Tampere University of Applied Sciences

Code Obfuscation

Methods and Practicality Within Automation

Kurtis Wilhoite

BACHELOR'S THESIS
August 2022

Software Engineering

ABSTRACT

Tampere ammattikorkeakoulu
Tampere University of Applied Sciences
Bachelor's Degree Programme in Software Engineering

Wilhoite, Kurtis:
Code Obfuscation
Methods and Practicality Within Automation

Bachelor's thesis 53 pages, appendices 6 pages
August 2022

This thesis discusses, analyses, and explains the four primary methods of code obfuscation: renaming, control flow, debug, and string obfuscation, as well as briefly covering two other notable protections: tamper-proofing and watermarking. Examples of implementations were outlined and explained, as well as considerations to the various forms that these implementations could take.

Four code obfuscation solutions were selected for their professionalism, cost, security, and being reliably updated when compared to other solutions: ConfuserEx2, Eziriz's .Net Reactor, Eazfuscator, and Babel.

Tests were then explained and implemented to assess the four code obfuscation solutions and their qualities of functionality preservation, usability, the level of obfuscation they deliver, and the performance cost prevalent in each. The obfuscation methods offered by each tool were outlined, as well as any other possibly unique protection features.

Functionality was performed as a pass or fail test using a real sample process prevalent in an automation company. Usability was scored using a base of credentials and qualities. Obfuscation level was analysed and various techniques the tools implement were noted and explained. Performance cost was tested through the timing of a sample process from start to finish, and then cross referenced with the non-obfuscated sample code's original performance.

Lastly, comparisons were drawn between the four compared tools, and each were ranked based on each test and their performance. Ultimately, it was concluded that .Net Reactor was the most favourable of these, using the typical environment of an automation company, due to the strong level of obfuscation, low performance cost, and easy to use UI (User Interface).

Key words: obfuscation, automation, security, code, protection

CONTENTS

GLOSSARY

Build	The process of converting source code into a standalone software program which can be run. Compilation is a part of the building process.
Compilation	Translation of source code from a human readable and understandable language into low level binary understood by computers.
Debug	To identify or remove errors from software.
Decompilation	The act of transforming a compiled program or assembly back into readable source code
Deployment	The act or process of providing a software publicly or to a customer.
Dynamic	Used to signify something's ability to be adapted or changed.
Encryption	The act or process of encrypting something: a conversion of something (such as data) into a code or cipher.
Environment	A set of programs, libraries, files, and utilities used in the creation of a cohesive software program.
IDE	An integrated development environment: a software which combines development and building tools into one UI.
Obfuscate	To make obscure: not readily understood or clearly expressed.
Pipeline	An automated series of processes used to compile, build, test, or deploy code.
Project	All assemblies and files that make up a program
Source code	A program's human-readable base code before being compiled or otherwise changed.

1 INTRODUCTION

The field of technology is well known for constantly expanding and twisting in new and inventive directions. Companies, both large and small, work diligently to expand their programs and innovations with new features, protections, and optimizations. Some programs are large, simulating entire worlds for users to get lost in. Some are small, perfecting features down into fewer and fewer lines as they go along. Whether the program is made only with the intent to hone the developer's skills or to offer an invaluable tool like no other, all programs have two things in common: 1. They all have time, effort, and/or knowledge put into them and 2. Their source code can all be understood, if given enough time, by nearly anyone with a working knowledge of coding.

That second part is worrisome and maybe even terrifying depending on who you ask. What do you do if you are selling this code as a product? What happens if a customer takes it upon themselves to alter your company's code themselves? Anyone who can reverse engineer a program may be able to see this source code and copy or change it as they please.

While it may be impossible to completely prevent this possibility, there are many layers of security that companies and individuals can go through to minimize this risk. Encryption, nondisclosure agreements, firewalls, multi-factor authentication, and data loss prevention tools (just to name a few), all exist for the purpose of giving more control over who sees a company's source code. One additional security layer is called "Code Obfuscation".

1.1 What is Code Obfuscation?

To obfuscate something is to make it obscure: not readily understood or clearly expressed (Merriam-Webster n.d.). Applying this to the context of software, code obfuscation is the act of making source code more difficult to understand.

There are many methods to achieve this, which will be discussed in depth later, but in essence they all try to transform source code in some way to make it less

readable while keeping its original purpose. Many methods can even be used in combination with each other for extra layers of coverage.

One rudimentary example would be a program swapping variable names to remain consistent but more difficult to read (Figure 1). While this is not an example of a secure level of obfuscation (which will be explained further later), this exemplifies why, over the span of an entire program and in addition to other forms of obfuscation, this would result in the code being mostly indecipherable.

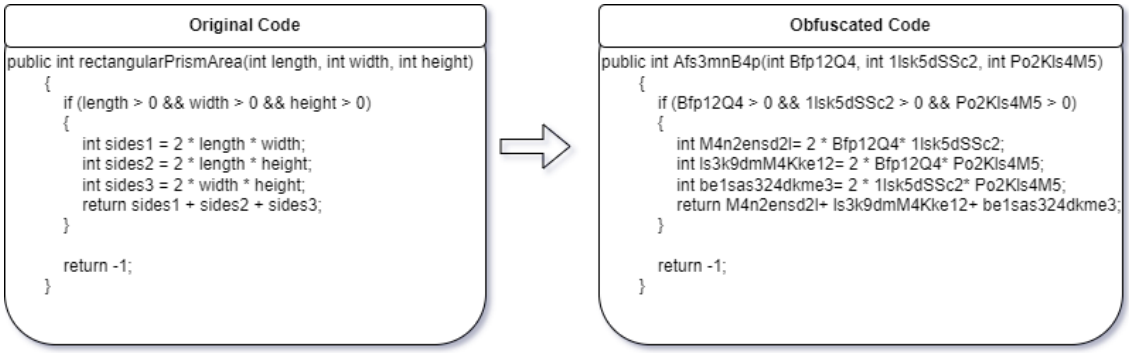

Figure 1. Rudimentary example of obfuscated code using a renaming technique

1.2 Why is code obfuscation beneficial?

Obfuscation acts as a layer of protection and a deterrent. The code can be mixed to the point where it is incomprehensible while still being fully functional by machines. The intent is to make the code overly troublesome to read/modify in case of the program is being reverse engineered, while still allowing the code to function as it was intended to.

Some methods may add a bit of randomness to their obfuscation and trusting that the developer has the source code copied on a more secure platform. Thus, making an obfuscated version much more demanding to reverse engineer. In this case, not even the developer obfuscating the code may know what the final product will be or be able to understand it.

Many methods have their own separate benefits as well: some are easy to use, can be automated using already existing software building and deployment pipelines or development features, and some can even increase efficiency by shortening code, compressing files, or removing unused functions or variables.

Like the locks we use on our homes, it is there to dissuade others who may try to enter, but an experienced crook may have ways around it. In either case, the level of entry to access our belongings is heightened. The NIST (National Institute of Standards and Technology, part of the U.S. Department of Commerce) lists obfuscation among a list of other valid techniques for cyber resiliency, when used alongside other approaches (Ross et al. 2021).

1.3 How useful is obfuscation within automation?

Code obfuscation can be particularly useful in the automation industry. Highly customized programs can be prevalent inside an international customer's systems for decades. Combined with the market being heavily competitive, the source code only becomes that much more valuable making heightened levels of security perpetually needed.

Code obfuscation's ability to be automated using a pipeline or applied during deployment is a valuable feature to note in the automation industry as well. Once setup, obfuscation programs can be run with use of a pre-existing command and simply become a quick way of adding more protection when giving customers access to software.

In this work, the environment of an automation company was used to provide a more realistic, consistent, and real-world example of what is sought after within this field. As well as to answer the foremost questions prevalent to those within this area.

2 OBFUSCATION METHODS

There are many kinds of code obfuscation, some are easier to implement, and some need more time and complex logic. Many types of obfuscation also consider the deobfuscation tools used to reverse engineer code back into readable source code, such as destroying code patterns that decompilation software often use or removing string references which can be searched to find the purpose of the outputs of the program (PreEmptive Solutions n.d.). Obfuscation methods can be used alongside one another as well, meaning each method can potentially supply added effort and experience to break the security.

2.1 Renaming

Renaming obfuscation is a technique in which variables, function and class names, and their corresponding implementations are swapped with less intelligible names. This produces code which is less readable and troublesome to follow while still being fully functional, as these names are used purely for the sake of developer understanding.

2.1.1 Methodology

There are two important things to consider that may affect overall security: What happens when the obfuscation program runs into a variable with the same name, and should new names be unique or reused characters in a new context?

Perhaps the simplest way to implement renaming obfuscation would be a simple program that, upon encountering something which it can rename, creates a scrambled sequence that it replaces all other instances with.

With that simple method of rename obfuscation in mind, the “Complicated & Statistically exchanged Names” (Figure 2) would be one possible outcome of the given source code.

While it does look confusing and not easily read, it is leaving a breadcrumb trail of hints behind it, which hackers may look for. Take the renaming of the variable “Total” as an example: we can tell it is an integer which is being added to in a while-loop and is the output of the function. It is not too difficult to find that “bl2b”

indicates the total of something due to it being initialized as 0 and added onto inside of a loop. Also, because the program has seen that word previously, it swaps all instances of the word “Total” into “bl2b”, meaning the name of the function “CalculateTotal” also holds the same phrase at the end (“tst3*bl2b*”). From this, a hacker can then infer that any time the phrase “bl2b” is encountered, it is simply the renaming of the word “Total”.

