.1, the renaming of the fields proved to be a static technique in Salamander .NET. It was a surprise that there was no control flow obfuscation performed in the results.



```
// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit a.A
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  A(string[] A_0) cil managed
  {
    .entrypoint
    // Code size       72 (0x48)
    .maxstack  4
    .locals init (int32 V_0,
             int32 V_1,
             int32 V_2,
             class [mscorlib]System.Random V_3,
             bool V_4)
    IL_0000:  nop
    IL_0001:  newobj     instance void [mscorlib]System.Random::.ctor()
    IL_0006:  stloc.3
    IL_0007:  ldc.i4.1
    IL_0008:  stloc.2
    IL_0009:  ldloc.3
    IL_000a:  callvirt   instance int32 [mscorlib]System.Random::Next()
    IL_000f:  stloc.0
    IL_0010:  ldloc.3
    IL_0011:  callvirt   instance int32 [mscorlib]System.Random::Next()
    IL_0016:  stloc.1
    IL_0017:  ldloc.0
    IL_0018:  ldloc.1
    IL_0019:  cgt
    IL_001b:  ldc.i4.0
    IL_001c:  ceq
    IL_001e:  stloc.s    V_4
    IL_0020:  ldloc.s    V_4
    IL_0022:  brtrue.s   IL_002a
```

Figure 4.3: CIL code after obfuscation with setting two from Table 3.1

Figure 4.3 shows the CIL code with default obfuscation and the extra ILDASM option turned on (the second setting from Table 3.1). As the figure shows, it is identical to the previous one. This is a little surprising since the code was simulated with different options.



```
    IL_003a:  stloc.s    V_4
    IL_003c:  ldloc.s    V_4
    IL_003e:  brtrue.s   IL_002c

    IL_0040:  ldloc.2
    IL_0041:  call       void [mscorlib]System.Console::WriteLine(int32)
    IL_0046:  nop
    IL_0047:  ret
  } // end of method A::A

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method A::.ctor

} // end of class a.A


// =============================================================

// *********** DISASSEMBLY COMPLETE ***********************
```

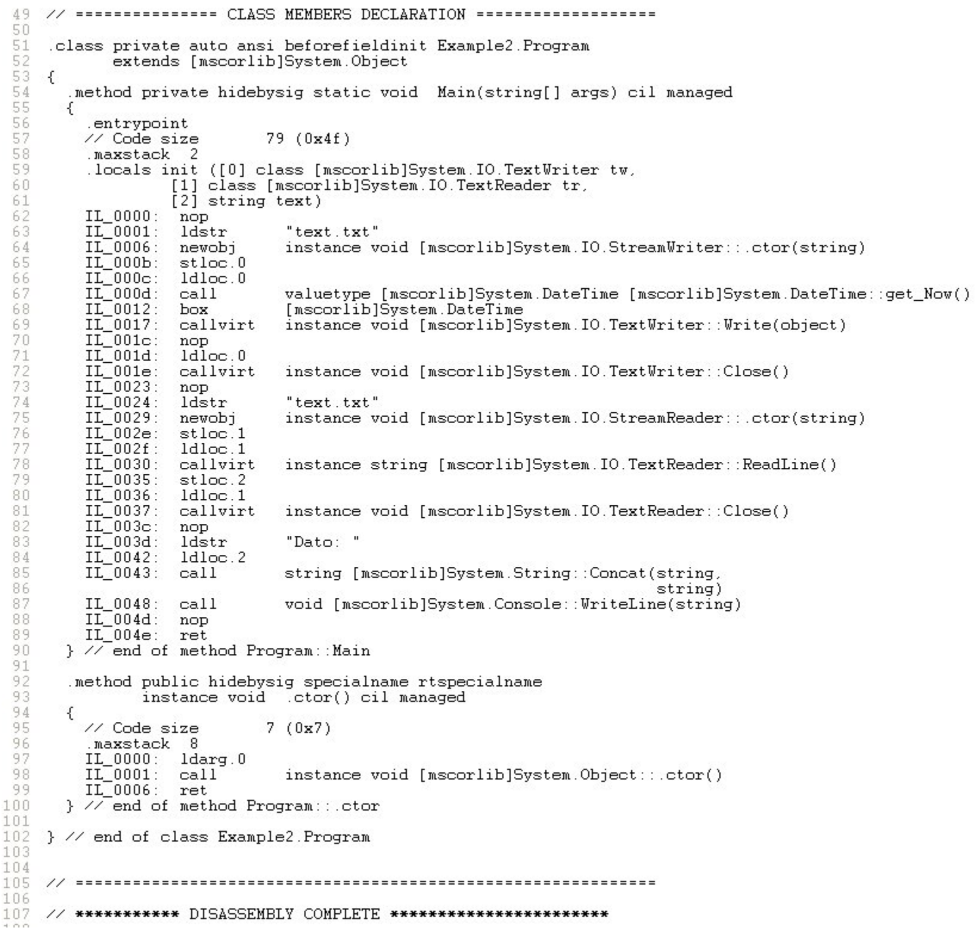
Figure 4.4: The end fragment of example one obfuscated with ILDASM protection

Figure 4.4 shows the end of the CIL code obfuscated with the extra anti ILDASM option as described in setting two in Table 3.1. All the obfuscated code in this simulation was identical to the first simulation, which was surprising since this meant that there was no extra code included for protection against ILDASM. This could have been added at the end of the obfuscated CIL code but as shown in Figure 4.4 there are no extra instructions at the end of the file either.

Since this simulation gave the same obfuscated CIL code as the first simulation, no control flow obfuscation technique was used in this simulation either.

4.1.2 CIL code of example 2

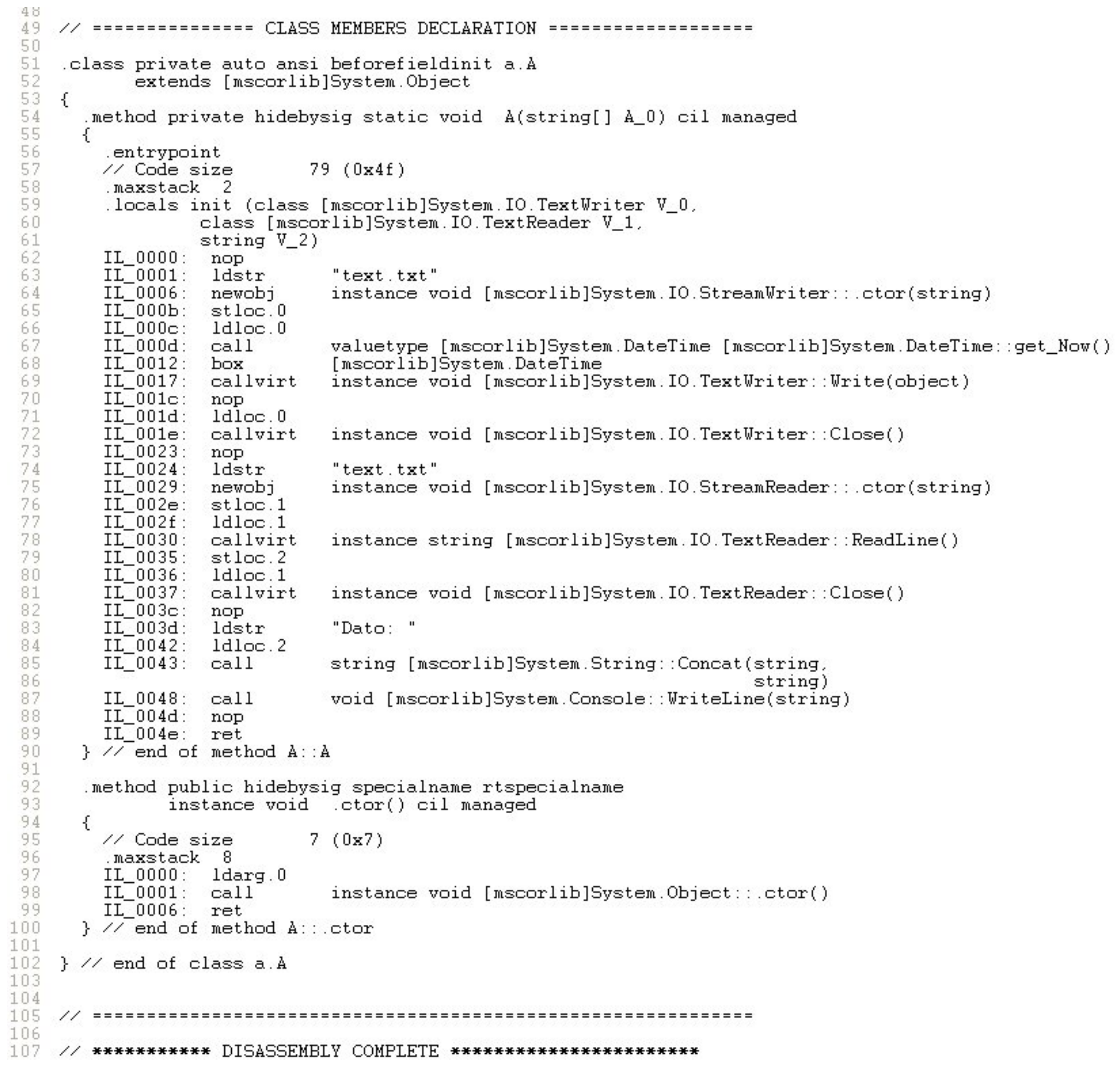
As described in section 3.2.2 the main interest for the second code example is to see whether the control flow obfuscation can be applied to a program that does not contain any loops or other conditions. Control flow obfuscation is usually is applied to loops and conditions.



```
 49 // ============== CLASS MEMBERS DECLARATION ===================
 50
 51 .class private auto ansi beforefieldinit Example2.Program
 52        extends [mscorlib]System.Object
 53 {
 54   .method private hidebysig static void  Main(string[] args) cil managed
 55   {
 56     .entrypoint
 57     // Code size       79 (0x4f)
 58     .maxstack  2
 59     .locals init ([0] class [mscorlib]System.IO.TextWriter tw,
 60              [1] class [mscorlib]System.IO.TextReader tr,
 61              [2] string text)
 62     IL_0000:  nop
 63     IL_0001:  ldstr      "text.txt"
 64     IL_0006:  newobj     instance void [mscorlib]System.IO.StreamWriter::.ctor(string)
 65     IL_000b:  stloc.0
 66     IL_000c:  ldloc.0
 67     IL_000d:  call       valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()
 68     IL_0012:  box        [mscorlib]System.DateTime
 69     IL_0017:  callvirt   instance void [mscorlib]System.IO.TextWriter::Write(object)
 70     IL_001c:  nop
 71     IL_001d:  ldloc.0
 72     IL_001e:  callvirt   instance void [mscorlib]System.IO.TextWriter::Close()
 73     IL_0023:  nop
 74     IL_0024:  ldstr      "text.txt"
 75     IL_0029:  newobj     instance void [mscorlib]System.IO.StreamReader::.ctor(string)
 76     IL_002e:  stloc.1
 77     IL_002f:  ldloc.1
 78     IL_0030:  callvirt   instance string [mscorlib]System.IO.TextReader::ReadLine()
 79     IL_0035:  stloc.2
 80     IL_0036:  ldloc.1
 81     IL_0037:  callvirt   instance void [mscorlib]System.IO.TextReader::Close()
 82     IL_003c:  nop
 83     IL_003d:  ldstr      "Dato: "
 84     IL_0042:  ldloc.2
 85     IL_0043:  call       string [mscorlib]System.String::Concat(string,
 86                                                              string)
 87     IL_0048:  call       void [mscorlib]System.Console::WriteLine(string)
 88     IL_004d:  nop
 89     IL_004e:  ret
 90   } // end of method Program::Main
 91
 92   .method public hidebysig specialname rtspecialname
 93           instance void  .ctor() cil managed
 94   {
 95     // Code size       7 (0x7)
 96     .maxstack  8
 97     IL_0000:  ldarg.0
 98     IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
 99     IL_0006:  ret
100   } // end of method Program::.ctor
101
102 } // end of class Example2.Program
103
104
105 // =============================================================
106
107 // *********** DISASSEMBLY COMPLETE ***********************
```

Figure 4.5: The original CIL code from example two

The CIL code from the simulation in Salamander .NET Obfuscator, with setting one from Table 3.1, is shown in Figure 4.6. The same changes were done to the CIL code as the ones made by the Salamander .NET Obfuscator in example one. The variable renaming is on the lines 59-61, class renaming is on lines 51 and 102 and method renaming is on lines 54, 90 and 100. There was no signs of control flow obfuscation here either.



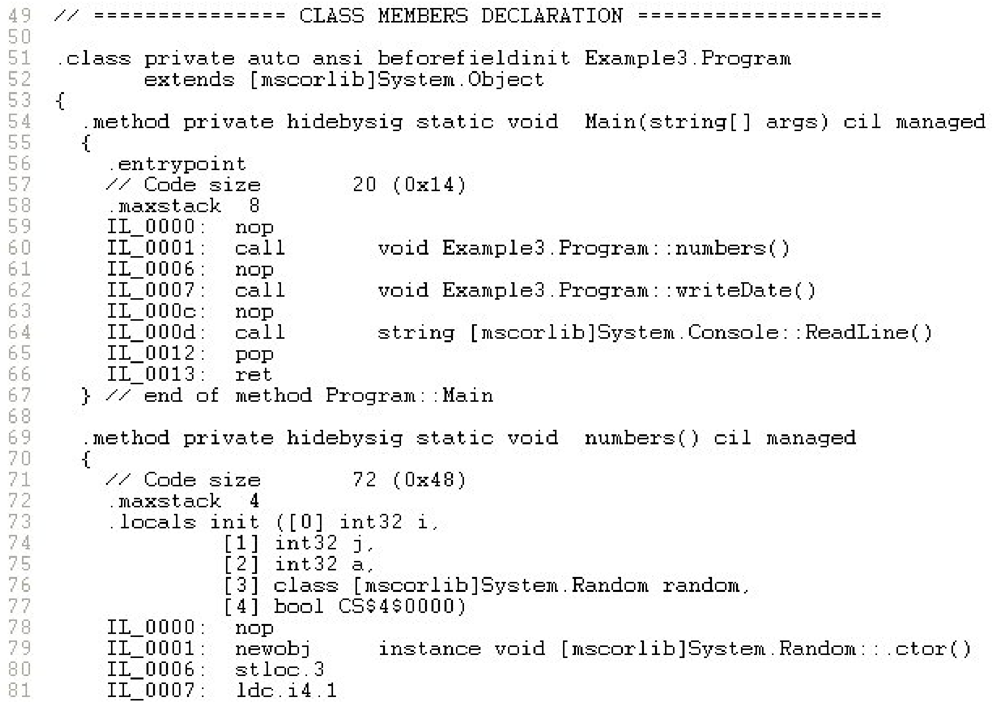
```
// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit a.A
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  A(string[] A_0) cil managed
  {
    .entrypoint
    // Code size       79 (0x4f)
    .maxstack  2
    .locals init (class [mscorlib]System.IO.TextWriter V_0,
             class [mscorlib]System.IO.TextReader V_1,
             string V_2)
    IL_0000:  nop
    IL_0001:  ldstr      "text.txt"
    IL_0006:  newobj     instance void [mscorlib]System.IO.StreamWriter::.ctor(string)
    IL_000b:  stloc.0
    IL_000c:  ldloc.0
    IL_000d:  call       valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()
    IL_0012:  box        [mscorlib]System.DateTime
    IL_0017:  callvirt   instance void [mscorlib]System.IO.TextWriter::Write(object)
    IL_001c:  nop
    IL_001d:  ldloc.0
    IL_001e:  callvirt   instance void [mscorlib]System.IO.TextWriter::Close()
    IL_0023:  nop
    IL_0024:  ldstr      "text.txt"
    IL_0029:  newobj     instance void [mscorlib]System.IO.StreamReader::.ctor(string)
    IL_002e:  stloc.1
    IL_002f:  ldloc.1
    IL_0030:  callvirt   instance string [mscorlib]System.IO.TextReader::ReadLine()
    IL_0035:  stloc.2
    IL_0036:  ldloc.1
    IL_0037:  callvirt   instance void [mscorlib]System.IO.TextReader::Close()
    IL_003c:  nop
    IL_003d:  ldstr      "Dato: "
    IL_0042:  ldloc.2
    IL_0043:  call       string [mscorlib]System.String::Concat(string,
                                                                string)
    IL_0048:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_004d:  nop
    IL_004e:  ret
  } // end of method A::A

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method A::.ctor

} // end of class a.A


// =============================================================

// *********** DISASSEMBLY COMPLETE ***********************
```

Figure 4.6: Obfuscated CIL code after simulation with setting one from Table 3.1

The CIL code from example two was also tested with the ILDASM option turned on as described in setting two in Table 3.1. Also in this scenario, the CIL code was completely identical to example one, and not protection against ILDASM was offered. Since it were no loops or conditions in this example, it was no surprise that there was no signs of control flow obfuscation.

4.1.3 CIL code of example 3

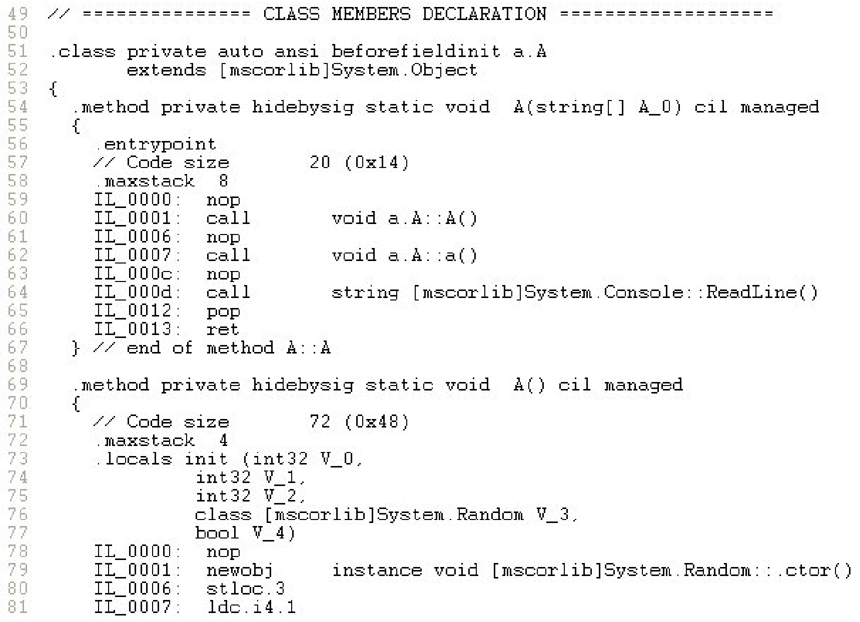
As described in section 3.2.3 the motivation behind these simulations is to see if the changes made in the obfuscated CIL code are the same as for the first two code examples. This will give some indication about whether the obfuscation techniques used by Salamander .NET Obfuscator is dynamic or static. It is also interesting to see if the larger size of the code will trigger some other techniques. Hopefully the control flow obfuscation techniques and also the ILDASM protection should be included in this simulation.



```
// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit Example3.Program
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  Main(string[] args) cil managed
  {
    .entrypoint
    // Code size       20 (0x14)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  call       void Example3.Program::numbers()
    IL_0006:  nop
    IL_0007:  call       void Example3.Program::writeDate()
    IL_000c:  nop
    IL_000d:  call       string [mscorlib]System.Console::ReadLine()
    IL_0012:  pop
    IL_0013:  ret
  } // end of method Program::Main

  .method private hidebysig static void  numbers() cil managed
  {
    // Code size       72 (0x48)
    .maxstack  4
    .locals init ([0] int32 i,
             [1] int32 j,
             [2] int32 a,
             [3] class [mscorlib]System.Random random,
             [4] bool CS$4$0000)
    IL_0000:  nop
    IL_0001:  newobj     instance void [mscorlib]System.Random::.ctor()
    IL_0006:  stloc.3
    IL_0007:  ldc.i4.1
```

Figure 4.7: Original CIL code of example three

Figure 4.7 shows the original CIL code, and displays the new main function, the **`numbers()`** function is declared below.

