Universiteit Leiden
The Netherlands

Opleiding Informatica

Difference in code obfuscation between different programming languages

Jeroen Massar

Supervisors:
Tim Cocx

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
`www.liacs.leidenuniv.nl`

14/07/2017

Abstract

Some small developers are depended on the use of code obfuscation as an protection method for their source code. In this research we look at the result from obfuscation tools on four different programming languages. Those for programming languages are Java, PHP, VB.NET and JavaScript. First we look at which programming language is best and most suitable for obfuscation. In this research, 48 programs of different complexities have been put through five obfuscation tools for each programming language. The result was that VB.NET was easiest to obfuscate and is also the most suitable for obfuscation.
After that, we try to reverse engineer the obfuscated code with deobfuscation tools. For Java two tools are used, and for the other languages three tools were used. For each programming language the deobfuscating tools were not able to reverse engineer the code from at least one obfuscation tool. The code obfuscated by JSImg (a JavaScript obfuscator) could not be reversed engineered at all. With VB.NET three obfuscation tools where somewhat successful in obfuscating the code to something that was still difficult to understand after deobfuscating.
The research concludes in that Visual Basic .NET is the best programming language to protect source code.

Contents

Chapter 1

Introduction

In the current era, where software development is still growing, the issue of source code protection is becoming more and more important [15]. Organizations are increasingly having trouble in protecting their intellectual property against software piracy[4]. Generally speaking, there are two forms of code protection: legal or technical protection. While the first can be a good solution for larger companies with high legal budgets, for smaller organizations it is much more difficult to protect themselves against bigger and more powerful competitors[6]. These companies are thus dependent on technical protection methods to protect their intellectual properties.

One of these technical methods is the use of code obfuscation. Code obfuscation is about transforming readable code into unreadable code, while still functioning and making it as hard as possible to reverse-engineer the code back to the original source code. Code obfuscation is a common adopted method and there are many approaches[4].

The purpose of this research is to find if any differences occur in the difficulty and effectiveness of implementing code obfuscation using different programming languages. While there is a lot of research done into the field of code obfuscation, including efficiency and effectiveness of obfuscation methods, as far as we are aware of there has not been any research about differences in efficiency or effectiveness between different programming languages. However, if there is a difference, this information could be really useful for an organization in deciding which programming language they are going to use.

Supporting this main question, this research is also split up into two sub questions. The first being, if there is a differences in quality of obfuscation between different languages. The other question is, is there is one languages better suited for code obfuscation then others.

The thesis is split into multiple chapters. This chapter contains the introduction; Chapter 2 will go deeper into what code obfuscation is and mentions the related work. Chapter 3 explains the approach of the research for this thesis. In Chapter 4 we will look at the results from obfuscating and deobfuscating different programs. Chapter 5 will provide a conclusion based on the results. In the end, we will have a discussion in Chapter 6 and look at possibilities for further research.

This thesis was written as a bachelor thesis at the Leiden Institute of Applied Computer Sciences (LIACS) of the University of Leiden for the bachelor Computer Science and Economics. The thesis was supervised by Tim Cocx.

Chapter 2

Preliminaries

In this chapter we will have a more in depth look into the method of code obfuscation. First we will define a more precise definition of code obfuscation. Second, we will look into different methods within code obfuscation followed by a look at some tools available for code obfuscation. We will also look at the previous work done related to code obfuscation and this research.

2.1 Code obfuscation

2.1.1 What is code obfuscation?

In simple words, code obfuscation is a technique to create code that is difficult for humans to understand. Collberg et al. [6] gives the definition "Code obfuscation refers to a class of techniques that transforms a source program into a target program, such that both programs have the same behaviour, but the targeted program is difficult to reverse engineer by an attacker".

The main use of code obfuscation is to protect source code. It is commonly used by small software companies to protect their application against misuse. Its main purpose is to help organizations protect their intellectual property. A good code obfuscator makes it as hard as possible for buyers or other organizations to be able to understand to processes within source code or copy source code for their own usage. An example is that some companies want to use some algorithms for their own product without the approval of the creator. With code obfuscation it is much harder to decode the source code in comparison to source

code not obfuscated. However, a flaw of code obfuscation is that if a individual is willing enough, in most cases, he will still be able to crack to code. However, the cost and the time needed makes it unprofitable in most cases.
In some cases, code obfuscation is also used for code optimization. This is because code obfuscation is similar to code optimization as [7] states. The main difference is that code optimization is mainly used to minimize execution time while code obfuscation main goal is to maximize the ambiguity of the code.

2.1.2 Code obfuscation techniques

There are many different ways to obfuscate code. Collberg et al. [7] divide the different techniques into four classes: lexical transformations, control transformations, data transformations and a group with other techniques.
Lexical transformations are code transformations that change the lexical structure of the code. An example for this is changing the names of variables, classes and functions from logical names, most programmers use, to incomprehensible names. While the processes within programs are still readable if only techniques of this group are used, if names are transformed as ambiguous as possible, a considerable amount of time will be needed to reverse engineer the code.
Control transformations are obfuscation methods that try to obscure the control-flow of the original code. Control flow statements are for example a while-loop or an if-else statement. In [6] three sorts of control transformations are defined: Aggregation transformations, which break up or merge computations, ordering transformations, which changes the order of computations, and computation transformations, which insert new functions or changes algorithms [6].
Data transformations techniques transform the data within a program. An example for this sort of transformation is splitting the value of an variable into multiple variable[7]. A variable of value of 5, for example, can be split up into two variables of value 2 and 3, and when the original value is called in the original source code, in the obfuscated code the addition of the two new variables will be called. Other examples are data aggregation, a method which changes the way data is grouped, and data ordering that changes the order of the data[24].

The final class consist of all the techniques used in code obfuscation which do not belong to one of the other three groups. Examples for other possible techniques are methods like antidisassembly and anti-debugging. These techniques are further discussed in [7].

2.1.3 Code obfuscation tools

For the most common programming languages there are multiple tools available for code obfuscation. Most of these tools are only able to obfuscate one programming language. The company Stunnix has tools for multiple languages, namely C/C++, JavaScript, Perl and VBscript. A tool from Semantic can be used for many programming languages including the eight biggest languages[38].
Although the biggest tools, such as the previously mentioned tools, are licensed-based, there are also a couple of open-source tools. Examples are obfuscar, CodeDefender and confuser for .NET assemblies, intelliguard for java bytecode, Phpobfuskator and Hix-php-obfuscator for PHP, Pyhidecode for python [42]. For this research we will mostly use open-source tools, free tools or demo versions of licensed-based tools. More about this in the next chapter.

2.2 Related work

In [6], techniques for technical protection for software are reviewed and they concluded that at the moment code obfuscation is the most viable method against reverse-engineering. In [4], research has been done on different techniques for code protection implemented in assembly code. Wrobleswski gives a general assessment about code obfuscation in [47]. A paper that has done research on effectiveness and efficiency of code obfuscation is [5].

Chapter 3

Approach

In this chapter we will look at the approach of the research. First, we will look at the choices made before the research about which data to use for the research. Second, we will look at the process of the research.

3.1 Creating the research data

Before we are able to look at which language is the best to obfuscate, we first need to determine which programming languages are the most interesting and relevant to use during the research and which components the programs we are going to obfuscate will have.

3.1.1 Programming languages

For this research, multiple programming languages will be used. To determine these programming languages, three factors are being considered: How much a language is being used, how the language is compiled and how easy the source code is readable without obfuscation.

To determine which programming languages are the most used languages, the TIOBE-index [38] will be used. This is an index based on how many hits a programming language has in the most common search engines. We will look at the programming language with a rating higher than 2,5%. [1]. This results in nine programming languages: Java, C, a C++,

[1]This rating has been chosen because a higher rating resulted in just a few languages and would skip some languages important for some application-types, while a lower rating would result into a far to wide

C#, Python, PHP, Visual Basic.NET (VB.NET), JavaScript and Delphi.
From this selection, Delphi has the most protected method of compilation, because Delphi cannot be decompiled[2]. C and C++ are compiled into native machine code, which makes it relatively hard to get usable information from the source code back. C# uses the Common Intermediate Language (CIL), which means a lot of information from the original source code can be obtained from reading CIL. Something similar goes applies to Java and .NET. Java is compiled to bytecode, which includes a lot of information of the source code and .NET is compiled into IL which can be easy to reverse-engineer [36]. Python is also compiled to bytecode, however there are some differences in comparison to the bytecode you get from compiling a Java application. This is because compiled Python code can be decompiled more easily. The result of this, is that Python is highly discouraged for the sort of applications we will be looking at in the research. Because of this Python, will not be used in this research. For this research will we also not look at C# because there are not that many tools available for obfuscating C#, which would make further steps of this research much more difficult.
Giving the results from above, we will examine four programming languages: Java, PHP, Visual Basic .NET (VB.NET) and JavaScript. Java because it is the most used language and also it is quite important to obfuscate the code because it is easily to reverse-engineer. For PHP and VB.NET there are a lot of tools available for obfuscation and these languages are not relatively unsafe[36], making code obfuscation important. Finally JavaScript, because the full JavaScript source code can be seen by all users, making obfuscation really important.