This can also potentially blow the cover of multiple other functions using this one simple one, many of our calculations could now be found by a simple search of that new name. Simple one-to-one exchanges of names are not a good thing within obfuscation as it can leave hints like this behind.

What if another direction was taken, and any time a new variable, class, or function was met, it entirely got a simpler new name, and only the iterations were changed? “Simple & Dynamically exchanged Names” (Figure 2) displays this, and in addition to removing some of the hints of the first iteration, it also lowers the overall file size as a bonus. However, this can be taken one step further yet.

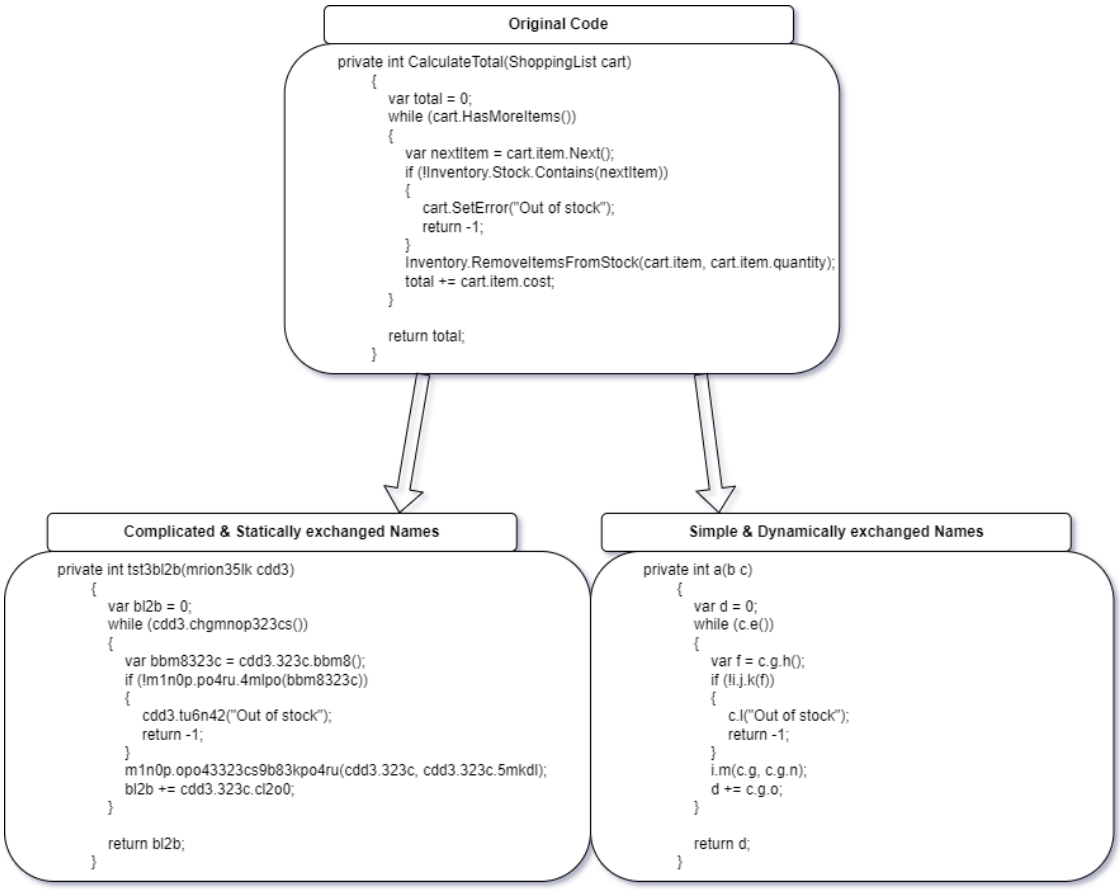

Figure 2. Complicated and Static vs Simple and Dynamic name changes

Taking the idea of simplification one step further, what if we check each class, function and variable and name it as simple as possible with consideration to the fact that these things can share names in certain contexts (For example, a class and variable can both have the name “a”, if the machine can interpret them separately). This can increase the level of confusion even further as what is being referred to by the divergent functions, classes, and variables is not easily understood.

Figure 3 displays the “Unique names” we had previously in Figure 2 but compare it to the new “Reused Names” where there are multiple elements with the name “a”, “b”, and “c”. This additional complexity is all due to them being used in different contexts but having no connection to one another. The renaming is applied as the simplest answer every time it meets something that can feasibly be changed to the simplest option without breaking the code. This is a concept referred to as Overload Induction by PreEmptive Solutions (n.d.).

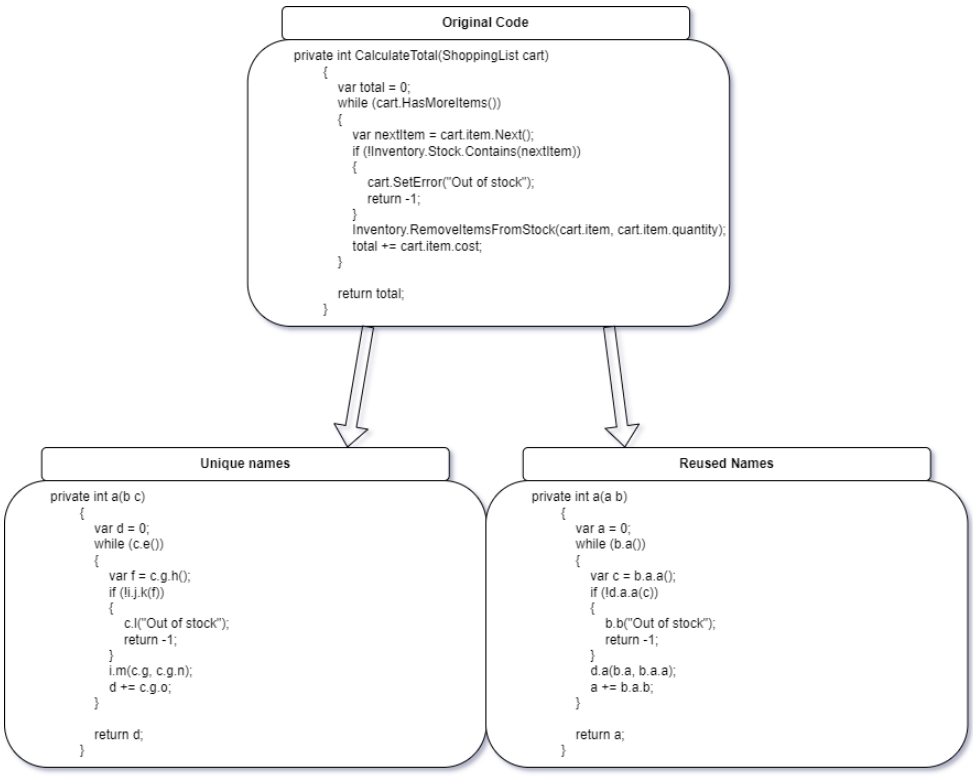

Figure 3. Unique vs Reused naming

2.1.2 Analysis

Renaming obfuscation is often considered the most quintessential, but basic, form of code obfuscation. It certainly does, as all code obfuscation methods

should, scattering the code and making it harder to decipher. Depending on the complexity it can be tricky to tell what the code is trying to do at all.

However, as shown by “Methods”, this is highly dependent on the complexity of obfuscation, otherwise a breadcrumb trail of clues is left behind that could potentially risk the security of recurring names. In addition to this potential issue, renaming does little to stop decompilation, as decompilation software can still detect a consistent pattern in the code and IDEs can still maintain a semblance of understanding as to how the variables link together.

Renaming obfuscation is quite easy to implement, and it does keep some level of protection if decompilation of a project’s code does happen. It can also be done alongside any other method of obfuscation, so it is not necessary for it to be the only level of security. Lastly, it can be automated using pre-existing build processes, and its high potential to make it dynamic for it to work with nearly any project makes it a necessary and consistent foundation of protection.

2.2 Control Flow: Dummy Code, and Opaque Predicate Insertion

A common term used in the field of software engineering is “spaghetti code”, which refers to code in which the order of events is tangled up and awkward to follow. This is usually considered an unintentional bad practice. Control flow obfuscation is meant to mirror this but in an intentional way.

Control flow obfuscation refers to the use of dummy code and opaque predicate insertion to scatter the way that code may work into multiple different “dead-end” routes, which can confuse both hackers and the decompilation tools they may be using to recreate source code (PreEmptive Solutions n.d.).

Dummy code insertion refers specifically to the addition of code with the intent to deceive any tools or malicious users with code that may seem important or relevant but ultimately has no effect on the output.

Opaque Predicate insertion refers to the addition of unnecessary conditional statements (if, while, switch, etc.) which do not detract from the original intent of the code, to offer deceptive “alternative” routes the code could take, some of which it never does. Like dummy code insertion, the intent is to fool malicious users and the tools they may use.

2.2.1 Methodology

Opaque predication and dummy code insertion go together as usually conditional expressions do not work well if the leading up to a condition can easily explain which route will be taken (Collberg C. & Thomborson C. 2000, 10).

In Figure 4, we can see control flow obfuscation taking place through dummy code and opaque predicate insertion. Emboldened in each insertion are the new parts that have been added but will not be followed when the code is executed.

