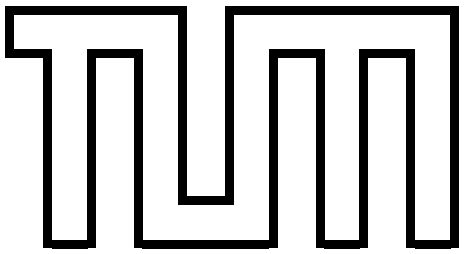
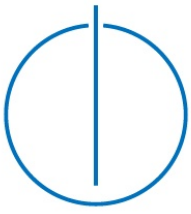


Technische Universität München

Software Protection by Virtualization Obfuscation

User's Manual



Contents

1. Tool Configuration **1**
1.1. Obfuscator Architecture 1
1.1.1. Modularity 1
1.1.2. Method Annotation 2
1.2. Virtualization Algorithm 3
1.2.1. Virtualization Transformations 4

2. Obfuscation Settings **15**
2.0.2. Interpreter Level 15
2.0.3. Randomization Points 19
2.0.4. Default Operation 22
2.0.5. Junk Code 22
2.0.6. Maximum Operands and Invocations 25
2.0.7. Readable and Refactor Mode 25

3. Obfuscator Limitations **27**
3.0.8. Interpreter 27
3.0.9. Class Attributes 27
3.0.10. Try/Catch Statements 28
3.0.11. Compiler 28

Bibliography **31**

List of Abbreviations **39**

List of Figures **41**

List of Tables **43**

List of Listings **45**

A. Appendix **47**
A.1. Levenshtein distance 47
A.2. Resilience 48
A.2.1. Factorial Recursive 48
A.2.2. Resilience trend 49
A.3. Quick Sort 50
A.3.1. Source Code 50
A.3.2. Runtime 52
A.4. Binary Search 54
A.4.1. Source code 54
A.4.2. Runtime 55

Chapter 1

Tool Configuration

1.1. Obfuscator Architecture

In this section we will go into some of the details of the architecture of the tool and how it was implemented. The requirements were to have a tool which is easily extensible in the future with other types of obfuscations and to have a tool which is able to obfuscate any fragment of C# source code.

One of the requirements of the obfuscation tool is that it should be able to work on any given C# program. C# features and source code samples depend on the .NET framework used. At the moment of the completion of the thesis, Jungheinrich's software components were written with .NET 2.0 framework but in the near future they will be built on .NET 4.5. Therefore the source code samples presented used only features supported by .NET 2.0. For the implementation of the tool we have used Visual Studio 2015 with the .NET Compiler Platform ("Roslyn") [54] extension which enables us to easily manipulate C# source code.

1.1.1. Modularity

In order to facilitate the extension of the tool we have used a component based architecture [17]. This architectural style makes it easy to add in the future additional obfuscation techniques.

The extensibility of the tool has been achieved by making the virtualization algorithm a component of the tool. Roslyn project enables us to load a Visual Studio solution, i.e. `.sln` file and extract the syntax tree of the source code. The virtualization algorithm receives the syntax tree as an input and returns a modified syntax tree.

In the future, other obfuscation techniques can be added easily to the tool as other components by using Roslyn's API [54]. These new components will receive as input the syntax tree of the source code, virtualized or not, and return an updated syntax tree, which again can serve as input for another component.

The virtualization obfuscation tool can be run automatically in a build script by specifying as input the project's Visual Studio solution file, i.e. `.sln` file, to be obfuscated and the specific obfuscation settings for that project. The tool will modify the existing source code without creating any additional files or modifying existing dependencies.

1.1.2. Method Annotation

We select the methods to be obfuscated by annotating them with an obfuscation attribute. Here we also specify if we want a class interpreter or a method interpreter. The interpreter level is specified in the annotation's `Feature` attribute as a `string` value separated by semi-column. The selection of the `class static` or `class instance` is done automatically by the tool. Please refer to listing 1.1 for an example.

Listing 1.1: Virtualization annotation

```
1         //method interpreter
2         [Obfuscation(Exclude = false, Feature = "virtualization; method;")]
3         public long FactorialIterative_1(int num)
4         { ... }
5
6         //class interpreter
7         [Obfuscation(Exclude = false, Feature = "virtualization; class;")]
8         public long FactorialIterative_2(int num)
9         { ... }
```

In this chapter we will go into the details of the design and implementation of the virtualization obfuscation tool. We begin with a motivation for the choice of the layer of abstraction where we applied the obfuscation. Then we proceed with the details of the virtualization algorithm we have used for our implementation of the tool. The obfuscation tool features a number of obfuscation settings in order to mitigate reverse engineering. These settings are presented later in the chapter. Next, the architecture of the virtualization tool is presented as a free alternative to commercial products. We conclude the chapter with the limitations of the tool and the methodology used during development to ensure observation equivalence between the original and the obfuscated programs. The design

choices will be motivated in the following sections.

Source Code

The second option to apply the obfuscation is directly at source code level, right before compilation. The source code being available, modifying it right before compilation makes it a more natural choice.

Another reason for modifying the source code is the support offered by Microsoft for .NET framework. The Roslyn project [54] is the open source version of the compiler used also for C# applications which is also included in Visual Studio 2015. Roslyn project provides a syntax tree view of the source code and makes it much easier to manipulate source code.

Finally, with the use of Roslyn project and Visual Studio 2015 it is possible to pack the obfuscation tool as a Visual Studio extension. A Visual Studio extension can provide an integrated view of the obfuscation process into the development process by adding context menu to obfuscate source code elements, i.e. methods, classes.

1.2. Virtualization Algorithm

In [45] László and Kiss presented a step by step algorithm and a prototype implementation of applying control-flow obfuscation to C++ programs. The algorithm assumes that the abstract syntax tree of the function to be obfuscated is available. It start with a preprocessing phase where various code transformations are applied, transformations such as moving variables at the beginning of the function to avoid visibility issues or possible variables renaming to avoid name conflicts. Then the algorithm traverses the syntax tree and generates the control-flow-flattening structure by processing step by step every statement of the preprocessed function. In a similar manner we have two main phases for the virtualization obfuscation process. These phases are `refactoring` and `virtualization` and are described in detail in the following subsections.

As a general structure of the source code of the interpreter, we have followed the basic ideas presented in **??**. An *instruction* is also called an *operation*, an indivisible unit of code which represents an executable code segment. A source code statement can contain multiple instructions, e.g. an object with multiple method invocations. We have used an `object` array which we call `DATA` to store all the variables, constants and any other data used in the execution of the original code. The other fundamental element of virtualization is the *bytecode*. For this we have

used an integer array which we call `CODE`. The bytecode contains the logic of the original program and it is stored in `CODE`. Each instruction supported by the virtualization interpreter is represented by an `OPCODE`. Opcodes are stored in the bytecode sequence alongside other elements. The pointer to the next instruction is the virtual program counter, `VPC` which is a variable of type `int`. The interpreter is composed of an infinite loop surrounding a `switch` construct. The key of a `switch case` represents an `OPCODE` of the bytecode. The `switch case` interprets a unique instruction pointed by the current position of the `VPC`. More details about the interpreter and other data structures will follow in the next subsections.

In section **??** we have presented Collberg et al.'s catalog of obfuscation transformations which can be employed in the building of an obfuscation tool [16]. The transformations types are lexical, control, data and preventive. Our virtualization algorithm employs all these types of code transformations in order to improve the resilience of the obfuscated program to deobfuscation. We have grouped these transformations in two sections: *refactoring* and *virtualization* transformations. Refactoring transformations are independent transformations which can be applied also in other contexts, while virtualization transformations are specific code transformations needed to achieve virtualization.

1.2.1. Virtualization Transformations

After the code transformations of the refactoring phase have been applied, we can proceed to the virtualization phase. The virtualization phase has the following basic steps:

Step 1. ***Select method body to virtualize.*** The obfuscator tool provides the option to virtualize a selection of critical code segments and not the entire program. The exact details of code selection are presented in section 1.1. Because we are dealing with object-oriented source code, any critical algorithm is always stored in method. Therefore, at the moment we consider only obfuscating the body of the methods.