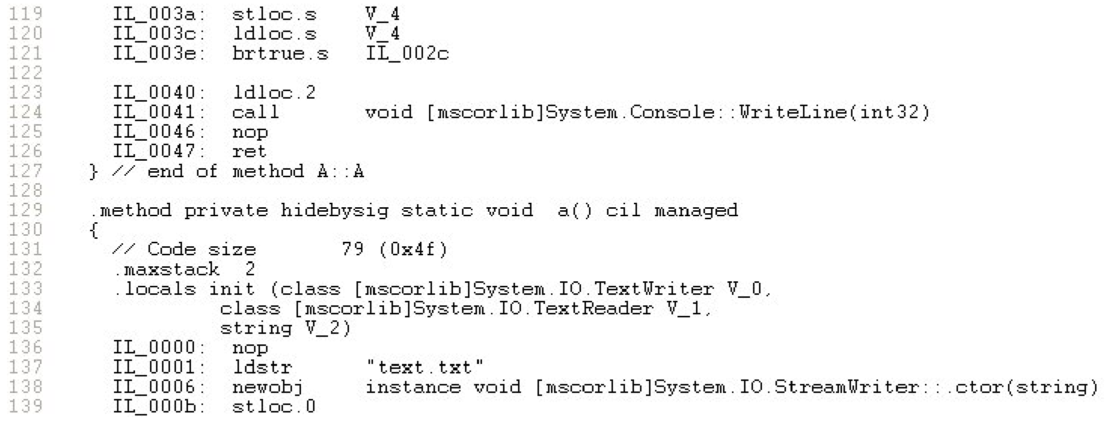


```
// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit a.A
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  A(string[] A_0) cil managed
  {
    .entrypoint
    // Code size       20 (0x14)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  call       void a.A::A()
    IL_0006:  nop
    IL_0007:  call       void a.A::a()
    IL_000c:  nop
    IL_000d:  call       string [mscorlib]System.Console::ReadLine()
    IL_0012:  pop
    IL_0013:  ret
  } // end of method A::A

  .method private hidebysig static void  A() cil managed
  {
    // Code size       72 (0x48)
    .maxstack  4
    .locals init (int32 V_0,
             int32 V_1,
             int32 V_2,
             class [mscorlib]System.Random V_3,
             bool V_4)
    IL_0000:  nop
    IL_0001:  newobj     instance void [mscorlib]System.Random::.ctor()
    IL_0006:  stloc.3
    IL_0007:  ldc.i4.1
```

Figure 4.8: CIL code of example three with default setting

The CIL code in the Figure 4.8 shows the obfuscation result from the simulations in Salamander .NET Obfuscator with the first setting from Table 3.1. The renamings in the figure are obviously, like the previous examples, on lines 51, 54, 60, 62, 67, 69 and 73-77. It is also easy to see that the **`numbers()`** function has been renamed to **`A()`** and the class is called **`a.A`**.

The renaming of variables on lines 73 to 77 is also identical with the renaming performed on the lines 59 to 63 in Figure 4.2. There is no control flow obfuscation or ILDASM protection triggered in this simulation either.



```
119      IL_003a:  stloc.s    V_4
120      IL_003c:  ldloc.s    V_4
121      IL_003e:  brtrue.s   IL_002c
122
123      IL_0040:  ldloc.2
124      IL_0041:  call       void [mscorlib]System.Console::WriteLine(int32)
125      IL_0046:  nop
126      IL_0047:  ret
127    } // end of method A::A
128
129    .method private hidebysig static void  a() cil managed
130    {
131      // Code size       79 (0x4f)
132      .maxstack  2
133      .locals init (class [mscorlib]System.IO.TextWriter V_0,
134               class [mscorlib]System.IO.TextReader V_1,
135               string V_2)
136      IL_0000:  nop
137      IL_0001:  ldstr      "text.txt"
138      IL_0006:  newobj     instance void [mscorlib]System.IO.StreamWriter::.ctor(string)
139      IL_000b:  stloc.0
```

Figure 4.9: Obfuscated CIL code of example three with ILDASM protection

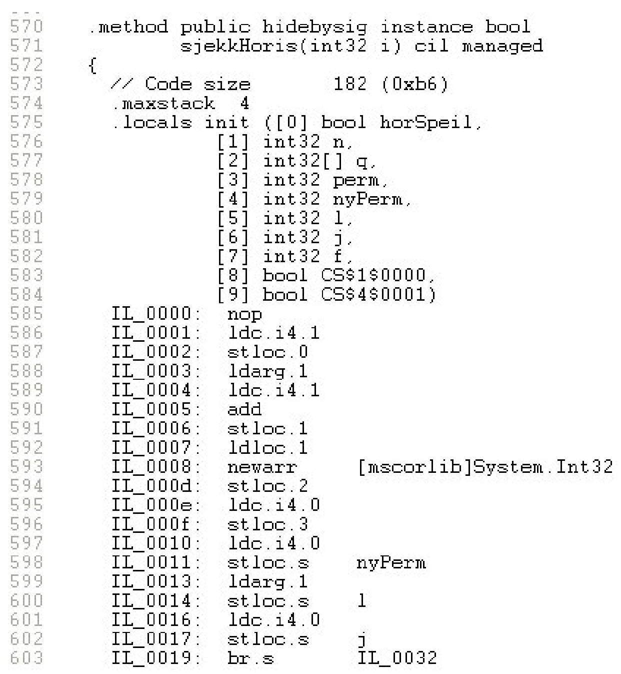
In Figure 4.9, renaming is present on lines 119, 120, 127, 129 and 133-135.
The CIL code was simulated with the second setting from Table 3.1. Again, the code generated with the default obfuscation and with the default ILDASM option is identical.

The variable renaming on the lines 133-135 is identical with the renaming performed in the lines 59-61 in the Figure 4.6.

The presence of control flow obfuscation and ILDASM protection is still missing in the obfuscated CIL code. The size of this code example should have been enough to trigger these techniques and also others if there were some.

4.1.4 CIL code of example 4

The motivation behind the fourth example code is to investigate the alterations on the control flow technique due to the high number of loops and conditions in the CIL code. The size should also initiate all the obfuscation techniques that are implemented in the Salamander .NET Obfuscator, because of a large and complex code.

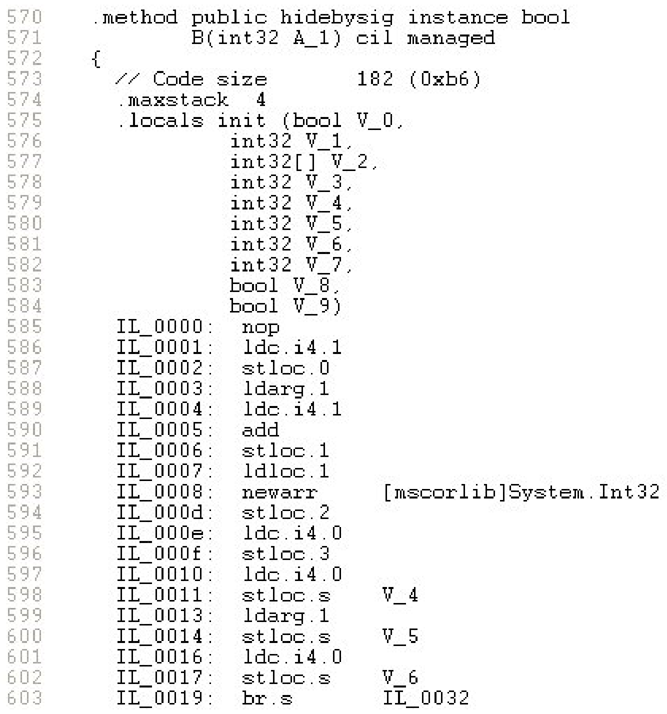


```
570   .method public hidebysig instance bool
571           sjekkHoris(int32 i) cil managed
572   {
573     // Code size       182 (0xb6)
574     .maxstack  4
575     .locals init ([0] bool horSpeil,
576              [1] int32 n,
577              [2] int32[] q,
578              [3] int32 perm,
579              [4] int32 nyPerm,
580              [5] int32 l,
581              [6] int32 j,
582              [7] int32 f,
583              [8] bool CS$1$0000,
584              [9] bool CS$4$0001)
585     IL_0000:  nop
586     IL_0001:  ldc.i4.1
587     IL_0002:  stloc.0
588     IL_0003:  ldarg.1
589     IL_0004:  ldc.i4.1
590     IL_0005:  add
591     IL_0006:  stloc.1
592     IL_0007:  ldloc.1
593     IL_0008:  newarr     [mscorlib]System.Int32
594     IL_000d:  stloc.2
595     IL_000e:  ldc.i4.0
596     IL_000f:  stloc.3
597     IL_0010:  ldc.i4.0
598     IL_0011:  stloc.s    nyPerm
599     IL_0013:  ldarg.1
600     IL_0014:  stloc.s    l
601     IL_0016:  ldc.i4.0
602     IL_0017:  stloc.s    j
603     IL_0019:  br.s       IL_0032
```

Figure 4.10: The original CIL code of example four

The CIL code simulated with the first setting from Table 3.1 is shown in Figure 4.11. There is renaming of variables and method on lines 571, 575-584, 598, 600 and 602. Like the earlier examples, all the changes made by the obfuscator is in form of renaming. No control flow obfuscating techniques were used.

The obfuscated CIL code after the simulations in Salamander .NET Obfuscator did not trigger any extra obfuscation techniques. The whole obfuscated CIL code with the ILDASM option is in fact identical to the default obfuscation. The conclusion is therefore that both control flow obfuscation and ILDASM protection is not implemented in the Salamander .NET Obfuscator.



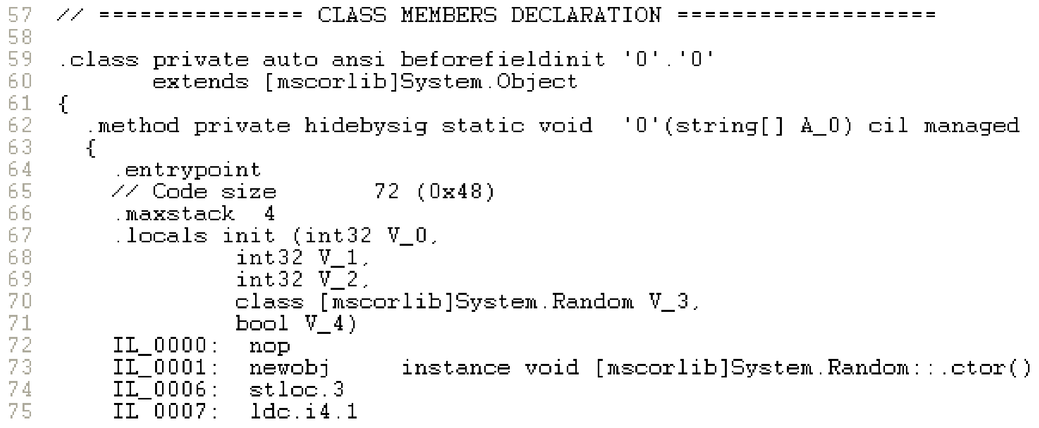
```
.method public hidebysig instance bool
        B(int32 A_1) cil managed
{
  // Code size       182 (0xb6)
  .maxstack  4
  .locals init (bool V_0,
           int32 V_1,
           int32[] V_2,
           int32 V_3,
           int32 V_4,
           int32 V_5,
           int32 V_6,
           int32 V_7,
           bool V_8,
           bool V_9)
  IL_0000:  nop
  IL_0001:  ldc.i4.1
  IL_0002:  stloc.0
  IL_0003:  ldarg.1
  IL_0004:  ldc.i4.1
  IL_0005:  add
  IL_0006:  stloc.1
  IL_0007:  ldloc.1
  IL_0008:  newarr     [mscorlib]System.Int32
  IL_000d:  stloc.2
  IL_000e:  ldc.i4.0
  IL_000f:  stloc.3
  IL_0010:  ldc.i4.0
  IL_0011:  stloc.s    V_4
  IL_0013:  ldarg.1
  IL_0014:  stloc.s    V_5
  IL_0016:  ldc.i4.0
  IL_0017:  stloc.s    V_6
  IL_0019:  br.s       IL_0032
```

Figure 4.11: Obfuscated CIL code with the default setting

4.2 Spices .NET Obfuscator

The following sections will present simulation results from the Spices .NET Obfuscator. Various obfuscation techniques applied in the obfuscated CIL code will be briefly described.

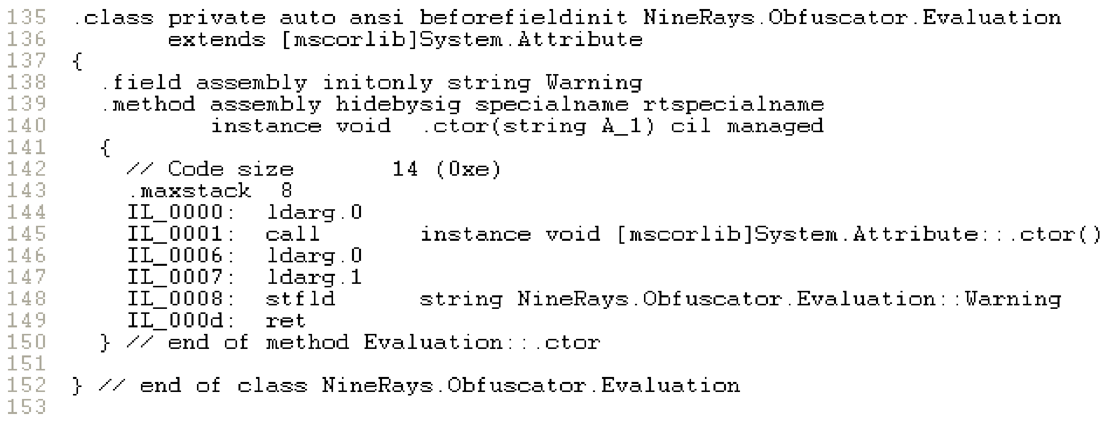
4.2.1 CIL code of example 1



```
// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit '0'.'0'
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  '0'(string[] A_0) cil managed
  {
    .entrypoint
    // Code size       72 (0x48)
    .maxstack  4
    .locals init (int32 V_0,
             int32 V_1,
             int32 V_2,
             class [mscorlib]System.Random V_3,
             bool V_4)
    IL_0000:  nop
    IL_0001:  newobj     instance void [mscorlib]System.Random::.ctor()
    IL_0006:  stloc.3
    IL_0007:  ldc.i4.1
```

Figure 4.12: Obfuscated CIL code from example one with default setting

Figure 4.12 and 4.13 shows two different parts of the CIL code after obfuscation with the first setting from Table 3.2 has been applied. The renaming of variables is done in the same manner as the Salamander .NET Obfuscator, and can be seen on the lines 67-71 in Figure 4.12. The same applies to the parameter renaming in the methods. The class and method are, however, renamed with different symbols. Where the Salamander .NET Obfuscator used letters starting with A, a, B, b and so on (see lines 51 and 54 in Figure 4.2), the Spices .NET Obfuscator uses numbers, starting at 0 and increasing (see lines 59 and 62 in Figure 4.12). The renaming scheme is although still static for these fields.

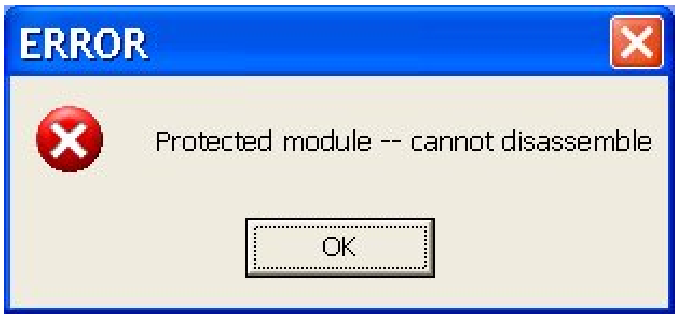


```
.class private auto ansi beforefieldinit NineRays.Obfuscator.Evaluation
       extends [mscorlib]System.Attribute
{
  .field assembly initonly string Warning
  .method assembly hidebysig specialname rtspecialname
          instance void  .ctor(string A_1) cil managed
  {
    // Code size       14 (0xe)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Attribute::.ctor()
    IL_0006:  ldarg.0
    IL_0007:  ldarg.1
    IL_0008:  stfld      string NineRays.Obfuscator.Evaluation::Warning
    IL_000d:  ret
  } // end of method Evaluation::.ctor

} // end of class NineRays.Obfuscator.Evaluation
```

Figure 4.13: An extra class added in the end of the obfuscated CIL code

The Spices .NET Obfuscator also inserts its own class at the end of the CIL code to inform that the obfuscation here is performed by this particular obfuscator, as seen in the Figure 4.13. This class does not have any further functionality and is never called.

When CIL code was run with the with the ILDASM protection set to *true* (see setting two from Table 3.2) through the Spices .NET Obfuscator, the result makes the ILDSM crash, as shown in Figure 4.14, just as expected.

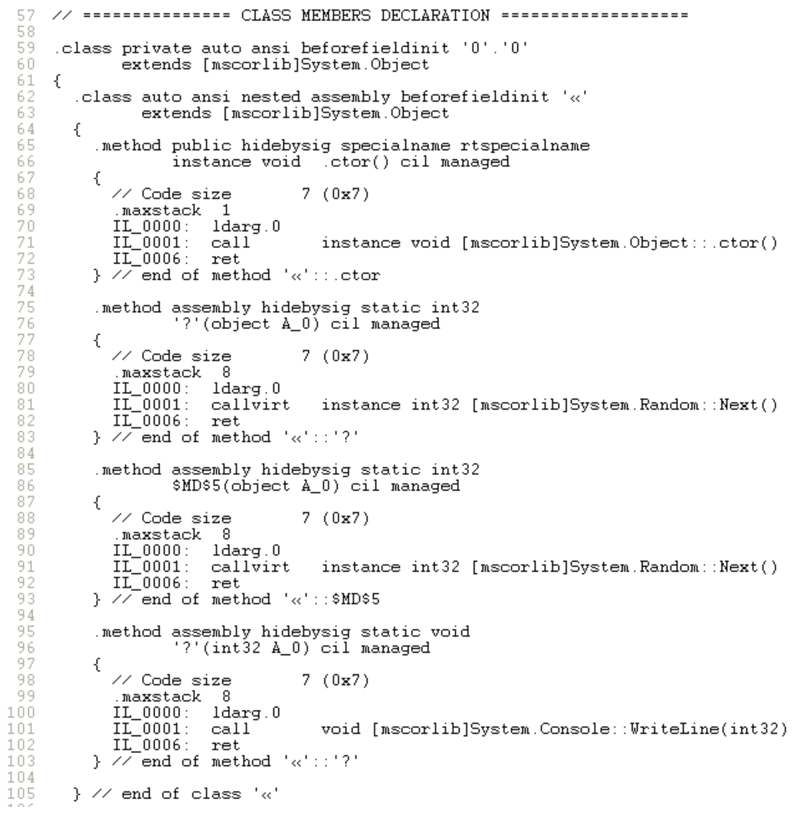


Protected module -- cannot disassemble

Figure 4.14: Error message when running the MSIL Disassembler

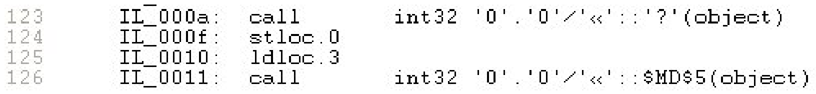
When the CIL code was simulated with the third setting from Table 3.2, with ILDASM protection set to *complete*, the code was identical to the code with the default configuration. A figure of this simulation is therefore not presented here, but the CIL code from the simulation is available on the CD. It is although a little surprising that the

ILDASM protection works if the option is set to *true* but not *complete*. The *complete* option might be understood in such a way that it should give even more protection than the *true* option.