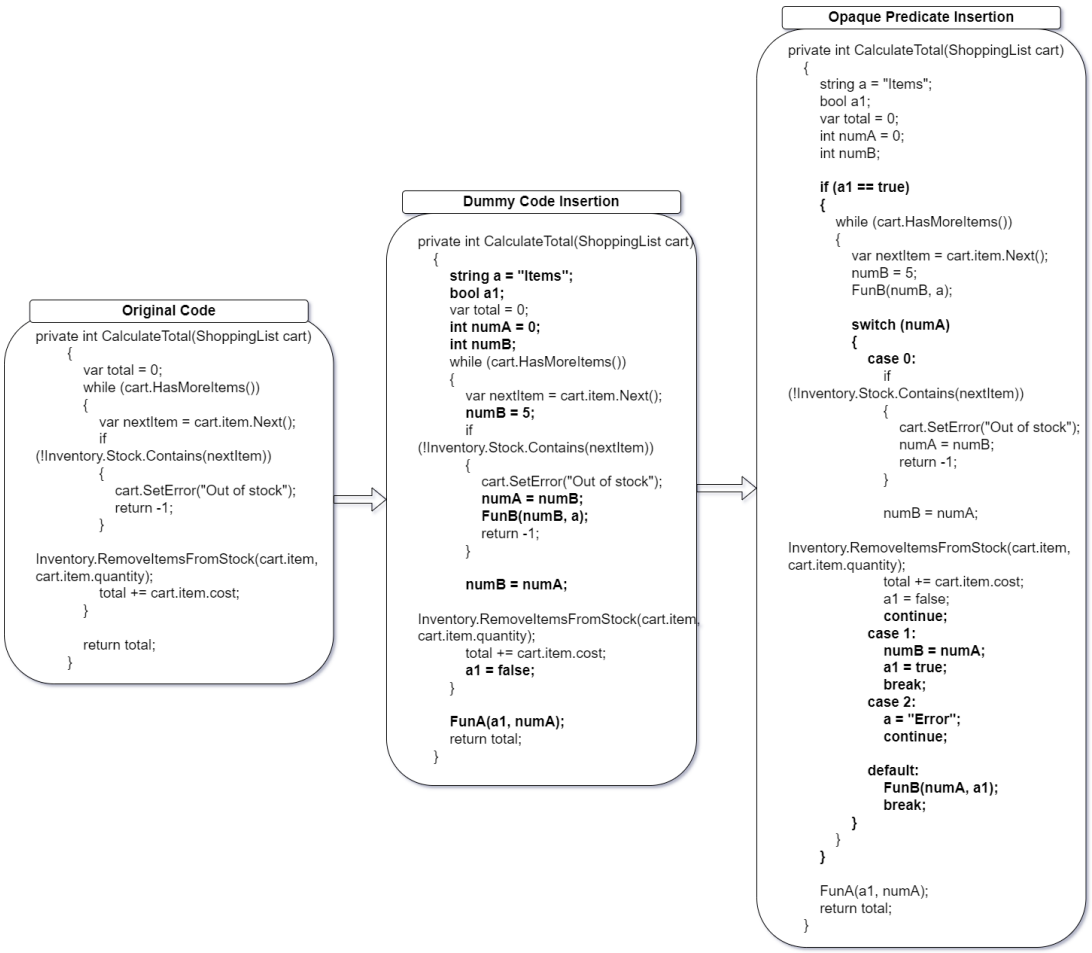

Figure 4. Control flow obfuscation through dummy code and opaque predicates

Dummy code's insertion of variable and functions which will lead off into further unused code works mostly due to their vague names. This is also why renaming obfuscation goes very well with control flow obfuscation: Two variables could be named "a" and "b", and one or even both may not have any actual function inside of the code, but both appear to be used. In this example they are named suspiciously vague things such as "numA" or "a1" for the sake of clarity in the transformation process, but it is important to consider that control flow obfuscation can potentially make renaming obfuscation stronger and vice versa.

Opaque predicates then create possible avenues that programs could travel to using conditional "If" and "Switch" statements. In the example (Figure 4), it is rather easy to follow that the two new conditions will follow the original intent. However, along with renaming and over the span of an entire program, these can become far more difficult to understand, as variables may be passed between

functions or classes and interacted with and simply have no actual input on the outcome.

2.2.2 Analysis

For control flow obfuscation to work properly and to affect deobfuscation software, the code unfortunately needs to react to the new dummy variables and functions on some level. Depending on the depth of the interactions and the overall size of the project, this can have a varying outcome on the program's performance.

Control flow obfuscation also requires more testing than some other methods, primarily because it can affect performance and to ensure the program still functions completely as intended. This also means that control flow obfuscation could possibly introduce more bugs into a system if done improperly, in addition to generally increasing file size.

On the positive side, control flow obfuscation may inhibit decompilation software's usage of code patterns to deduce the source code of a program, which is a large benefit. It also fits well with the renaming obfuscation method, as now new and completely irrelevant variables may leave hints to dead ends while looking no different from variables which may be important. Lastly, it has potential for dynamic integration in similar projects and could be automated into existing development processes.

2.3 Debug and Unused Code Removal

Debugging logs and tools are often used with a project's code for the sake of developers' understanding into what went wrong with a program and why. These logs often give the direct output and path of travel a program is taking to perform various actions.

When reverse-engineering and trying to piece together what a program does, debug information can be immensely valuable. Simply make a small alteration to the code and see how it effects the rest using the debugging logs. Thus, making the removal of debugging files and tools potentially particularly important to securing source code.

Unused code, left intentionally or otherwise, can also present important context to the development process a project followed, providing valuable hints on how a program works or even the development practices a company follows. Unused code is usually inconsequential to remove, but potentially consequential to leave.

2.3.1 Methodology

Removing unused code is one of the easiest forms of obfuscating a code's original purpose. Removal of unused code before deployment is a standard practice but creating an automated system of removal can be used as an added assurance. Exceptions may be necessary in environments using seemingly dead-end code such as data contracts.

Debug removal can be more complex depending on the environment needed. Debug logging does have a purpose after all, it helps developers understand what went wrong in case of a bug or issue. To remove this, in most circumstances, would make a bug far troublesome to track and fix. So, a level of obfuscation needs to be decided.

There are four practical options when considering removing debugging: Not removing it at all, removing it entirely, obfuscation through removing error locations, or swapping/encrypting key information of debugging. In Figure 5, a typical error message is shown under "No Obfuscation". This was brought about by forcing a simple program to try addition on a character value. It shows the exact line and

function in which the failure occurs, a useful thing for developers and hackers alike.

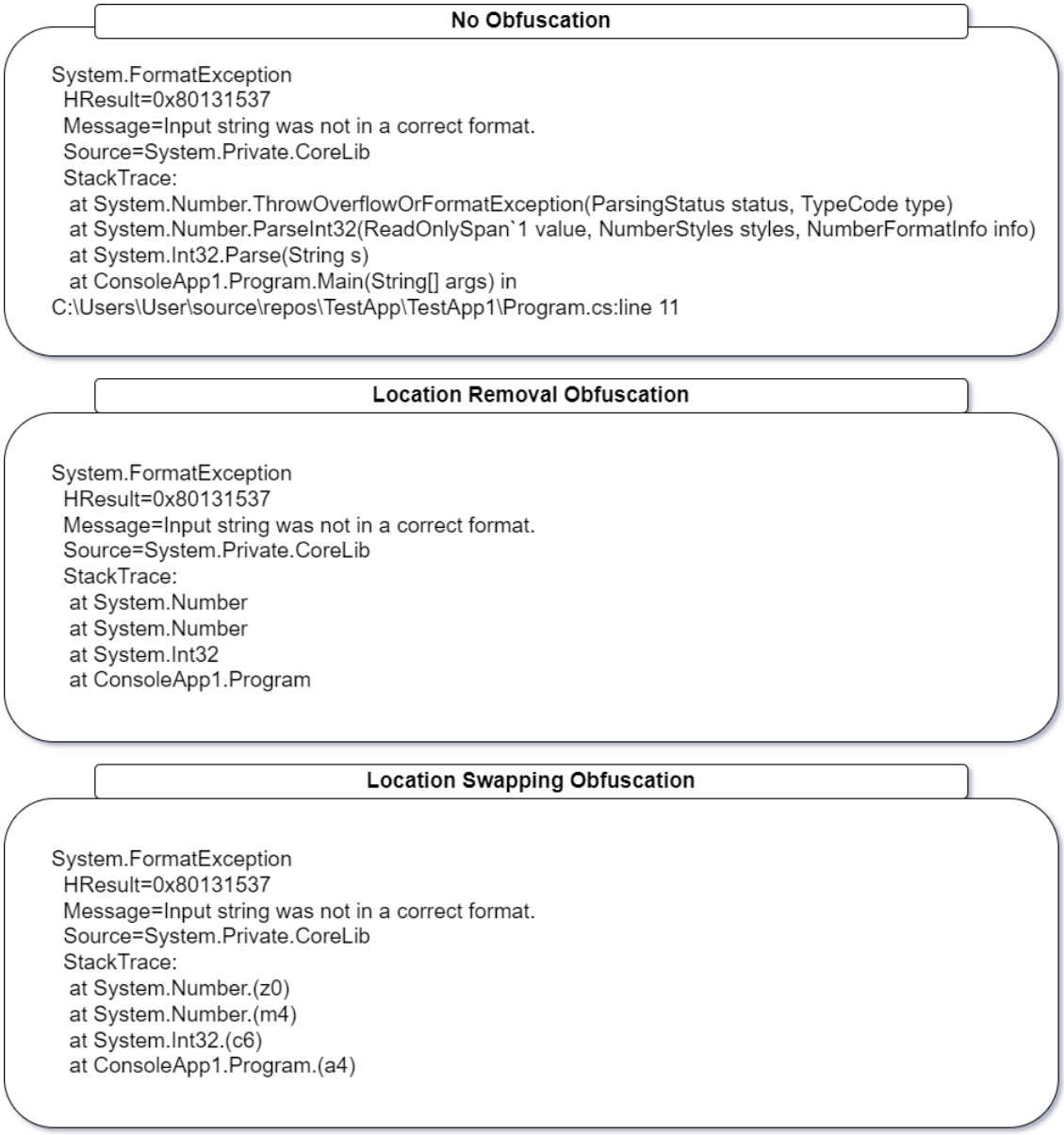

Figure 5. Debug logging obfuscation types

Simply removing all debugging is a possibility. Keeping in mind that it is important that debugging be entirely removed and not simply disabled through a setting or commenting out the code. An attacker is likely to target any debugging system present and reenable it if it remains.