Step 2. ***Virtualize constants, local arguments, parameters.*** represents data-type transformations (see Collberg's transformation's taxonomy). In this step we go through the code and select all elements which will be stored in the `DATA` object array of the virtualization.

Step 3. ***Process method's body statements.*** is the parsing of the statements which will become virtualized in the interpreter. Also at this step the bytecode which incorporates the logic of the original program is generated. The bytecode

will be stored in the `CODE` array.

Step 4. ***For compound structures go to previous step and virtualize the statement's body.*** This step is similar with step 3. The only difference is that it is only applied to the body of complex statements, e.g. `if`, `while`. We consider complex statements those statements which have a body, which limits the scope of other variables or statements. Examples are `if` statements, `while`, `try-catch`, `switch` and others. In the current version we deal only with `if` and `while` statements.

In the next paragraphs we will go into the details of virtualizing the body of a method. As a running example for all the steps involved in the virtualization we will use a simple *factorial iterative* implementation which can be found in listing 1.2. After the refactoring phase, it has the structure which can be seen in listing 1.3.

Listing 1.2: Factorial iterative

```
1  public long FactorialIterative(int num)
2  {
3      long result = 1;
4      if (num == 0)
5      {
6          return 1;
7      }
8      else
9      {
10         for (int i = 2; i <= num; i++)
11         {
12             result *= i;
13         }
14         return result;
15     }
16 }
```

Listing 1.3: Factorial iterative - refactored

```
1  public long FactorialIterative(int num)
2  {
3      long result = 1 + 0L;
4      bool var_ifCondition_0 = num == 0;
5      if (var_ifCondition_0)
6      {
7          return 1;
8      }
9      else
10     {
11         int var_forIndex_0 = 2;
12         bool var_whileCondition_0 = var_forIndex_0 <= num;
13         while (var_whileCondition_0)
14         {
15             result = result * var_forIndex_0;
```

```
16              var_forIndex_0 = var_forIndex_0 + 1;
17              var_whileCondition_0 = var_forIndex_0 <= num;
18          }
19
20          return result;
21      }
22  }
```

Virtual Data

Supposing that we have selected the method which implements factorial to obfuscate, we can proceed to step 2 of the virtualization phase. At this point, the local variables, method parameters and constants will all be stored in the `DATA` object array which is one of the key elements of virtualization obfuscation. We call an element of this array a *virtual data*.

In order to keep track of all these elements during virtualization, we have defined a class `VirtualData` which stores all the information that is needed to transform the original code into a virtualized one. The most important attributes of a VirtualData object are:

- Index - position in the `DATA` array
- Name - identifier of the data
- Type - e.g. `int`, `string`, `Car`
- Value - 3, "hello world", object instance
- Default value - random value

When we identify an element in the original source code that will become a *virtual data*, the first thing that we do is assign a position in the `DATA` array for that element. This will become the `index` position of the variable. We also give it a `name` to be able to track it easier later in the virtualization process, e.g. the identifier of a variable. Next information that is extracted is the `type` of that element which is needed in order to generate code that will cast an object to that specific type, e.g. `long`, `Driver`. Another key information needed is the `value` of that element before virtualization. This is represented by the source code expression statement, e.g. `MethodB() + Counter - 10`. Finally, if there are local variables which are not initialized, we can initialize them with a `default value` which is either predefined for every type or it can be a random value, e.g. `null`, `43254232`, `"sdfgaxcgd"`.

In the current implementation the DATA array is created and initialized at the beginning of the method's body. The size of the virtual data array must be large enough for all local variables, method parameters and other constants. The size can be set as a configuration parameter, i.e. MAX_DATA_KEY, and currently we use as a default value a random number between 4000 and 5000. We considered this to be a reasonable maximum number of local variables since the limitation for local variables in C# is 65536 [25, 59]. A complete list with all the configuration parameters can be found in section 2.

Constants, Local Variables and Method Parameters

The next transformation towards virtualization is to store the constants into DATA array. For each constant present in the original code, we assign a position in the DATA array. If there are constants which are repeated in the original code, then we store it only once in the data array. This makes the code more resilient to tampering because if the attacker modifies one constant value while trying to affect one statement or only a region of the code, he or she will end up affecting also other regions of protected code.

Once the constants are identified, assignment statements to DATA positions are generated for each constant with it's corresponding type. The next step is to replace each appearance of the constants in the original code with the values from the virtual data array.

Method parameters are virtualized in the same steps as constants. There is no need for additional initialization.

Local variables are virtualized in a similar manner with constants. The additional step is that if there are local variables which are not initialized when declared in the original code, then these variables will be initialized with a default random value. The next step, as in the case of constants, is to replace each appearance of the variable in the original code with the values from the virtual DATA array.

Applying this transformation step on listing 1.3 will result in the code that can be seen in listing 1.4. The mapping from the name of the variable to the position in the DATA array is only an internal structure of the obfuscator and will not appear in the output code. However, for the ease of explanation we have made this mapping explicit in the comments of the array initialization statements.

Listing 1.4: Factorial iterative - virtualized data

```
1 public long FactorialIterative(int num)
2 {
```

```
3         //Virtualization variables
4     object[] data = new object[4820];
5
6      //Data init
7     data[741]=num; //num
8     data[824]=1 ; //1 constant
9     data[213]=(long)0L; //0L constant
10    data[53]=0; //0 constant
11    data[2830]=(long)1; //1 constant
12    data[1193]=2; //2 constant
13    data[2807]=(long)-91L; //result
14    data[1341]=false; //var_ifCondition_0
15    data[2033]=15; //var_forIndex_0
16    data[2084]=false; //var_whileCondition_0
17    data[1492]=708; //jmpDestinationName_1492 constant
18    data[1520]=70; //if_GoTo_True_1520 constant
19    data[296]=184; //if_GoTo_False_296 constant
20    data[659]=512; //if_FalseBlockSize_Skip_659 constant
21    data[410]=645; //jmpWhileDestinationName_410 constant
22    data[3635]=70; //while_GoTo_True_3635 constant
23    data[1131]=317; //while_GoTo_False_1131 constant
24    data[2991]=-317; //while_FalseBlockSkip_2991 constant
25
26    data[2807] = (int) data[824] + (long) data[213];  // virtual operation key 9336
27    data[1341] = (int) data[741] == (int) data[53]; // virtual operation key 5420
28    if ((bool) data[1341]) // virtual operation key 4823
29    {
30        return (long) data[2830]; //virtual operation key 2635
31    }
32    else
33    {
34        data[2033] = (int) data[1193]; // virtual operation key 1931
35        data[2084] = (int) data[2033] <= (int) data[741]; // virtual operation key 3747
36        while ((bool) data[2084]) // virtual operation key 4823
37        {
38            data[2807] = (long) data[2807] * (int) data[2033]; // virtual operation key
                  2478
39            data[2033] = (int) data[2033] + (int) data[824]; // virtual operation key
                  1920
40            data[2084] = (int) data[2033] <= (int) data[741]; // virtual operation key
                  3747
41        }
42
43        return (long) data[2807]; //virtual operation key 2635
44    }
45 }
```

Because the mapping from the identifier of original program to its location in the DATA array is not explicit, we can consider this transformation of type *data transformation* because it hides the original data structure into the virtual data array.

Virtual Operation

Once step 2 of the virtualization phase is complete, we can proceed to the next step, step 3 and process the statements, some of which were updated in the previous step. We start with identifying and selecting all statements which will become virtualized in the interpreter. These statements will contain operations like assignment, addition, subtraction, method invocation, return statements, loop statements, conditional statements and others.

Virtualization provides us with more freedom by allowing us to design our own instruction set architecture. This freedom is used for two main goals: first to increase tamper-resistance and, then to introduce diversity [2]. Tamper-resistance is achieved by the use of a new instruction set architecture which forces the reverse engineer to learn the meaning of at least a few new `opcodes` before being able to change any part of the obfuscated code.

In order to be consistent with *virtual data*, we call the instructions of the new architecture a *virtual operation*. A virtual operation is the bytecode representation of the original instruction. The bytecode is stored in the `CODE` which is an int array. The virtualization interpreter will execute in each branch a virtual operation. For the virtualization algorithm we have also defined a `VirtualOperation` class which contains the following important attributes:

- Key - the information by which the interpreter will know how to translate and execute the virtualized operation. It is the key for the `case` sections of the interpreter.
- List<Virtual Data> - the list of operands of the new instruction.
- Size - how many entries in the bytecode array that instruction requires.
- Prefix Size - number of positions available for operands before the operation key.
- Postfix Size - number of positions available for operands after the operation key.
- Offset - random number to simulate variable size.
- Frequency - represents how many times this instruction occurred in the original program.