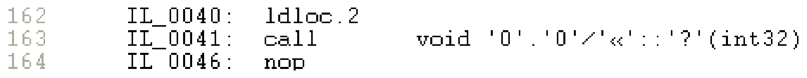
```
// ============== CLASS MEMBERS DECLARATION ==================

.class private auto ansi beforefieldinit '0'.'0'
       extends [mscorlib]System.Object
{
  .class auto ansi nested assembly beforefieldinit '«'
         extends [mscorlib]System.Object
  {
    .method public hidebysig specialname rtspecialname
            instance void  .ctor() cil managed
    {
      // Code size       7 (0x7)
      .maxstack  1
      IL_0000:  ldarg.0
      IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
      IL_0006:  ret
    } // end of method '«'::.ctor

    .method assembly hidebysig static int32
            '?'(object A_0) cil managed
    {
      // Code size       7 (0x7)
      .maxstack  8
      IL_0000:  ldarg.0
      IL_0001:  callvirt   instance int32 [mscorlib]System.Random::Next()
      IL_0006:  ret
    } // end of method '«'::'?'

    .method assembly hidebysig static int32
            $MD$5(object A_0) cil managed
    {
      // Code size       7 (0x7)
      .maxstack  8
      IL_0000:  ldarg.0
      IL_0001:  callvirt   instance int32 [mscorlib]System.Random::Next()
      IL_0006:  ret
    } // end of method '«'::$MD$5

    .method assembly hidebysig static void
            '?'(int32 A_0) cil managed
    {
      // Code size       7 (0x7)
      .maxstack  8
      IL_0000:  ldarg.0
      IL_0001:  call       void [mscorlib]System.Console::WriteLine(int32)
      IL_0006:  ret
    } // end of method '«'::'?'

  } // end of class '«'
```

Figure 4.15: System calls obfuscated into separate methods



```
123      IL_000a:  call       int32 '0'.'0'/'«'::'?'(object)
124      IL_000f:  stloc.0
125      IL_0010:  ldloc.3
126      IL_0011:  call       int32 '0'.'0'/'«'::$MD$5(object)
```

Figure 4.16: Obfuscation of the calling of the newly crated methods



```
162      IL_0040:  ldloc.2
163      IL_0041:  call       void '0'.'0'/'«'::'?'(int32)
164      IL_0046:  nop
```

Figure 4.17: Obfuscation of the calling of the newly crated methods

Figure 4.15 – 4.17 shows the CIL code after obfuscation with members option on full and anonymizer on high (the fifth setting from Table 3.2). All of the CIL code in Figure 4.15 is generated by the Spices .NET Obfuscator and the generated methods are used to obfuscate system calls.

The non-obfuscated version of the system calls made on line 123 and 126 in Figure 4.16 can be viewed on lines 70 and 73 in Figure 4.1, and the non-obfuscation of line 163 is found on line 110 in Figure 4.4. The 32 different simulations of this setting proved that the name given to the new created methods was dynamic.

The code in example 1 was also simulated in Spices .NET Obfuscator with the fourth setting from Table 3.2 (with members and anonymizer options set to default, but without the two string encryption options). When the obfuscated results was compared to the CIL code from the simulation, the only difference was that the results had named the class with another symbol, which is no surprise since the renaming is dynamic.

From the Figure 4.15 – 4.17 it is clear that the obfuscating of different system calls will make the control flow of the program different from the original one. The rewriting can be viewed as an outlining technique described in section 2.2.2.6. There is although no other trace of other control flow obfuscation techniques.

4.2.2 CIL code of example 2

The CIL code in example two was run with the Spices .NET Obfuscator using the same five different settings from Table 3.2.
The simulation with the first setting showed the same changes as in the previous section. The renaming of variables, methods, classes and parameters is identical to the result in the first CIL code example, see Figure 4.12. The same CIL code as in Figure 4.13 was added to the code in this simulation.

The second and third setting from Table 3.2 were executed with the Spices .NET Obfuscator. The second setting with ILDASM option set to *true* still made the CIL code impossible to retrieve for the MSIL Disassembler, and gave the same error message as shown in Figure 4.14. When the ILDASM option was set to *complete* this gave the same surprising result as it did in the first CIL code example.

When the example two CIL code was obfuscated with the last two settings from Table 3.2, the fourth and the fifth, the result of these simulations differed in the renaming of symbols.

In the last two simulations, all the system calls were obfuscated and turned into separate methods as shown in Figure 4.15. The only difference in this simulation was that the CIL code in example two contained more system calls than the first code example and hence the number of created methods was larger. In example one there were only four generated methods, but with example two the total number of generated methods was eight (this also includes the constructor) with one method per system call.
No other signs of control flow obfuscation were found in the simulation of example two, besides the outlining technique.

4.2.3 CIL code of example 3

All the simulations of the third code examples were performed with the same five settings done with example one and two. Since the third code example is an combination of the first two code examples, the simulation results should resemble the results of the first two code examples, although there might be some differences in the symbols used by the Spices .NET Obfuscator.

The simulations of the first and the third setting returned the same results as they did in the previous sections. The classes, methods and variables were renamed in the same way as they were earlier (see Figure 4.12). The class generated by the Spices .NET Obfuscator as seen in Figure 4.13 was also added to the code in these two simulations. Since the simulation with the *complete* option returned the same result as the default setting did, it was no surprise that the MSIL Disassembler could retrieve the CIL code without problems.

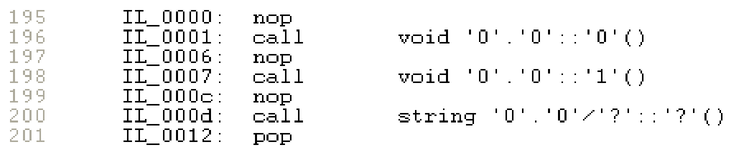
In the simulation with the third setting from Table 3.2, the ILDASM option was set to *true* and the results of this setting were the same as before: MSIL Disassembler could not retrieve the CIL code and gave the same error message as seen in Figure 4.14.

When the last two settings were simulated with the third code example, the result differed from the result in the previous section 4.4.2. In the simulation with the fourth setting, the number of created methods for the system calls was 12. In the result from the simulation with Spices .NET Obfuscator and the fifth setting, the number was 14.

The reason for this is as follows: With the fourth setting, the system calls is the only methods that are obfuscated. From section 4.4.1 and 4.4.2 it is clear that the first code example generates three methods and the second code example generates seven methods when the constructor methods are excluded. In example three there is one additional system call in the new main method (see section 3.4.3 for more information about the original CIL code). By also including the constructor in the generated internal class, this makes the total number of methods 12.

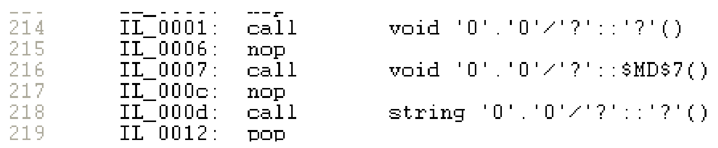
When the last simulation is executed, the fifth setting from Table 3.2, the results are different. The two method calls made in the main method of example three (see lines 60 and 62 in Figure 4.7) were only renamed with the other settings (1-4), as shown in the Figure 4.18, but with the new setting, these two method calls are obfuscated just like the system calls. The amount of created methods is thus 14, one extra for each of these methods. Figure 4.19 shows how the method calls are presented in this last simulation.

Throughout the simulations of example three there were no other obfuscation techniques applied on the obfuscated CIL code besides the ones that also were used on the first two examples.



```
IL_0000:  nop
IL_0001:  call       void '0'.'0'::'0'()
IL_0006:  nop
IL_0007:  call       void '0'.'0'::'1'()
IL_000c:  nop
IL_000d:  call       string '0'.'0'/'?'::'?'()
IL_0012:  pop
```

Figure 4.18: Calls to local methods are just renamed



```
IL_0001:  call       void '0'.'0'/'?'::'?'()
IL_0006:  nop
IL_0007:  call       void '0'.'0'/'?'::$MD$7()
IL_000c:  nop
IL_000d:  call       string '0'.'0'/'?'::'?'()
IL_0012:  pop
```

Figure 4.19: Calls to local methods are outlined to their own methods

4.2.4 CIL code of example 4

The CIL code in example four was obfuscated with the same five settings from Table 3.2 as the previous section 4.4.1 – 4.4.3. Since this example is the largest code example, all the different obfuscation techniques implemented in the Spices .NET Obfuscator should be extensively used throughout the results.

In the first simulation of example four, the first setting was used. Since the code example contains four different classes with their respectively methods, parameters and variables, the earlier results in section 4.4.1 – 4.4.3 suggests that these fields would be renamed like in the previous simulations. The first class in example four is renamed like the other three code examples, but the results surprisingly showed that the following three sub-classes were not renamed at all in any fields. Neither the name of the classes, methods nor parameters are altered, although the local variables are renamed.

When the simulation with the second setting was obfuscated with the Spices .NET Obfuscator, the results showed no difference from the same setting with the previous three code examples. The error shown in Figure 4.14 was still the only output from the MSIL Disassembler.

For the next simulation, using setting three from Table 3.2, where the ILDASM option is set to *complete*, the result was not different from the earlier simulations. The result was identical to the results from the simulation of example four with the default options (setting one from Table 4.1).

The results from the simulation with the fourth setting showed the same obfuscating of the system calls as discussed in the section 4.4.3. The calls to none system methods were not obfuscated by generating new methods. All the four classes were given their own internal class when the system calls were obfuscated. The variables were also renamed in all four classes.

The result of the simulation showed some surprising results. Just as the simulation with the first setting did not rename the three last classes or their methods and parameters, the same result appeared when the fourth setting was used in this simulation.

In the final simulation with the fifth setting from Table 3.2, the result was as expected. All the four classes were renamed, just as their respective methods, variables and parameters had been too. Every class was given their own internal class which contained the new generated methods made by the obfuscator.

This code example is the largest example used in the simulations, but the Spices .NET Obfuscator does not apply any other form of control flow obfuscation other than the outlining of calls.

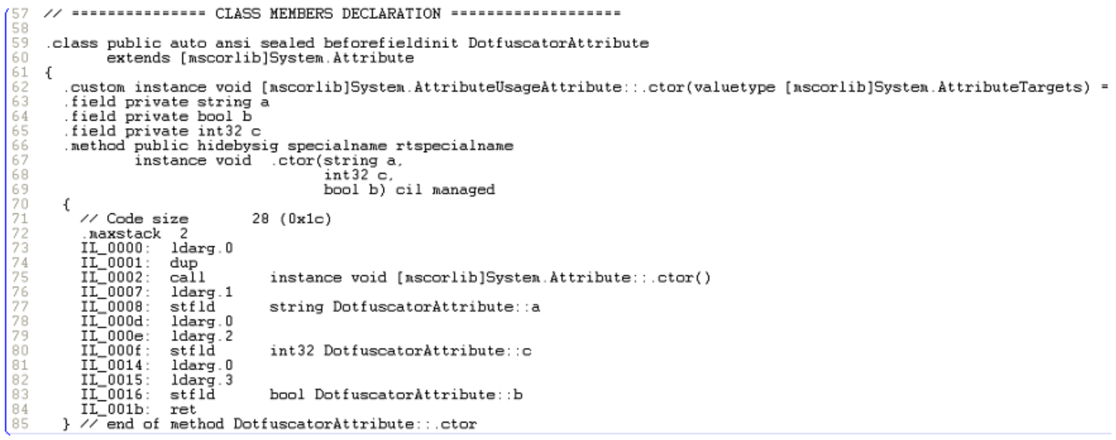
Because the obfuscation of method calls generates more code, the CIL code from the last simulation contained 2341 lines of CIL code compared to 1625 lines of CIL code in the original non-obfuscated version of the example.

4.3 DotFuscator

4.3.1 CIL code of example 1

Since the control flow and renaming techniques were the only two techniques that were interesting for the simulations with DotFuscator (see section 3.3.3), the desired obfuscation techniques could be isolated by testing these two techniques separately.

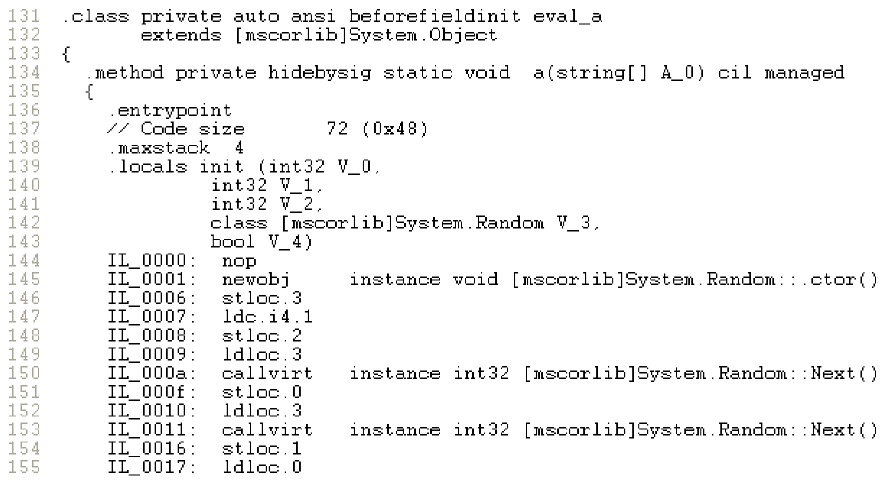
The first setting from Table 3.3 (only default options) showed the same obfuscated CIL code as the second (only renaming) and fifth (control flow set to high) setting did when the results were combined. Thus no figure for the first setting is presented in this chapter but can be found on the CD if desired.



```
// =============== CLASS MEMBERS DECLARATION ===================

.class public auto ansi sealed beforefieldinit DotfuscatorAttribute
       extends [mscorlib]System.Attribute
{
  .custom instance void [mscorlib]System.AttributeUsageAttribute::.ctor(valuetype [mscorlib]System.AttributeTargets) =
  .field private string a
  .field private bool b
  .field private int32 c
  .method public hidebysig specialname rtspecialname
          instance void  .ctor(string a,
                               int32 c,
                               bool b) cil managed
  {
    // Code size       28 (0x1c)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  dup
    IL_0002:  call       instance void [mscorlib]System.Attribute::.ctor()
    IL_0007:  ldarg.1
    IL_0008:  stfld      string DotfuscatorAttribute::a
    IL_000d:  ldarg.0
    IL_000e:  ldarg.2
    IL_000f:  stfld      int32 DotfuscatorAttribute::c
    IL_0014:  ldarg.0
    IL_0015:  ldarg.3
    IL_0016:  stfld      bool DotfuscatorAttribute::b
    IL_001b:  ret
  } // end of method DotfuscatorAttribute::.ctor
```

Figure 4.20: The class created by DotFuscator

One commonality was found in all the results from the simulations of code example one: The DotFuscator always creates a new class called DotfuscatorAttribute in the beginning of the obfuscated CIL code, as seen Figure 4.20.



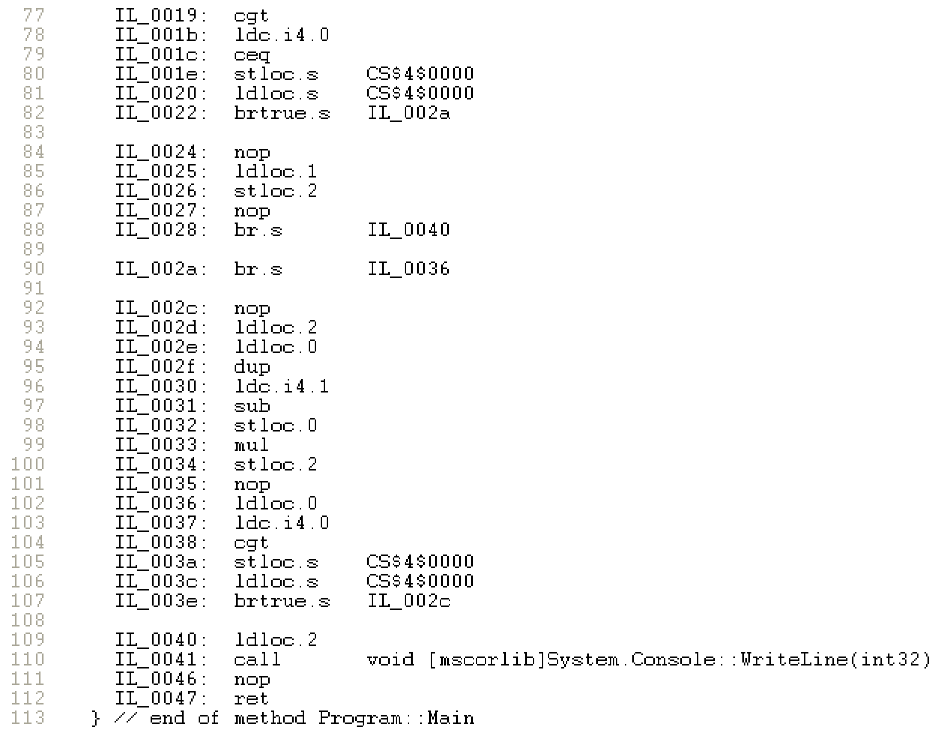
```
.class private auto ansi beforefieldinit eval_a
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  a(string[] A_0) cil managed
  {
    .entrypoint
    // Code size       72 (0x48)
    .maxstack  4
    .locals init (int32 V_0,
             int32 V_1,
             int32 V_2,
             class [mscorlib]System.Random V_3,
             bool V_4)
    IL_0000:  nop
    IL_0001:  newobj     instance void [mscorlib]System.Random::.ctor()
    IL_0006:  stloc.3
    IL_0007:  ldc.i4.1
    IL_0008:  stloc.2
    IL_0009:  ldloc.3
    IL_000a:  callvirt   instance int32 [mscorlib]System.Random::Next()
    IL_000f:  stloc.0
    IL_0010:  ldloc.3
    IL_0011:  callvirt   instance int32 [mscorlib]System.Random::Next()
    IL_0016:  stloc.1
    IL_0017:  ldloc.0
```

Figure 4.21: CIL code of example one with normal renaming

Figure 4.21 shows results when only renaming is turned on, setting two from Table 3.3. The renaming of fields was static, just as the other two obfuscators.

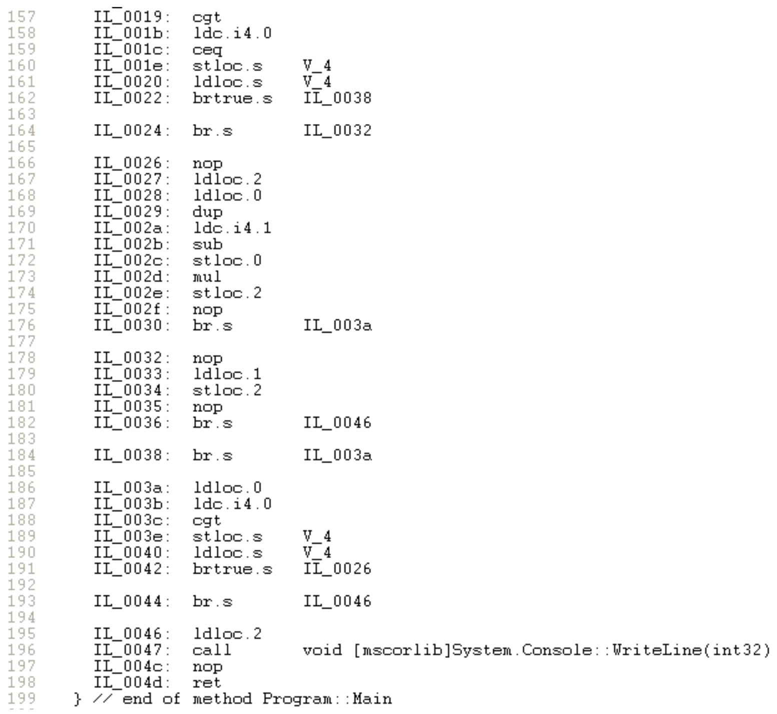
The CIL code from example one was thereafter simulated with the third setting from Table 3.3. The results were identical to the results from the previous settings. This was a bit surprising since the difference in the setting was that the enhanced overload of renaming should have been applied to the CIL code and therefore altered some of the names compared to the results from setting two.