Leaving only part of the information can be a choice as well (“Location Removal Obfuscation”; Figure 5), particularly in the case of using other debugging methods in addition to this. This will have the result of giving developers a solid idea of

where the issue is found in the source code, but only a slight hint to attackers. Since hackers will be dealing with an obfuscated form of the code, this makes it extra time consuming for them. It is not ideal, but this marks a potential middle ground, particularly in cases where other tools are available to developers.

Lastly, locations could be swapped with a scrambled string which represents each class and function (“Location Swapping Obfuscation”; Figure 5) and translated back by tools available exclusive to developers. One step further to this is the possibility to encrypt and de-encrypt logging files when needed.

2.3.2 Analysis

Removing unused code from a delivered product results only in benefits, a system which would remove functions which are not used anywhere in a project would not only result in removing vital hints but reduce file size as well.

Debug logging can be valuable to developers, as well as potential attackers. If it is possible for the system to outright remove debugging after delivering a product, that likely should be done as debugging can be particularly useful for reverse-engineering. However, there are middle grounds that can be met which will throw a wrench in a less-experienced attackers reverse-engineering process while still pointing a more knowledgeable developer in the correct direction of an issue. Taken a step further, the encryption or protection of these log files is also possible.

This is highly dependent on the environment and company at hand and their own development practices. If a system can log how a user got to a particular issue, then debug logs may only be secondary and removable. However, in many systems, it could be the only way a developer has of understanding how to fix any issues. It may be necessary to proceed with caution or investigate other methods.

2.4 String Encryption

String Encryption is about cutting off a key aspect of reverse-engineering. Hackers use output strings given by a program and seek them out inside of the code to better understand how the code works. Any given string surely must exist somewhere in the code, and these can be used as landmarks for revealing the logic of a program.

String encryption tries to solve this by encrypting strings available in a program and decrypting them on runtime. This can also go in-hand with the previously mentioned “Debug removal” method of obfuscation.

2.4.1 Methodology

A common tactic when developers discover an unknown error present in a system is to take a segment of the error that is particularly distinct. For example, an error saying, “Error occurred: Items in the cart are currently out of stock!”, the words “out of stock” may be searched using typical features of an IDE to find points where the issue may occur. On the negative side of this, hackers may try the same thing when trying to dissect a program by intentionally causing errors and then finding more info on how the code processes such issues.

The best way around this is through encryption, and then giving the program the tools to decrypt the strings it needs, as it needs them. If the strings are all encrypted when it is compiled, the resulting reverse-engineered project will also have only encrypted strings, which look like an extreme mix of characters, numbers, and symbols (Figure 6). However, since the system has the tools that it needs to decrypt only when it needs to, the strings will still show appropriately when accessed. Resulting in the system showing the intended message but the message not being searchable in the code.

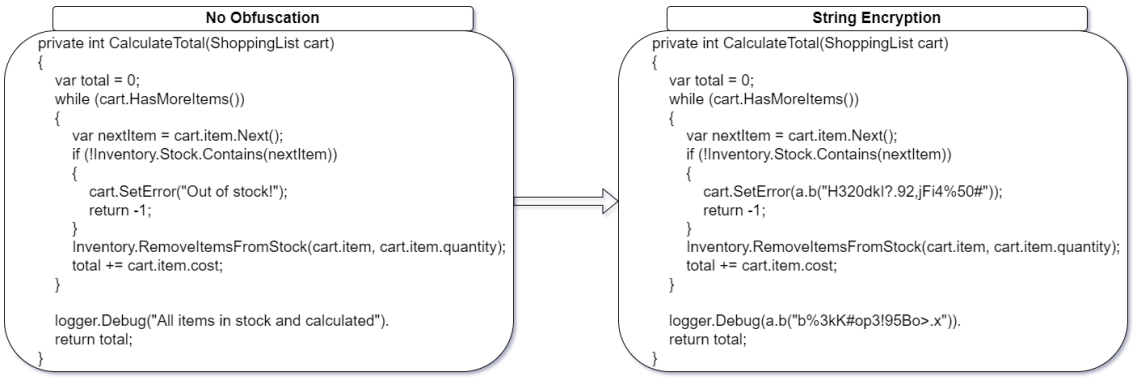

Figure 6. Simple string encryption example

2.4.2 Analysis

There are a couple of downsides to this form of obfuscation. The first being that because the system originally simply gave strings, and now must first decrypt all strings before their usage, this adds some level of performance degradation. Secondly, if the system is creating strings dynamically it will experience a much higher performance degradation as the system, then must decrypt a series of strings as opposed to one-time and direct decryptions.

String encryption can remove an immensely valuable tool out of a hacker's arsenal in a very cut and dry manner. However, encryption and decryption also rely on keys and the validity of the protection can also hinge on reliability of the encryption method being used.

Its overall value as an obfuscation method is heavily reliant on the environment it is built in. Projects which produce heavy string output or dynamically alter strings may lose more in performance than is gained in security, but for systems that have general one-to-one direct string usage, this can raise security highly while only cutting minorly into performance.

3 OTHER NOTABLE PROTECTIONS

There are a couple of other notable protection methods available to software developers which would be amiss to not briefly discuss: Watermarking and Tamper-proofing. Both of which add layers of protection that do not quite fit into the description of an obfuscation method, as they do not restrict the code's readability, but can still afford it valuable protection from outside threats.

3.1 Tamper-Proofing

Much like how we do not use items from a store where the seal of the item has been tampered with, a program can refuse to run once modified. Tamper-proofing can supply more protection if obfuscation fails to protect from reverse-engineering. Hackers often reverse-engineer code by making changes and analysing how that affected the results. The goal of tamper-proofing is to limit or completely stop this process, as when the software is edited outside of the original form or performs atypical behaviour, it can hinder this process in any number of ways defined by the original developer.

3.1.1 Methodology

Tamper-proofing requires two functions: Detecting the alteration of the program it is defending and the procedure to perform upon detection. The latter is up to the developer, and it can be as simple as closing the program or as complex as alerting the company itself to know their software has been tampered with.

The detection of software modification typically falls under one of three primary methods via: Examination, Program checking, and Procedural decryption. Figure 7 gives an example of each of these methods' means of detection.