3.1.2 Code to obfuscate

For this research a great amount of applications will be tested and obfuscated. These applications range from very simple applications to more complex applications. For the applications a list of ten possible components is selected. The complexity of the code is defined by how many of the these ten components are used:

- The amount of variables used in a program
- The amount of different data types used in a program

range of programming languages

- The amount of (different) operators used in a program
- The amount of (different) control flow statements used in a program
- The usage of array's
- The usage of classes or not within a program
- The usage of a constructor within a class
- The usage of an abstract class
- The usage of inheritance between classes
- The usage of composition

The first five components are chosen because these form the basement of each application. However, the way these components are used can result in differences in behaviour with obfuscation. An application with fewer variables may be harder to make unreadable, than an application with many different variables. The same might apply to different data types, the use of operators, the amount of control flow statements and the amount of arrays use. We will also look at the differences of programs with and without classes and the inclusion of different types of classes and some of the possible components of classes. A distinction will be made between applications with or without constructors, abstract classes, inheritance and composition. These are the most common components of classes used in the programming languages.

In total, this result in 280 possible applications for each language to examine. These possibilities can be found in the decision tree in Appendix A. With this tree, we assume that an application with only a few variables will not have a class, because application with both few variables and classes are uncommon. We also assume that if there are few variables, there will also only be few different data types. The reason for this is that in most languages the number of different data types is less than or equal to the amount of variables. Furthermore, we assume that we can only have abstract classes if there are classes and that classes are also needed for the use of constructor, inheritance and composition. For further reduction in the amount of programs, we will divide the possible programs into two groups: programs with few, and programs with many variables. We will look at every program with few variables, because of the small amount of programs. However, because of the large amount of comparable programs, we will mostly only look at all programs containing 2 other (next to the amount of variables) components from the list above. Also, for simplification, we state that if there is composition a constructor is required and an

abstract class needs inheritance. This results in a list of 48 programs. The list can be found in appendix B.

3.2 The research process

3.2.1 Obfuscate the code

The obfuscation process will be done by putting the source code through five different obfuscation tool for each language. We will look at how the code is obfuscated by each obfuscator, and which techniques are used during the obfuscation. After obfuscating, we will look at the performance in each of the languages by using the four classes [6] determined. Finally, we will look at which languages is best suitable for obfuscation. This we will conclude with looking at how common each of the transformation classes are in the languages.

Multiple open source obfuscation tools will be used to obfuscate the tools. The obfuscated code resulting from this will be used later to find out how easy it is to reverse-engineer code in each language. For each language, five different obfuscation tools will be used. For this research the following where chosen:

- Java: Jshrink[12], ProGuard[22], Obfuscat.ion.land[18], DashO[32], Allatori[1]
- PHP: FOPO[43], Mobilefish.com[26], phpencode[48], PHPProtect[17], pipsomania[31]
- VB.NET: .NET Reactor[14], confuser[40], Crypto Obfuscator[23], Skater.NET[34], SmartAssembly6[33]
- Javascript: Dan's Tools Javascript Obfuscator[9], Herokuapp.com[35], Jasob[16], JavaScript2img[44], javascriptobfuscator.com[19]

3.2.2 Reverse-engineer the code

After obfuscating the code, the code will be tested by how well they are resistant against reverse-engineering. This will mainly be done by using open-source deobfuscating tools. We will determine how effective these are in reverse-engineering the obfuscated code. For each language three deobfuscation tools will be used (2 for java, because of the lack for obfuscation tools) to try to reverse-engineer the obfuscated code. The following tools will

be used:

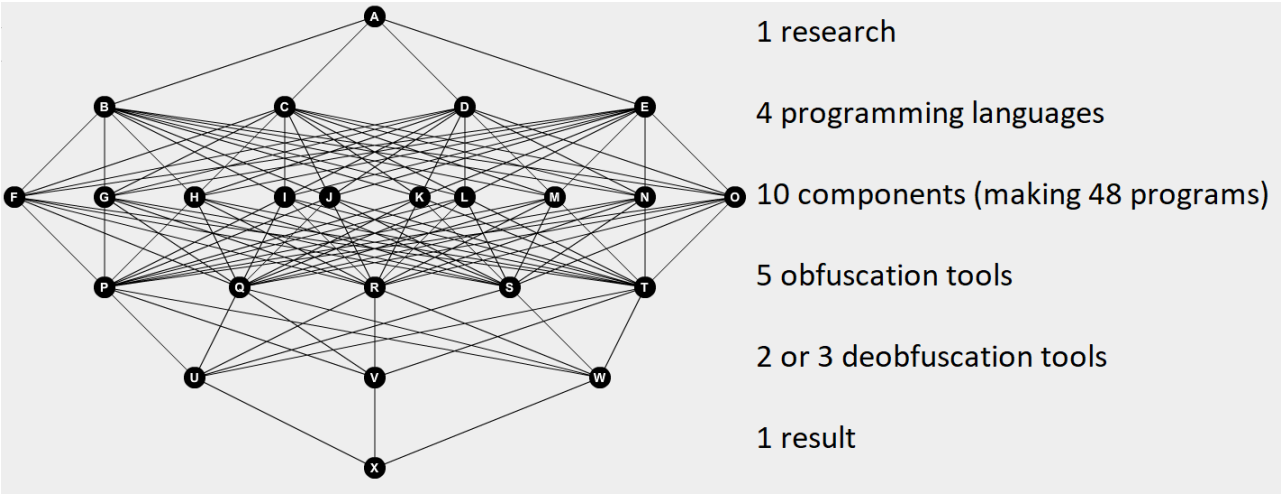
- Java: JDO[45], Enigma[8]
- PHP: unPHP[11], PHP decoder[39], PHP deobfuscator from aWebAnalysis[3]
- VB.NET: Dis#[27], Script Deobfuscator v0.2[21], De4Dot[41]
- JavaScript: jsbeautifier[13], deobfuscatejavascript.com citedeobfscatejavascript.com, jsnice[37]

After deobfuscating code from all obfuscation tools, we will examine which language produced obfuscated code that was the most difficult to reverse engineer. We will rank the deobfuscated code on readability using five ranks. An easy readability means the code is the same as, or almost the same as the original code before obfuscating. Somewhat easy means that only lexical transformations were used or some other small alterations remained. Somewhat hard means that, next to lexical transformations, also some control and data transformations were not deobfuscated. Some reverse engineering by hand is needed. A hard readability means it is possible to further reverse engineer the deobfuscated by hand, but a lot of time is needed to do is. Finally, an impossible readability means that the deobfuscated code is impossible to read and impossible to reverse engineering further by hand.

3.2.3 Comparing the programming languages

After obfuscating and trying to deobfuscate the obfuscated code, we will have a final delve into the performance of each programming language. After that we will answer the main and sub question and determine which programming language works best with obfuscation.

In short: One research will be done. In this research we will examine four programming language. In those programming languages, programs will be written based on ten components, which will result in 48 programs. For each programming language we will than use 5 obfuscator tools to obfuscate these programs. After that, we will use two or three deobfuscation tools to deobfuscate the obfuscated code back. In the end everything will result in one conclusion.



Chapter 4

Results

In this chapter we will examine the result of the research. First we will examine the obfuscation process. Second we will examine the reverse-engineering of the obfuscated programs.

As mentioned in Chapter 3 a test set of 48 programs in four different programming language where used for this research. All code was manually written in Java, and then translated into the other three languages. Many of the programs are comparable with each other. For example: Program 48 contains almost all code from the programs 15 to 47. For this reason, this program will be the first benchmark in this research. The code of the program can be found in Appendix C

The steps within the obfuscation and deobfuscation processes were sometimes done by hand and sometimes done automatically. For JavaScript and PHP, in most cases, the code itself needed to be copied and then be pasted into the obfuscator, and than the obfuscated code could be copied back to a new file. The obfuscators for Java and VB.NET were mostly automatic. In some cases one, but in most cases all program(s) could be selected and obfuscated by a simple click of a button, and saved as a new file.

4.1 Obfuscating the test programs

In this part, we will examine the different programming languages determine which programming language is the best suited for code obfuscation of the four. We will examine what each of the obfuscation tools did with the code for each programming language. We will also examine if the programs are still compileable and executable after obfuscation. Finally, there will be a small comparison between the different language.

4.1.1 Java

Java files can be obfuscated in two ways: By obfuscating the jar-file, which contain all the classes within in java applications. This is the most common method. The other method is by obfuscating the original java-file. To find out what the obfuscators did with the code, the decompiler from javadecompiler.com [10] was used.

Jshrink only changed the names of the classes, variables and functions, and therefor only used lexical transformations. It doesn't add any control flow statements, new functions, and does not use data transformations, resulting in code relatively easy to read and understand. Also, some variables are given a value at a different moment. An example can be found in figure 4.1.

```
class MoreOperators extends Operators{
    public boolean greater_than_or_equal_to;
    public boolean smaller_than_or_equal_to;
    public boolean exponentiation;
    public boolean take_root;
    public int omega;

    public MoreOperators(){
        addition = true;
        multiply = true;
        greater_than_or_equal_to = true;
        exponentiation = true;
        omega = 8;
    }

    void morePytagoras(){
        if (addition & multiply && greater_than_or_equal_to && exponentiation)
            answer = alpha * alpha + omega * omega;
            System.out.println(answer);
        }
    }
}
```

```
class C
  extends B
{
  public boolean I = true;
  public boolean Z = true;
  public int out = 8;

  public C()
  {
    C = true;
    B = true;
  }

  final void I()
  {
    if (((C & B)) && (I) && (Z))
    {
      S = (D * D + out * out);
      System.out.println(S);
    }
  }
}
```

Figure 4.1: An example of the class MoreOperators. Left is the original code. Right is the code obfuscated by JShrink

ProGuard obfuscated the jar-file almost exactly the same as JShrink. One big difference is that ProGuard sometimes splits one big class into two smaller classes. Also, instead of renaming variables to for example `I`, as JShrink did, ProGuard changes the name of that example to `jdfield_a_of_type_Boolean`. However, it is still somewhat easy to understand how the algorithms within the programs work.