The fourth setting from Table 3.3 was then simulated in the DotFuscator. Figure 4.22 shows the original CIL code and Figure 4.23 shows the same code segment from the obfuscated result. As the figures show, there are clear differences in the CIL code. The instructions on lines 84-88 in Figure 4.22 are not present in Figure 4.23. The same instructions should have been on line 164. The instruction block starting at line 92 in Figure 4.22 has been split up into two blocks in Figure 4.23. The corresponding instructions are now found on lines 166-176 and 186-191.



```
    IL_0019:  cgt
    IL_001b:  ldc.i4.0
    IL_001c:  ceq
    IL_001e:  stloc.s    CS$4$0000
    IL_0020:  ldloc.s    CS$4$0000
    IL_0022:  brtrue.s   IL_002a

    IL_0024:  nop
    IL_0025:  ldloc.1
    IL_0026:  stloc.2
    IL_0027:  nop
    IL_0028:  br.s       IL_0040

    IL_002a:  br.s       IL_0036

    IL_002c:  nop
    IL_002d:  ldloc.2
    IL_002e:  ldloc.0
    IL_002f:  dup
    IL_0030:  ldc.i4.1
    IL_0031:  sub
    IL_0032:  stloc.0
    IL_0033:  mul
    IL_0034:  stloc.2
    IL_0035:  nop
    IL_0036:  ldloc.0
    IL_0037:  ldc.i4.0
    IL_0038:  cgt
    IL_003a:  stloc.s    CS$4$0000
    IL_003c:  ldloc.s    CS$4$0000
    IL_003e:  brtrue.s   IL_002c

    IL_0040:  ldloc.2
    IL_0041:  call       void [mscorlib]System.Console::WriteLine(int32)
    IL_0046:  nop
    IL_0047:  ret
  } // end of method Program::Main
```

Figure 4.22: Fragment of the original CIL code from example one

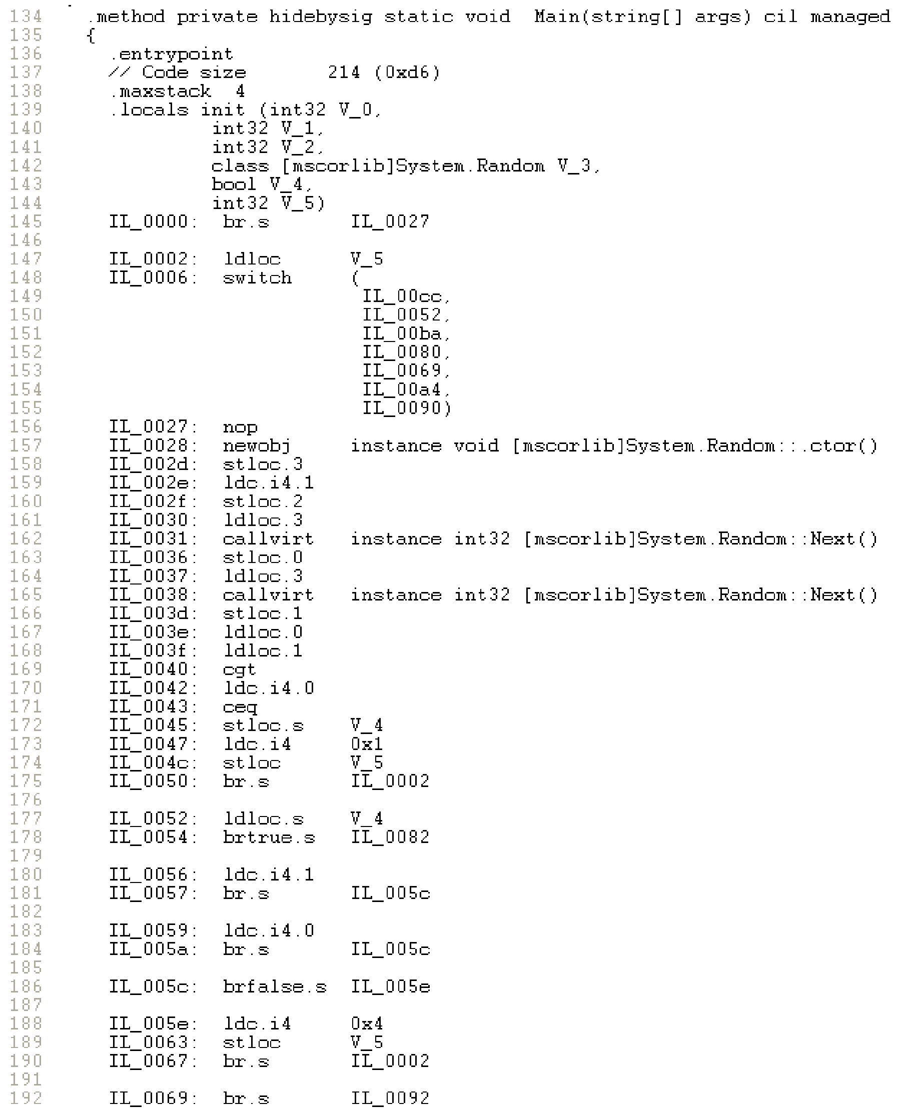


```
    IL_0019:  cgt
    IL_001b:  ldc.i4.0
    IL_001c:  ceq
    IL_001e:  stloc.s    V_4
    IL_0020:  ldloc.s    V_4
    IL_0022:  brtrue.s   IL_0038

    IL_0024:  br.s       IL_0032

    IL_0026:  nop
    IL_0027:  ldloc.2
    IL_0028:  ldloc.0
    IL_0029:  dup
    IL_002a:  ldc.i4.1
    IL_002b:  sub
    IL_002c:  stloc.0
    IL_002d:  mul
    IL_002e:  stloc.2
    IL_002f:  nop
    IL_0030:  br.s       IL_003a

    IL_0032:  nop
    IL_0033:  ldloc.1
    IL_0034:  stloc.2
    IL_0035:  nop
    IL_0036:  br.s       IL_0046

    IL_0038:  br.s       IL_003a

    IL_003a:  ldloc.0
    IL_003b:  ldc.i4.0
    IL_003c:  cgt
    IL_003e:  stloc.s    V_4
    IL_0040:  ldloc.s    V_4
    IL_0042:  brtrue.s   IL_0026

    IL_0044:  br.s       IL_0046

    IL_0046:  ldloc.2
    IL_0047:  call       void [mscorlib]System.Console::WriteLine(int32)
    IL_004c:  nop
    IL_004d:  ret
  } // end of method Program::Main
```

Figure 4.23: Obfuscated CIL code of example one with control flow option *low*

When simulating the fifth setting from the Table 3.3 with the control flow option set to medium, the results of the two simulations (low and medium) were identical. The simulation is therefore not presented, but can be found on the CD. It is although surprising that the change of control flow setting did not change the obfuscated result.

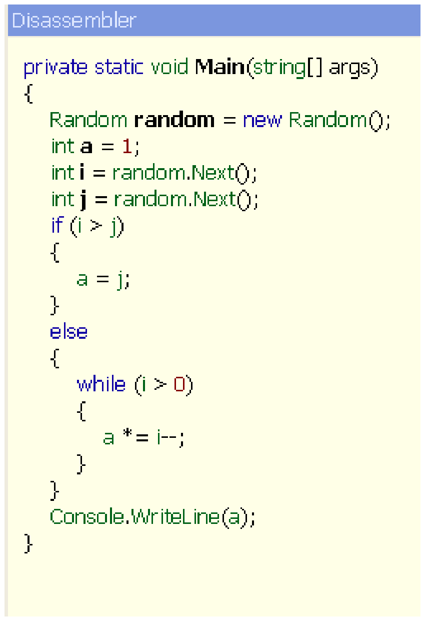
The sixth setting resulted in considerable changes after simulation with DotFuscator. Compared to the original code, the obfuscated CIL code was now changed as shown in Figure 4.24, with major differences compared to the original one. The whole structure of the program was changed compared to the original one.



```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       214 (0xd6)
  .maxstack  4
  .locals init (int32 V_0,
           int32 V_1,
           int32 V_2,
           class [mscorlib]System.Random V_3,
           bool V_4,
           int32 V_5)
  IL_0000:  br.s       IL_0027

  IL_0002:  ldloc      V_5
  IL_0006:  switch     (
                         IL_00cc,
                         IL_0052,
                         IL_00ba,
                         IL_0080,
                         IL_0069,
                         IL_00a4,
                         IL_0090)
  IL_0027:  nop
  IL_0028:  newobj     instance void [mscorlib]System.Random::.ctor()
  IL_002d:  stloc.3
  IL_002e:  ldc.i4.1
  IL_002f:  stloc.2
  IL_0030:  ldloc.3
  IL_0031:  callvirt   instance int32 [mscorlib]System.Random::Next()
  IL_0036:  stloc.0
  IL_0037:  ldloc.3
  IL_0038:  callvirt   instance int32 [mscorlib]System.Random::Next()
  IL_003d:  stloc.1
  IL_003e:  ldloc.0
  IL_003f:  ldloc.1
  IL_0040:  cgt
  IL_0042:  ldc.i4.0
  IL_0043:  ceq
  IL_0045:  stloc.s    V_4
  IL_0047:  ldc.i4     0x1
  IL_004c:  stloc      V_5
  IL_0050:  br.s       IL_0002

  IL_0052:  ldloc.s    V_4
  IL_0054:  brtrue.s   IL_0082

  IL_0056:  ldc.i4.1
  IL_0057:  br.s       IL_005c

  IL_0059:  ldc.i4.0
  IL_005a:  br.s       IL_005c

  IL_005c:  brfalse.s  IL_005e

  IL_005e:  ldc.i4     0x4
  IL_0063:  stloc      V_5
  IL_0067:  br.s       IL_0002

  IL_0069:  br.s       IL_0092
```

Figure 4.24: Obfuscated CIL code of example one with control flow set to *high*

As shown in Figure 4.24, a lot of major changes are present in the obfuscated CIL code. There is a new variable declared, and the flow of the program is completely different. This is because the DotFuscator has rewritten the if-else statements and the last line of the main function in Figure 4.25 into the switch case in Figure 4.26.

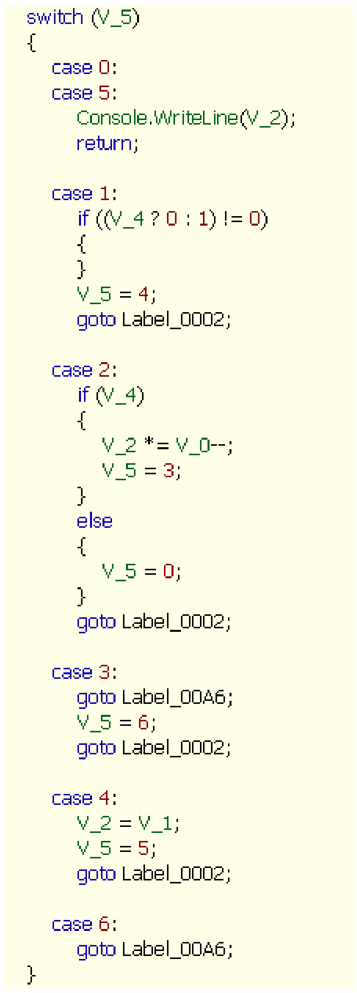


```
private static void Main(string[] args)
{
    Random random = new Random();
    int a = 1;
    int i = random.Next();
    int j = random.Next();
    if (i > j)
    {
        a = j;
    }
    else
    {
        while (i > 0)
        {
            a *= i--;
        }
    }
    Console.WriteLine(a);
}
```

Figure 4.25: Original program of example one

The code executed in lines 156 – 172 in Figure 4.24 is the same code as the lines 64-80 in Figure 4.1. The obfuscation technique used with setting six, corresponds to the basic control flow flattening technique described in section 2.2.2.2. Other than this technique, no other control flow obfuscation techniques were found in the obfuscated CIL code.

Figure 4.25 shows the original program from example one viewed in .NET Reflector. Figure 4.26 shows the obfuscated version with control flow set to high as setting in the DotFuscator.



```
switch (V_5)
{
   case 0:
   case 5:
      Console.WriteLine(V_2);
      return;

   case 1:
      if ((V_4 ? 0 : 1) != 0)
      {
      }
      V_5 = 4;
      goto Label_0002;

   case 2:
      if (V_4)
      {
         V_2 *= V_0--;
         V_5 = 3;
      }
      else
      {
         V_5 = 0;
      }
      goto Label_0002;

   case 3:
      goto Label_00A6;
      V_5 = 6;
      goto Label_0002;

   case 4:
      V_2 = V_1;
      V_5 = 5;
      goto Label_0002;

   case 6:
      goto Label_00A6;
}
```

Figure 4.26: The obfuscated version of example one

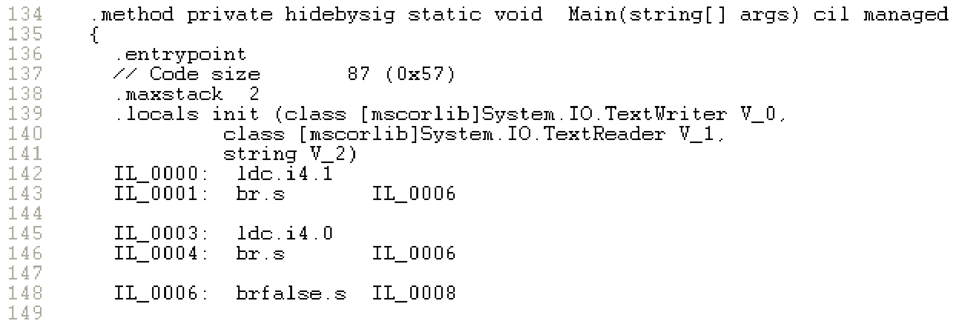
4.3.2 CIL code of example 2

The simulation of the second setting from Table 3.3 gave the same results in terms of the variables that were renamed as it did with the first example. The extra code that was generated by DotFuscator as shown in Figure 4.20 was also included in all the results after obfuscating example two.

When the enhanced overload renaming was tested in simulation (see setting three Table 3.3) there was no differences compared to the first setting, just as there was no differences in these simulations in the obfuscated CIL code of example one.

In the simulations with the fourth setting from Table 3.3 the results were identical to the original CIL code if the extra class generated by DotFuscator is not considered. This is a bit surprising, but the reason is probably that the second example does not contain any control statements and hence the control flow of the program is difficult to disguise with control flow flattening. The result from the fifth setting was identical to the fourth setting, just as it was when the first code example was obfuscated.

Just as the flow low setting did not manage to change the flow of the program in example two, the sixth setting from Table 3.3 did not alter the flow either. The only difference from the results with the previous setting was 5 extra instructions in the CIL code as shown on lines 142 – 148 in Figure 4.27. These five lines did not alter the flow of the code, but they manage to make the .NET Reflector unable to retrieve the corresponding C# code. These lines are therefore probably inserted to break decompiling tools.



```
134    .method private hidebysig static void  Main(string[] args) cil managed
135    {
136      .entrypoint
137      // Code size       87 (0x57)
138      .maxstack  2
139      .locals init (class [mscorlib]System.IO.TextWriter V_0,
140               class [mscorlib]System.IO.TextReader V_1,
141               string V_2)
142      IL_0000:  ldc.i4.1
143      IL_0001:  br.s       IL_0006
144
145      IL_0003:  ldc.i4.0
146      IL_0004:  br.s       IL_0006
147
148      IL_0006:  brfalse.s  IL_0008
149
```

Figure 4.27: The extra lines in the obfuscate CIL code of example two

4.3.3 CIL code of example 3

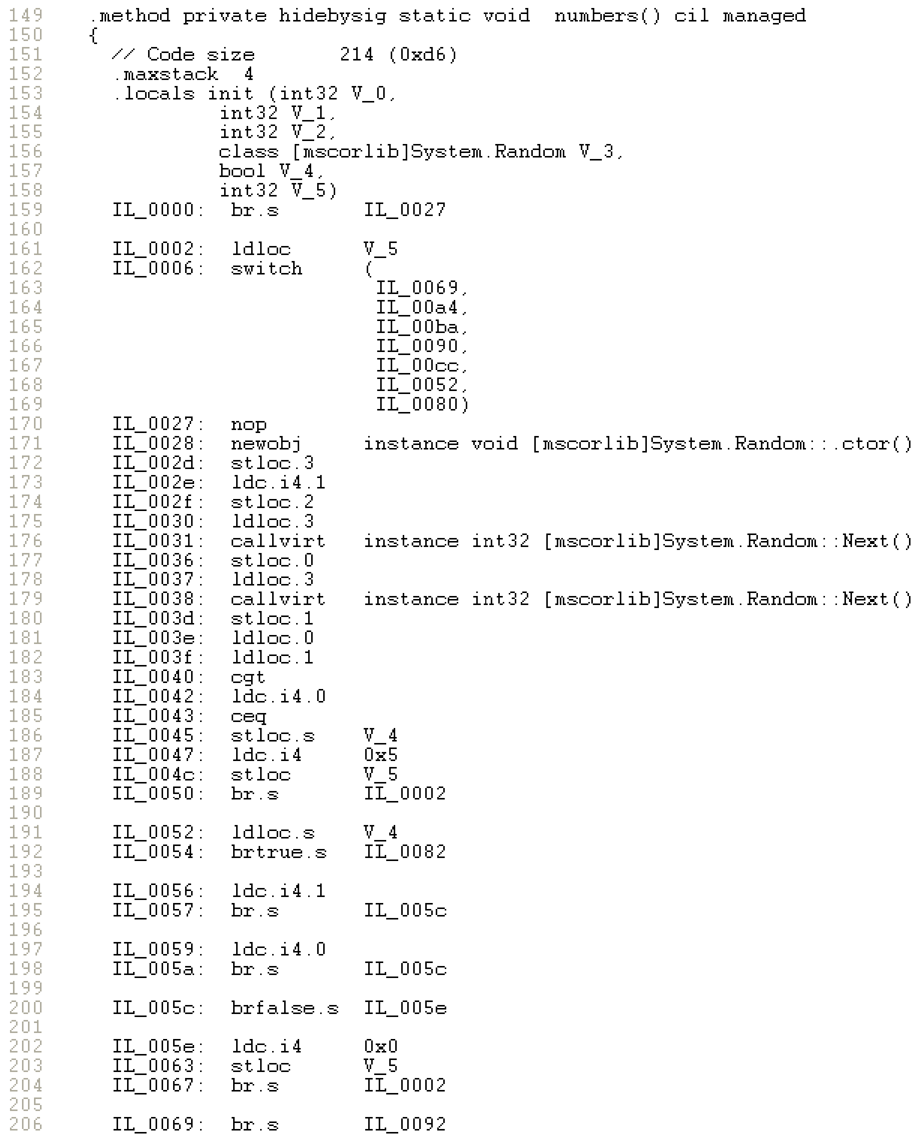
Since code example three is the two first code examples combined into one program (see chapter 3.2 for more information), the expectation for the simulation was that the alterations made by DotFuscator on this code example should be the combination of the obfuscated CIL code from the first two code examples. Maybe some small differences since the size of code example three is larger than the two earlier code examples.