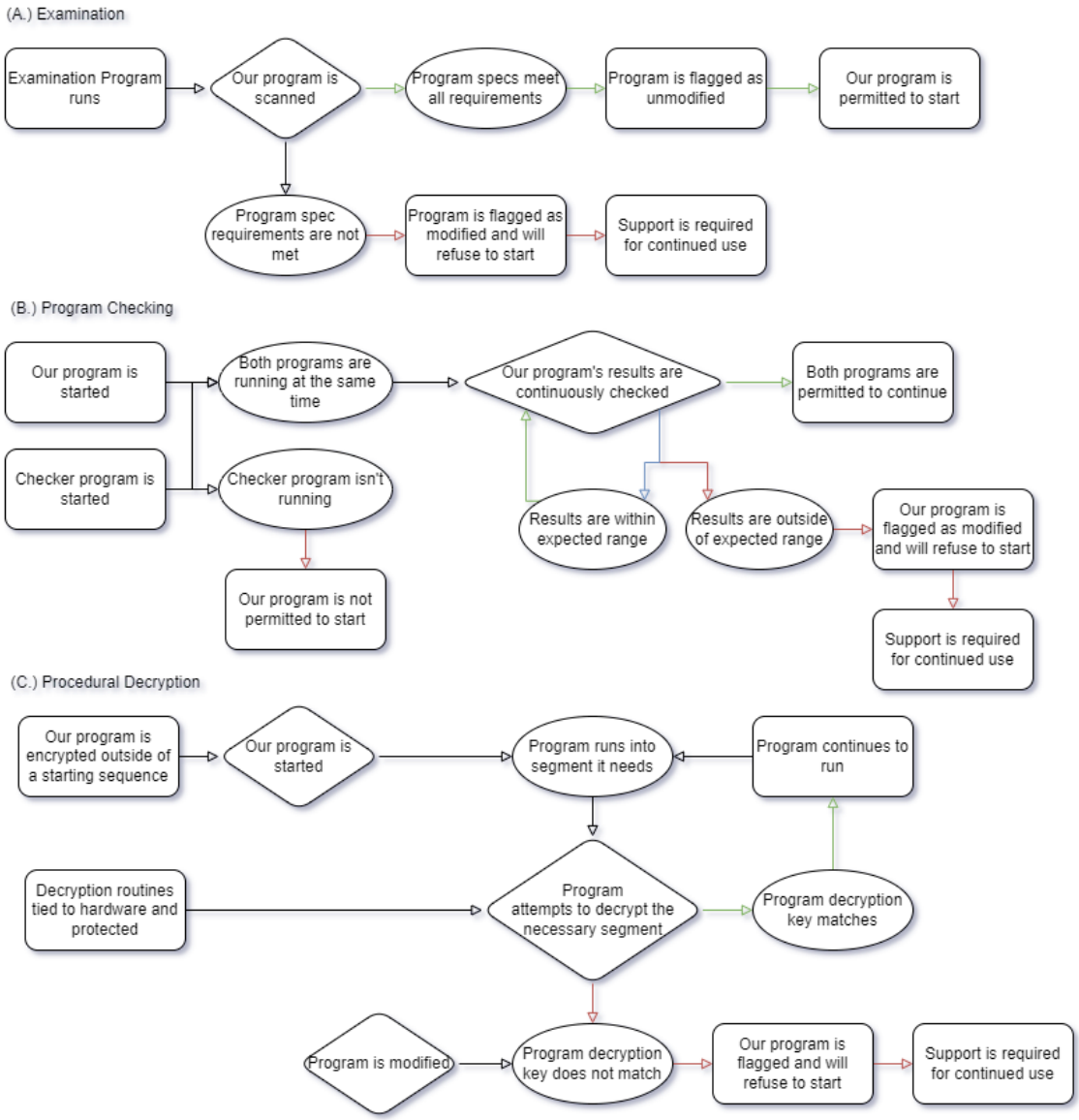

Figure 7. Tamper-proofing detection methods

a) **Examination** is done via requiring an approved examination of the program to be able to be started. The program doing the examination should verify either the whole program or key aspects of the program are identical to the original every time. For extra security, the checked original could be off-site.

b) **Program checking** is running a secondary program or enabling intermittent internal checks inside of the program to ensure behaviour and results are at expected levels continuously.

c) **Procedural decryption** is a program which has been mostly encrypted and decrypts itself as those functions are needed. Variation of any given point would result in a different decryption key for that segment and the

difference in keys could be then used to lock the system or otherwise prevent continued use.

3.1.2 Analysis

The topics covered here are simply the tip of the “Tamper-proofing” iceberg. It is a highly researched and broad topic which could easily be expounded upon further in a separate paper.

Not only does it grant an exterior layer of protection aside from obfuscation, but it can also help a company avoid or act against less reputable customers or users who try to change or use their proprietary software outside of the intended use.

The primary downside of tamper-proofing is that it typically must be taken into consideration at a base level. On-the-fly encryption and decryption of a program or intermittently checking the results of a program is something that may be difficult to successfully implement into an existing system. As well, certain coding languages or environments may not take well to tamper-proofing methods. For example, Java may be difficult to perform “Procedural decryption” with, as it cannot perform such actions stealthily, requiring calling an atypical class to perform encryption or decryption (Collberg & Thomborson 2000, 13).

3.2 Watermarking

Watermarks exist in nearly every facet of media these days. For television, Channels add a watermark at the bottom right or top right of a television program to signify the current program to be licensed under them and not for redistribution. For music, companies can add copyrighted sounds outside of the boundaries of human hearing spectrum which can be used to legally claim copied versions back to the original owner (Kirovski & Malvar 2003). For photography, stock image companies insert highly visible watermarks on copies which have yet to be paid for by customers.

In the realm of software however, more discretion is needed. Obvious signs of watermarking could be stripped from the code and thus potentially negate any legal basis of claiming ownership.

3.2.1 Methodology

Successful watermarking in software boils down into fooling the attacker that they were able to succeed in getting away with copying the code while providing the original owner a discrete legal basis to definitively prove ownership over a property. As such, simple measures taken by other forms of media cannot be so easily used, as those with knowledge enough to reverse-engineer a program likely possess the means to find and remove simplistic markings.

A more dynamic approach is necessary in this case and should be given either in the execution or the structure of the project itself. To maximize discretion of the watermark, output should not be altered. Dynamic watermarking is not as obvious as the more static forms we know, it is delving more into the analysis of the existing metadata of the code or bending it in such a specific way that a copied version could be found. In Figure 8, examples of dynamic watermarking are displayed: Easter eggs, Execution, and Structural watermarks.

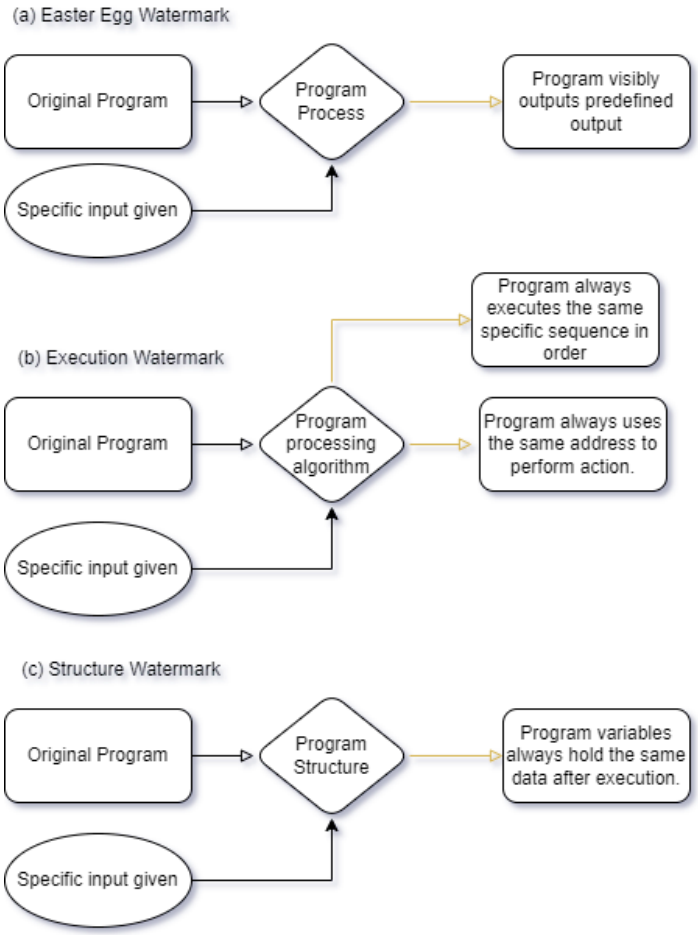

Figure 8. Dynamic Watermarking examples

“Easter egg” watermarks are the simplest of the three. The only thing separating them from a static form of watermark is that a particular input is needed to receive the hidden message. Easter egg watermarks heavily rely on being well hidden, otherwise they can be simply removed (Collberg & Thomborson 1999.).

Execution watermarks require a more technical analysis of the execution order of the code producing a specific output in the machine code of the program when given a specific input. This is less obvious due to the more hidden nature of machine code when compared to direct outputs.

Structure watermarks are left after the execution of specific inputs by the remaining values of the variables. These could then be extracted either through debugging or an external program and those values could be used to verify ownership if they provably pertain to the company in some way (i.e., a slogan or a specific pre-recorded customer code).

3.2.2 Analysis

Software watermarking is more difficult than other media's forms of watermarking. Typically, an attacker will have an understanding deep enough to analyse and remove surface level watermarking. Metadata of the code is potentially important in proving ownership in a discrete manner if it can be provably consistent. Though the watermarks may be overlooked, static and Easter egg watermarks may also be removed by keen-eyed attackers. Watermarking heavily relies on this subtlety to be successful.

Certain types of obfuscation may also break or interact with certain types of watermarking. Execution watermarking such as algorithm timings can be altered once a renaming obfuscation takes place or architectural watermarking may be added onto dynamically by a control flow obfuscation, thus negating these watermarks.

Watermarking could be a potential last line of defence for programs, providing hidden definitive legal proof of ownership. However, when considering watermarking and the potential positives, it is equally as important to think of its difficulties and the potential effects obfuscation could have on it.

4 TESTING PLAN

In the realm of automation, such constant effort is focused into the improvement of proprietary software that outside tools are far more likely to be implemented to protect the system, as opposed to a company creating an obfuscation tool from scratch. Preference would typically be to buy this service from another company.

As such, four tools were chosen to test the quality of the security they offer, usability, versatility in a multitude of environments, and their cost to both the wallet and the protected code's performance. These tools were: ConfuserEx2, .Net Reactor, Eazfuscator, and Babel.

All paid tools were tested using trial versions, and as such, the features available in them may have been subject to pay-walls and may extend outside of those mentioned here. Default settings were used for the tools to give an impression of what the results of most developers would be, as well as what the original creators had intended as their program's status quo.

4.1 Qualities and Tests

Besides the basic requirements of the tools such as professionalism and cost, there are quality based requirements which should be tested further. Those being functionality preservation, obscurity, resilience, performance cost, and overall usability. Without each of these things, an obfuscation tool may be easily negated, difficult to use, or simply require too much effort for the amount of security given.