Obfuscat.ion.land puts the code of the original java file in a comment and transforms the code into unicode. The alignment of the original file weas lost after obfuscating the code. While the maker of the obfuscator suggest the code should still work even if the code is put into a comment, the code did not compile.

DashO obfuscates the code in a way that is still readable, however instead of java, the code of the different functions were completely written in bytecode. While you have to know bytecode to understand the obfuscated code, if someone does have that knowledge, it won't be too difficult for that person to find out what the different algorithms do. The names of the classes are also changed to just letters of the alphabet and special exception within a class, such as an array out of bound, got their own class. An example can be found in figure 4.2.

```
public MoreOperators(){
    addition = true;
    multiply = true;
    greater_than_or_equal_to = true;
    exponentiation = true;
    omega = 8;
}
```

```
public eval_e()
{
  // Byte code:
  //   0: goto +8 -> 8
  //   3: athrow
  //   4: nop
  //   5: nop
  //   6: athrow
  //   7: athrow
  //   8: aload_0
  //   9: invokespecial 17    eval_g:<init>   ()V
  //   12: aload_0
  //   13: iconst_1
  //   14: putfield 20    eval_g:addition Z
  //   17: aload_0
  //   18: iconst_1
  //   19: putfield 23    eval_g:multiply Z
  //   22: aload_0
  //   23: iconst_1
  //   24: putfield 25    eval_e:greater_than_or_equal_to Z
  //   27: aload_0
  //   28: iconst_1
  //   29: putfield 27    eval_e:exponentiation   Z
  //   32: aload_0
  //   33: bipush 8
  //   35: putfield 29    eval_e:omega    I
  //   38: return
}
```

Figure 4.2: An example of the class MoreOperators. Left is the original constructor. Right is the constructor obfuscated by DashO

Allatori is an obfuscator which transforms the complete code of the program. Classes and variables get new names, declaration is done not a the beginning of an class, but more at the end. Functions are completely altered and are sometimes split into multiple functions. In some cases, code is translated into bytecode, just as with DashO. Almost nothing of the original code can be retraced and it is hard understand the code does within a small amount of time.

In short, three of the obfuscators mainly used lexical transformations, one by transforming

Table 4.1: Overview of the used techniques by the obfuscators for Java

Obfuscator	Lexical	Control	Data	comments
ProGuard	+	+	-	
jshrink	+	-	-	
Obfuscat.ion.land	+	-	-	Everything in Unicode
DashO	+	+	+	Everything in byte-code
Allatori	+	+	+	

the complete code into Unicode, and the other two only changed the names of variables, classes and functions. DashO changed the code to byte-code, which can make it harder to read for some people. However, for someone with the right knowledge this will not be a barrier. Of all the obfuscators for Java, Allatori altered the code the most, using almost all forms of transformations.

4.1.2 PHP

The PHP code was obfuscated using the original PHP files. In most cases the code had to be copied to an online obfuscator. The five obfuscators used for PHP are: FOPO, an obfuscator from mobilefish, phpencode, PHPprotect and Best PHP Obfuscator from pipsomania.

FOPO transforms the code into unreadable text. However, all the obfuscated programs cannot be run by any of the compilers [28][29][30][46] the code was tested in. This may be because the transformed code uses functions not compatible with these compilers. PhpFiddle [29] reports that some parts of the code could not be compiled because the code contained disabled function(s). The first five rows of program 48 obfuscated by FOPO can be found in figure 4.3 on the next page. The files obfuscated by mobilefish.com were also not compilable by the used compilers. This obfuscator uses nested combination to

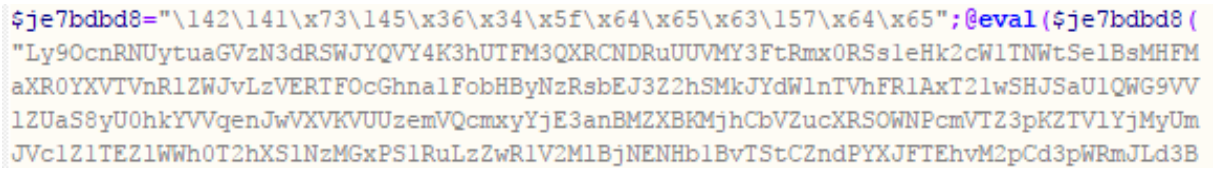


Figure 4.3: The first five rows of program48 obfuscated by FOPO

obfuscate the PHP code (see [26]). The site mentions that the code can only be decoded by their own tool, a eval gzinflate base_64 decoder.

The obfuscator from phpencode also works by translating the code to a form of base_64, however the code itself looks more like the code from FOPO. Just as the previous two, the obfuscated programs obfuscated by this obfuscator did not work in any of the compilers the code was tested in.

The obfuscator PHPProtect only gives variables a different name. The flow of applications can still easily be found, meaning that it is not a good solution to hide algorithms within the program. In contrast to the other obfuscator, the programs could be compiled.

The obfuscator from pipsomania, which they call Best PHP Obfuscator themselves, transforms the code as much as possible into unicode. Because of this the code is at first not easily readable, but because it mainly only uses unicode, there is a big chance it is not that hard to reverse-engineer the code. Just as PHPProtect the code could still be compiled.

Table 4.2: Overview of the used techniques by the obfuscators for PHP

Obfuscator	Lexical	Control	Data	comments
FOPO	+	+	+	Did not compile
mobilefish.com	+	+	+	base64, did not compile
phpencode	+	+	+	base64, did not compile
PHPProtect	+	-	-	
best PHP Obfuscator	+	-	-	unicode

Three of the PHP obfuscators used techniques from all transformation classes. The problem is that in the tested PHP compilers the obfuscated code from all these three obfusctors would not compile. The other obfuscators only use lexical transformations, with pipsomania only translating names to unicode, which is possibly very easy to reverse engineer. PHPprotect does not alter the code a lot as well.

4.1.3 VB.NET

With VB.NET the executable of the code is obfuscated, not the original source code. The obfuscated executables were decompiled using dotPeek[20]. As mentioned in the previous chapter, the obfuscatosr used for VB.NET are .NET Reactor, confuser, Crypto obfuscator, Skater.NET and SmartAssembly 6.

The obfuscator .NET Reactor adds a lot of new modules (comparable to files in other programming languages) next to the original one with long names consisting of random numbers and letters. Nothing of the original code can be traced back using dotPeek, because all functions only consist of the error "unable to decompile the method.". Function declarations and the variables declared outside of functions can be seen. These functions and variables, as well as the classes, got long names with random numbers and letters just as `PFOYuMcChrVM3jnglI`.r The files consist of much more functions than the original files. The code does run, however a warning is given that the code is protected by an unregistered version of Ezizi's .NET Reactor.

Figure 4.4: Part of program 48, obfuscated by confuser

The obfucator confuser also adds a lot of files next to the original files. It also adds a lot of functions, and changes a lot of names into random Chinese and Korean characters. At first it looks like the original file is almost untraceable. However, one of the about thirty files contains almost the exact code from the original file, only with most (variable) names

transformed into random names consisting of non roman characters. With some time, you probably can find out how the algorithms of the programs work. In figure 4.4 a part of program 48 can be found after being obfuscated by confuser.

Crypto obfuscator follows a similar approach to confuser, in that it adds new modules (however not as much as confuser). The main difference is that Crypto obfuscator just uses letters and numbers, instead of characters more difficult to recognize. Crypto obfuscator also adds new functions and control flows. However, in a similar way as with confuser, one of the modules still contains the full original code, with the original variable names replaced with long names consisting of letters and numbers. Algorithms can be traced back quite easily. However, this will still cost some time.

The VB.NET obfuscator from Skater.Net leaves the original code almost intact. In contrast to the previous obfuscators, the obfuscator does not add new modules. In the original code new code has been added only in a few occasions, and only a couple of variables received new names, and those changes do not make it harder to understand the code.

SmartAssembly 6 adds some new modules and all the modules have, what appears, invisible names. In the original module, a lot of new control flow statements are added. All variable names are gone. The flow statements of the algorithms are still traceable. However, without variable names it is hard to understand what the code exactly does. Just as with the other obfuscators, all obfuscated programs still worked.

Table 4.3: Overview of the used techniques by the obfuscators for VB.NET

Obfuscator	Lexical	Control	Data	comments
.NET Reactor	+	?	?	functions are not decompilable
confuser	+	+	+	
Crypto obfuscator	+	+	+	
Skater.NET	+	+	-	Control and data only in few occasions
SmartAssembly 6	+	-	-	variable names are gone

Two of the obfuscators use techniques from all of the main classes. The code from SmartAssembly 6, while doing relatively less to the code, also produced unreadable code. The code from .NET Reactor could not be checked. The code from Skater.NET, while using techniques from multiple classes, did not produce code hard to reverse engineer.

4.1.4 JavaScript

The JavaScript code was obfuscated using the original JavaScript files. In most cases the code had to be copied to an online obfuscator. The five obfuscators used for JavaScriot are: Javascriptobfuscator.com, Jasob, an obfuscator from herokuapp.com, Dan's Tools JavaScript obfuscator and JavaScript2img.

With JavaScript, one of the tools was not able to obfuscate the programs successfully. Javascriptobfuscator.com couldn't obfuscate Javascript programs with classes. This might be because Javascript usally does not use classes. However, all the other tools were successful in obfuscating these programs. All obfuscated programs did work using the compiler Node.js.