The renaming of fields in the simulations with setting two and three resulted in the same results as from example one and two. It was no surprise that the second setting was identical to the earlier results, but it was more interesting to notice that the third setting was identical to the results obtained before. Since the size of this third code example is larger, it could be expected that the enhanced overload option would make some extra changes in this result, but no alterations were found.

As expected the simulations of setting four and five from Table 3.3 resulted in identical CIL code as before. This was no surprise since the simulations with the two different settings were identical for both code example one and two.

The simulations of the sixth setting on the third code example resulted in the same alterations in the `numbers()` function compared to the alterations made in the CIL code of example one. The difference between example one and three was that the sequence of different cases in the switch statement was altered, and the values given to the condition in the `switch()`was altered accordingly, so that the flow of the program did not change.

Figure 4.28 shows the obfuscated CIL code from example three and the switch on the lines 162–169 can be compared to the switch on the lines 148-155 in Figure 4.24. This means that the control flow flattening technique creates the same blocks for the cases each time, but the number for any given case is dynamic. The values for the dispatcher variable will thus also be different so the control flow within the switch will not change .



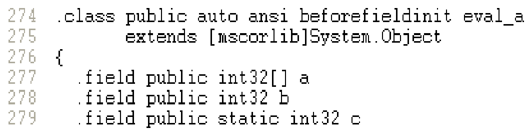
```
149    .method private hidebysig static void  numbers() cil managed
150    {
151      // Code size       214 (0xd6)
152      .maxstack  4
153      .locals init (int32 V_0,
154               int32 V_1,
155               int32 V_2,
156               class [mscorlib]System.Random V_3,
157               bool V_4,
158               int32 V_5)
159      IL_0000:  br.s       IL_0027
160
161      IL_0002:  ldloc      V_5
162      IL_0006:  switch     (
163                            IL_0069,
164                            IL_00a4,
165                            IL_00ba,
166                            IL_0090,
167                            IL_00cc,
168                            IL_0052,
169                            IL_0080)
170      IL_0027:  nop
171      IL_0028:  newobj     instance void [mscorlib]System.Random::.ctor()
172      IL_002d:  stloc.3
173      IL_002e:  ldc.i4.1
174      IL_002f:  stloc.2
175      IL_0030:  ldloc.3
176      IL_0031:  callvirt   instance int32 [mscorlib]System.Random::Next()
177      IL_0036:  stloc.0
178      IL_0037:  ldloc.3
179      IL_0038:  callvirt   instance int32 [mscorlib]System.Random::Next()
180      IL_003d:  stloc.1
181      IL_003e:  ldloc.0
182      IL_003f:  ldloc.1
183      IL_0040:  cgt
184      IL_0042:  ldc.i4.0
185      IL_0043:  ceq
186      IL_0045:  stloc.s    V_4
187      IL_0047:  ldc.i4     0x5
188      IL_004c:  stloc      V_5
189      IL_0050:  br.s       IL_0002
190
191      IL_0052:  ldloc.s    V_4
192      IL_0054:  brtrue.s   IL_0082
193
194      IL_0056:  ldc.i4.1
195      IL_0057:  br.s       IL_005c
196
197      IL_0059:  ldc.i4.0
198      IL_005a:  br.s       IL_005c
199
200      IL_005c:  brfalse.s  IL_005e
201
202      IL_005e:  ldc.i4     0x0
203      IL_0063:  stloc      V_5
204      IL_0067:  br.s       IL_0002
205
206      IL_0069:  br.s       IL_0092
```

Figure 4.28: The obfuscated numbers() function with the switch.

The alteration in the obfuscated version of the **`writeDate()`** was identical to the alterations described in section 4.3.2 with the same setting used in the simulations. Since the third code example contains the **`main()`** method, the obfuscation by DotFuscator was a little surprising since this method is not altered at all. The calls to these function could for example had been transformed in the same way as the outlining technique used in Spices .NET Obfuscator.

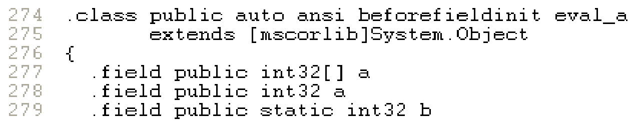
4.3.4 CIL code of example 4

The results from the simulation in DotFuscator with the first setting from Table 3.3 showed the same renaming of field as the results from the previous three code example. But unlike earlier simulations, the second setting from Table 3.3 showed difference in the renaming by the enhanced overload option. The difference between the two first settings is that the latter uses the same name on methods and variables in a broader extent than the results from the obfuscations with the first setting. Examples from the different settings are shown in Figure 4.29 – 4.32.



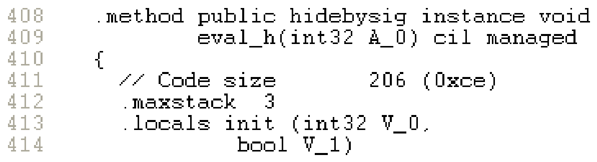
```
.class public auto ansi beforefieldinit eval_a
       extends [mscorlib]System.Object
{
  .field public int32[] a
  .field public int32 b
  .field public static int32 c
```

Figure 4.29: Renaming of variables from the first setting



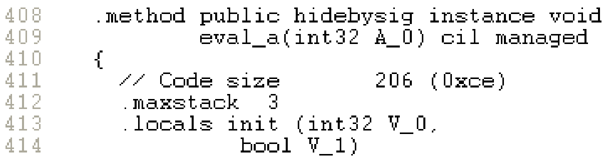
```
.class public auto ansi beforefieldinit eval_a
       extends [mscorlib]System.Object
{
  .field public int32[] a
  .field public int32 a
  .field public static int32 b
```

Figure 4.30: Renaming of variables from the second setting



```
  .method public hidebysig instance void
          eval_h(int32 A_0) cil managed
  {
    // Code size       206 (0xce)
    .maxstack  3
    .locals init (int32 V_0,
             bool V_1)
```

Figure 4.31: Renaming of method from the first setting

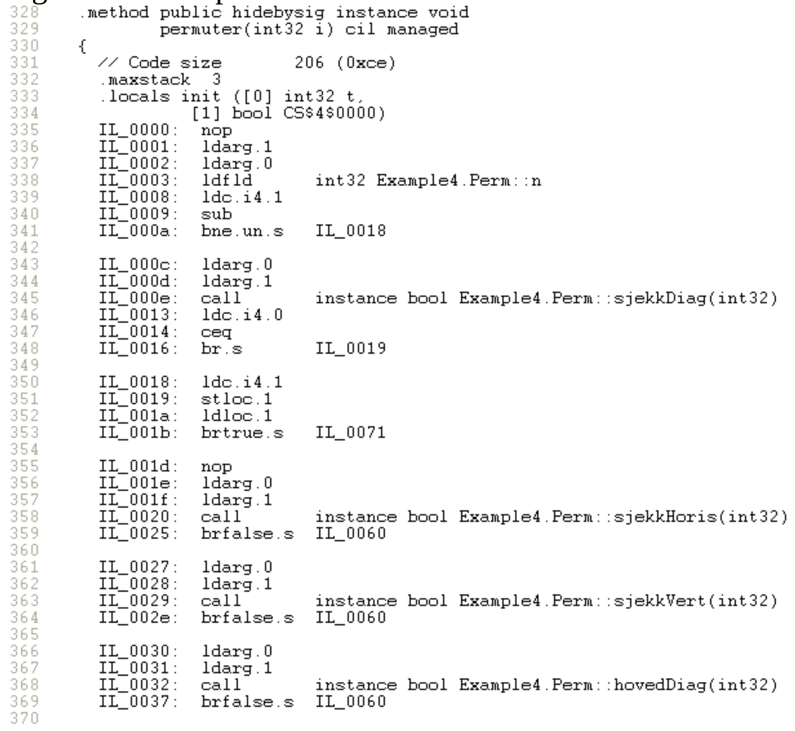


```
  .method public hidebysig instance void
          eval_a(int32 A_0) cil managed
  {
    // Code size       206 (0xce)
    .maxstack  3
    .locals init (int32 V_0,
             bool V_1)
```

Figure 4.32: Renaming of method from the second setting

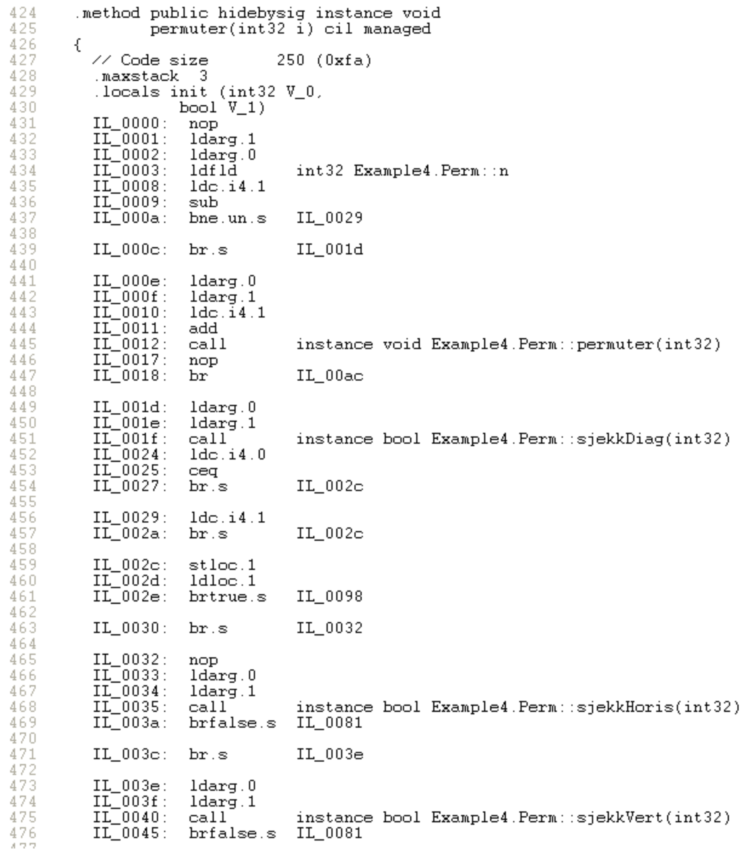
The fourth code example was then obfuscated with the fourth and fifth setting from Table 3.3. Surprisingly, the result from these simulations demonstrated that the DotFuscator did not alter the control flow differently than the changes made with the third setting, even though the aggressiveness for the control flow option was set to two different values. If the *low* and *medium* setting is supposed to be different, this should have been visible in the obfuscated CIL code of example four, which it was not.

However, the obfuscated CIL code from the fourth setting showed major differences compared to the original one. The control flow of some of the methods in example 4 is altered considerably by the DotFuscator. A small example of differences is shown in Figure 4.33 and 4.34. Like the figures show, the instructions are ordered in another sequence in Figure 4.34 compared to 4.33.



```
.method public hidebysig instance void
        permuter(int32 i) cil managed
{
  // Code size       206 (0xce)
  .maxstack  3
  .locals init ([0] int32 t,
           [1] bool CS$4$0000)
  IL_0000:  nop
  IL_0001:  ldarg.1
  IL_0002:  ldarg.0
  IL_0003:  ldfld      int32 Example4.Perm::n
  IL_0008:  ldc.i4.1
  IL_0009:  sub
  IL_000a:  bne.un.s   IL_0018

  IL_000c:  ldarg.0
  IL_000d:  ldarg.1
  IL_000e:  call       instance bool Example4.Perm::sjekkDiag(int32)
  IL_0013:  ldc.i4.0
  IL_0014:  ceq
  IL_0016:  br.s       IL_0019

  IL_0018:  ldc.i4.1
  IL_0019:  stloc.1
  IL_001a:  ldloc.1
  IL_001b:  brtrue.s   IL_0071

  IL_001d:  nop
  IL_001e:  ldarg.0
  IL_001f:  ldarg.1
  IL_0020:  call       instance bool Example4.Perm::sjekkHoris(int32)
  IL_0025:  brfalse.s  IL_0060

  IL_0027:  ldarg.0
  IL_0028:  ldarg.1
  IL_0029:  call       instance bool Example4.Perm::sjekkVert(int32)
  IL_002e:  brfalse.s  IL_0060

  IL_0030:  ldarg.0
  IL_0031:  ldarg.1
  IL_0032:  call       instance bool Example4.Perm::hovedDiag(int32)
  IL_0037:  brfalse.s  IL_0060

```

Figure 4.33: Original CIL code from example 4

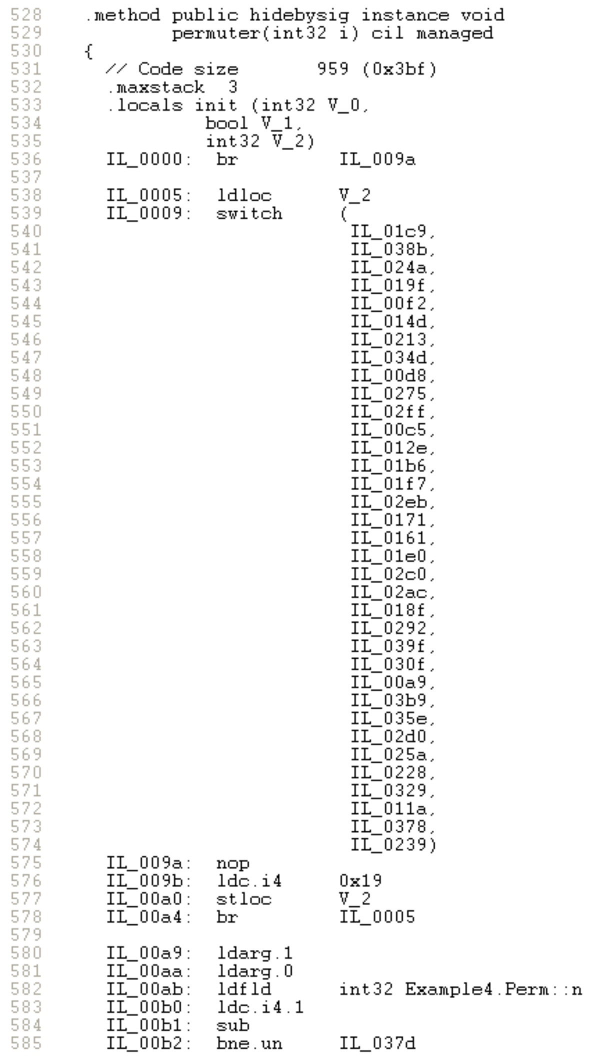


```
.method public hidebysig instance void
        permuter(int32 i) cil managed
{
  // Code size       250 (0xfa)
  .maxstack  3
  .locals init (int32 V_0,
           bool V_1)
  IL_0000:  nop
  IL_0001:  ldarg.1
  IL_0002:  ldarg.0
  IL_0003:  ldfld      int32 Example4.Perm::n
  IL_0008:  ldc.i4.1
  IL_0009:  sub
  IL_000a:  bne.un.s   IL_0029

  IL_000c:  br.s       IL_001d

  IL_000e:  ldarg.0
  IL_000f:  ldarg.1
  IL_0010:  ldc.i4.1
  IL_0011:  add
  IL_0012:  call       instance void Example4.Perm::permuter(int32)
  IL_0017:  nop
  IL_0018:  br         IL_00ac

  IL_001d:  ldarg.0
  IL_001e:  ldarg.1
  IL_001f:  call       instance bool Example4.Perm::sjekkDiag(int32)
  IL_0024:  ldc.i4.0
  IL_0025:  ceq
  IL_0027:  br.s       IL_002c

  IL_0029:  ldc.i4.1
  IL_002a:  br.s       IL_002c

  IL_002c:  stloc.1
  IL_002d:  ldloc.1
  IL_002e:  brtrue.s   IL_0098

  IL_0030:  br.s       IL_0032

  IL_0032:  nop
  IL_0033:  ldarg.0
  IL_0034:  ldarg.1
  IL_0035:  call       instance bool Example4.Perm::sjekkHoris(int32)
  IL_003a:  brfalse.s  IL_0081

  IL_003c:  br.s       IL_003e

  IL_003e:  ldarg.0
  IL_003f:  ldarg.1
  IL_0040:  call       instance bool Example4.Perm::sjekkVert(int32)
  IL_0045:  brfalse.s  IL_0081
```

Figure 4.34: Obfuscated CIL code with the low control flow option

The simulations of the sixth setting showed major modifications in the obfuscated CIL code. The DotFuscator alters the control flow in extensively, and as the Figure 4.35 shows, the CIL code is very different and more difficult to understand compared to the original one in Figure 4.33.

For each method the whole block, except from the declaration in the beginning of the code, has been rewritten into a switch using basic control flattening technique (like with example one and three). However, not all methods in example four are rewritten; rewriting of methods without control statements would not make it any harder to understand, since the flow of the method has to be kept as it is. This is for example the case with the **`public void bytt(int i, int j)`**method in code example four.



```
.method public hidebysig instance void
        permuter(int32 i) cil managed
{
  // Code size       959 (0x3bf)
  .maxstack  3
  .locals init (int32 V_0,
           bool V_1,
           int32 V_2)
  IL_0000:  br         IL_009a

  IL_0005:  ldloc      V_2
  IL_0009:  switch     (
                        IL_01c9,
                        IL_038b,
                        IL_024a,
                        IL_019f,
                        IL_00f2,
                        IL_014d,
                        IL_0213,
                        IL_034d,
                        IL_00d8,
                        IL_0275,
                        IL_02ff,
                        IL_00c5,
                        IL_012e,
                        IL_01b6,
                        IL_01f7,
                        IL_02eb,
                        IL_0171,
                        IL_0161,
                        IL_01e0,
                        IL_02c0,
                        IL_02ac,
                        IL_018f,
                        IL_0292,
                        IL_039f,
                        IL_030f,
                        IL_00a9,
                        IL_03b9,
                        IL_035e,
                        IL_02d0,
                        IL_025a,
                        IL_0228,
                        IL_0329,
                        IL_011a,
                        IL_0378,
                        IL_0239)
  IL_009a:  nop
  IL_009b:  ldc.i4     0x19
  IL_00a0:  stloc      V_2
  IL_00a4:  br         IL_0005

  IL_00a9:  ldarg.1
  IL_00aa:  ldarg.0
  IL_00ab:  ldfld      int32 Example4.Perm::n
  IL_00b0:  ldc.i4.1
  IL_00b1:  sub
  IL_00b2:  bne.un     IL_037d
```

Figure 4.35: Obfuscated CIL code from example 4 with the sixth setting from Table 3.3

The same obfuscation techniques used in example four as with the previous code examples; control flow flattening, renaming of symbols and also protection against decompilation.

5 Analysis of Obfuscated CIL code

This chapter contains an evaluation of the obfuscation techniques that were discovered in the previous chapter. The evaluations are done for each obfuscator tool, and a basic discussion about the effects of the techniques is presented. Possible suggestions for enhancements are also presented where it is suited.

5.1 Salamander .NET Obfuscator

As described in section 3.3.1, there were some expectation to which obfuscation techniques that should be applied in the obfuscation process based on the information provided about the Salamander .NET Obfuscator. The features were renaming of different symbols, protection against ILDASM and control flow obfuscation.

5.1.1 Renaming of fields

The different results extracted from the Salamander .NET Obfuscator were interesting, even though some unexpected. The technique of renaming different information is widely used, with renaming of variables, methods and classes, as seen in the numerous examples (See Figure 4.2 - 4.11). Variable renaming makes it harder to read the code, but it is far from impossible. The different variables are renamed with different names and do not use the same name (where it is possible). This makes the structure of the program rather easy to read, since the names are distinct through the different functions. It is also worth noticing that the renaming technique in Salamander .NET is static, which means that the renaming is performed in the same way in all simulations.