4.1.1 Functionality Preservation

Of primary concern is the quality of functionality preservation. This is the ability of the tool to obscure the given program, while not tampering with the base functionality of the code. If any obfuscation tool were to vary, break, or otherwise hinder the program's ability to run as intended, it would defeat the point of obfuscation entirely.

This will be performed as a pass or fail test. Our sample project will be obfuscated using each tool and then run as usual. If this process is performed the same as the non-obfuscated form, without any errors or issues present in the system, the

tool has passed this test. Obfuscation of the program was confirmed using an open-source program called ILSpy, which is a tool used to disassemble compiled code and convert it back into source code.

The sample code used is a segment of proprietary code. This means that while it cannot be displayed, it is a real and in-use example of a process prevalent in an automation company. It meets the following standards for a balanced test performance:

- The sample code does not rely on user input which may affect timings.
- The sample code uses multiple predicates (at least three) and can produce an error if not executed correctly.
- The sample code branches out into other classes and namespaces of the project both in input and output.
- The sample code is of reasonable length (20+ lines) and takes at least 100 ms on average to perform.
- The sample code does not use threading which would make timing checks unreliable.
- The non-obfuscated program performs a task with reliable consistency with no major outliers in timing.
- The program functions as intended, without error or crashing, with reliable consistency before obfuscation.

4.1.2 Obscurity and Resilience

Obscurity is referring to the human readability of the disassembled source code. Resilience refers to the obfuscated code's ability to resist disassembly or deobfuscation using malicious software or disassembly tools.

This is an analysis-based test and many aspects of each type of obfuscation are considered. For example, if Renaming obfuscation has been used: Are the characters limited to those found in the English language? Are extra symbols used? Is the renaming consistent? These would all be things to take note of when analysing and considering the level of obfuscation.

A basic C# sample code (Figure 9) was created for the purposes of giving a clear and clean example of code before and after obfuscation. The only thing this code

does is output the area of a rectangular prism of 3 units in length, 5 units in width, and 8 units in height, providing an output of “The calculated area of a rectangular prism is: 158” into a command prompt.

No Obfuscation

```
using System;

namespace ExampleApp
{
    class Program
    {
        public int RectangularPrismArea(int length, int width, int height)
        {
            if (length > 0 && width > 0 && height > 0)
            {
                int sides1 = 2 * length * width;
                int sides2 = 2 * length * height;
                int sides3 = 2 * width * height;
                return sides1 + sides2 + sides3;
            }

            return 0;
        }

        static void Main(string[] args)
        {
            var area = new Program();
            var number = area.RectangularPrismArea(3, 5, 8);
            Console.WriteLine("The calculated area of a rectangular prism is: ");
            Console.WriteLine(number);
        }
    }
}
```

Figure 9. Non-Obfuscated sample code

The sample code is simplistic but built in a particular way to test certain elements of renaming, control flow obfuscation, and string encryption/obfuscation, the three most common types of obfuscation. Math was intentionally introduced in sections to test if control flow obfuscation would break the necessary order of events to get a proper output. The rectangular prism’s measurements were tested to not be 0 in an “if” statement (despite not being necessary here) to ensure control flow obfuscation would not dismiss this check. Lastly, Console was used to output to see how the obfuscation would handle a system function, if it would make any effort to obfuscate such a base-level function and if String obfuscation would interact with the first line written, as it should always remain consistent.

4.1.3 Performance Cost

Like functionality preservation, the optimization of the program must also be considered. If obfuscation causes the quality of a product to drop too significantly, this causes new issues both in functionality and value to the customer. Some level of performance cost may be impossible to avoid, but not at the cost of making the product inferior to others on the market.

This test is measured by introducing an internal clock to process a task in the sample program ten times with, and without, obfuscation being introduced. These values will be calculated then into a percentage of expected estimated performance loss.

The sample code used during this test is the same present in the Functionality Preservation test for the sake of reliability, consistency, and real-world applicability.

4.1.4 Usability

Perhaps the most opinionative of these qualities, but important nonetheless, is that of Usability. In testing this, things such as the various forms of integration methods available, level of usability present in the program's UI (if it has one), and difficulty of configuration will all be considered.

The usability will have a criteria-based score given on a scale of zero to five for aspects of Integration, Configuration, and UI usability (appendix 1). Usability is generally highly subjective, but an analysis of a program's positives and negatives in an objective matter alongside a score for the sake of comparison between the tools may be helpful to less experienced developers or those with time constraints.

5 RESULTS

5.1 ConfuserEx2

ConfuserEx 2 is the successor to the Confuser and ConfuserEx projects. It is an open-source project which is intended for the general protection of .Net Framework projects from 2.0 to 4.8. Being an open-source project, update activity is prone to fluctuation. Its potential reliability is less when compared to a paid product, but at the exchange of having no monetary cost.

For testing purposes: Anti-Debug, Anti-ILDASM (an attribute which tries to deny disassembly through common tools such as ILSpy), Control Flow, Renaming, Anti-Tamper were all settings which were activated and used as their default settings.

5.1.1 Functionality Preservation and Usability

Functionality was preserved by ConfuserEx2, and a functional form of the program was achievable rather quickly after configuration and adjustment of settings. The program received the following scores:

UI: 3/5

- UI is simple and rather organized.
- Documentation is needed to understand the purpose of the various settings.
- Editing of configuration files outside of the UI may be necessary to achieve desired results.
- Build logging clearly shows what file is being worked on and with what form of obfuscation.

Configuration: 3/5

- Documentation is descriptive, though tricky to find as it is not listed on the tool's website, rather the GitHub the tool is downloaded from.
- Setting parameters are clearly explained, as well as possible incompatibilities, inside of the documentation. Remarks on certain settings give valuable hints for further customization and integration.
- Most settings can be modified.

- System comes with a list of pre-set protection options.

Implementation: 3.5/5

- Repeated obfuscations using the same settings produced slight variations, this is worse for testing and implementation but possibly better for security.
- Offers both a UI and MSBuild integration.
- Configurations are saved and easily passed.
- Stack tracing decoder tool is available within the program.
- Build produces no errors in full use.

5.1.2 Obscurity and Resilience

Features such as Anti-Tamper and Anti-ILDASM aided in making ILSpy and other deobfuscation tools more difficult to use because an error was produced when attempting to disassemble the code. This helped in both obscurity and resilience as, without further investigation, basic tools were not able to access the code. Obfuscation also varied with repeated attempts, which could improve the security of the application against deobfuscation tools.

Using ConfuserEx's obfuscation (Figure 10; appendix 2) the primary purpose of the code can still be seen with analysis. Emboldened are the renamed and altered, but still maintained, variables of the non-obfuscated code. For the most part, variables were properly renamed to something vaguer, such as "side" becoming "num" or "length" becoming "P_0". Unfortunately, this was not always the case, as the result variable was simply called "result" and gave an obvious hint as to the path of the code, despite the control flow being transformed.

Control flow was heavily changed. If not for having the original code, the path of execution may have been much tougher to follow in the calculation function. However, it was much easier to decipher is the main function, which was simply encased in a switch and two while cases.

System functions were also left untouched, as well as their strings, leaving behind the full intent of the code along with it. As such, an obvious hint was left behind by the program when it ignored changing the string of the WriteLine function.