All the tools had a different approach in obfuscating the code. The obfuscator Jasob only changed the name of some variables, resulting in that the algorithms are still relatively easy to understand.

The obfuscator from javascriptobfuscator.com changes the text of some commands into a variable written in unicode. While it is a bit better in hiding the code resulting in that the program is still quite easy to read.

The obfscators from herokuapp.com and Dan's Tools changes the complete flow of the program, instead of only changing the names of the variables. Dan's tools uses a method whereby all functions of the program are mutated in a way, where it is not possible to understand the functions of the program immediately. All variables are declared at the end of the code. The obfuscator from herokuapp adds a lot of control flows and also uses a lot of Unicode instead off regular text. An example for code obfuscated by Dan's Tools JavaScript obfuscator can be found in figure 4.5

```
eval(function(p,a,c,k,e,d){while(c--){if(k[c]){p=p.replace(
    new RegExp('\\b'+c+'\\b','g'),k[c])}}return p
    }('2(1=0;1<3;1++){4.5(1)}',6,6,'|x|for|10|console|log'.
    split('|')))
```

Figure 4.5: Program 1 obfuscated by Dan's Tools JavaScript obfuscator

The fifth obfuscator, JavaScript2Img, uses a method where the JavaScript code is embedded into an img file. The maker claims that the code cannot be reversed-engineering using the most common tools such as JsBeautifier. An example can be found in figure 4.6

```
v9590e2cf04e941a01b43d16391df12b0=[ function(
    v16202774b383344b7c3c5a794266f2d1){return '8
    cc981aa1ecd37ee3376cacc031a214e24416c28fcca7fc550bdb3a2065482bbaf49830b
    ';}, function(v16202774b383344b7c3c5a794266f2d1){return
    vd57f593c9c5db784e6ceca5702b0e6d0.createElement(
    v16202774b383344b7c3c5a794266f2d1);}, ...
```

Figure 4.6: As small piece of program 1, obfuscated by JavaScript2Img

From the tools used for obfuscating JavaScript, three of them did a good job obfuscating the code as far as we can conclude at this point. Javascriptobfuscator also delivered more unreadable code. However, the tool was not able to obfuscate all the programs. JavaScript2Img used a special method by translating the code into an img file. They promise, the deobfuscators will not be able to deobfuscate the code.

Table 4.4: Overview of the used techniques by the obfuscators for JavaScript

Obfuscator	L	C	D	comments
Jasob	+	-	-	
javascriptobfuscator	+	+	-	Only program 1-20
herokuapp.com obfuscator	+	+	-	
JavaScript2img	+	+	+	transforms the code into an image
Dan's Tools JavaScript Obfuscator	+	+	-	

4.1.5 Techniques used in the different languages

As we can see in the table below, all obfuscators in every language used lexical transformations to obfuscate the code. Control transformations were less common overall, but were a bit more common with PHP and VB.NET. Data transformations were least common of the three main groups. The code from three of the PHP obfuscators did not compile.

Table 4.5: Amount of obfuscators used each transformation class in each language

Obfuscator	Lexical	Control	Data	comments
Java	5/5	2/5	2/5	
PHP	5/5	3/5	3/5	3/5 did not compile
VB.NET	5/5	3/5	2/5	1 obfusctator was not decompilable
JavaScript	5/5	2/5	1/5	

4.2 Deobfuscating

After examining the obfuscation process, we are now going to investigate the deobfuscation process. For each language three (two in the case of Java) deobfuscation tools where chosen (also see chapter 3). We will look at the obfuscated code from each tool used in the obfuscation process and find out how far the deobfuscation tools succeeded in reverse engineering the code to the original code. Finally, we will compare the four languages.

4.2.1 Java

For Java, two deobfuscation tools were used. These are Enigma and JavaDeObfuscator, which is also known as JDO. JDO only single classes could be deobfuscated. Enigma could deobfuscate the complete the jar file, with all the classes, in one go.
With Enigma, the code from jshrink did not change using the deobfuscator. The reason for this is that jshrink mostly changes the names from variables into something short, and deobfuscators cannot reverse engineer back the names from variables in most cases. Because the variables only consist of a single alphabetical letter, it is not that easy to read the code immediately, but with some time the functions of the algorithms can still easily be found. One big difference is that the obfuscated code doesn't use a constructor (everything is declared before the constructor, within the regular text of the class), but after deobfuscating the code, all declaration are done within the constructor. Using the obfuscated code from jshrink, JDO only changes some names. These consist mostly only out of one letter. By deobfuscating the name gets a bit longer. For example the function call `ds.add()` was obfuscated to `localI.I()` and deobfuscated back to `localClass_I.sub_1ff();`
Proguard changed the names of the variables into for example jdfield_b_of_type_Boolean. Enigma translates this back to regular variable names with one alphabetical letter such as in this example to `b`. This makes the readability a lot better than in the obfuscated code. The code after deobfuscation is comparable to the code from jshrink after deobfuscation. Using JDO results in a similar situation as using JDO on obfuscated code by jshrink . The main difference is that using the code from proguard, only the variable and class names are changed (in the same way as with jshrink), while with the code from jshrink, more names where changed. An example is the boolean variable `addition` was obfuscated to `jdield_c_of_type_Boolean` and deobfuscated to `var_2ec`.

Engima could completely reverse engineer the code from obfuscat.ion.land. Using JDO, only some of the alignment was lost.

Using the obfuscated code from DashO and deobfuscting with Enigma, the code written in byte code is reverse engineered back to code in java. All variables have different names in comparison to the original code. While it is possible to find out the function of the algorithms within the application, some time is needed to figure out what does what, because a lot of extra code in inserted aswell, and some rows of code are split into multiple rows of code, which makes it much harder to read. JDO could not deobfuscate the code.

```
package none;

class n extends a
{
    @Override
    void e() {
        System.out.println(/*EL:213*/this.B + ALLATORIxDEMO("\u0
    }

    @Override
    void ALLATORIxDEMO() {
        System.out.println(/*EL:235*/this.B + this.L);
    }

    @Override
    void A() {
        System.out.println(/*EL:208*/this.B - this.L);
    }

    public static String ALLATORIxDEMO(final String al) {
        final int n = 4 << 3;
        final int n2 = 0x2 ^ 0x5;
        final int length = al.length();
        final char[] array = new char[length];
        int n3;
        int i = n3 = length - 1;
        final char[] array2 = array;
        final char c = (char)n2;
        final int n4 = n;
        while (i >= 0) {
            final char[] array3 = array2;
            final int n5 = n3;
            final char char1 = al.charAt(n5);
            --n3;
            array3[n5] = (char)(char1 ^ n4);
            if (n3 < 0) {
                break;
            }
            final char[] array4 = array2;
            final int n6 = n3--;
            array4[n6] = (char)(al.charAt(n6) ^ c);
            i = n3;
        }
        return new String(array2);
    }
```

```
package none;

public class eval_p
{
    public static String copyValueOf(final String s, int n) {
        try {
            final int n2 = 4;
            final int n3 = 1 + n2;
            final boolean b = false;
            final char[] charArray = s.toCharArray();
            final int length = charArray.length;
            final char[] array = charArray;
            int n4 = b ? 1 : 0;
            final int n5 = (n2 << n3) - 1 ^ 0x20;
            char[] array2;
            while (true) {
                array2 = array;
                if (n4 == length) {
                    break;
                }
                final int n6 = n4;
                final int n7 = array2[n6] ^ (n & n5);
                ++n;
                ++n4;
                array2[n6] = (char)n7;
            }
            return String.valueOf(array2, 0, length).intern();
        }
        catch (eval_r eval_r) {
            return null;
        }
    }
}
```

Figure 4.7: On the left code obfuscated by Allotori and reverse engineered by Enigma. On the right code obfuscated by DashO and reverse engineered by Enigma

Using Enigma, the code from Allatori is debofuscated back somewhat correctly. There is a lot of extra code in comparison to the original code. For example, it is now clear that unnecessary while loops are added to make the code look more complex. After deobfuscating the obfuscated code it is possible to find out what the programs do, however more time will be needed than with the original source code.

With JDO, the code was deobfuscated in a similar way as JDO deobfuscated proguard.

Table 4.6: Readability of the code in Java after deobfusacting by each obfuscator

Obfuscator	Enigma	JDO
ProGuard	somewhat hard	somewhat hard
jshrink	somewhat hard	somewhat hard
Obfuscat.ion.land	Easy	somewhat easy
DashO	hard	hard
Allatori	somewhat hard	somewhat hard

If we look at the best deobfuscator for each obfuscator, we have one obfuscator that obfuscated the code resulting in code that is hard after using deobfuscation tools. Three obfuscators were somewhat hard to read after deobfuscating and the code from one obfuscator was deobfuscated back to the original code.

4.2.2 PHP

As mentioned in the approach, the three deobfuscators used for the obfuscated PHP code are UnPHP, PHP decoder and the deobfuscator from aWebAnalysis. The deobfuscator from aWebAnalysis could not reverse engineer anything. In every instance, the deobfuscator returned an error. The reason for this may be that the code is written in a format the deobfuscator doesn't recognize. PHP decoder had a similar problem. Only with the obfuscated code from phpencode the deobfusctor did not return an error message.
The obfuscated code from FOPO can only be reverse-engineered by the deobfuscator from FOPO themselves. For this, a cipher key has to be declared beforehand as an extra protection method. Only with this key, the deobfuscator will give the exact original code back. UnPHP could find the declaration of a few variables, and could find a couple of functions to align regularly, but almost everything else was still an unreadable document with random characters.
When reverse engineering the code from the obfuscator from mobilefish, at first it looked like the deobfuscator was not successful, because the top of the code had still unreadable text. However, at the bottom of the reversed-engineered code, the exact code of the original source code can be found. This was the case for every program, except for program 1. Using this program, nothing of the original code could be found back. A reason may be that the code of the program only consist of a few rows. Mobilefish also provides an own tool to reverse-engineer their code. This deobfuscation tool could reverse engineer the code

to the exact original code of the programs.