Each `VirtualOperation` will have a predefined `size` which represents the positions in the bytecode. For example, we can allocate 50 positions for each virtual operation, value which can be configured as a setting of the obfuscator. This means that a virtual operation can have at most 49 operands and 1 instruction key. For example, the following statement, line 40 from the previous example, has three operands and needs one position in the bytecode for the interpreter key and another three position for the position of the data:

```
1         data[2084] = (int) data[2033] <= (int) data[741];
```

The other fundamental virtualization element is the virtual program counter, `VPC`. This is used to point to the next instruction but is also used to point to the operands of the virtual operation. This is done in the following way.

The next virtualization transformation requires the following. Each operand location in the `DATA` array will be virtualized by storing its position in the bytecode, the `CODE` int array. The bytecode position of the operands is assigned in a random way alongside the virtual operation key. Two other properties of `VirtualOperation` are `Postfix` and `Prefix` size, which can also be configured as as a setting of the obfuscator, i.e. `INSTRUCTION_PREFIX` and `INSTRUCTION_POSTFIX`. `Postfix` size represents how many position after the current `VPC` position we can use to allocate values in the bytecode representation. `Prefix` size specifies how many positions we can use before `VPC` positions for the same purpose. The virtual instruction key is always at the position pointed by the virtual program counter when entering the interpreter section and it always occupies one position in the bytecode.

Therefore, an initial bytecode sequence of a virtual operation consists of operation key, always one position and surrounded by operands, then a number of operands on random positions and the remaining empty positions, as seen in figure 1.1.

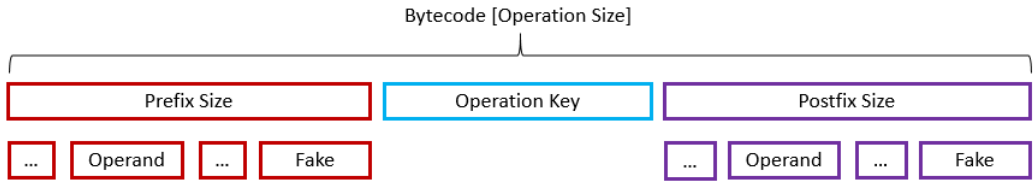


Figure 1.1.: Initial bytecode sequence

To access the operand we will have to access first the bytecode and extract the data index from the code array. The bytecode is accessed by offsetting the

virtual program position in the allocated space, i.e. the size, of the virtual operation. For example, after this transformation, the previous statement, line 40 from listing 1.4 will have the following structure with the corresponding bytecode representation.

```
1 //local variables, constants are virtualized
2         data[code[vpc+3]] = (int) data[code[vpc+(-5)]] <=
             (int) data[code[vpc+1]];
3
4 //bytecode representation of the while condition expression
5         code[468]=3747; //ExpressionStatement_6
6         code[486]=2084; //var_whileCondition_0
7         code[449]=2033; //var_forIndex_0
8         code[474]=741; //num
```

The numbers 3, -5, 1 which offset the `VPC` position are random numbers used to point to random positions in the bytecode. In the bytecode representation the value 3747 represents a `case` of the interpreter, while the values 2084, 2033 and 741 represent different position ins the `DATA` array.

Interpreter

The next step is to generate the interpreter. In order to do that we have to select the virtualized statements which will become the body of the interpreter. As we have already mentioned, the interpreter is an infinite loop which encloses a `switch-case` structure over the bytecode keys. Each interpreter `case` section represents the translation and execution of a virtual operation.

For this version of the obfuscator, each `case` section of the interpreter will process a unique virtual operation. In order to get the unique statements we will go over again the modified source code, modified by the previous steps, and we extract the statements. With the use of Roslyn it is possible to compare statements and check their equivalence. Two statements are equivalent if they have the same source code elements but without regarding aspects like comments or formatting differences.

At the end of the virtualization phase, the body of the obfuscated method will be replaced with the obfuscated body. In listing 1.5 we have presented our iterative factorial example with the entire obfuscation. The virtualized body starts with the initialization of the virtualization variables, `CODE`, `DATA`, `VPC` and then we see the initialization of data and bytecode arrays. We have added the com-

ments to ease the identification of the unique statements and their position in the interpreter.

Listing 1.5: Factorial iterative - complete virtualization

```
public long FactorialIterative(int num)
{
        //Virtualization variables
    int[] code = new int[100306];
    object[] data = new object[4820];
    int vpc = 99;

    //Data init
    data[741]=num; //num
    data[824]=1 ; //1 constant
    data[213]=(long)0L; //0L constant
    data[53]=0; //0 constant
    data[2830]=(long)1; //1 constant
    data[1193]=2; //2 constant
    data[2807]=(long)-91L; //result
    data[1341]=false; //var_ifCondition_0
    data[2033]=15; //var_forIndex_0
    data[2084]=false; //var_whileCondition_0
    data[1492]=708; //jmpDestinationName_1492 constant
    data[1520]=70; //if_GoTo_True_1520 constant
    data[296]=184; //if_GoTo_False_296 constant
    data[659]=512; //if_FalseBlockSize_Skip_659 constant
    data[410]=645; //jmpWhileDestinationName_410 constant
    data[3635]=70; //while_GoTo_True_3635 constant
    data[1131]=317; //while_GoTo_False_1131 constant
    data[2991]=-317; //while_FalseBlockSkip_2991 constant

    //Code init

    code[99]=9336; //ExpressionStatement_0 # ExpressionStatement_0
    code[96]=2807; //result
    code[90]=824; //1
    code[118]=213; //0L

    code[161]=5420; //ExpressionStatement_1 # ExpressionStatement_1
    code[174]=1341; //var_ifCondition_0
    code[184]=741; //num
    code[188]=53; //0

    code[221]=4823; //IfStatementSyntax_2 # IfStatementSyntax_2
    code[209]=1492; //jmpDestinationName_1492
    code[248]=1341; //var_ifCondition_0
    code[244]=1520; //if_GoTo_True_1520
    code[242]=296; //if_GoTo_False_296

    code[291]=2635; //ReturnStatement_3 # ReturnStatement_3
    code[303]=2830; //1