If the name of all variables were set to the same, where it is possible, for example **`a`**, it would be more confusing to understand which variable that is referenced where.

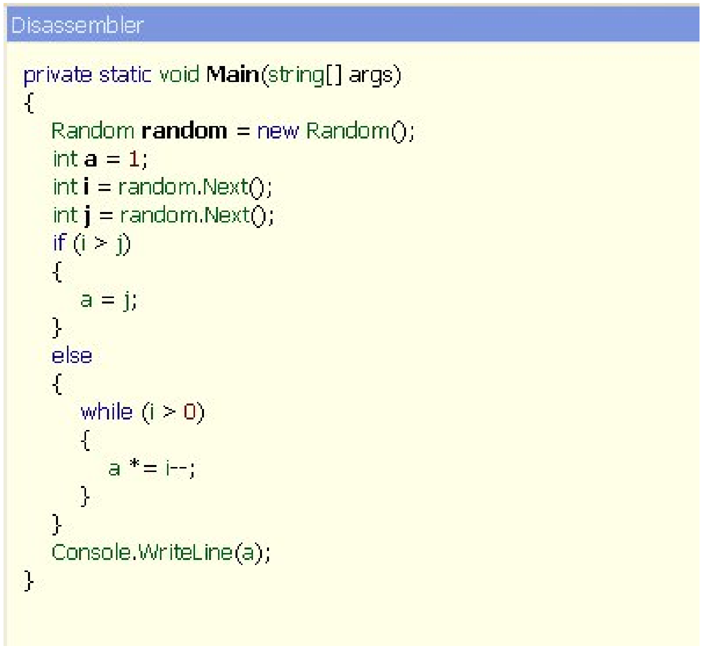
The variable name starts at **`V_0`** and are further renamed by changing only the number, so the next name is **`V_1`**, as shown in Figure 4.3 – 4.12. These variable names could instead be set to different symbols and characters chosen randomly to make it more difficult to follow the variables throughout the code. Like a#¤%fsdw instead of **`V_0.`**

It was a surprising finding that the different method calls were not obfuscated, as seen on line 70 in Figure 4.3 or line 138 in Figure 4.9. Obfuscating these calls would make the CIL code more confusing and difficult to read.

The Salamander .NET Obfuscator renames all classes and methods to the same symbol, either an "**`A`**" or "**`a`**". The tool will try to use as few symbols as possible and it will continue to rename classes to **`A`**, **`B`**, **`C`** and so on in both upper and lower case. However, as stated before, it is not difficult to follow the flow of the program. This renaming is therefore performed statically.

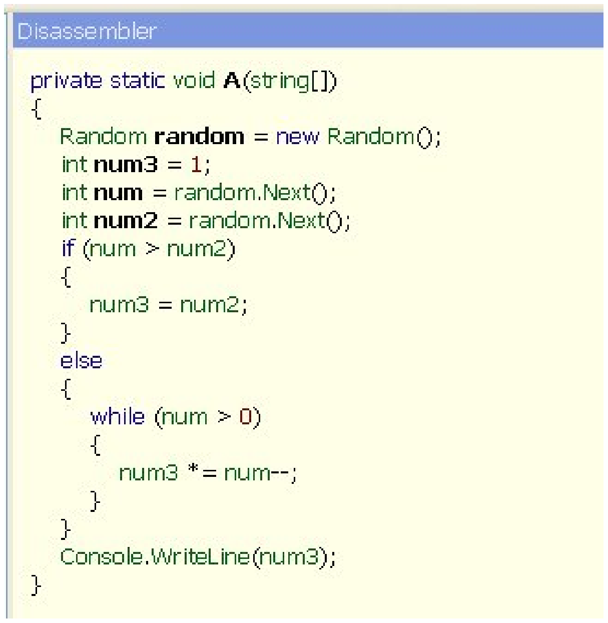
An enhancement to the Salamander .NET Obfuscator would be to do this by using method overloading [50] . One method can take a boolean value as a parameter, another method can take a int value as parameter and so on. If two functions of the same type occur, they can be given the same name as long as they are different in the parameters of the function. For example: **`bool a(int 32 A_0)`** and **`bool a(int32 A_0, int32 A_1).`**

To show the impact of variable renaming, the Lutz Roeder's .NET Reflector was used to show what information it would be able to obtain given only the obfuscated CIL code. The following Figure 5.1 and 5.2 shows the results from the reflector with the main function of the program. The alteration in the obfuscated results are not major; the only thing changed is that the if condition has been reversed and hence the if end else statement in Figure 5.1 has changed places in Figure 5.2.



```
private static void Main(string[] args)
{
    Random random = new Random();
    int a = 1;
    int i = random.Next();
    int j = random.Next();
    if (i > j)
    {
        a = j;
    }
    else
    {
        while (i > 0)
        {
            a *= i--;
        }
    }
    Console.WriteLine(a);
}
```

Figure 5.1: Decompiling the original program in example one



```
private static void A(string[])
{
   Random random = new Random();
   int num3 = 1;
   int num = random.Next();
   int num2 = random.Next();
   if (num > num2)
   {
      num3 = num2;
   }
   else
   {
      while (num > 0)
      {
         num3 *= num--;
      }
   }
   Console.WriteLine(num3);
}
```

Figure 5.2: Decompiling the obfuscated program of example one

5.1.2 ILDASM protection

It is obvious that the ILDASM protection does not work. For the first, the obfuscated CIL code is identical with the default obfuscation in all the four examples. There is not a single variation when using the ILDASM protection, which has also been confirmed by the *diff* function in Linux.

A proof that supports this observation is that the actual CIL code of the files obfuscated with this option is shown in their full, see Figure 4.4, 4.5 and 4.9. Since the ILDASM protection is not applied in any of the code examples that were simulated, there seems to be a bug in the Salamander .NET Obfuscator. This is an option of the tool as shown in Figure 3.1, so it should be turned on as a default.

Another thing to consider is how well the ILDASM protection would work even if it made it impossible to view the CIL code in this tool. There are many other tools that might be used for decompiling the CIL code besides ILDASM, as shown in Figure 5.1 and 5.2. So the protection will only work for one specific tool.

5.1.3 Control flow

Salamander .NET Obfuscator did not use control flow obfuscating techniques in any of the examples we used. Even on the early stages of the simulations, it was noticed that the settings for the Salamander .NET Obfuscator did not contain any options regarding control flow features. However, the product page of Salamander claims to contain this technique [32], so it was unexpected that no control flow obfuscating techniques were applied during simulation.

The results found during simulation emphasized the fact that no control flow obfuscation was implemented in the Salamander .NET Obfuscator, since there is no alteration of the control flow in the test code examples. Hence the impression given by the developer is wrong compared to the actual product.

On a later stage it was discovered that the user manual [34] of the product stated that the control flow feature was not implemented in the downloadable version. This feature has still not been implemented.

5.2 Spices .NET Obfuscator

Based on the documented features of the Spices .NET Obfuscator, described in section 3.3.2, there were some expectation of obfuscation techniques that should be applied during the obfuscation of the different code examples. These features included entity renaming, control flow obfuscation, method call anonymization and ILDASM protection.

5.2.1 Renaming of fields

The results from the simulations performed with the Spices .NET Obfuscator brought some surprises. After the simulation of the first three code examples, the obfuscators renaming property seemed to work as it should, regardless of the settings that were tested. The renaming of fields were although static as with the Salamander .NET Obfuscator.

The renaming of variables was performed like with the Salamander .NET Obfuscator, where they were given names V_0, V_1 and so on. All the parameters in the different methods were also renamed in the same manner as the Salamander .NET Obfuscator, starting at A_0 and increasing.

Instead of the letters ranging from a/A and upwards as with the Salamander .NET Obfuscator when renaming classes, the results from the Spices .NET Obfuscator showed that it used numbers starting at 0 and increasing. None the less, the same technique was used, just with different names.

In the last code example some unexpected results were revealed. The obfuscated CIL code from the first four settings from Table 3.2, showed that only the first class was renamed as expected, even though the variables were renamed in all four classes.

However, with the fifth settings (see Table 3.2), it was expected that all classes should be renamed. This was not the case, as the results showed (see section 4.2.4). Since the last two classes were not renamed, information in the CIL code that should have been removed was revealed. This makes the technique less resistant towards reverse engineering. This is probably a bug in the Spices .NET Obfuscator, since the final setting from Table 3.2 renamed these fields as expected.

All the results from the simulations had interesting findings regarding the protection provided by the Spices .NET Obfuscator. If the CIL code that is obfuscated is small, i.e. contains one class, the renaming of this class works at it should regardless of the settings of the obfuscator. But as revealed, the setting of the Spices .NET Obfuscator could be very crucial if the code example is larger.

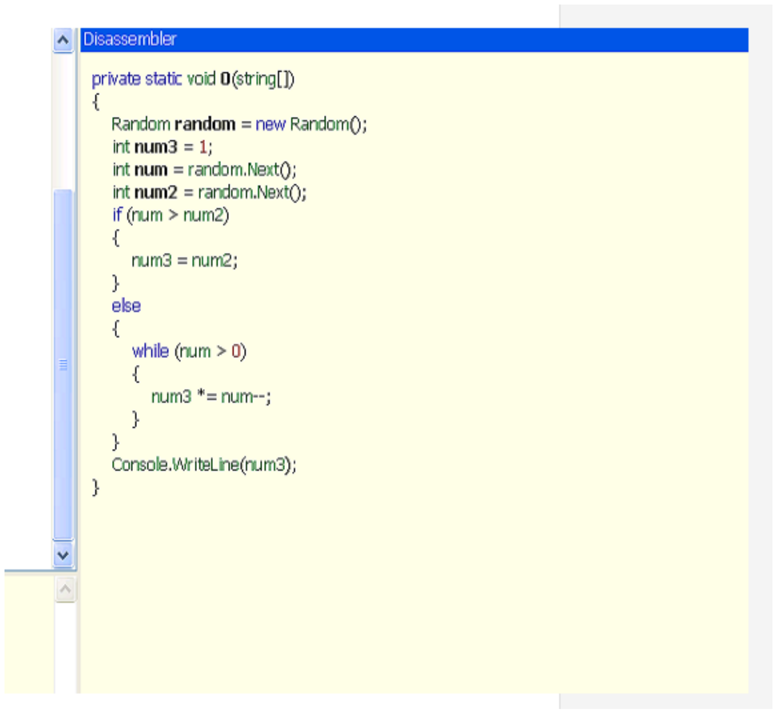
It is a weakness that the default options of Spices .NET Obfuscator does not provide sufficient renaming of all classes in a program, although this is probably a bug in the Spices .NET Obfuscator's code. The Salamander .NET Obfuscator does not have any problems with renaming all classes so the same technique can be implemented in the Spices .NET Obfuscator.

5.2.2 ILDASM protection

In Spices .NET Obfuscator, the ILDASM protection works as expected when the option is set to *true*, as shown in Figure 4.14. It is a bit confusing that when the ILDASM option is set to *complete*, it means that the protection against the MSIL Disassembler is the same as if it was set to *false*. There are no changes made in the CIL code.

The *complete* option would for many imply that it should offer more protection of the code, instead of the complete opposite. Since the option does not have any effect on the obfuscation of the Spices .NET Obfuscator, it is a bit unclear why this option is available at all. The documentation of the obfuscator does not explain the differences between the true and complete setting, it only gives the impression that both will give protection against the ILDASM. It is therefore very likely that it is a bug in the Spices .NET Obfuscator, since is does not add ILDASM protection with the *complete* setting.

As discussed in the section 4.4.2, the ILDASM protection might mislead the users, if they believe that the ILDASM protection adds to the security of obfuscated code. Even though the protection works with the ILDASM, it does not protect the original CIL code from other decompilers. The extraction of the main method of the code was decompiled without trouble with Reflector. The Figure 5.3 shows the output from the Lutz Roder's .NET Reflector.


Disassembler

```
private static void 0(string[])
{
    Random random = new Random();
    int num3 = 1;
    int num = random.Next();
    int num2 = random.Next();
    if (num > num2)
    {
        num3 = num2;
    }
    else
    {
        while (num > 0)
        {
            num3 *= num--;
        }
    }
    Console.WriteLine(num3);
}
```

Figure 5.3: Decompiling the CIL code of example one obfuscated with ILDASM protection

5.2.3 Control flow / anonymizer

As discussed in section 4.5.1, the last two settings in the simulations revealed results where system calls and other method calls are respectively obfuscated by the Spices .NET Obfuscator, as shown in Figure 4.5 – 4.7.

This method can be seen as the outlining obfuscation technique described in section 2.2.2.6, since every call is created into new methods. It is worth noticing that this technique should be applied to random selected statements, if it is supposed to be effective. This is not done in the way that Spices .NET Obfuscator uses outlining. It is therefore not making the control flow very much harder to understand, since it is clear that all calls are rewritten by generation new wrapper methods.

This outline obfuscation technique has a drawback, since it generates more code. After the obfuscation was applied with the Spices .NET Obfuscator (code example four), the original 1624 lines of CIL code was replaced with 2341 lines. This could reduce the efficiency of the program, although this was not tested in this scenario. As discussed earlier, the program that will be obfuscated with this obfuscator will probably be much larger than the examples used, so the effects make a major difference regarding the size of the CIL code, which can be a problem if size is important

One last thing that the Spices .NET Obfuscator adds to the original CIL code is the extra class shown in Figure 4.13. Because the version using during simulation was a trial version, this class was added to the CIL code. It is unclear what the function of this class is. The documented reason from 9Rays is to ensure that the obfuscated code can not be distributed [1]. However, when the obfuscated executables is executed, they work as they should (without any output that says this has been obfuscated by the Spices .NET Obfuscator). Since the functionality of the class is never initiated it does not alter the control flow of the program either.

The CIL code itself is obfuscated with all the techniques that Spices .NET Obfuscator includes, so the warning does not constrain the result. The only way to notice the warning from the obfuscator is if the executables is disassembled. This information would tell the intruder that the code he is viewing has been obfuscated by this particular obfuscator, but the intruder still needs to figure out what the Spices .NET Obfuscator has used of obfuscation techniques

5.3 DotFuscator

Features that were expected in the DotFuscator included control flow obfuscation, renaming of fields and protection against various decompilers.

As described in section 4.3.1 and seen in Figure 4.20, one commonality was found in all the obfuscations of the code examples: an extra class in the beginning of each program, called DotfuscatorAttribute, is added by DotFuscator. The reason for this extra class was not found with the different settings shown in Table 3.3, since the class is not initiated in any of the results.

It is possible that this is a class that is used when string encryption or watermarking is turned on, since one would have to store this information somewhere in the CIL code. Another possibility is that it is used when the string encryption needs to be decrypted under execution. But these settings were, as described in section 3.3.1.3 not simulated in this thesis, so the motivation of this class is therefore unclear.

It was surprising that this class is not excluded if it is not needed. Though the class adds some extra code to the results it does not alter the control flow of the actual program since the methods of the class is never called. It is therefore not difficult to understand the extra CIL code nor is it difficult to see that this bit of code can be overlooked when trying to understand a program. There are no descriptions of this class in the documentation of DotFuscator [40].

5.3.1 Renaming of fields

The simulations with the four different code examples were obfuscated a little different in DotFuscator compared to the other two obfuscators. Since it was possible to isolate the different obfuscation techniques, the renaming technique was tested on all four test cases with two different settings (the normal renaming and with the enhanced overload option on as described in setting one and two in Table 3.3).

The standard renaming worked as expected. The classes were renamed with **`eval_a`**, **`eval_b`** and so forth as shown in Figure 4.21. It is possible that this name would only be one character if the edition of the DotFuscator was not an evaluation copy, since **`eval`** may be short for evaluation (trial). But it would not make any difference to the quality of the obfuscation, since the original name is lost nevertheless.

The methods are renamed in different ways. Some has only one letter such as **`a`**, and some has **`eval_a`**. Both of the renaming schemes changes **`a`** to **`b`** and so on when renaming the rest of the methods. Figure 4.21 shows a method with only the single letter as a name and Figure 4.29-4.32 shows different renaming with the latter name. The renaming in DotFuscator is, however, static. The same names are used every time.

The parameter of the methods is renamed in the same way as in both the Salamander .NET and the Spices .NET Obfuscator. They start at **`A_0`** and increases. The different variables in the obfuscated code is also renamed in the same way as before, starting with **`V_0`** and ranging upwards to **`V_1, V_2`** and so on. The variable renaming can be seen on lines 139-143 in Figure 4.21.

As presented in chapter 4.3.3, where the results of the DotFuscator was presented, the effects of the enhanced overload setting seems to be very limited. There were only differences in the renaming in the simulations of the fourth and largest code example, where some of the changes can be seen in Figure 4.29-4.32. The major differences was that the simulations with the enhanced overload setting tried to rename as many fields as possible to the same name.

It is also a bit unclear why the renaming of methods uses two different types of names, both a single letter and **`eval`** plus a letter. The user manual to DotFuscator [40] does not contain any information about the limitations of the evaluation copy ,so it is not possible to say if the reason is caused by the evaluation copy.

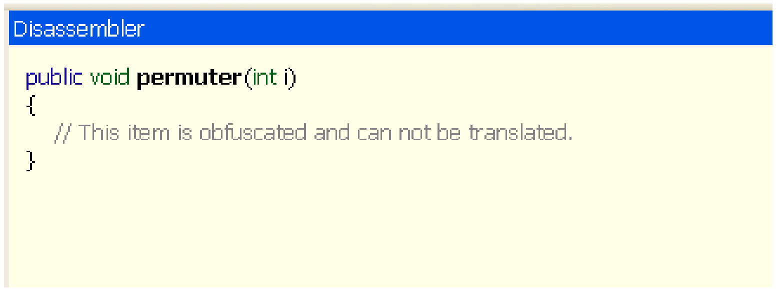
Further enhancements of the renaming could be by using more confusing names, and also by renaming as many fields to the same name legally in terms of the scope rules.

5.3.2 Protection against decompilers

Surprisingly, the DotFuscator did not have any option regarding protection against ILDASM or other decompilers, as can be seen in Figure 3.3. The obfuscated CIL code is therefore not difficult to decompile with the MSIL Disassembler. The FAQ on DotFuscator's homepage explains that protection against decompilers is removed, because they can not keep up with the development of the ILDASM. This was an option in the earlier versions of the DotFuscator (see [39]).

The DotFuscator does although fool the .NET Reflector. In some cases the only information provided by the .NET Reflector is shown in Figure 5.4. It should be possible to remove the CIL code that is making the decompiler unable to retrieve the C#, since

ILDASM still can decompile and extract the CIL code. It is therefore some legal CIL code which forms one (or several) illegal C# statements, that must be removed.


Disassembler

```
public void permuter(int i)
{
    // This item is obfuscated and can not be translated.
}
```

Figure 5.4: Decompilation of an obfuscated method

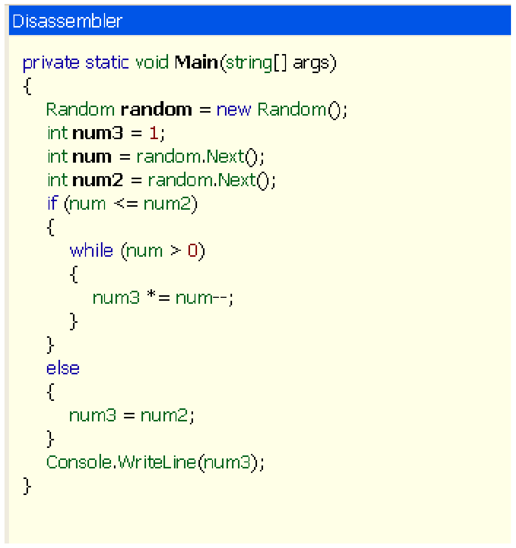
5.3.3 Control flow

The simulations in DotFuscator with the three last settings from Table 3.3 resulted in control flow obfuscation techniques, like those discussed in section 2.2. The modifications of the code examples resulted in very different control flows compared to the original code.