The code from the obfuscator phpencode could be reverse engineered by both UnPHP and PHP decoder. Both came with a comparable result, with Unphp giving a more complex code. However, the result from deobfuscating the different programs, resulted in the exact same code for each program. There may be a chance that the obfuscator did not really obfuscate the code, but changed the code to something completely different. As previous mentioned, the code did not work in any of the tested compilers, so it is not easily possible to check this. In figure 4.9 the code resulting from obfuscating and deobfuscating program 1 can be found.

```php
<?php function YiunIUY76bBhuhNYIO8($g, $b = 0) {
    $a = implode("
", $g);
    $d = array(655, 236, 40);
    if ($b == 0) $f = substr($a, $d[0], $d[1]);
    elseif ($b == 1) $f = substr($a, $d[0] + $d[1], $d[2]);
    else $f = trim(substr($a, $d[0] + $d[1] + $d[2]));
    return ($f);
}
if (!function_exists("YiunIUY76bBhuhNYIO8")) {
    function YiunIUY76bBhuhNYIO8($g, $b = 0) {
        $a = implode("
", $g);
        $d = array(655, 236, 40);
        if ($b == 0) $f = substr($a, $d[0], $d[1]);
        elseif ($b == 1) $f = substr($a, $d[0] + $d[1], $d[2]);
        else $f = trim(substr($a, $d[0] + $d[1] + $d[2]));
        return ($f);
    }
}
```

Figure 4.8: Program 1 obfuscated by phpencode and deobfuscated by UnPHP

The code from phpprotect could not be reverse engineered further by unPHP. This is mainly because there where not a lot of elements to reverse engineer. The obfuscator only changed names from variables and functions, and the original names of variables and functions can almost never be found back. With some programs, instead of giving the code resulting from the reverse engineering, the deobfuscator gave the output of the code.

The code from pipsomania could easily be reversed engineered, because it mostly uses unicode. The only noticeable thing, was that sometimes the code from one row in the original source code, now used multiple rows.

Table 4.7: Readability of the code in PHP after deobfusacting by each obfuscator

Obfuscator	UnPHP	PHP decoder	aWebAnalysis
FOPO	Unreadable	Did not work	Did not work
mobilefish.com	Easy	Did not work	Did not work
phpencode	Different code	Different code	Did not work
PHPProtect	Somewhat easy	Did not work	Did not work
best PHP Obfuscator	Somewhat easy	Did not work	Did not work

Only one of the three deobfuscators was able to deobfuscate any of the obfuscated code back to something resambling the original code. The code from FOPO was still unreadable after deobfuscating. The code from PHPProtect and best PHP obfuscator was deobfusacted somewhat easily readable. The code from mobilefish was deobfuscated back to the original.

4.2.3 VB.NET

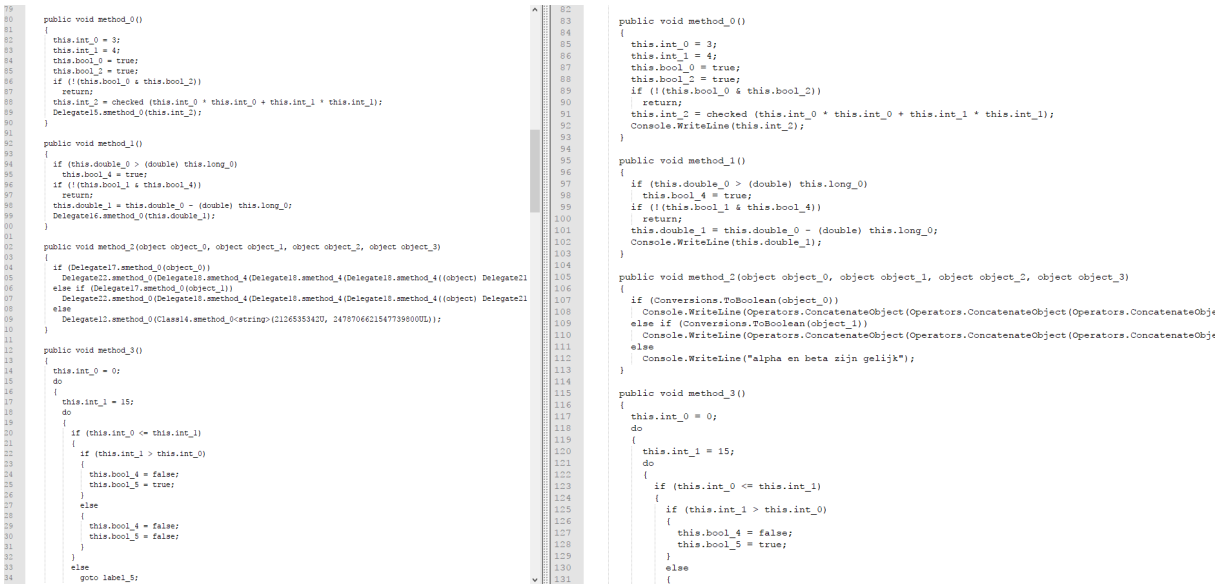
The three deobfuscators used for VB.NET are Dis#, Script Deobfuscator v0.2 and De4Dot. The first and last deobfuscate the executables produced by the obfuscators. Script Deobfuscator only deobfuscate VB.NET code.

The obfuscated code from skater.net is deobfuscated back by Dis# to almost the original code. Script Deobfuscator v0.2 could not do anything visible. The code was not really unreadable, only some code was injected and altered, and the deobfuscator was not able to delete this. Variable names were changed into a character. De4Dot changes these characters to a name consisting of the type the name represent and a number, determined by number of variables already used of the type.

Using Dis#, obfuscated code from SmartAssembly was translated back to normal visual basic code. However, it is still really hard to understand the code. With Script Deobfuscator v0.2 the missing names are found back, but written in unicode. It is still quite hard to read the code, but it is not impossible. Finally with De4Dot the order of the code is put back to the original place. Names of variables, functions and classes where deobfuscated from unicode numbers, to the data type with a number in the case for variables, classes to the word class with a number and functions to method with a number. Where the obfuscated code used a lot more control flow statements such as do and while statements, the deobfuscator shortens the code to only the necessary parts, making it much better readable. Many of the functions have become easily readable after obfuscation. However, it looks like the obfuscator did some strange things on some part, because a few functions

do still include some strange things after deobfuscating.

The code obfuscated by Net Reactor could not be deobfuscated with any of the deobfuscation tools.



```
public void method_0()
{
  this.int_0 = 3;
  this.int_1 = 4;
  this.bool_0 = true;
  this.bool_2 = true;
  if (!(this.bool_0 & this.bool_2))
    return;
  this.int_2 = checked (this.int_0 * this.int_0 + this.int_1 * this.int_1);
  Delegate15.smethod_0(this.int_2);
}

public void method_1()
{
  if (this.double_0 > (double) this.long_0)
    this.bool_4 = true;
  if (!(this.bool_1 & this.bool_4))
    return;
  this.double_1 = this.double_0 - (double) this.long_0;
  Delegate16.smethod_0(this.double_1);
}

public void method_2(object object_0, object object_1, object object_2, object object_3)
{
  if (Delegate17.smethod_0(object_0))
    Delegate22.smethod_0(Delegate18.smethod_4(Delegate18.smethod_4(Delegate18.smethod_4((object) Delegate21
  else if (Delegate17.smethod_0(object_1))
    Delegate22.smethod_0(Delegate18.smethod_4(Delegate18.smethod_4(Delegate18.smethod_4((object) Delegate21
  else
    Delegate12.smethod_0(Class14.smethod_0<string>(2126535342U, 2478706621547739800UL));
}

public void method_3()
{
  this.int_0 = 0;
  do
  {
    this.int_1 = 15;
    do
    {
      if (this.int_0 <= this.int_1)
      {
        if (this.int_1 > this.int_0)
        {
          this.bool_4 = false;
          this.bool_5 = true;
        }
        else
        {
          this.bool_4 = false;
          this.bool_5 = false;
        }
      }
      else
        goto label_5;
```

```
public void method_0()
{
  this.int_0 = 3;
  this.int_1 = 4;
  this.bool_0 = true;
  this.bool_2 = true;
  if (!(this.bool_0 & this.bool_2))
    return;
  this.int_2 = checked (this.int_0 * this.int_0 + this.int_1 * this.int_1);
  Console.WriteLine(this.int_2);
}

public void method_1()
{
  if (this.double_0 > (double) this.long_0)
    this.bool_4 = true;
  if (!(this.bool_1 & this.bool_4))
    return;
  this.double_1 = this.double_0 - (double) this.long_0;
  Console.WriteLine(this.double_1);
}

public void method_2(object object_0, object object_1, object object_2, object object_3)
{
  if (Conversions.ToBoolean(object_0))
    Console.WriteLine(Operators.ConcatenateObject(Operators.ConcatenateObject(Operators.ConcatenateObje
  else if (Conversions.ToBoolean(object_1))
    Console.WriteLine(Operators.ConcatenateObject(Operators.ConcatenateObject(Operators.ConcatenateObje
  else
    Console.WriteLine("alpha en beta zijn gelijk");
}

public void method_3()
{
  this.int_0 = 0;
  do
  {
    this.int_1 = 15;
    do
    {
      if (this.int_0 <= this.int_1)
      {
        if (this.int_1 > this.int_0)
        {
          this.bool_4 = false;
          this.bool_5 = true;
        }
        else
        {
```

Figure 4.9: Left the code from CryptoObfuscator and right from confuser, deobfuscated by De4Dot

Using Dis# with the code obfuscated by CryptoObfuscator the obfuscated code is enormously altered. However instead of making the code readable, the code is made even more unreadable. The complete order of the code is altered, variable names are altered even more. Properties of function and control flow of functions is made unreadable much and much more than it was. However some parts of the codes could not be compiled. and so deobfuscated at all. Script Deobfusctor v0.2 could not reverse engineer anything of the code. First, because only raw code could be decompiled it was hard to find the right file, because the original code was hidden between other pieces or random code. But even with the right code, the deobfuscator could not do anything with it. De4dot did a good job in reverse engineering the code from CryptoObfuscator. The result from deobfuscating was comparable of that from smartassambly. All the long random names where replaced by names consisting of the type and a number, based on how many of these type were already declared. The deobfuscator was also able to reconstruct some of the functions back to their original control flow, however is some cases do...while statements where still in the deobfusctad code, while these where not used in the original code. The obfuscator

also reduced the number of modules used, and gave the remaining ones short clear names consisting of class and a number.