```

```
49      code[353]=2429; //ExpressionStatement_4 # ExpressionStatement_4
50      code[379]=659; //if_FalseBlockSize_Skip_659
51
52      code[405]=1931; //ExpressionStatement_5 # ExpressionStatement_5
53      code[398]=2033; //var_forIndex_0
54      code[401]=1193; //2
55
56      code[468]=3747; //ExpressionStatement_6 # ExpressionStatement_6
57      code[486]=2084; //var_whileCondition_0
58      code[449]=2033; //var_forIndex_0
59      code[474]=741; //num
60
61      code[538]=4823; //IfStatementSyntax_2 # WhileStatementSyntax_7
62      code[526]=410; //jmpWhileDestinationName_410
63      code[565]=2084; //var_whileCondition_0
64      code[561]=3635; //while_GoTo_True_3635
65      code[559]=1131; //while_GoTo_False_1131
66
67      code[608]=2478; //ExpressionStatement_8 # ExpressionStatement_8
68      code[622]=2807; //result
69      code[609]=2807; //result
70      code[634]=2033; //var_forIndex_0
71
72      code[673]=1920; //ExpressionStatement_9 # ExpressionStatement_9
73      code[662]=2033; //var_forIndex_0
74      code[655]=2033; //var_forIndex_0
75      code[692]=824; //1
76
77      code[733]=3747; //ExpressionStatement_6 # ExpressionStatement_10
78      code[751]=2084; //var_whileCondition_0
79      code[714]=2033; //var_forIndex_0
80      code[739]=741; //num
81
82      code[803]=2429; //ExpressionStatement_4 # ExpressionStatement_11
83      code[829]=2991; //while_FalseBlockSkip_2991
84
85      code[855]=2635; //ReturnStatement_3 # ReturnStatement_12
86      code[867]=2807; //result
87
88      while(true)
89      {
90          switch(code[vpc])
91          {
92                  case 4823: //frequency 2 IfStatementSyntax_2
93                          data[code[vpc+(-12)]] = (bool)data[code[vpc+(27)]] ?
                                (int)data[code[vpc+(23)]] : (int)data[code[vpc+(21)]];
94                          vpc+=(int)data[code[vpc+(-12)]];
95                          break;
96                  case 1920: //frequency 1 ExpressionStatement_9
97                          data[code[vpc+(-11)]]= (int)data[code[vpc+(-18)]]+
                                (int)data[code[vpc+(19)]];
98                          vpc+=60;
99                          break;
100                 case 1931: //frequency 1 ExpressionStatement_5
101                         data[code[vpc+(-7)]]= data[code[vpc+(-4)]];
102                         vpc+=63;
103                         break;
```

```
104                 case 2429: //frequency 2 ExpressionStatement_4
105                         vpc += (int)data[code[vpc+(26)]];
106                         vpc+=52;
107                         break;
108                 default: //frequency 0
109                         break;
110                 case 9336: //frequency 1 ExpressionStatement_0
111                         data[code[vpc+(-3)]]= (int)data[code[vpc+(-9)]]+
                                (long)data[code[vpc+(19)]];
112                         vpc+=62;
113                         break;
114                 case 2478: //frequency 1 ExpressionStatement_8
115                         data[code[vpc+(14)]]= (long)data[code[vpc+(1)]]*
                                (int)data[code[vpc+(26)]];
116                         vpc+=65;
117                         break;
118                 case 3747: //frequency 2 ExpressionStatement_6
119                         data[code[vpc+(18)]]= (int)data[code[vpc+(-19)]]<=
                                (int)data[code[vpc+(6)]];
120                         vpc+=70;
121                         break;
122                 case 5420: //frequency 1 ExpressionStatement_1
123                         data[code[vpc+(13)]]= (int)data[code[vpc+(23)]]==
                                (int)data[code[vpc+(27)]];
124                         vpc+=60;
125                         break;
126                 case 2635: //frequency 2 ReturnStatement_3
127                         return (long)data[code[vpc+(12)]];
128                         vpc+=62;
129         }
130     }
131
132     return 0;
```

By the use of the interpreter with a `switch-case` structure with a controlling variable we have also used a control-flow transformation in the virtualization algorithm. Therefore, so far the virtualization algorithm uses lexical, control-flow and data transformations. Lexical transformations are used in the refactoring phase, data transformations are done by storing local variables and constants in the `DATA` array. The final type of transformations, preventive transformations, will be mentioned in the next section.

Chapter 2

Obfuscation Settings

Commercial obfuscators like Themida [63], Agile.NET [1] and open source obfuscators like ConfuserEx [18] offer a number of options to the user to choose from. Some settings configurations can affect the number of obfuscation techniques that are applied, but other configurations can affect the strength or the depth of a particular obfuscation technique. For example, in renaming obfuscation one could specify if one wants to rename only the class names or also the methods, in control flow obfuscation if one wants to have a switch structure or a chained `if` statement, or in virtualization obfuscation if one wants to have only one interpreter or multiple interpreters.

In the next section we will present the obfuscation settings of the virtualization tool accompanying this thesis and also we will talk about some of its features which affect the potency of the obfuscation. A list of all settings than ca be configured in our virtualization tool can be found in table 2.1.

2.0.2. Interpreter Level

The first setting we address is the level of the interpreter. In other words, this refers to the position of the virtualization interpreter in the obfuscated source code. In the previous examples we have always put the code of the interpreter inside the obfuscated method.

For the interpreter we have the following three options:

1. Method
2. Class Instance
3. Class Static

Name	Default Value	Description
CODE_IDENTIFIER	code	the name of the bytecode (`CODE`) array variable
DATA_IDENTIFIER	data	the name of the `DATA` array variable
DEFAULT_MOST_FREQUENT_OPERATION	TRUE	move the most frequent operation to the default section of the interpreter
INSTRUCTION_POSTFIX	30	controls the number of positions allocated for operands after the operation key
INSTRUCTION_PREFIX	20	controls the number of positions allocated for operands before the operation key
INSTRUCTION_SIZE_OFFSET	20	controls the size offset of each instruction; simulates a random size for virtual operations
MAX_INVOCATIONS	1	limit the maximum number of invocations that are left in a statement without splitting it further into smaller statements
MAX_OPERANDS	2	limit the maximum number of operands that are left in a statement without splitting it further into smaller statements
MAX_JUNK_CODE	10	controls the maximum number of CODE and DATA positions filled with junk values
MAX_DATA_KEY	3999	controls how large is the DATA array
MAX_CODE_KEY	99999	controls how large is the CODE array
MIN_SWITCH_KEY	1000	controls the minimum range value of the operation key range, i.e. `case` key
MAX_SWITCH_KEY	9999	controls the maximum range value of the operation key range, i.e. `case` key
VPC_IDENTIFIER	vpc	the name of the virtual program counter (`VPC`) variable

Table 2.1.: Obfuscation tool settings

Method interpreter means that the interpreter code is inside the method. It has access to the local variables and the class attributes. For more details regarding class attributes and interpreters, please refer to section 3.0.9. For an example of method level interpreter please refer to listing 1.5.

Class instance interpreter means that the interpreter code is extracted and put into a separate method in the parent class of the obfuscated method. The reason for doing so is that now the interpreter can be extended to cover multiple obfuscated methods. In other words, we can have an interpreter for all virtualized methods of the class.

The class interpreter is defined in a private method. It has as input parameters the `VPC`, the `DATA` object array and the bytecode, `CODE` array. As a return value it returns an object which will be cast to the proper type in the method from where it is called. The class interpreter is invoked from every method that is virtualized and was marked to have a class interpreter. The interpreter is invoked as a call to a regular method, see listing 2.1.

Listing 2.1: Interpreter invocation

```
public long FactorialIterative(int num)
{
    //Virtualization variables
    int[] code = new int[100754];
    object[] data = new object[4475];
    int vpc = 24;
    //Data init
    data[273]=num;
        // ...
        //Code init
        code[24]=7887;
        // ...

    return (long)InstanceInterpreterVirtualization_Iterative_3054(vpc, data, code);

}
```

For example, let us consider the factorial method from listing 1.2 and the method ForSimple_Array found in listing **??**. By virtualizing them and we have only one interpreter at the class level. The interpreter will have elements from both methods but if there are equivalent statements between the methods, then these will appear only once in the interpreter sections; the interpreter has only unique case sections. Please refer to listing 2.2 for the class interpreter of these two methods. We have included only a fragment of the interpreter and we added also the frequency of each statement that is virtualized as a comment to the interpreter `case` structures. For example in the `case` 3105 we note the frequency to be 3. This is a simple assignment statement which occurred multiple times in

both of the virtualized statements. Also the `case` 9095 is part of the computation of the jump of a `while` or `if` structure. The jump computation occurred 3 times in total, one `while` in listing **??**, one `while` and one `if` in listing 1.2.