There were some surprising findings in the results, however. The simulation with the control flow option set to *low* and *medium* resulted in identical results, which was unexpected. The result was the same for every test code example, so the length of the code does not seem to be the explanation.

The user manual for the DotFuscator [40] was investigated for a reason, but the only information the manual provided, was that the three different options only sets the aggressiveness of the control flow. In order words, it should have been differences in the simulations results. The unexpected results may be because by a bug in the DotFuscator. It is also possible that the medium setting only triggers at specific code instructions that are not present in our code examples.

Obfuscation results with the fourth setting, i.e. the control flow option set to *low*, does alter the control flow in profound ways. The changes made with the low setting do not make the actual obfuscated CIL code to difficult to understand. Figure 5.1 shows the original C# code and Figure 5.5 shows the obfuscated version of the C# code.



```
private static void Main(string[] args)
{
    Random random = new Random();
    int num3 = 1;
    int num = random.Next();
    int num2 = random.Next();
    if (num <= num2)
    {
        while (num > 0)
        {
            num3 *= num--;
        }
    }
    else
    {
        num3 = num2;
    }
    Console.WriteLine(num3);
}
```

Figure 5.5: Example 1 with control flow set on low

Comparison of the two figures shows that the *if test* has changed operators from > to <= and so the contents of the if and else section has switch places. This is the only change applied by the control flow techniques, when the setting is set to *low*. As the Figure 5.5 shows, the actual flow and content of the program is not very difficult to understand.

An unexpected result from the simulations was revealed in the simulation of code example two. The DotFuscator did not alter the original control flow, but it just added 5 extra lines of code to the obfuscated result. These lines seem to be inserted to confuse or break the .NET Reflector, and possibly other decompilation tools. Figure 4.27 shows these extra lines of code which causes the .NET Reflector to not be able to decompile the CIL code. But if these few lines are excluded from the obfuscated CIL code, the rest of the program would be identical to the original one.

The obfuscations with the sixth setting revealed major alteration of the control flow. The entire body of the first code example was modified. Instead of *if else* statements and loops, they were split up in a switch statement, which also included variables and instructions performed on these variables that could never be executed. This is in context with the basic control flow flattening technique described in section 2.2.2.2.

Close examination of the switch shows that the original code of example one is present in case five, two and four, where the original content of the *if test* from Figure 3.13 is present in case four, the else content is in a rewritten form in case two and the rest of the original code is in case five.

The alterations made by DotFuscator make the control flow much harder to understand, especially with larger code examples. The complete obfuscated result from the fourth code example can be found on the CD. It is worth mentioning that DotFuscator only uses the basic control flow flattening and not the enhancements presented in section 2.2.2.3

and 2.2.2.4. This does not make the control flow too hard to understand, since the values that are being assigned to the dispatcher value is still constants in the different cases (or program blocks as they are called in 2.2.2.2).

An unexpected result occured with the control flow set on high, when obfuscating code example two. The alteration made in the actual results was the same as with the low setting, and the same lines of code was included in the results. This may be caused by the fact that moving different statements into switch cases, when the control flow between them can not be altered, would not make the control flow of the program much harder to understand.

It was also a bit surprising to see that when the third code example was obfuscated, the modification performed on the obfuscated CIL code was the same as the combination of the alteration from the simulations of code example one and two. The only difference was that the contents of the different switch cases shown in Figure 4.26 were rearranged with different labels.

6 Reversing Obfuscation Techniques

This chapter presents several algorithms for reversing obfuscated CIL code. Based on the evaluation from Chapter 5 and the known theory from reversing obfuscated code (see Chapter 2) along with some new approaches, some interesting algorithms for reversing obfuscated code have been developed.

Section 6.1 will discuss the obfuscation techniques that were discovered in Chapter 4 and connected them with possible reversing techniques. In section 6.2 an algorithm that reverses the outline obfuscation technique is presented. This algorithm is general and can be applied to different outlining techniques, as shown in this section. Following in 6.3, the ILDASM protection is discussed and one approach for removing this protection is shown. Finally, in section 6.4, a basic control flow flattening algorithm is outlined, which can reverse several advanced obfuscation techniques.

6.1 Found obfuscation techniques

The results and evaluation of the obfuscated code from the three different obfuscators used in this thesis, showed that there were surprisingly fewer obfuscation techniques used than one might expect. Renaming of fields such as classes, methods, parameters and variables was the only common technique used by all three obfuscator. More advanced obfuscation techniques like control flow obfuscation were only presented in two of three obfuscators tested.

The renaming itself can be a little confusing for one who is reading the CIL code, but as discussed in section 2.1, this alone is not enough to make the CIL code much harder to understand. It might take a person some more time before mapping out which reference a particular name points to, but because of the scope of the programming language there are restrains to how efficient this obfuscation technique is.

Two of the obfuscators offered techniques that were intended to protect the obfuscated code against ILDASM and also other decompilers. In the first obfuscator, Salamander .NET, this technique was not observed, even though the program stated if should have this protection. . The Spices .NET does manage to crash ILDASM, but it does not fool the Reflector. DotFuscator also offer protection against decompilers, however, not ILDASM. The .NET Reflector does not manage to show the obfuscated code correctly when some extra lines of code is included in the CIL code as shown in Figure 5.4.

Both Spices .NET and DotFuscator have implemented control flow obfuscation techniques as a feature, but they uses different techniques. Spices .NET uses a technique called outlining, which is presented in section 2.2.2.6. This techniques takes sections of the code and rewrites them into own methods. DotFuscator on the other hand uses basic control flow flattening, which is described in section 2.2.2.2. This can be recognized by the way all methods are transformed into a switch statement.

All in all there are only three obfuscation techniques used by the obfuscators that can be said to be common, even though they are implemented in different ways:

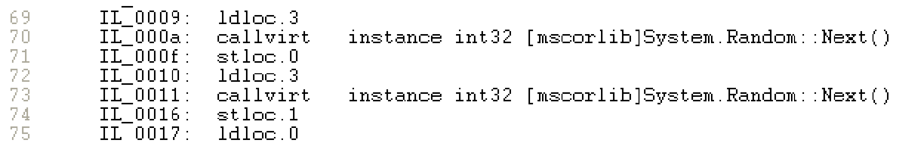
1. Renaming of variables. This is a **one way** technique, since the original names are lost for good when the alterations are performed. It might although be possible to redo some of the renaming, so that the obfuscated code is easier to read as described in section 2.1
2. Protection against decompilers. – although it is not clear whether or not this is an actual obfuscation technique, it does in fact protect the CIL code from some basic decompilation. This technique is **reversible,** since the crashing is done by adding some extra code into the obfuscated CIL code that the decompiler is not able to understand. See section 2.3.
3. Control flow obfuscation. These are also **reversible** technique as discussed in section 2.2, since the changes are performed on the original control flow

6.2 Outlining

As described in section 4.2.1 the results from simulations with Spices .NET Obfuscator showed that the anonymizer setting rewrote different types of calls, by wrapping them into a new method. This does create a lot of extra code, but it also confuses someone who is trying to understand the code.

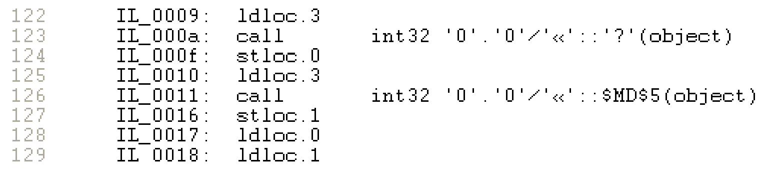
In section 5.2.3 this technique was described as a version of the outlining technique explained in section 2.2.2.6. The technique used by Spices .NET is not used optimal, as it could be more efficient by including randomness. Randomness can be increase the efficiency of the outlining technique, by rewriting random sections of the code into new methods. The Spices .NET Obfuscator only rewrites system or other calls into a new method and does not include the randomness.

Also shown in the results from Spices .NET was that the rewriting of the methods was performed in two different levels according to the chosen option in the obfuscator. One only lead to the system calls to be rewritten, the other rewrote every call that was made throughout the whole code. But the rewritings was done in the same way with both cases, so our reversing method can be applied to both.



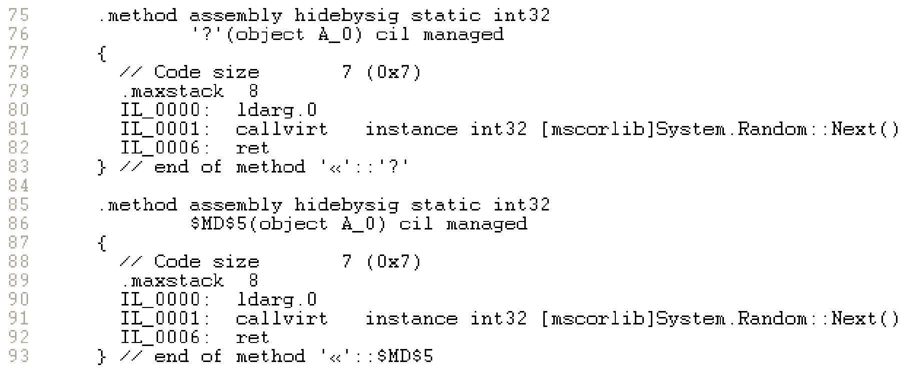
```
IL_0009:  ldloc.3
IL_000a:  callvirt   instance int32 [mscorlib]System.Random::Next()
IL_000f:  stloc.0
IL_0010:  ldloc.3
IL_0011:  callvirt   instance int32 [mscorlib]System.Random::Next()
IL_0016:  stloc.1
IL_0017:  ldloc.0
```

Figure 6.1: Original CIL code from example one



```
122      IL_0009:  ldloc.3
123      IL_000a:  call        int32 '0'.'0'/'«'::'?'(object)
124      IL_000f:  stloc.0
125      IL_0010:  ldloc.3
126      IL_0011:  call        int32 '0'.'0'/'«'::$MD$5(object)
127      IL_0016:  stloc.1
128      IL_0017:  ldloc.0
129      IL_0018:  ldloc.1
```

Figure 6.2: Fragment of the obfuscated CIL code from example one



```
75      .method assembly hidebysig static int32
76              '?'(object A_0) cil managed
77      {
78        // Code size       7 (0x7)
79        .maxstack  8
80        IL_0000:  ldarg.0
81        IL_0001:  callvirt   instance int32 [mscorlib]System.Random::Next()
82        IL_0006:  ret
83      } // end of method '«'::'?'
84
85      .method assembly hidebysig static int32
86              $MD$5(object A_0) cil managed
87      {
88        // Code size       7 (0x7)
89        .maxstack  8
90        IL_0000:  ldarg.0
91        IL_0001:  callvirt   instance int32 [mscorlib]System.Random::Next()
92        IL_0006:  ret
93      } // end of method '«'::$MD$5
```

Figure 6.3: The correspond methods to Figure 6.2

Figure 6.1 shows a fragment of the original CIL code of example one where the two system calls are performed on line 70 and 73. Figure 6.2 shows the corresponding fragment from the obfuscated version of the same code example with the calls performed on line 123 and 126, and Figure 6.3 shows the methods which correspond to the calls in Figure 6.2.

It should be clear that the instruction IL_0001 in both of the methods in Figure 6.2 is identical IL_000a and IL_0011 in Figure 6.1. One can also see that the methods called '?' and '$MD$5' are the two methods called in Figure 6.1. It is therefore possible to obtain the original call with the following algorithm:

(i) locate all the calls performed in the different methods in the original classes.

(ii) go through every call and for each:

    (1) find the corresponding method in the internal class. It is important to get a full match with the obfuscated once, since many of the new methods might have the same name. If they do have the same name, they will be differently in the type of method (int32, void and so forth) or in their number of parameters.
    (2) If the corresponding is not within the internal class, then the call is towards another method that was there originally. Repeat from ii
    (3) locate the call in the new method – it will only be one call per method
    (4) copy the whole instruction from the new method and replace the original call with this one.

(iii) the method created by the obfuscator can be removed if desired

This algorithm is based on the inlining technique, which is also discussed in section 2.2.2.6. The inlining technique does the exact opposite of the outlining technique so the effect outlining can therefore be removed this way.

In the Figure 6.2 one of the calls that need to be replaced is IL_000a on line 123. The whole instruction here reads:

```
call  int32 ‘0’.’0’/’«’::’?’(object)
```

The method here is the one which starts at line 75 in Figure 6.3. From the instruction above, it is clear that the instruction is looking for an **`int32`** method called **`‘?’`** in the internal class **`‘«’`** in the class called **`‘0’.’0’`**, and it only takes one argument; **`(object)`**. The only call in this new method is located on line 81 and the whole instruction (IL_0001) reads:

```
callvirt  instance int32 [mscorlib]System.Random::Next()
```

So if the original instruction IL_000a in Figure 6.2 is replaced with the instruction above, it will read:

```
IL_000a: callvirt  instance int32 [mscorlib]System.Random::Next()
```

and it will look exactly like the original one located in line 70 in Figure 6.1.

As described in section 4.2.1 all the methods that are obfuscated in a class are rewritten to a new internal class that contains all the new methods. The Spices .NET Obfuscator obfuscated all these classes the same way each time. The same algorithm can therefore be used every time for every obfuscated call. This algorithm is thus generic and should be applicable for outlining obfuscation as long as the outlined sections are not randomly chosen. Further research of this algorithm is needed if there exist obfuscators that use outlining randomly. However, all obfuscators that use the outline obfuscation technique without randomness, can be reversed using our algorithm.

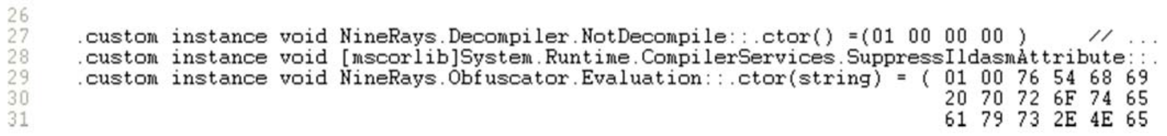
6.3 ILDASM protection

Spices .NET managed to crash the ILDASM disassembler as described in section 4.2.1. The only way to force this crash is to add some extra code into the obfuscated CIL code so that the ILDASM can not decompile the code properly, as described in section 2.3. There is no known (to the best of our knowledge) technique for reversing this kind of obfuscation. The only way to remove this protection is to find another decompiling tool which can extract the CIL code from the executable.

The Salamander .NET Obfuscator came in a package when it was downloaded as described in section 3.3.1, called the Remotesoft .NET Explorer. This explorer also contains a disassembler so any executables can be loaded in order to view the CIL code. The Remotesoft .NET Explorer did not encounter any problem when decompiling the CIL code from the code examples. It was possible to use the .NET Reflector to obtain the CIL code, but it was easier to obtain the CIL code from the Remotesoft .NET Explorer, because it does not split the CIL code into so many different parts as the .NET Reflector does.

Hence we were able to retrieve the CIL code which could not be viewed in ILDASM. The full CIL code can be found on the CD, which was obtained by extracting all the CIL code from each class involved in the obfuscated executables and putting them together in a new file.

Figure 6.4 shows the CIL code which caused the ILDASM to crash:



```
26
27    .custom instance void NineRays.Decompiler.NotDecompile::.ctor() =(01 00 00 00 )     // ...
28    .custom instance void [mscorlib]System.Runtime.CompilerServices.SuppressIldasmAttribute::.
29    .custom instance void NineRays.Obfuscator.Evaluation::.ctor(string) = ( 01 00 76 54 68 69
30                                                                            20 70 72 6F 74 65
31                                                                            61 79 73 2E 4E 65
```

Figure 6.4: SuppressIldasmAttribute

On line 28 in the figure above one can see a system call:

`System.Runtime.CompilerServices.SuppressIldasmAttribute`

This line is the reason that the ILDASM can not disassemble the CIL code.

The SuppressIldasmAttribute is a class in the .NET framework in the System.Runtime.CompilerService library [26]. It is a class that specifies that if it is used, it will prevent the ILDASM from disassembling the code where it is applied. The documentation also states that it does not prevent the assembly from being viewed by reflectors.

If one is able to retrieve the CIL code from an obfuscated executable, a general search can be made to see if the SuppressIldasmAttribute is used in the assemblies and thus crashes ILDASM. One can therefore create a pattern recognition technique as described in section 2.4 to reverse the protection towards ILDASM if this method is used.

The protection against decompilers featured in DotFuscator is described together with the algorithms for removing the control flow flattening effect in the next section.

6.4 Basic control flow flattening

All the results in section 4.3 showed that there were two of the options in the DotFuscator which could be viewed in the CIL code after the simulations. The different findings showed that if the control flow option was set to low or high, it did in fact alter the control flow of the program. The low option's changes were only minor compared to the high option – i.e. if it is possible to reverse engineering the changes made in the high option, it will also undo the alteration made with low option, since the high option also contains all the alterations from the low option.

In section 4.3 it was shown briefly that DotFuscator rewrites the content of a method into a switch case when obfuscating with the high option. This is in line with the basic control flow flattening technique described in section 2.2.2.2. It also adds some extra lines of code which only seems to have the purpose of confusing a decompiler/reflector (see section 2.3).

Our attempt to reverse the effects of the obfuscator is performed in two parts. First of all there are some instructions that can be removed from the CIL code. It is not any problem to obtain the CIL code from the obfuscations performed by DotFuscator, since it does not have any protection against ILDASM. The instructions that DotFuscator inserts for protection against decompilation is also included in this first algorithm.

The second part of the reversing technique is performed on the C# level and not on CIL code. It is easier to describe an algorithm on how to reverse the effects of the higher level, but it is of course possible to implement the same algorithm on the CIL level. It is, however, a lot more difficult to implement the algorithm on the CIL level, since the structure of the CIL code makes it a difficult to resolve.

Changing the CIL code

In the obfuscated CIL code, there are several lines which can be removed. The removal of these lines will also make the job of reversing the C# code easier, because it will make the flow within the switch easier to read. The algorithm described here is based upon the reversing technique that *Collberg et.al.* [11] call pattern recognition and is briefly explained in section 2.4. This technique can be used since the simulations from DotFuscator showed that it is the same code which is inserted in the obfuscated CIL code everytime.

The following algorithms are designed to reverse the decompiler protection and the control flow flattening technique used by DotFuscator. The first algorithm will use pattern recognition to find specific lined that the DotFuscator inserts. The second algorithm will reverse the switch statement into loop statements based on dataflow analysis.

The first algorithm relies on functionality which goes through the complete CIL code of a program and maps out the different methods. All the obfuscators used in the simulation for this thesis obfuscated each method separately, so the algorithm will therefore expect one method as input. The class/function will then be able to search through the lines of CIL code after the regular expressions (reg.ex.) which corresponds to the patterns found in the simulations of DotFuscator. The variable '**`$1`**' implies that it is a variable named 1, which stores the part of the reg.ex.