With the code from confuser using Dis#, the names written in special characters are completely gone after deobfuscating. The flow of the code was also a bit altered. The function calls could not be traced back. Using the decompiler we can see that some parts of the code were not visible using the deobfuscation tool. It looks like this may be because a demo version of confuser was used for this research. The code from confuser was reverse engineered back to a relatively readable state by De4Dot. The names of the variables, classes and functions where altered the same way as with code from other obfuscators. Where in the original code, you could see the control flow, but could not read it because of the names, the code is now quite easily readable. We are now also able to determine that something called delegates were added, which are objects that refer to methods [25] This is the reason the obfuscated code had, what looked like, a lot of modules, but these where those delegates which are visualized in a similar way in DotPeek. The code from three of

Table 4.8: Readability of the code in VB.NET after deobfusacting by each obfuscator

Obfuscator	Dis#	Scipt Obfuscator	De4Dot
.NET Reactor	Did not work	Did not work	Did not work
confuser	Hard	Hard	Hard
Crypto obfuscator	Hard	Hard	Hard
Skater.NET	Easy	Somewhat Easy	Easy
SmartAssembly 6	Hard	Hard	Hard

the obfuscators was still hard to read after deobfuscating. The code from one obfuscator was easy to read after obfuscating. None of the deobfuscators worked with the code from .NET Reactor

4.2.4 JavaScript

For JavaScript the jsbeautifier, deobuscatejavascript.com, and JSnice were used for trying to reverse engineer the code. Jsbeautifier and deobfuscatejavascript.com could deobfuscate the obfuscated code form Dan's Tools JavaScript obfuscator perfectly back to the original code. JSnice was not able to reverse engineering the code completely. De deobufscator could only restore the alignment from the obfuscated code, which originally only consisted of 1 row. The code processed by Herokuapp.com was almost perfectly reverse engineered

by jsbeautifier. There were only differences with the names some variables and functions. Some parts of the code were still in unicode. JSNice could only reverse engineer the code from the programs without classes. However, with the programs it could reverse engineer, it did a better job than jsbeautifier. The deobufscator was successful in translating all the unicode to original code. The deobfuscator from deobfuscatejavascript.com was not able to reverse-engineer anything.

The code from the obfuscator Jasob could be completely reversed back to the original code, with the exception of variable and function names, by jsbeautifier. JSNice was also able to reverse engineer, but only the programs without classes. Also the placement of a few functions were altered and the deobfuscated code had comments about the type of each variable, when a variable was declared. Deobfuscatejavascript.com was not able to reverse-engineer any code.

None of the deobfuscators were able to reverse engineer the code from JavaScript2img. JSbeautifier could find some functions within the code, but could not reverse engineer anything recognizable. Deobfuscatejavascript.com could not reverse engineer the code, however it did return the word ””canvas” in some situations. JSNice could only reverse engineer some things from the first thirteen programs. The other programs resulted in an error message. Just as with JSbeautifier, JSNice could find some functions, but it could also find the names of some of the variables and functions.

With the code from Javascriptobfuscator, Jsbeautifer reversed the code almost exactly, except for variable and some function names. We are now also able to conclude that the obfuscator put all the variables in an array, and changed the name of the variables in unicode. JSNice had the same result as jsbeautifier, with the difference that it could reverse engineer the names of the variables within the array. With a little big of extra work, it is easy to get the exact original code back. Also some comments about data types were left behind in jsnice, while jsbeautifier deleted these. Deobfuscatejavascript could not do anything with the code.

Table 4.9: Readability of the code in JavaScript after deobfusacting by each obfuscator

Obfuscator	Jsbeautifier	JSnice	deobfuscatejavascript
Jasob	Somewhat easy	Somewhat easy	Did not work
javascriptobfuscator	Somewhat hard	Somewhat easy	Did not work
herokuapp.com	Somewhat easy	Somewhat easy	Did not work
JavaScript2img	Impossible	Impossible	Did not work
Dan's Tools	Easy	Hard	Easy

Three of the obfuscator tools produced code that was after deobufscating easy readable or somewhat easily readable. Only the code from JavaScript2img was the only code of which the deobfuscation tools were not able to deobfuscate the obfuscated code to something readable. The tools were not able to get anything near readable from the obfuscated code.

4.2.5 Readability after deobfuscating

As seen below, every programming language had one obfuscation tool that provided code that was still easily readable after reverse engineering. Only PHP and JavasScript had an obfuscated which was able to obfuscate the code of which none of the deobfuscators were able to reverse engineer the code to something anything remotely readable. Three of the VB.Net obfuscators and one of the java deobfuscators delivered obfuscated code that still needs a lot of time for someone to fully understand. PHP and JavaScript both have two obfuscators that could be deobfuscated almost completely and three Java obfuscators and one JavaScript obfuscator obfuscated the code to something that was not directly readable.

Table 4.10: Readability of the code by each language, based on the best deobfuscator

Language	Easy	Somewhat easy	Somewhat hard	Hard	Impossible
Java	1	-	3	1	-
PHP	1	2	-	-	1
VB.NET	1	-	-	3	-
JavaScript	1	2	1	-	1

Chapter 5

Conclusion

With the results from the previous chapter we can conclude that Java is the worst suited for code obfuscation with the used obfuscators. Except for one obfuscator, the obfuscated code is still relatively readable. Also, it is relatively hard to get the obfuscation tools working for Java, in contrary to the other programming languages. PHP is also not really suited. While the code is made unreadable in most cases, the obfuscated code could not be compiled, or it is easy to reverse-engineer the code. JavaScript and VB.Net are better suited for obfuscation, but JavaScript obfuscators rely quite heavenly on unicode. With VB.NET, obfuscaters do a lot more work. New control flow statements, functions and even files are easily made to make it as hard as possible to read the original code. This results in that we can conclude with that VB.NET is the programming language best suited of the four tested.

The deobfuscators where not able to obfuscate the code from two obfuscators to anything readable at all. These are the codes obfuscated by FOPO for PHP and for JavaScript2Img for JavaScript. However the code from FOPO could not be compiled by the compiler used. The obfuscated code from DashO for Java was also obfuscated into something hard to read, but with a lot of time on hand, would still be able to be reversed-engineered to something more readable. The code obfuscated by CrytoObfuscator, SmartAssembly 6 and Confuser, all VB.NET obfuscators were also obfuscated to something were deobfuscators produced something which was not readable directly.

This means that for all tested programming languages there was at least one tool that could obfuscate the code into something that was completely not or not directly readable.

JavaScript is the only language with an obfuscator which delivered something unreadable and uncompilable, while VB.NET was the only language with multiple obfuscation tools that could obfuscate the code into somewhat unreadable after reverse engineering.
Code programmed in VB.NET performed well with the obfuscating process as well as the deobfuscating process. Following the results of this research we can say, that VB.NET delivers an effective result with obfuscation in more cases than the other languages and is also the most efficient of the languages. If a small company wants to write its code, the best programming language would be VB.NET.

Chapter 6

Discussion and future work

It is important to mention that this research has been done with open-source obfuscators and trials of paid obfuscators. There is a chance that expensive obfuscators are able to obfuscate, for example Java, into working unreadable code, while this was not the case with the used obfuscators. However, small developers, for whom code obfuscation is one of the only ways to protect their code, also do not have the resources to get these expensive obfuscators and rely on the obfuscators used in this research.

Another important thing to mention is, while we rejected the VB.NET obfuscate .NET Reactor, because it could not decompile and the deobfuscation tools did not work with the code from the deobfuscators, there is also the possibility that the obfuscation tool does the best job of them all. However, this can not be checked. The code did still work, however we can not check if this is because the code still contain the original code. Something similar applies to the rejected PHP obfuscators of which the obfuscated code did not compile.

Other possibilities for comparing the different languages than those done in this research, is looking at how much from the original code is still in the obfuscated code. While, the current comparison gives an indication of this. However, it is possibly interesting to find out the exact percentage of the original code in the obfuscated code.

Another possibility is comparing the size of the original file to the size of an obfuscated file. As mentioned in chapter 2, code obfuscation is sometimes also used for compression of a program. Also, in some cases obfuscating a program makes the program much bigger is size. An much larger program could result in further complications, such as, for example, a shortage of space.