Listing 2.2: Class instance interpreter

```
private object
private object InstanceInterpreterVirtualization_Iterative_3054(int vpc, object[] data,
    int[] code)
{
    while (true)
    {
        switch (code[vpc])
        {
            case 9095:  //frequency 3 ExpressionStatement_4
                vpc += (int)data[code[vpc + (13)]];
                vpc += 63;
                break;
            case 7887:  //frequency 1 ExpressionStatement_0
                data[code[vpc + (22)]] = (int)data[code[vpc + (-6)]] +
                    (long)data[code[vpc + (20)]];
                vpc += 56;
                break;
            case 7902:  //frequency 2 ReturnStatement_3
                return (long)data[code[vpc + (3)]];
                vpc += 60;
            case 7779:  //frequency 1 ExpressionStatement_16
                data[code[vpc + (-15)]] = (ConsoleCalculator.Piston)
                    (((System.Collections.Generic.List<ConsoleCalculator.Piston>)
                    data[code[vpc + (9)]]).First());
                vpc += 64;
                break;
            case 8351:  //frequency 1 ExpressionStatement_24
                data[code[vpc + (4)]] = ((ConsoleCalculator.Piston)data[code[vpc +
                    (-19)]]).ToString();
                vpc += 70;
                break;
            // ...
            case 3105:  //frequency 3 ExpressionStatement_5
                data[code[vpc + (-16)]] = data[code[vpc + (-14)]];
                vpc += 52;
                break;
            case 6619:  //frequency 1 ExpressionStatement_23
                data[code[vpc + (8)]] =
                    (ConsoleCalculator.Piston)(((ConsoleCalculator.Engine)data[code[vpc
                    + (23)]]).GetPiston((int)data[code[vpc + (11)]] -
                    (int)data[code[vpc + (20)]]));
                vpc += 67;
                break;
            case 7369:  //frequency 2 ExpressionStatement_6
                data[code[vpc + (29)]] = (int)data[code[vpc + (23)]] <=
                    (int)data[code[vpc + (25)]];
                vpc += 67;
                break;
            default:  //frequency 0
                break;
```

```
42              case 6442:  //frequency 3 IfStatementSyntax_2
43                  data[code[vpc + (-15)]] = (bool)data[code[vpc + (17)]] ?
                        (int)data[code[vpc + (11)]] : (int)data[code[vpc + (24)]];
44                  vpc += (int)data[code[vpc + (-15)]];
45                  break;
46              case 8233:  //frequency 1 ReturnStatement_42
47                  return (string)data[code[vpc + (-6)]];
48                  vpc += 56;
49              case 4960:  //frequency 1 ExpressionStatement_22
50                  data[code[vpc + (29)]] =
                        ((System.Collections.Generic.List<ConsoleCalculator.Piston>)data[code[vpc
                        + (-19)]]).Count;
51                  vpc += 63;
52                  break;
53              // ...
54          }
55      }
56
57      return null;
58  }
```

The other type of interpreter is **Class static interpreter**. This is the same with the class instance interpreter with the only difference that the method where the interpreter code resides is a static method, and not an instance method. We have this additional option for the following reason. If we have the interpreter always in a static method, then it does not have access to instance attributes of an object. On the other hand, if we always have the interpreter in an instance method, and if the virtualized methods are all static, then we cannot invoke the interpreter from the virtualized method, you cannot call an instance method from a static method. Therefore we have these two options which are automatically selected by the obfuscator tool by the following simple rule: static methods will always have static interpreters, while instance methods will have always instance interpreters.

2.0.3. Randomization Points

Because diversity is a key enabler in minimizing the impact of an attack [2], we have used randomization at multiple points in the virtualization algorithm. This is a means to automatically add diversification to distributed copies of the original program. The points where we added randomization and which we will discuss next are:

- Operation key
- Operands position
- Operation size

- Virtual program counter
- Array initialization
- Switch sections

The *key of the operation* is a random number, which can be configured to be always between a minimum and a maximum value, i.e. `MIN_SWITCH_KEY` and `MAX_SWITCH_KEY`. This key will have to have a correspondent in the interpreter case sections. The operation key holds the mapping from the original statements to the obfuscated bytecode.

The next randomization point is the operands positions in the virtual operation's bytecode representation. We have already mentioned that there is a prefix size and a post fix size, see Figure 1.1. These properties are configured by modifying the values of `INSTRUCTION_PREFIX` and `INSTRUCTION_POSTFIX`. The position of the operation key is always the reference point to pre/post fix size. Rolles [52] mentioned the existence of interpreters that read the bytecode backwards for some instructions. By randomly distributing the operands with regard to operation key the bytecode is traversed in a random order with regard to the elements of a single operation. The operands positions are different for every unique statement.

```
1  switch(code[vpc])
2  {
3      case 4774: //frequency 1 WhileStatementSyntax_23
4          data[code[vpc+(-14)]]=(bool)data[code[vpc+(-13)]]?(int)data[code[vpc+(-6)]]:(int)data[code[vpc+(-9)]];
5          vpc+=(int)data[code[vpc+(-14)]];
6          break;
7      case 7385: //frequency 4 ExpressionStatement_0
8          data[code[vpc+(23)]]= (string)data[code[vpc+(-18)]]+ (int)data[code[vpc+(11)]];
9          vpc+=61;
10         break;
11     case 9872: //frequency 2 ExpressionStatement_21
12         data[code[vpc+(-4)]]= ReturnArg_Array((int)data[code[vpc+(-16)]]);
13         vpc+=52;
14         break;
15     case 7635: //frequency 2 ExpressionStatement_18
16         data[code[vpc+(9)]]= data[code[vpc+(25)]];
17         vpc+=63;
18         break;
19     default: //frequency 10 ExpressionStatement_2
20         data[code[vpc+(2)]]= (string)data[code[vpc+(4)]]+ (string)data[code[vpc+(20)]];
21         vpc+=69;
22         break;
23 }
```

Figure 2.1.: Virtual Operation Size

We have mentioned that the operation size is fixed. The operation size is equal with the `prefix` value plus the `postfix` value plus one for the operation key. In

order to compensate for that we add a virtual operation size offset for each unique statement, i.e. `INSTRUCTION_SIZE_OFFSET`. This means that in the bytecode array the virtual operations do not have predetermined starting points based on their order like first on position 0, second starts on position 50, third starts on position 100 and so on. The next instruction's starting point depends on the previous instruction's size and random offset. Please refer to figure 2.1 where in the red boxes we have highlighted the different starting positions based on the unique statements.

The virtual program counter is initialized with a different starting point for every virtualized method. This means that, for example, the initial of `VPC` can be 10 or 23 or 2 or any other random number; see line 6 in listing 2.1.

To further obfuscate things, the `DATA`, and `CODE` arrays are initialized randomly at different positions so that we have a sparse allocation of elements. Additionally, the initialization of these elements in the source code is not done in any particular order, neither based on virtual data, nor data array first and then bytecode. We shuffle the initialization statements so that these initializations are completely random, see figure 2.2.

```
1   code[438]=3590;    data[2038]=-979;
2       data[2495]=67;
3       data[3724]=max;
4   code[827]=3148;code[1068]=3590;code[479]=3382;code[859]=3590;code[915]=3382;
5   code[344]=3147;code[390]=2070;code[1192]=9352;code[1145]=3382;
6   code[404]=121;code[750]=2495;    data[2686]=-886;
7   code[355]=2070;code[638]=7133;code[757]=347;    data[1529]=899;
8   code[687]=3590;code[854]=1208;code[1079]=2038;code[525]=2686;
9   code[766]=7383;code[117]=8297;    data[2542]=false;
10  code[962]=9352;    data[1271]=key;
11  code[1055]=3148;    data[121]=2; data[1855]=67;
12  code[386]=3590;code[793]=914;code[586]=9352;code[519]=7383;code[345]=7504;code[162]=702;
13      data[2174]=false;
14      data[917]=551; data[569]=-1;
15  code[1112]=2897;    data[3369]=533;
16      data[347]=303; data[914]=false;
```

Figure 2.2.: Virtualization variables random initialization

Finally, the switch sections order is also shuffled in order not to preserve any information with regard to which operation was first processed.

2.0.4. Default Operation

Another obfuscation setting which makes use of the frequency property of the virtual operation is the default operation option, i.e. `DEFAULT_MOST_FREQUENT_OPERATION`. The frequency counter shows which is the most frequent operation in the virtualized code. The most frequent operation will also have its key most often in the bytecode array. Therefore, a frequency analysis on the bytecode could highlight the most frequent operation and also easily identify its operands.

In order to mitigate a frequency analysis attack we decided to move the most frequent operation from its assigned case section to the default section of the interpreter. By doing so for every copy of that unique statement we can assign different operation keys which do not exist in the interpreter. Now, in the bytecode, the most frequent operation will have a different instruction key for every invocation of that instruction. Therefore, the most frequent operation will not have its operation key as the most frequent `opcode` in the bytecode array. Please refer to figure 2.1 where at the bottom we have highlighted the default section.

2.0.5. Junk Code

In order to further complicate the mapping of information gained from one instance to another instance we also decided to insert a number of junk values into the `DATA` and `CODE` arrays. We call these junk values also *fake values*.