(1) Search through the method after labels matching the following expression:

```
”IL_$1: ldc.i4.1
 IL_$2: br.s *
 \n
 IL_$3: ldc.i4.0
 IL_$4: br.s *
 \n
 IL_$5: brfalse.s IL_$6”
```

(2) Search through the method after labels which contains the variables **`$1`** to **`$5`**. The search can for example look like “**`IL_*: * IL_$1`**” when searching for **`$1`**. If this search returns lines the **`$1`** needs to be replaced with **`$6`** to ensure that the flow of the program continues. If these lines where just removed the CIL code could not have been compiled.
(3) Search through the method after the declaration of the switch. This search can look like ”**`IL_*: switch (`**“. If this is found in the method;
(4) Then search through the each line of the switch and see if the label match **`$1-$5`** until the line equals ‘**`)`**’. If a match is found, the line needs to be replaced with the **`$6`**
(5) The method can look like this

```
while( line != ‘)’ ){
    if ( line == ‘IL_$1’ || … || line == ‘IL_$6’)
       then line = “IL_$6”
}
```

(6) thereafter the lines which matched the reg.ex in (1) can be deleted from the method

After this, the different cases in the switch need to be changed, so the flow within all the cases will be the same:

(1) Again find the switch declaration using ”**`IL_*: switch (`**“
(2) Store all the labels found in the switch in a list, here called **`caseList[]`**
(3) Go through the **`caseList[]`** and for each entry search through the method to find the label corresponding to the entry. If another jump is at this label, then follow the jump until the instruction is not a jump. When it is not a jump replace the entry in **`caseList[]`** with the label of that instruction.
(4) This method can be viewed this way:

```
Foreach (entry in caseList){
      Foreach(line in method){
            $1 = caseList[entry];
            while(line == “IL_$1: br.s $2”) {
                  line = nextLine();
                  $1 = $2;
            }
            caseList[entry] == $1;
       }
```

(5) All the lines between the original entry and the label found that is not a jump can thereafter be deleted.

Reversing control flow flattening for loops

When the algorithms above have finished, the CIL code can now be decompiled into C# language. The algorithm proposed for retrieving the code from the C# files was tested on single methods presented in the .NET Reflector, after the CIL code was altered. The following algorithm is also meant to be used on a single method at a time. This algorithm is limited; it only rewrites methods which contains loops (while and for loops). If the loops contains any if-else statements it only can resolve these if they are shown in full (does not contain any branching within the switch case).

As described in section 2.2.2.2, 2.4 and also in [45], basic static analyses can be used to help the reverse engineering of the control flow flattening. By using data flow analyzing it is possible to map the control flow between the different cases in the switch. This is not difficult, since the constant values that are assigned to the dispatcher variable are available within the different switch cases.

The following algorithm relies on input from a class/function to describe how the control flow of the switch looks like. The class must also define the entry and exit points of each switch case within one method. By examining these entries and exits it is possible to map which cases that are executed before and after a particular switch case. It will also show the exit from the loop, since only one case will have two exits and neither is to the previous executed switch case. This was analyzed manually when writing this algorithm, but this is an area that should be automated in future research.

The algorithm is as follows;

(1) Search through the method to see that there actually is a switch present in the C# code. If there are no switches the C# code has not been altered and there is no need for following this algorithm further. If the switch is found then store the variable that the switch is controlled by as switchVariable and also store the label where the switch starts as switchLabel. Create a list called caseVisited[] and create a empty file which will contain the rewritten method called rewrittenFile

(2) Read the name and the declarations first in the method and write these to the rewritten file. If:
   - a label follows the first delclarations or follows directly after the switch then go to (3)
   - there is a default case found in the switch. Write any content of the switch to the rewrittenFile except a given switch value or the goto statement before continuing then continue to (4)
   - there is a goto statement right after the switch go to (5)

(3) Read the content of the label. Write the content to the rewritten file except the value given to the switchVariable and the goto statement (if any).
   - If it contains a goto statement see if it contains the switchLabel.
      - If it does then go to the switch case that corresponds to the value given to the switch variable. Thereafter continue to (6)

        - If is does not have the switchLabel as goto then find the corresponding label in the method. Thereafter continue to (3)
    - If it does not contain a goto statement, go to the corresponding case in the switchVaraible and continue from (6)

(4) See if the default case's goto statement is the switchLabel.
    - If it does then go to the switch case corresponding to the given value and continue from (6)
    - If it does not then find the corresponding label in the method and continue to (3)

(5) See if the goto statement corresponds to the switchLabel.
    - If it does then go to the switch case corresponding to the given value and continue from (6)
    - If it does not then find the corresponding label in the method and continue to (3)

(6) See if the current case value is stored in the caseVisited[]
    - If no then store the value and continue
    - If yes jump to (10)

(7) Check if the case is empty.
    - If not empty, continue.
    - If empty then search for the next case that is not empty and go to this case. Then start again at (6)

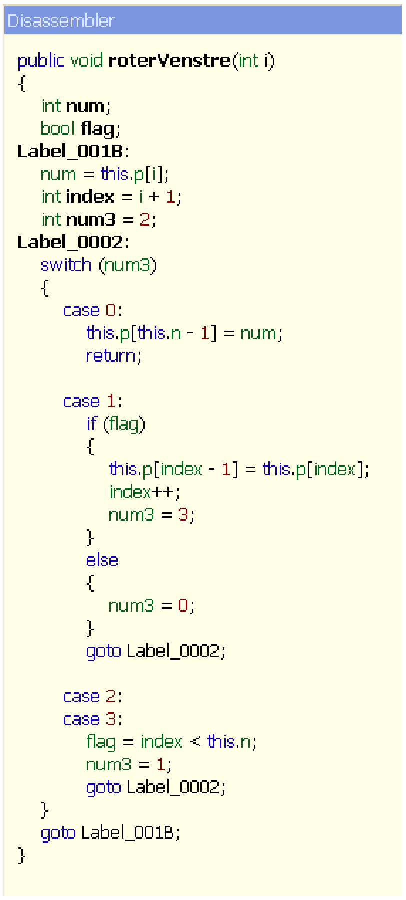
(8) See if this case has one or two entries.
    - If it has two entries then write the content of the case into the rewrittenFile except the switch value and the goto statement. Then go to the next case and continue from (9)
    - If one entry then write the content from the case into the rewrittenFile except the switch value and goto statement (if any).
        - If there is a *return* statement then quit;
        - If a goto statement check if it corresponds to the switchLabel.
            - If it does then follow the switch variable to the next case and start again at (6)
            - If it does not then find the corresponding label and go to (3)

(9) Write a while() with the if variable as condition into the same file. Write the rest of content of the if statement as the body of the while loop minus the value given to the switch value and the goto statement. Thereafter check if the goto statement is the switchLabel.
    - If it is then
        - See if the switch value is stored in the caseVisited[]:
            - If not then go to the next case and start again at (6)
            - If yes then goto (10)
    - If it is not then find the corresponding label in the method an continue from (3)

(10) Check if this case has one or two exits from the control flow chart.

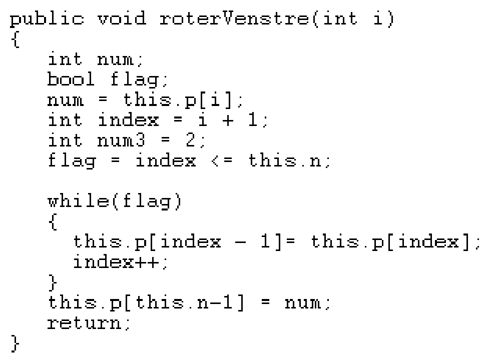
- If one exit then write a ‘}’ and content of the case except the switch value and goto statement. See if the goto statement is the switchLabel
    - If it is then go to the next case and start again from (6)
    - If it is not then find the corresponding label and continue from (3)
- If there are two exits then write a ‘}’. Then write the content of the else statement except the given switch value and goto statement. Check if the goto statement corresponds to the switchLabel
    - If it does then go to the next case and continue from (6)
    - If it is not then find the corresponding label and continue from (3)

The following Figures 6.5 and 6.6 represents a method in its obfuscated version and the corresponding method after our reversed algorithm has been applied to it. This method is one of the many in the Perm class in code example four. The control flow graph showed Figure 6.7 was used during the algorithm.


Disassembler

```
public void roterVenstre(int i)
{
    int num;
    bool flag;
Label_001B:
    num = this.p[i];
    int index = i + 1;
    int num3 = 2;
Label_0002:
    switch (num3)
    {
        case 0:
            this.p[this.n - 1] = num;
            return;

        case 1:
            if (flag)
            {
                this.p[index - 1] = this.p[index];
                index++;
                num3 = 3;
            }
            else
            {
                num3 = 0;
            }
            goto Label_0002;

        case 2:
        case 3:
            flag = index < this.n;
            num3 = 1;
            goto Label_0002;
    }
    goto Label_001B;
}
```

Figure 6.5: Obfuscated version



```
public void roterVenstre(int i)
{
   int num;
   bool flag;
   num = this.p[i];
   int index = i + 1;
   int num3 = 2;
   flag = index <= this.n;

   while(flag)
   {
     this.p[index - 1]= this.p[index];
     index++;
   }
   this.p[this.n-1] = num;
   return;
}
```

Figure 6.6: The same method reversed

Notice that this algorithm will transform every switch into loops in order to make the program easier to understand. As discussed in section 2.3 there is a possibility that this alters original code also, but it is although unlikely that the obfuscated code given to the algorithm contains similar switch statements, since the original source code is normally optimized.

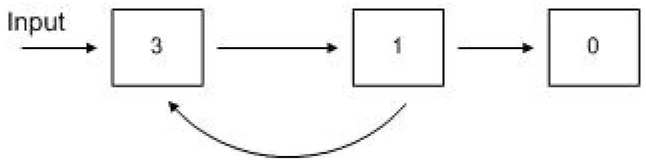


Figure 6.7: Control flow graph

7 Conclusion

In this thesis three questions has been addressed regarding obfuscating on the .NET platform:

- Which obfuscation techniques are one-way/not reversible?
- Which obfuscation techniques are vulnerable to reverse engineer?
- Is it possible to reverse engineer these obfuscation techniques in practice?

Our research presents numerous simulation results and analysis of these in Chapter 4 and 5. The results showed that there were three main types of obfuscation techniques used in the three obfuscators that were tested:

- ❖ renaming of fields
- ❖ protection against disassemblers and decompilers
- ❖ control flow obfuscation

Two different forms of control flow obfuscation were found: Outlining obfuscation and basic control flow flattening obfuscation. In section 6.1 these were classified based on the reversing theory in section 2.4, and this showed that renaming of fields was the only one-way technique found, and that both control flow techniques and also the disassembly protection feature were reversible.

The protection, outlining and control flow flattening obfuscation techniques were presented in section 6.2-6.4 respectively. An approach to reverse engineering each of the techniques was proposed, with background in the different methods explained in section 2.4, and a algorithm was created for all three techniques. These algorithms were based on pattern recognition and also data flow analysis. All three obfuscation techniques were successfully reversed, as shown with the examples.

This thesis has presented an overview of the techniques used in obfuscators on the .NET platform in general. It was surprising that so few techniques were implemented compared to the amount of techniques that has been proposed in research literature. However, most of the research techniques have been tested on the Java platform and it looks like the obfuscators on .NET are not yet as efficient as the Java obfuscators. The impression that .NET obfuscators are not as developed was also supported by the fact that there seemed to be obvious bugs in all the three obfuscators that were tested.

It was also surprising that the implementation of the control flow flattening technique did not include the two enhancements suggested in section 2.2.2.3 and 2.2.2.4. These would have made the reverse engineering much harder and the algorithm in 6.4 would not have been able to remove all of the obfuscation's modifications. Another surprising finding was that none of the obfuscator's obfuscated code examples which did not contain control statements. Using for example opaque predicates to alter the control flow of these blocks would have made the program harder to decompile and understand. It is also worth noting that if one obfuscator had included the techniques from the two others, the resilience would have been stronger towards reverse engineering, even though a combination of the purposed algorithms in Chapter 6 could have been used for reverse engineering.

The conclusion is therefore that there is still a lot of development needed for obfuscators on the .NET platform to implement all the obfuscation techniques that are available in the current research. Today’s obfuscators are also clearly vulnerable in terms of reverse engineering, like shown in Chapter 6.

8 Future Work

The algorithms presented in chapter 6 could have been implemented in a deobfuscation tool for the .NET platform. To our knowledge there has not been implemented a automated deobfuscation tools. This would require an implementation of the data flow analysis and pattern recognition functionality that these algorithms are based on. The pattern recognition should also contain a database of all known changes performed in CIL code by obfuscators.

In this thesis only three obfuscators was tested due to time limitations. Even though the market leading obfuscation tool (Obfuscator) was tested, more obfuscation tools on the .NET platform should be tested before concluding the efficiency of the obfuscators. A survey on which obfuscators that are actually used for software protection in the industry could also be useful to map out which features that vendors wants.

More general research should also be done as to combining the different methods that are being used in today's obfuscators. The impression from this thesis is that only some methods are implemented in each tool, and it would be interesting to see the results if more of the obfuscation techniques were combined into one single tool.

There has been a proposal in research that a parser can be used to resolve the renaming of fields, and rename them into more understandable and unique names. This should be investigated further to check if one can create an algorithm for reversing the obfuscated names, and thus making the code more understandable during reverse engineering.

References

1. 9Rays.NET, *FAQ Spices .NET Obfuscator.* http://www.9rays.net/Products/Spices.Net/FAQ.aspx#l2.
2. 9Rays.NET, *Spices .NET Obfuscator - customer list.* http://www.9rays.net/products/Spices.Net/Customers.aspx.
3. 9Rays.NET, *Spices .NET Obufscator.* http://www.9rays.net/products/Spices.Obfuscator/.
4. L. Badger, L. D'Anna, D. Kilpatrick, B. Matt, A. Reisse and T. Van Vleck, *Self-proteckting mobile agents obfuscation techniques evaluation report.* Technical Report #01-036, NAI Labs, March 2002
5. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan and K. Yang, *On the (Im)possibility of Obfuscating Programs.* Proc. 21th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '01), 2001 http://www.aladdin.cs.cmu.edu/papers/pdfs/y2001/obfuscators.pdf.
6. Business Softeware Alliance (BSA), *Global Study 2006.* http://w3.bsa.org/globalstudy/.
7. S. Chandrasekharan, S. Debray and C. Collberg, *Deobfuscation: Improving Reverse Engineering of Obfuscated Code.*
8. W. Cho, I. Lee and S. Park, *Against intelligent tampering: Software tamper resitance by extended control flow obfuscation.* Proc. World Multiconference on Systems, Cybernetics, and Informatics (SCI'2001), 2001
9. M. Christodorescu and S. Jha, *Static Analysis of Executables to Detect Malicious Patterns.* Proceedings of the 12th USENIX Security Symposium (Security'03), August 2003 http://www.cs.wisc.edu/wisa/papers/security03/cj03.html.
10. C. Collberg, C. Thomborson and D. Law, *Manufacturing Cheap, Resilient and Stealthy Opaque Contructors.* Principles of Programming Languages 1998 (POPL'98), January 1998 http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow98a/A4.pdf.
11. C. Collberg, C. Thomborson and D. Law, *A Taxonomy of Obfuscating Transformations.* Technical Report #148, The Department of Computer Science, University of Auckland, July 1997
12. C. Collberg and C. Thomborson, *Software watermarking: Models and dynamic embeddings.* Proc. 26th ACM Synopsium on Principles of Programming Languages, January 1999
13. C. Collberg and C. Thomborson, *Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection.* Technical Report #170, The Department of Computer Science, University of Arizona, February 2000
14. L. Cullen and S. Debray, *Obfuscation of Executable Code to Improce Resistance to Static Disassembly.* CCS'03, 2003
15. S. Drape, *Obfuscation of Abstract Data-Types*. 2004, University of Oxford.
16. ECMA, *ECMA-335.* http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf.

17. E. Eilam, *Reversing: Secrets of Reverse Engineering*. 2005: Wiley Publishing.
18. ISO, *ISO/IEC 23270:2003 C# Standard*. http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER= 36768.
19. ISO, *ISO/IEC 23271:2003 C# Standard*. http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=36769.
20. R. Jain, *The Art of Computer Systems Performance Analysis*. 1991
21. Microsoft, *ILASM*. http://msdn2.microsoft.com/en-us/library/496e4ekx(VS.80).aspx.
22. Microsoft, *Microsoft .NET Framework - JIT compiler*. http://msdn2.microsoft.com/en-us/library/ht8ecch6(VS.71).aspx.
23. Microsoft, *MSIL Disassembler - ILDASM*. http://msdn2.microsoft.com/en-us/library/f7dy01k1(VS.80).aspx.
24. Microsoft, *.NET Framework*. http://msdn2.microsoft.com/en-us/library/ms644566.aspx.
25. Microsoft, *Obfuscation tools for C#*. http://msdn2.microsoft.com/en-us/vcsharp/aa336818.aspx#obfuscators.
26. Microsoft, *The SupressIldasmAttribute Class*. http://msdn2.microsoft.com/en-us/library/system.runtime.compilerservices.suppressildasmattribute.aspx.
27. Microsoft, *Visual Studio*. http://msdn2.microsoft.com/en-us/vstudio/default.aspx.
28. Microsoft, *Visual Studio, DotFuscator Community Edition*. http://msdn2.microsoft.com/en-us/library/ms227240(VS.80).aspx.
29. C# Online .NET, *CIL instruction Set*. http://en.csharp-online.net/CIL_Instruction_Set.
30. F. Nielson, H. Riis Nielson and C. Hankin, *Principles of Program Analysis*. 2005
31. Linux MAN Page, *diff method*. http://linux.die.net/man/1/diff.
32. Remotesoft, *Salamander .NET Obfuscator*. http://www.remotesoft.com/salamander/obfuscator.html.
33. Remotesoft, *Salamander .NET Obfuscator - trial specifications*. http://www.remotesoft.com/salamander/obfuscator/try.html.
34. Remotesoft, *User Manual Salamander .NET Obfuscator*. http://www.remotesoft.com/salamander/obfuscator/manual/index.html.
35. Lutz Roeder, *Homepage*. http://www.aisto.com/roeder/.
36. Lutz Roeder, *Lutz Roeder's .NET Reflector*. http://www.aisto.com/roeder/DotNet/.
37. Sun Developer Networdk (SDN), *Java Bytecode*. http://java.sun.com/docs/white/langenv/Neutral.doc1.html#402.
38. PreEmtive Solutions, *DotFuscator*. http://www.preemptive.com/products/dotfuscator/.
39. PreEmtive Solutions, *DotFuscator FAQ*. http://www.preemptive.com/products/dotfuscator/FAQ.html#break ildasm1.
40. PreEmtive Solutions, *DotFuscator user manual*. http://www.preemptive.com/products/dotfuscator/manuals/userguide.pdf.
41. PreEmtive Solutions, *Features of DotFuscator* http://www.preemptive.com/products/dotfuscator/Features.html.

42. W. Stallings, *Cryptography and Network Security - Principals and Practice*. Third edition ed. 2003: Prentice Hall, Pearson Education Inc.
43. T. Ogiso, Y. Sakabe, M. Soshi and A. Miyaji, *Software obfuscation on a theoretical basis and its implementation.* IEEE Trans. Fundamentals, January 2003
44. Paul Tyma, *The New Obfuscation.* 2004 http://today.java.net/pub/a/today/2004/10/22/obfuscation.html.
45. S. K. Udupa, S. K. Debray and M. Madou, *Deobfuscation - Reverse Engineering Obfuscated Code.* Proc. 12th Working Conference on Reverse Engineering (WCRE'05), 2005
46. C. Wang, J. Davidson, J. Hill and J. Knight, *Protection of software-based suvivability mechanism.* Proc. International Conference of Dependable Systems and Networks, July 2001
47. Wikipedia, *Code Coverage.* http://en.wikipedia.org/wiki/Code_coverage.
48. Wikipedia, *Common Language Infrastructure (CLI).* http://en.wikipedia.org/wiki/Common_Language_Infrastructure.
49. Wikipedia, *Eight Queen Puzzle.* http://en.wikipedia.org/wiki/Eight_queens_puzzle.
50. Wikipedia, *Method Overloading.* http://en.wikipedia.org/wiki/Method_overloading.
51. Wikipedia, *Reverse Engineering.* http://en.wikipedia.org/wiki/Reverse_engineering.
52. G. Wroblewski, *General Method of Program Code Obfuscation*, in *Institute of Engineering Cybernetics*. 2002, Wroclaw University of Technology.

* Reference from Wikipedia is subject to change, and is therefore used for detailed explanations and not scientific evidence.