It may also be interesting to look at more different code components and other more complex programs. Examples could be more graphical applications instead of only command line applications.
An interesting fact is that according to the TIOBE-index [38], VB.NET is also growing the most in popularity of the four language. VB.NET grew 0,53% between July 2016 and July 2017. This while the other three programming language had a decline in popularity. Java had a decline of 6,03%, PHP a decline of 0,18% and JavaScript had a decline of 0,04% in popularity.

Ultimately, if someone wants to publish their self made application and does not have the resources to fight legal battles and the only remaining option is the use of technical protection methods, and specifically source code obfuscation, they can best use the programming language Visual Basic .NET as the programming language to program their code in. As previous mentioned VB.NET delivers the most effective and efficient results of the languages tested and hereby the best languages when obfuscation is the best solution for protection.

Bibliography

[1] Allatori. Allatori java obfusctor. http://www.allatori.com/. First accessed: 2017-20-06.

[2] Anderson, T. (2008). Further down the code. *Personal Computer World*.

[3] aWebAnalysis. Php decrypt & decode online. https://awebanalysis.com/en/php-decoder/. First accessed: 2017-25-06.

[4] Behera, C. K. and Bhaskari, D. L. (2015). Different obfuscation techniques for code protection. *Procedia Computer Science*, 70:757–763.

[5] Ceccato, M., Di Penta, M., Falcarin, P., Ricca, F., Torchiano, M., and Tonella, P. (2014). A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19(4):1040–1074.

[6] Collberg, C., Thomborson, C., and Low, D. (1997). A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand.

[7] Collberg, C. S. and Thomborson, C. (2002). Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on software engineering*, 28(8):735–746.

[8] Cuchaz Interactive. Enigma. http://www.cuchazinteractive.com/enigma/. First accessed: 2017-29-06.

[9] Dan's Tools. Javascript obfuscator/encoder. http://www.danstools.com/javascript-obfuscate/. First accessed: 2017-16-06.

[10] Decompilers online. .JAR and .class to java decompiler. http://www.javadecompilers.com//. First accessed: 2017-19-06.

[11] Digital Ocean. Unphp. http://www.unphp.net/. First accessed: 2017-25-06.

[12] Eastridge Technology. Jshrink. http://www.e-t.com/jshrink.html. First accessed: 2017-20-06.

[13] Einar Lielmanis, L. N. Jsbeautifier. http://jsbeautifier.org/. First accessed: 2017-25-06.

[14] Eziriz. .NET Reactor. http://www.eziriz.com/dotnet_reactor.html/. First accessed: 2017-20-06.

[15] Falcarin, P., Collberg, C., Atallah, M., and Jakubowski, M. (2011). Guest editors' introduction: Software protection. *IEEE Software*, 28(2):24–27.

[16] Genesis Mobile. Jasob 4. http://jasob.com/JavaScript-Obfuscator.html. First accessed: 2017-16-06.

[17] Jacob, Y. Php protect: Free php obfuscator. http://www.phpprotect.info/. First accessed: 2017-16-06.

[18] Jaric, P. obfuscat.ion.land java obfuscator - lite. http://obfuscat.ion.land/. First accessed: 2017-20-06.

[19] Javascript Obfuscator. Javascript obfuscator. http://javascriptobfuscator.com/. First accessed: 2017-16-06.

[20] JetBrains. dotPeek: Free .NET decompiler and assembly browser. https://www.jetbrains.com/decompiler/. First accessed: 2017-22-06.

[21] Kahu Securiy. Script deobfuscator v0.2. http://www.kahusecurity.com/2016/script-deobfuscator-released/. First accessed: 2017-02-07.

[22] Lafortune, E. Proguard java optimizer and obfuscator. https://sourceforge.net/projects/proguard/. First accessed: 2017-20-06.

[23] LogicNP Software. Crypto obfuscator for .net. http://www.ssware.com/cryptoobfuscator/obfuscator-net.htm. First accessed: 2017-20-06.

[24] Low, D. (1998). Protecting java code via code obfuscation. *Crossroads*, 4(3):21–23.

[25] Microsoft. Delegates. https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/delegates/. Accessed: 2017-07-05.

[26] Mobilefish. Simple online PHP obfuscator. http://www.mobilefish.com/services/php_obfuscator/php_obfuscator.php. First accessed: 2017-16-06.

[27] NETdecompuler.com. Dis#. http://www.netdecompiler.com/. First accessed: 2017-02-07.

[28] Online PHP function(s). PHP sandbox. http://sandbox.onlinephpfunctions.com/. First accessed: 2017-15-06.

[29] PhpFiddle. PHP/MySQL in-browser IDE and online server. http://phpfiddle.org/. First accessed: 2017-15-06.

[30] PHPTester. http://phptester.net/. First accessed: 2017-15-06.

[31] Pipsomania. Best php obfuscator. http://www.pipsomania.com/best_php_obfuscator.do. First accessed: 2017-16-06.

[32] PreEmptive Solutions. Dasho: Java & android obfuscator & runtime protection. https://www.preemptive.com/products/dasho/overview. First accessed: 2017-20-06.

[33] Red Gate Software. Smartassembly 6. http://www.red-gate.com/dynamic/products/dotnet-development/smartassembly/download. First accessed: 2017-20-06.

[34] Rustemsoft. Skater .net obfuscator. http://rustemsoft.com/obfuscator.asp. First accessed: 2017-20-06.

[35] Serafim, T. Javascript obfuscator tool. https://javascriptobfuscator.herokuapp.com/. First accessed: 2017-16-06.

[36] Short, C. (2011). Source code revelation vulnerabilities. *SANS Institute*.

[37] Software Reliability Lab, Computer Science Department ETH Zürich. Jsnice. http://jsbeautifier.org/. First accessed: 2017-25-06.

[38] TIOBE. Tiobe index for march 2017. https://www.tiobe.com/tiobe-index/. Accessed: 2017-04-06.

[39] Toolki. Php decoder. https://toolki.com/en/php-decoder/. First accessed: 2017-25-06.

[40] Unknown. Confuser. https://confuser.codeplex.com/. First accessed: 2017-16-06.

[41] Unknown. De4dot. https://github.com/0xd4d/de4dot. First accessed: 2017-05-07.

[42] Unknown. Find best open source: obfuscators. http://www.findbestopensource.com/tagged/obfuscator. Accessed: 2017-04-06.

[43] Unknown. Free online PHP obfuscator. http://fopo.com.ar/. First accessed: 2017-16-06.

[44] Unknown. Javascript2img. http://javascript2img.com/. First accessed: 2017-16-06.

[45] Unknown. Jdo. http://javascriptobfuscator.com/. First accessed: 2017-04-07.

[46] Write php online. Start write and run your php code online. http://www.writephponline.com/. First accessed: 2017-15-06.

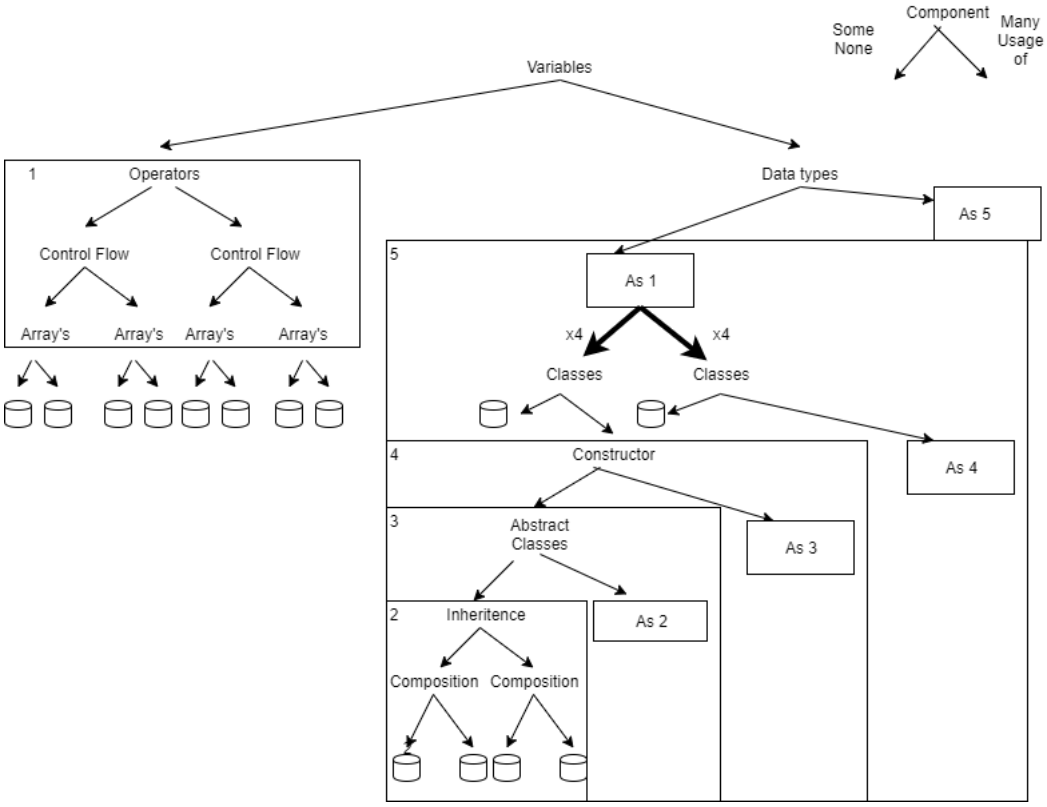
[47] Wroblewski, G. *General method of program code obfuscation (draft)*. PhD thesis, Citeseer.