For each virtual operation on the positions that are not already occupied by the operands and instruction key, we add random values. These random values can also be valid operation keys, but because that bytecode position is never accessed these values do not affect the logic of the obfuscated program. In the same manner, we also add junk values in the virtual data.

The number of junk code position is a setting of the obfuscator, i.e. `MAX_JUNK_CODE`. The algorithm will insert a random number of junk code valus up to maximum the number set in the setting. In this way every statement will have a different number of junk codes; also the number of junk values will be different.

A virtualized version of *iterative factorial* where the most frequent operation is on the `default` branch and we also use junk code can be seen in listing 2.3. Here we only show a section of the entire virtualized method with explanatory comments.

Listing 2.3: Factorial iterative - virtualization with junk code

```
public long FactorialIterative_junk(int num)
{
    //Virtualization variables
    int[] code = new int[100216];
    object[] data = new object[4490];
    int vpc = 30;

    //Data init
    data[2830]=num; //num
    //...
    data[3189]=(long)1; //1 constant
    data[2449]=2; //2 constant
    data[2593]=(long)406L; //result
    data[2788]=false; //var_ifCondition_0
    data[3826]=-585; //var_forIndex_0
    data[2780]=false; //var_whileCondition_0
    data[406]=-248; //fake-406
    // ...
    data[1043]=-743; //fake-1043
    data[770]=591; //fake-770
    data[3500]=794; //fake-3500
    data[2553]=-159; //jmpDestinationName_2553 constant
    data[1437]=68; //if_GoTo_True_1437 constant
    data[3590]=184; //if_GoTo_False_3590 constant
    data[1567]=504; //if_FalseBlockSize_Skip_1567 constant
    data[3753]=-312; //fake-3753
    data[3527]=-619; //fake-3527
    data[2202]=711; //fake-2202
    data[2254]=541; //fake-2254
    // ...

    //Code init
    code[30]=5457; //ExpressionStatement_0 # ExpressionStatement_0
    code[16]=2593; //result
    code[31]=926; //1
    code[50]=3104; //0L
    code[22]=3554; //fake-ExpressionStatement_0_3554_-8
    code[42]=1526; //fake-ExpressionStatement_0_1526_12
    code[17]=123; //fake-ExpressionStatement_0_123_-13
    code[35]=2724; //fake-ExpressionStatement_0_2724_5
    code[57]=1817; //fake-ExpressionStatement_0_1817_27

    code[97]=5941; //ExpressionStatement_1 # ExpressionStatement_1
    code[124]=2788; //var_ifCondition_0
    code[112]=2830; //num
    code[119]=2356; //0
    code[102]=3881; //fake-ExpressionStatement_1_3881_5
    code[100]=853; //fake-ExpressionStatement_1_853_3
    code[107]=2475; //fake-ExpressionStatement_1_2475_10

    // ...
    code[799]=2162; //ReturnStatement_3 # ReturnStatement_12
    code[813]=2593; //result
    code[809]=1045; //fake-ReturnStatement_3_1045_10
    code[825]=573; //fake-ReturnStatement_3_573_26
    code[821]=3864; //fake-ReturnStatement_3_3864_22
```

```
57
58      while(true)
59      {
60          switch(code[vpc])
61          {
62                  case 5941: //frequency 1 ExpressionStatement_1
63                          data[code[vpc+(27)]]= (int)data[code[vpc+(15)]]==
                                (int)data[code[vpc+(22)]];
64                          vpc+=72;
65                          break;
66                  case 9456: //frequency 2 ExpressionStatement_4
67                          vpc += (int)data[code[vpc+(-2)]];
68                          vpc+=58;
69                          break;
70                  // ...
71                  case 2162: //frequency 2 ReturnStatement_3
72                          return (long)data[code[vpc+(14)]];
73                          vpc+=58;
74                  default: //frequency 2 IfStatementSyntax_2
75                          data[code[vpc+(-6)]]= (bool)data[code[vpc+(10)]] ?
                                (int)data[code[vpc+(-11)]] : (int)data[code[vpc+(-2)]];
76                          vpc+=(int)data[code[vpc+(-6)]];
77                          break;
78                  case 5457: //frequency 1 ExpressionStatement_0
79                          data[code[vpc+(-14)]]= (int)data[code[vpc+(1)]]+
                                (long)data[code[vpc+(20)]];
80                          vpc+=67;
81                          break;
82          }
83      }
84
85      return 0;
86  }
```

A vulnerability that exists in the structure of the virtualized program is the bijective relation between interpreter and bytecode keys, `opcodes` in the `CODE` array. Without randomization points, default operation and junk code and even assuming no access to interpreter it would be much more straight forward to apply a frequency analysis technique, e.g. Caesar cipher analysis [12], in order to obtain the instruction size or obtain the most frequent operation, its operands, and then possibly identify the structure of every operation.

Collberg [15, 16] defined as preventive transformations those transformations which aim at making specific deobfuscation techniques more difficult to succeed. One example given, was inserting junk bytes into an instruction. By using *fake values* and the `default` operation we raise the bar for a frequency analysis attack. Although it does not prevent decompilation, it does hamper the reverse engineering of the bytecode structure. Therefore, we can consider this transformation a *preventive transformation*.

2.0.6. Maximum Operands and Invocations

The next setting we address is the maximum number of operands and the maximum number of invocations, i.e. `MAX_OPERANDS` and `MAX_INVOCATIONS`. The maximum number of operands means that one can limit the maximum number of operands that are left in a statement without splitting it further into smaller statements. This setting affects the refactoring phase, see section **??**. A similar setting is implemented also for method invocations and member accesses.

By changing these two parameters we affect the refactored code before virtualization. If the number of operands or the number of invocations is lower, this means that the refactoring is more aggressive, and vice-versa. If the refactoring is more aggresive this means the code which is about to be virtualized has already been changed from the original. The effect of these settings on the obfuscation will be discussed in the evaluation chapter, please refer to section **??**.

2.0.7. Readable and Refactor Mode

A final feature worth mentioning of the tool is the option to set the obfuscation into *readable mode* or *refactor mode* only.

Refactor mode means that the tool will apply only the refactoring transformation without any virtualization step. This is helpful to debug the transformations and see the effect of operand and invocation limitation. It also can be used as method of auto-formatting the code.

Readable mode means that the source code will be augmented with comments which help shed some light upon the mapping between bytecode positions and interpreter sections. Additionally the virtualization arrays initialization statements are not randomized any more, so one can follow operation by operation the values that are used.

Unless specified otherwise, most of the examples given in this chapter have been either in *refactor mode* or *readable mode*. These two modes can also be used in conjunction. For how to set them please refer to listing 2.4.

Listing 2.4: Transformation modes

```
1        //refactor mode
2        [Obfuscation(Exclude = false, Feature = "virtualization; refactor;")]
3        public long FactorialIterative_1(int num)
4        { ... }
5
6        //readable mode
7        [Obfuscation(Exclude = false, Feature = "virtualization; class; readable")]
8        public long FactorialIterative_2(int num)
```

```
9           { ... }
```

Chapter 3

Obfuscator Limitations

As with any other software product, perfection is almost impossible to attain, especially for complex projects or ones which are always open to change. Therefore, our solution is also affected by some limitations which we will present next.

3.0.8. Interpreter

The first limitation which is a security weakness is the visibility of the interpreter and its data structures. During development, one proposed solution was to have the interpreter compiled as a *lambda expression* [44]. This involved the following steps. First, write the interpreter in the form of a lambda expression, then use the `Compile` method in C# and finally serialize the result of the compilation and store it in an encrypted file. After having the interpreter in the file a component should only de-serialize it and call it.

Without going into many details there were two major issues with this proposal. The first one was time constraint issue. In the allocated time for the thesis there was not enough time to write a tool which automatically converts code to expression trees to be used later lambda expressions [56]. The second issue was a technical constraint: expression trees do not support all features of the C# language, e.g. field reassignment [33].

3.0.9. Class Attributes

Another limitation of the virtualization tool is that it cannot virtualize class attributes, meaning that it cannot store the class attributes in the `DATA` array.

The challenge occurs with regard to field reassignment. We illustrate the problem with the following example. Suppose we have a private attribute named

counter of type int. This attribute is used in method_A and in method_B. Method_A will be obfuscated while method_B remains the same. Both methods increment the value of counter. When method_A is obfuscated we assign counter to a position of DATA, but the field being of type int we store only the value. Method_A will behave correctly when executed but if method_B is executed after method_A, then the latter one will see an old value of counter.