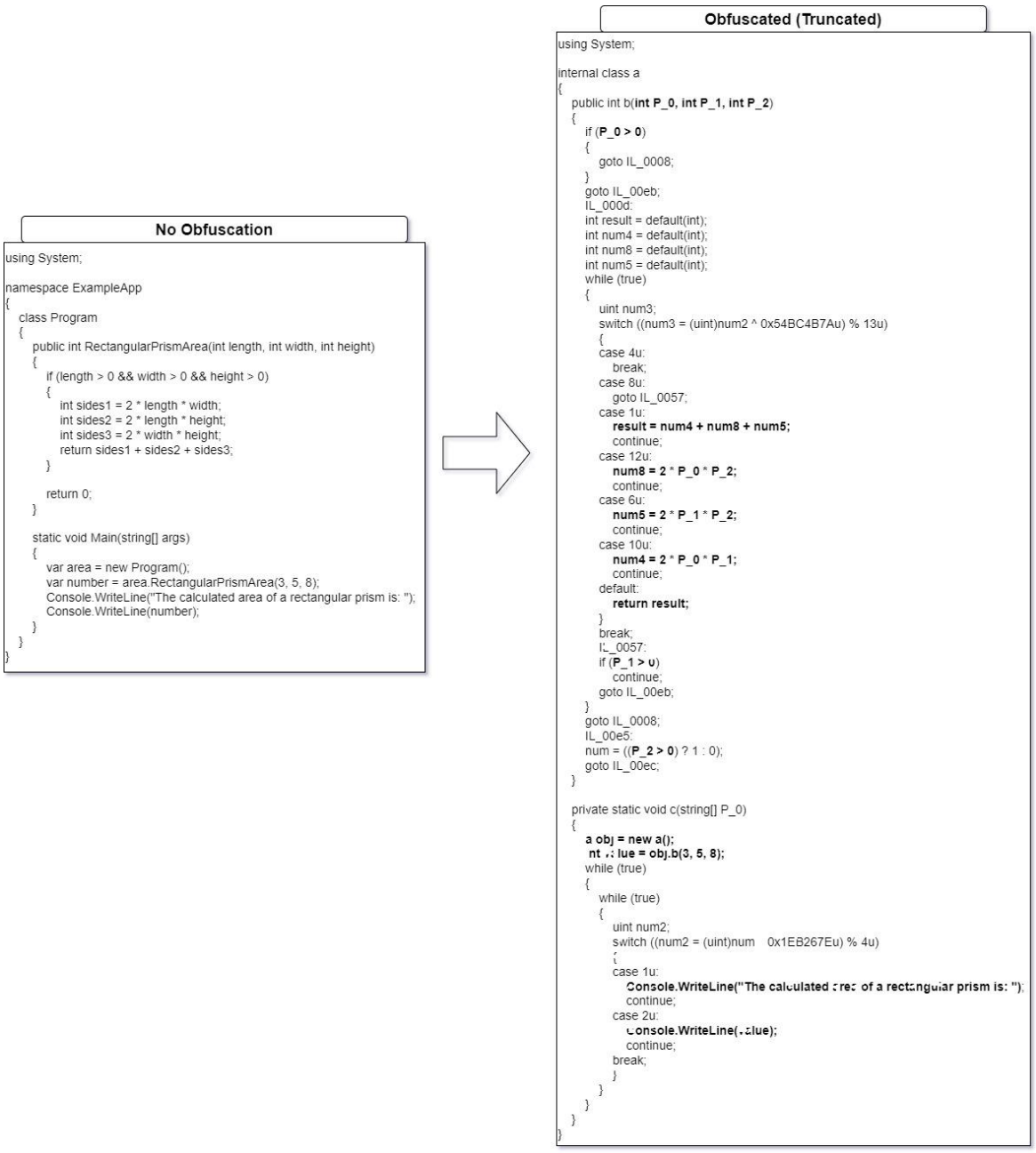

Figure 10. Truncated code obfuscated by ConfuserEx2

5.1.3 Performance Cost

Ten trials were run both with and without obfuscation of which Figure 11 shows the results. ConfuserEx performed the sample function at around **114.4** ms on average (**104** ms on average with no obfuscation). This is an estimated **10%** performance cost, which is quite high. ConfuserEx also had a range of **59** ms (**50** ms on average with no obfuscation), introducing slightly more inconsistency for performance.

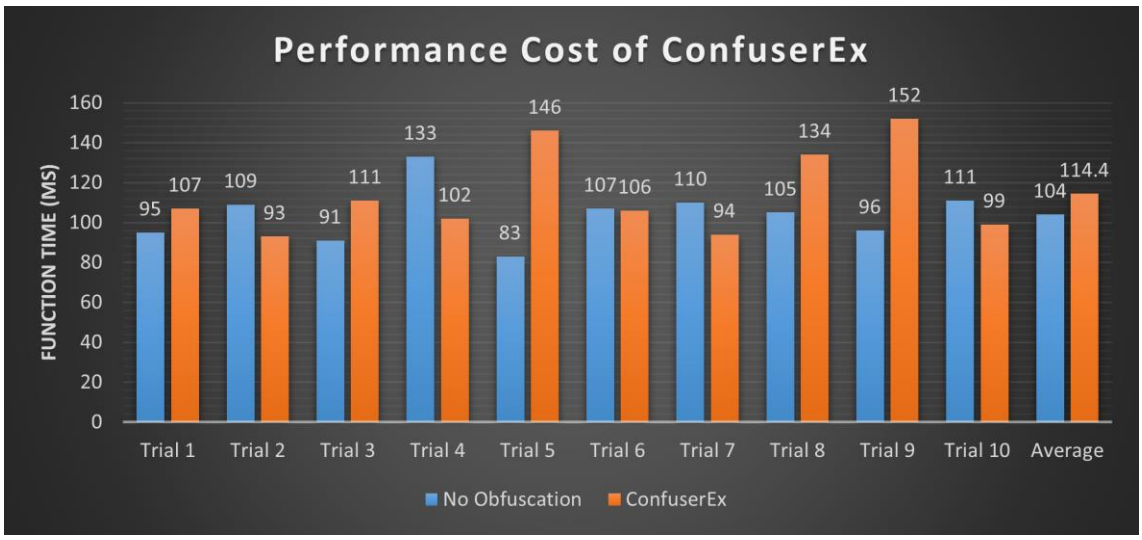

Figure 11. Performance cost of ConfuserEx vs normal performance

5.2 Eziriz's .Net Reactor

Eziriz's .Net Reactor is a versatile tool for both obfuscation and licensing of applications, with multiple forms of integration available, making it highly adaptive based on the environment. Currently this application is available for $200 for a single developer or $300 for a company-wide license.

.Net Reactor's Necrobit (unique to .Net Reactor and intended to prevent deobfuscation), Anti-ILDASM, Anti-Tamper, Anti-Debug, Control Flow, and String encryption features were all used during these tests.

5.2.1 Functionality Preservation and Usability

Functionality was preserved by .Net Reactor with default settings and implementation. The following user ratings were given of the tool's usability.

UI: 5/5

- Professional and well-organized UI.
- Protection building was clear and descriptive, along with options to make logging more verbose if necessary.
- Settings and options had provided descriptions within the UI itself.
- Usage of documentation was not necessary for usage of the program.

Configuration: 4.5/5

- Configuration to fit the required environment was quite easy, Configuration settings were clearly defined.
- Some settings were not completely clear as to their effect on their code, such as control flow obfuscation's level system.
- Documentation was not necessary for configuration.

Integration: 4/5

- Easily available through command line, UI, or through Visual Studio Addon.
- Build process was fast and produced no error within full use.
- Stack tracing decoder is available for use.
- Some level of optimization available through dead code removal.

5.2.2 Obscurity and Resilience

.Net Reactor has Anti-ILDASM, Anti-Tamper, and Necrobit features which inhibited deobfuscation tools. This caused the deobfuscated code, when using these features, to show odd results such as most functions returning only null which increased both obscurity and resilience.

Figure 12 features a highly truncated form of the obfuscated .Net Reactor code from the sample (appendix 3). Emboldened are the remains of the code which form the functions of the non-obfuscated code.

Renaming is quite like ConfuserEx2's sample, in that "side" is swapped to "num", and "result" is unfortunately correct. However, string obfuscation did take place, which helps greatly in creating confusion in the purpose of the code.

Control flow obfuscation was strong with this tool, the given example (Figure 12) is greatly truncated, and the original is long and loops through multiple different switches and while loops while still checking for nulls and performing the intended functions. The main function is also well obfuscated despite requiring less calculations than the area function.

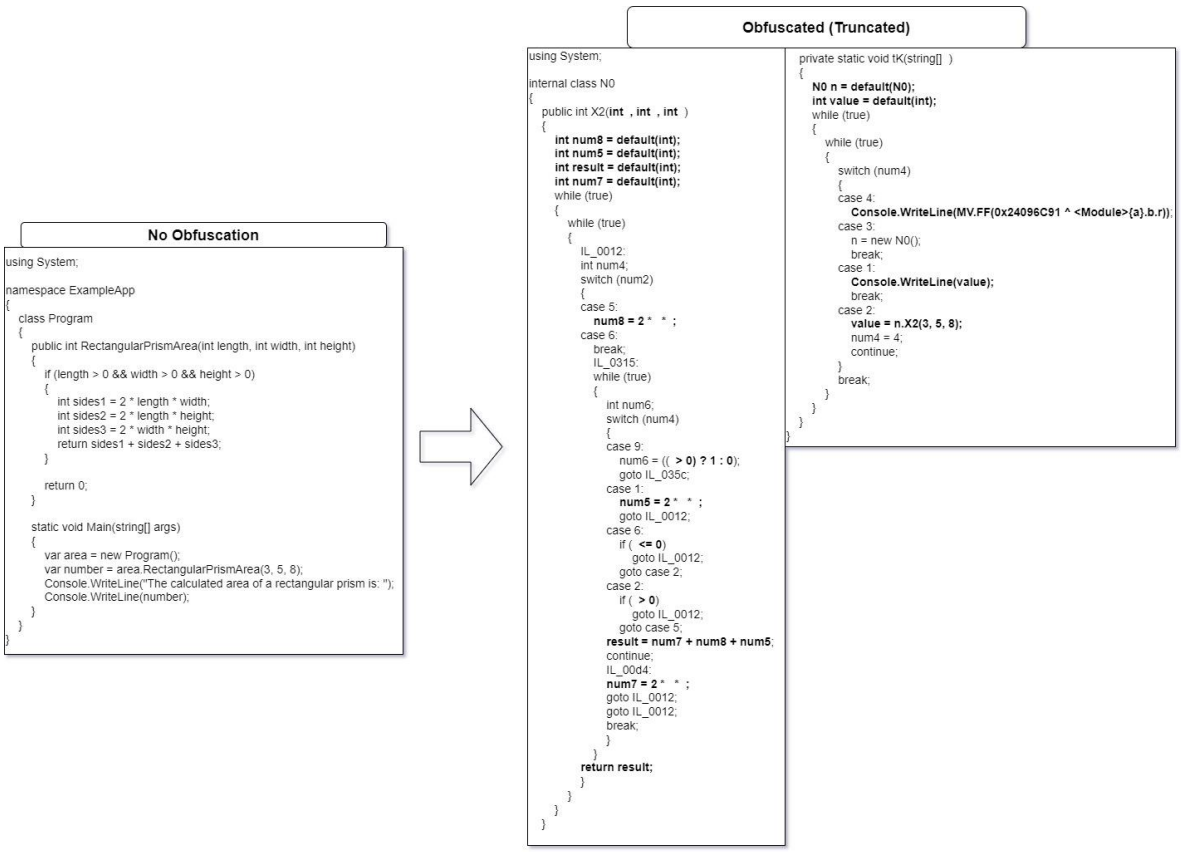

Figure 12. Truncated code obfuscated by .Net Reactor

5.2.3 Performance Cost

.Net Reactor performed the sample function at around **106.6** ms on average. Compared to normal functionality, this is an estimated **2.5%** performance cost. It also had a range of **42** ms, introducing slightly less inconsistency than the original function.