[48] Zeura.com. Php encode: Obfuscate your PHP scripts easily. http://www.phpencode.org/. First accessed: 2017-16-06.

Appendices

Appendix A

Application Tree



Appendix B

List of programs

In this table "s" stands for small amount, "m" for many, "-" for none and "+" for usage of.

	Variabelen	Data types	Operators	Flow statements	Arrays	Classes	Constructors	Inheritance	Abstract classes	Composition
Program 1	S	S	S	S	S	-	-	-	-	-
Program 2	S	S	M	S	S	-	-	-	-	-
Program 3	S	S	S	M	S	-	-	-	-	-
Program 4	S	S	S	S	M	-	-	-	-	-
Program 5	S	S	M	M	S	-	-	-	-	-
Program 6	S	S	S	M	M	-	-	-	-	-
Program 7	S	S	M	M	M	-	-	-	-	-
Program 8	S	S	M	S	M	-	-	-	-	-
Program 9	M	S	S	S	S	-	-	-	-	-
Program 10	M	M	S	S	S	-	-	-	-	-
Program 11	M	S	S	S	S	-	-	-	-	-
Program 12	M	S	S	M	S	-	-	-	-	-
Program 13	M	S	S	S	M	-	-	-	-	-
Program 14	M	M	M	M	M	-	-	-	-	-
Program 15	M	S	S	S	S	+	-	-	-	-
Program 16	M	M	S	S	S	+	-	-	-	-
Program 17	M	S	M	S	S	+	-	-	-	-
Program 18	M	S	S	M	S	+	-	-	-	-
Program 19	M	S	S	S	M	+	-	-	-	-
Program 20	M	M	M	M	M	+	-	-	-	-
Program 21	M	S	S	S	S	+	+	-	-	-
Program 22	M	M	S	S	S	+	+	-	-	-
Program 23	M	S	M	S	S	+	+	-	-	-
Program 24	M	S	S	M	S	+	+	-	-	-
Program 25	M	S	S	S	M	+	+	-	-	-
Program 26	M	M	M	M	M	+	+	-	-	-
Program 27	M	S	S	S	S	+	+	+	-	-
Program 28	M	M	S	S	S	+	+	+	-	-
Program 29	M	S	S	S	S	+	+	+	-	-

Program 30	M	S	S	M	S	+	+	+	-	-
Program 31	M	S	S	S	M	+	+	+	-	-
Program 32	M	S	S	S	S	+	-	+	-	-
Program 33	M	M	M	M	M	+	+	+	-	-
Program 34	M	S	S	S	S	+	+	+	+	-
Program 35	M	M	S	S	S	+	+	+	+	-
Program 36	M	S	M	S	S	+	+	+	+	-
Program 37	M	S	S	M	S	+	+	+	+	-
Program 38	M	S	S	S	M	+	+	+	+	-
Program 39	M	S	S	S	S	+	-	+	+	-
Program 40	M	M	M	M	M	+	+	+	+	-
Program 41	M	S	S	S	S	+	+	-	-	+
Program 42	M	M	S	S	S	+	+	-	-	+
Program 43	M	S	M	S	S	+	+	-	-	+
Program 44	M	S	S	M	S	+	+	-	-	+
Program 45	M	S	S	S	M	+	+	-	-	+
Program 46	M	S	S	S	S	+	+	+	-	+
Program 47	M	S	S	S	S	+	+	+	+	+
Program 48	M	M	M	M	M	+	+	+	+	+

Appendix C

Programm 48

```
public class programm48{
  public static void main(String[] args){
    WorkGroup workgroup = new WorkGroup();
    String name = workgroup.getFirstName();
    long studentnumber = workgroup.getStudentNumber();
    System.out.println(name);
    System.out.println(studentnumber);
    DoSomething ds = new DoSomething();
    DoSomethingElse dse = new DoSomethingElse();
    ds.add();
    ds.subtract();
    ds.print();
    dse.add();
    dse.subtract();
    dse.print();
    Operators operators = new Operators();
    operators.pytagoras();
    operators.subtract();
    operators.manyoperators();
    operators.manyflowstatements();
    operators.makearray();
    MoreOperators more_operators = new MoreOperators();
    more_operators.pytagoras();
    more_operators.morePytagoras();
  }
}

class Student {
   private String firstname;
   private long studentnumber;

   public String getFirstName() {
       return firstname;
   }
```

```
    public void setFirstName(String firstname) {
        this.firstname = firstname;
    }
    public long getStudentNumber() {
        return studentnumber;
    }
    public void setStudentNumber(long studentnumber) {
        this.studentnumber = studentnumber;
    }
}

class WorkGroup {
  private Student student;

  public WorkGroup(){
    this.student = new Student();
    student.setFirstName("James");
    student.setStudentNumber(123456);
  }

  public String getFirstName(){
    return student.getFirstName();
  }

  public long getStudentNumber(){
    return student.getStudentNumber();
  }
}

abstract class favoritenumbers {
  int a, b, c, d;
  favoritenumbers(){
    a = 5;
    b = 3;
    c = 10;
    d = 4;
  }
  abstract void add();
  abstract void subtract();
  abstract void print();

}

class DoSomething extends favoritenumbers {
  void add(){
    int answer3 = a + b;
    System.out.println(answer3);
```

```
  }
  void subtract(){
    int answer4 = a - b;
    System.out.println(answer4);
  }
  void print(){
    System.out.println(a + " " + b);
  }
}

class DoSomethingElse extends favoritenumbers {
  void add(){
    int answer3 = c + d;
    System.out.println(answer3);
  }
  void subtract(){
    int answer4 = c - d;
    System.out.println(answer4);
  }
  void print(){
    System.out.println(c + " " + d);
  }
}

class Operators{
  public boolean addition;
  public boolean subtraction;
  public boolean multiply;
  public boolean divide;
  public boolean greater_than;
  public boolean smaller_than;
  public boolean equal_to;
  public int alpha;
  public int beta;
  public float gamma;
  public int answer;
  public double answer2;
  public double delta;
  public long epsilon;
  public byte zeta;
  public char eta;
  public String theta = "theta";
  public int[][] array = new int[11][11];
  public int i;

  Operators(){
    alpha = 3;
```

```
  beta = 4;
  addition = true;
  multiply = true;
  gamma = 7.15f;
  delta = 3.14;
  subtraction = true;
  greater_than = false;
  smaller_than = false;
  equal_to = false;
  i = 0;
}


void pytagoras(){
  if (addition && multiply){
    answer = alpha * alpha + beta * beta;
    System.out.println(answer);
  }
}

void subtract(){
  if (gamma > delta)
    greater_than = true;
  if (subtraction && greater_than){
    answer2 = gamma - delta;
    System.out.println(answer2);
  }
}

void manyoperators(){
  for (alpha = 0; alpha < 20; alpha = alpha + 4){
    for (beta = 15; beta > 0; beta = beta - 3){
      if (alpha > beta){
        greater_than = true;
        smaller_than = false;
      }
      else if (beta > alpha){
        greater_than = false;
        smaller_than = true;
      }
      else{
        greater_than = false;
        smaller_than = false;
      }
      output(greater_than, smaller_than, alpha, beta);
    }
  }
```

```
}

void output(boolean greater_than, boolean smaller_than, int
    alpha, int beta ){
  if (greater_than){
    answer = alpha + (beta * alpha) + beta - (beta /alpha);
    System.out.println(answer + " " + alpha + " " + beta);
  }
  else if (smaller_than){
    answer = alpha + (beta * alpha) + beta - (alpha /beta);
    System.out.println(answer + " " + alpha + " " + beta);
  }
  else
    System.out.println("alpha en beta zijn gelijk");
}

void manyflowstatements(){
  for (alpha = 0; alpha < 10; alpha = alpha + 1){
    if (alpha < 6){
      if (alpha == 0){
        equal_to = true;
        System.out.println(alpha);
      }
      else if (alpha == 1){
        equal_to = true;
        System.out.println(alpha);
      }
      else if (alpha == 2){
        equal_to = true;
        System.out.println(alpha);
      }
      switch(alpha){
        case 3:
          System.out.println(alpha);
          break;
        case 4:
          System.out.println(alpha);
          break;
        case 5:
          System.out.println(alpha);
          break;
      }
    }
    else
      output2(alpha);
  }
}
```

```
  void output2(int alpha){
    equal_to = false;
    while (!equal_to){
      answer = alpha + i;
      System.out.println(answer + " " + alpha + " " + i);
      if (alpha == i){
        equal_to = true;
      }
      i++;
    }
    i = 0;
  }

  void makearray(){
    for(int alpha = 0; alpha < array.length; alpha = alpha + 1)
       {
      for(int beta = 0; beta < array.length; beta = beta + 1){
        if (alpha > beta)
          greater_than = true;
        if (greater_than)
          answer = alpha * beta - 15 + alpha;
        else
          answer = alpha * beta;

        array[alpha][beta] = answer;
       }
      }

      System.out.println(theta);

      for (int[] a : array){
        for (int i : a){
          System.out.print(i + "\t");
        }
        System.out.println("\n");
      }
  }
}

class MoreOperators extends Operators{
  public boolean greater_than_or_equal_to;
  public boolean smaller_than_or_equal_to;
  public boolean exponentiation;
  public boolean take_root;
  public int omega;
```

```
  public MoreOperators(){
    addition = true;
    multiply = true;
    greater_than_or_equal_to = true;
    exponentiation = true;
    omega = 8;
  }

  void morePytagoras(){
    if (addition & multiply && greater_than_or_equal_to &&
       exponentiation){
      answer = alpha * alpha + omega * omega;
      System.out.println(answer);
    }
  }
}
```