A solution to this problem would be to have another layer of data virtualization for the fields of the class. By doing this we ensure that the side effects are maintained across different invocations of the fields. Also when it comes to public fields or properties, a series of refactorings would have to take place in order to replace those fields with accessor methods to the virtualized data.

Struct ...

Multiple layer virtualization ... works but very inefficient

3.0.10. Try/Catch Statements

One important statement which the obfuscator does not support yet is the try/catch statement. After looking into a solution, we have stumbled across the following issues. First, how to store the target destination based on the exception that is thrown. The virtual program counter needs to be updated to point to the exception handling operations based on the thrown exception. The second challenge is how to treat the finally clause by interrupting the execution of code segment. If there is a return statement in the catch, the execution will be interrupted at return, the finally clause will be executed and only then the return statement will be executed.

3.0.11. Compiler

Another limitation is with regard to method parameters which have the type ref [47] and out [46]. The main problem here is that when virtualized, the compiler does not allow casting to pointer directly from the element of an array when casting is used [61].

This problem is illustrated in listing 3.1. Suppose we have a method SetValue with a parameter of type ref. This method is called from the method RefVirtual which is virtualized. The problem occurs when we want to pass the length parameter of method RefVirtual to method SetValue. The compiler does not allow referencing a cast construct and without a cast the types will not match when invoking the SetValue method. One solution around this

would be with a refactoring of the `SetValue` method with the use of generic types. The downside of this approach is that it potentially changes the interface of a module. The same applies for the `out` modifier.

Listing 3.1: Casting exceptions for `ref`

```
1         private void SetValue(int value, ref int dest)
2         {
3                 dest = value;
4         }
5         private void RefVirtual(string value, ref int length)
6         {
7             object[] dataObject = new object[10];
8
9             //when virtualized:
10            dataObject[1] = length;
11
12            SetValue(value.Length, ref  ((int) dataObject[1])); // compiler error.
13            SetValue(value.Length, ref dataObject[1]);          // compiler error.
14        }
```

Another limitation of the obfuscator is that we do not virtualize lambda expressions. The lambda expressions are left as is in order to avoid unwanted side effects due to variable scope. One solution would be to virtualize where possible only the body of a lambda expression.

Yet another limitation occurs with the use of `struct` types [26]. The invocation of the `struct` properties appears as a call from an object to its attributes, but their are called by value, not by reference. After storing the `struct`'s identifier to the `DATA` array, the compiler will give errors when trying to use casting when accessing the attribute of that `struct` element. A solution to this would be to store the elements of the `struct` one by one in the `DATA` array and make the virtualization of the local variables sensitive to `struct` types and not virtualize them as an object reference.

Finally, due to time constraints there are a few language features which are not yet supported. These are the conditional expressions, i.e. `result = condition ? true value: false value` and the `foreach` statement. Conditional expressions can be refactored as `if` statements with a concern to the scope where they are used. `Foreach` statement can be converted to a `while` statement and the virtualization is already implemented for `while` statements.

Not supported ...

Not implemented ...

Bibliography

[1] Agile.NET. Code Protection. http://secureteam.net/obfuscator.aspx. [Online; accessed 16-March-2015]. (cited on p. 15)

[2] B. Anckaert, M. Jakubowski, and R. Venkatesan. Proteus: Virtualization for Diversified Tamper-resistance. In *Proceedings of the ACM Workshop on Digital Rights Management*, DRM '06, pages 47–58, New York, NY, USA, 2006. ACM. (cited on pp. 9, 19)

[3] Android. Code style guidelines for contributors. https://source.android.com/source/code-style.html. [Online; accessed 22-August-2015].

[4] B. W. B. Yadegari, B. Johannesmeyer and S. Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy*, pages 674–691, 2015.

[5] S. Banescu, M. Ochoa, N. Kunze, and A. Pretschner. Idea: Benchmarking indistinguishability obfuscation ‚Äì a candidate implementation. In *Engineering Secure Software and Systems*, volume 8978 of *Lecture Notes in Computer Science*, pages 149–156. Springer International Publishing, 2015.

[6] S. Banescu, M. Ochoa, and A. Pretschner. A framework for measuring software obfuscation resilience against automated attacks. In *Proceedings of the 1st International Workshop on Software Protection*, SPRO '15, pages 45–51, Piscataway, NJ, USA, 2015. IEEE Press.

[7] S. Banescu, A. Pretschner, D. Battré, S. Cazzulani, R. Shield, and G. Thompson. Software-based protection against changeware. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 231–242, New York, NY, USA, 2015. ACM.

[8] S. Banescu, T. Wüchner, M. Guggenmos, M. Ochoa, and A. Pretschner. FEEBO: an empirical evaluation framework for malware behavior obfuscation. *CoRR*, abs/1502.03245, 2015.

[9] C#. Coding conventions (c# programming guide). https://msdn.microsoft.com/en-us/library/ff926074.aspx. [Online; accessed 22-August-2015].

[10] J. Cappaert. *Code Obfuscation Techniques for Software Protection*. PhD thesis, Katholieke Universiteit Leuven, Faculty of Engineering, April 2012.

[11] J. Cazalas, J. T. McDonald, T. R. Andel, and N. Stakhanova. Probing the limits of virtualized software protection. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, PPREW-4, pages 5:1–5:11, New York, NY, USA, 2014. ACM.

[12] R. F. Churchhouse. *Codes and Ciphers: Julius Caesar, the Enigma, and the Internet*. Cambridge University Press, 2001. (cited on p. 24)

[13] CIL. Introduction to IL Assembly Language. http://www.codeproject.com/Articles/3778/Introduction-to-IL-Assembly-Language. [Online; accessed 16-March-2015].

[14] F. B. Cohen. Operating system protection through program evolution. *Comput. Secur.*, 12(6):565–584, Oct. 1993.

[15] C. Collberg, I. Clark, and T. D. Low. D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In *In: Proc. of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196, 1998. (cited on p. 24)

[16] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations, 1997. (cited on pp. 4, 24)

[17] Component-Based Architectural Style. https://msdn.microsoft.com/en-us/library/ee658117.aspx. [Online; accessed 22-June-2015]. (cited on p. 1)

[18] ConfuserEx. free, open-source protector for .NET applications. https://yck1509.github.io/ConfuserEx//. [Online; accessed 20-May-2015]. (cited on p. 15)

[19] Converting a "do while loop" into a "for loop". https://www.gidforums.com/t-9546.html. [Online; accessed 22-August-2015].

[20] Converting a "while loop" into a "for loop". http://cboard.cprogramming.com/c-programming/138512-need-help-converting-while-loops-loops-my-code.html. [Online; accessed 22-August-2015].

[21] K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 275–284, New York, NY, USA, 2011. ACM.

[22] Crypto. Obfuscator For .Net. http://www.ssware.com/cryptoobfuscator/obfuscator-net.htm. [Online; accessed 20-May-2015].

[23] Design-Pattern. Pipes-and-filters pattern. https://msdn.microsoft.com/en-us/library/dn568100.aspx. [Online; accessed 12-October-2015].

[24] Dotfuscator. .NET Obfuscation. https://www.preemptive.com/products/dotfuscator/overview. [Online; accessed 20-May-2015].

[25] dotnetperls.com. Locals allowed. http://www.dotnetperls.com/locals-allowed. [Online; accessed 10-May-2015]. (cited on p. 7)

[26] dotnetperls.com. Struct types. http://www.dotnetperls.com/struct. [Online; accessed 12-July-2015]. (cited on p. 29)

[27] dotPeek. Free .NET Decompiler and Assembly Browser. https://www.jetbrains.com/decompiler/. [Online; accessed 20-May-2015].

[28] dotTrace. .NET Profiler. https://www.jetbrains.com/profiler/index.html. [Online; accessed 20-May-2015].

[29] Eazfuscator.NET. obfuscator and optimizer for .NET. http://www.gapotchenko.com/eazfuscator.net. [Online; accessed 16-March-2015].

[30] Eric Lippert. Representation and identity. http://blogs.msdn.com/b/ericlippert/archive/2009/03/19/representation-and-identity.aspx. [Online; accessed 22-August-2015].

[31] S. Garg, M. Raykova, C. Gentry, A. Sahai, S. Halevi, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *In FOCS*, 2013.

[32] J. Golić and M. Mihaljević. A generalized correlation attack on a class of stream ciphers based on the levenshtein distance. *Journal of Cryptology*, 3(3):201–212, 1991.

[33] M. Gravell. Expression as a compiler. http://www.infoq.com/articles/expression-compiler. [Online; accessed 12-July-2015]. (cited on p. 27)

[34] ILDasm. IL Disassembler. https://msdn.microsoft.com. [Online; accessed 20-May-2015].

[35] ILSpy. open-source .NET assembly browser and decompiler. http://ilspy.net/. [Online; accessed 20-May-2015].

[36] IntelPIN. a tool for the instrumentation of programs. https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/index.html. [Online; accessed 20-May-2015].

[37] G. J. Jain. Tracing with Code Injection, part 1. http://www.codeproject.com/Articles/445858/Tracing-with-Code-Injection. [Online; accessed 20-June-2015].

[38] G. J. Jain. Tracing with Code Injection, part 2. http://girishjjain.com/blog/post/Tracing-with-Code-Injection-Part-II.aspx. [Online; accessed 20-June-2015].

[39] Jungheinrich AG. http://www.jungheinrich.de/. [Online; accessed 15-April-2015].

[40] JustDecompile. .NET assembly browser and decompiler. http://www.telerik.com/products/decompiler.aspx. [Online; accessed 20-May-2015].

[41] JustTrace. memory and performance profiler for .NET and native apps. http://www.telerik.com/products/memory-performance-profiler.aspx. [Online; accessed 20-May-2015].

[42] J. Kinder. Towards static analysis of virtualization-obfuscated binaries. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, WCRE '12, pages 61–70, Washington, DC, USA, 2012. IEEE Computer Society.

[43] S. Krysmanski. The visitor pattern explained. http://www.codeproject.com/Articles/588882/TheplusVisitorplusPatternplusExplained. [Online; accessed 12-October-2015].

[44] Lambda-Expressions. C# programming guide. https://msdn.microsoft.com/en-us/library/bb397687.aspx. [Online; accessed 12-July-2015]. (cited on p. 27)

[45] T. László and A. Kiss. Obfuscating c++ programs via control flow flattening. In *Annales Univ. Sci. Budapest., Sect. Comp. 30*, pages 3–19, 2009. (cited on p. 3)

[46] Microsoft. out parameter modifier (c# reference). https://msdn.microsoft.com/en-us/library/ee332485.aspx. [Online; accessed 12-July-2015]. (cited on p. 28)

[47] Microsoft. ref parameter modifier (c# reference). https://msdn.microsoft.com/en-us/library/14akc2c7.aspx. [Online; accessed 12-July-2015]. (cited on p. 28)

[48] ProGuard. free Java class file shrinker, optimizer, obfuscator, and preverifier . http://proguard.sourceforge.net/. [Online; accessed 20-May-2015].

[49] Python. Style guide for python code. https://www.python.org/dev/peps/pep-0008/. [Online; accessed 22-August-2015].

[50] redgate. ANTS Performance Profiler. http://www.red-gate.com/products/dotnet-development/ants-performance-profiler/. [Online; accessed 20-May-2015].

[51] ResourceLib. C# File Resource Management Library. https://github.com/dblock/resourcelib. [Online; accessed 10-July-2015].

[52] R. Rolles. Unpacking virtualization obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association. (cited on p. 20)

[53] Roslyn. Can i rewrite source code within the compiler pipeline? https://github.com/dotnet/roslyn/wiki/FAQ. [Online; accessed 22-August-2015].

[54] Roslyn. The .NET Compiler Platform provides open-source C# and Visual Basic compilers with rich code analysis APIs. https://github.com/dotnet/roslyn. [Online; accessed 16-March-2015]. (cited on pp. 1, 2, 3)

[55] RuntimeFlow. monitors in real time and logs function calls and function parameters in a running .NET application and shows a stack trace tree. https://vlasovstudio.com/runtime-flow/index.html. [Online; accessed 20-May-2015].

[56] A. Rusina. Generating dynamic methods with expression trees. http://blogs.msdn.com/b/csharpfaq/archive/2009/09/14/generating-dynamic-methods-with-expression-trees-in-visual-studio-2010.aspx. [Online; accessed 12-July-2015]. (cited on p. 27)

[57] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 94–109, Washington, DC, USA, 2009. IEEE Computer Society.

[58] Stackoverflow.com. Converting a "do while loop" into a "while loop". http://stackoverflow.com/questions/26446193/converting-a-do-while-loop-into-a-while-loop. [Online; accessed 22-August-2015].

[59] stackoverflow.com. How to increase limit of local variables. http://stackoverflow.com/questions/20657278/how-to-increase-limit-of-local-variables. [Online; accessed 10-May-2015]. (cited on p. 7)

[60] stackoverflow.com. How to partially update compilation with new syntax tree? http://stackoverflow.com/questions/30670881/how-to-partially-update-compilation-with-new-syntax-tree. [Online; accessed 22-August-2015].

[61] stackoverflow.com. Passing an explicit cast as a ref parameter (c#). http://stackoverflow.com/questions/2165892/passing-an-explicit-cast-as-a-ref-parameter-c. [Online; accessed 12-July-2015]. (cited on p. 28)

[62] Z. Su, B.-R. Ahn, K.-Y. Eom, M.-K. Kang, J.-P. Kim, and M.-K. Kim. Plagiarism detection using the levenshtein distance and smith-waterman algorithm. In *Innovative Computing Information and Control, 2008. ICICIC '08. 3rd International Conference on*, pages 569–569, June 2008.

[63] Themida. advanced windows software protection system . http://www.oreans.com/themida.php. [Online; accessed 20-May-2015]. (cited on p. 15)

[64] N. Tillmann and J. de Halleux. Pex-white box test generation for .net. In B. Beckert and R. Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin Heidelberg, 2008.

[65] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, pages 45–54, Washington, DC, USA, 2005. IEEE Computer Society.

[66] yGuard. Java Bytecode Obfuscator and Shrinker. https://www.yworks.com/en/products_yguard_about.html/. [Online; accessed 20-May-2015].

[67] L. Yujian and L. Bo. A normalized levenshtein distance metric. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(6):1091–1095, June 2007.

List of Abbreviations

CFG	control flow graph
CIL	common intermediate language
CODE	the array variable which stores the bytecode of the virtualized program
DATA	the array variables which store the variables and other data is processed by the virtualization interpreter
ISA	instruction set architecture
MATE	man-at-the-end
OPCODE	the key of an instruction from the bytecode
VPC	virtual program counter

List of Figures

1.1. Initial bytecode sequence 10

2.1. Virtual Operation Size 20
2.2. Virtualization variables random initialization 21

A.1. Distance trend based obfuscation settings 49
A.2. Resilience to Settings correlation 49

List of Tables

2.1. Obfuscation tool settings 16

A.1. Quick Sort - 100000 elements 52
A.2. Quick Sort - 75000 elements 52
A.3. Quick Sort - 25000 elements 52
A.4. Quick Sort - 12500 elements 53
A.5. Binary Search performance - 3000000 elements 55
A.6. Binary Search performance - 2000000 elements 55
A.7. Binary Search performance - 1000000 elements 56
A.8. Binary Search performance - 500000 elements 56

Listings

1.1. Virtualization annotation 2
1.2. Factorial iterative 5
1.3. Factorial iterative - refactored 5
1.4. Factorial iterative - virtualized data 7
1.5. Factorial iterative - complete virtualization 12

2.1. Interpreter invocation 17
2.2. Class instance interpreter 18
2.3. Factorial iterative - virtualization with junk code 23
2.4. Transformation modes 25

3.1. Casting exceptions for `ref` 29

A.1. Levenshtein distance - Python implementation 47
A.2. Factorial - recursive implementation 48
A.3. QuickSort - iterative implementation 50
A.4. QuickSort - recursive implementation 50
A.5. BinarySearch - iterative implementation 54
A.6. BinarySearch - recursive implementation 54