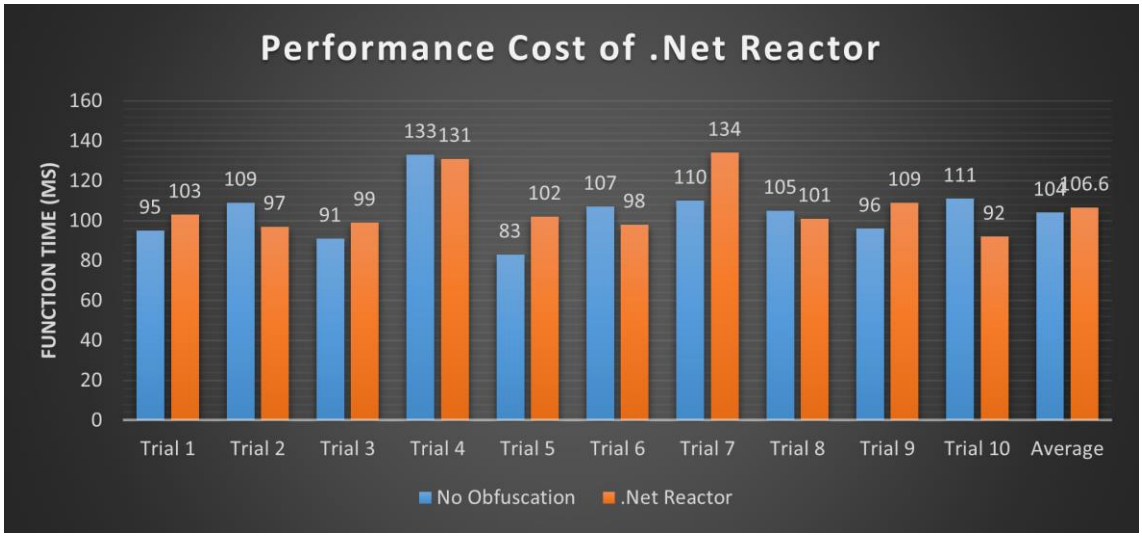

Figure 13. Performance cost of .Net Reactor vs normal performance

5.3 Eazfuscator

Eazfuscator focuses on ease of use and simplicity. Its UI is a simple drag-and-drop area which accepts both whole projects and assembled exe or dll files. Configuration is easy, as there is little of it, preferring to exclude classes, functions or namespaces through declarations which must be made within the source code. This program's licensing currently costs $400 for a single developer or $1700 for a site-wide license with unlimited developers.

5.3.1 Functionality Preservation and Usability

Functionality was preserved with prerequisite alteration of the code to ignore namespaces and classes which should not be obfuscated. The following user ratings were given on the tool's usability.

UI: 2/5

- The UI is simple, only using a drag-and-drop system.
- Possibility to drop either full projects or just assemblies.
- Build logging is unclear and only provides a default "Obfuscating assembly: 'ExampleApp.dll'" message.

Configuration: 0/5

- There are no configuration options.
- The only options are to remove certain files or classes by adding declarations inside of the code, meaning changes to the code are necessary for functionality preservation typically.
- No documentation available, as there are no settings to be changed.

Integration: 3/5

- Has options through both command-line, UI, and Visual Studio Addon.
- Configurations are not saved.
- Modification of base code is necessary for deployment of obfuscated code.
- Build process produced no errors in full use.
- Stack tracing decoder tool is available by default.

5.3.2 Obscurity and Resilience

There were no options nor built in protection against deobfuscation. ILSpy was uninhibited in its effort to disassemble the sample code.

Eazfuscator introduced rudimentary renaming obfuscation and string encryption. Renaming obfuscation and strings were both replaced with Unicode names. This may help more in systems reliant on more strings or printing functionalities, however, the original intent of the sample code is still noticeably clear and follows the same order.

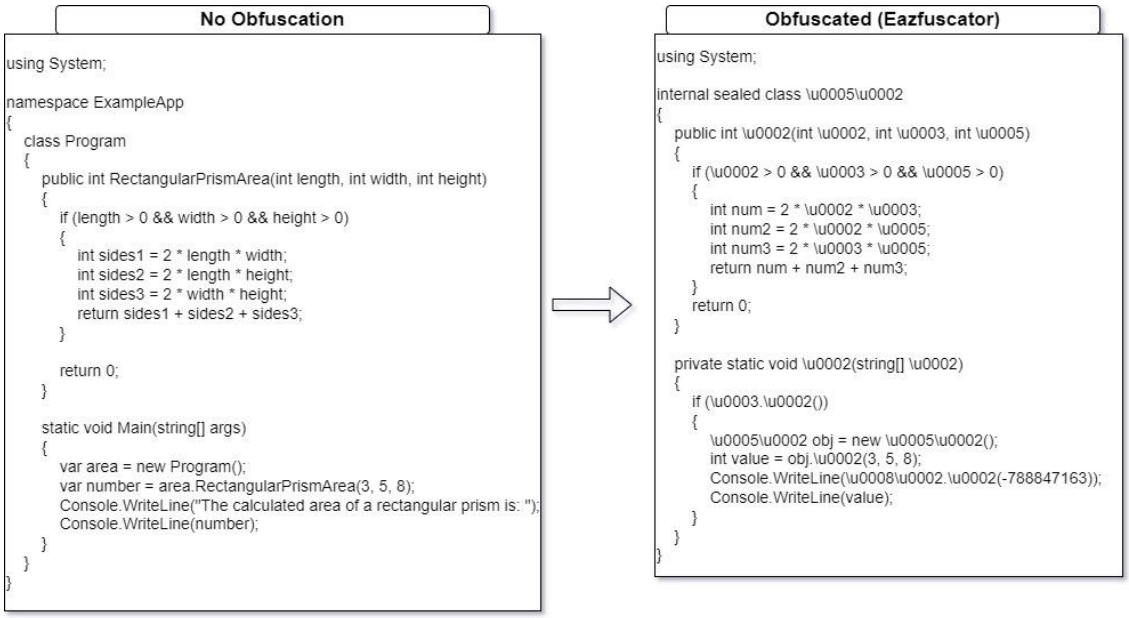

Figure 14. Code obfuscated by Eazfuscator

5.3.3 Performance Cost

Eazfuscator obfuscated code was performed ten times at around **106.4** ms on average, this is an estimated **2.31%** performance cost. Eazfuscator also had a range of **83** ms. This may be due to the outlier in Trial 7, without which the Range severely decreases.

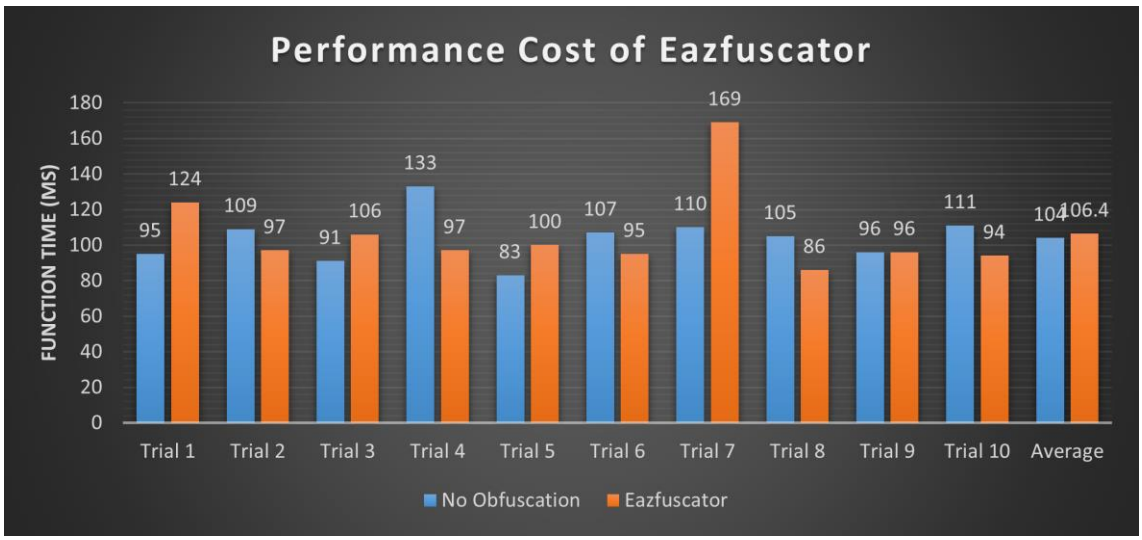

Figure 15. Performance cost of Eazfuscator vs normal performance

5.4 Babel

Babel focuses on a large amount of customization in its obfuscation methods. It also offers various other useful tools, such as the ability to merge or embed assemblies and obfuscate/deobfuscation stack traces, like debug obfuscation. This tool has license options for a professional version at 185 euros, 245 euros for an enterprise version, or 1250 euros for a site-wide license.

For these tests, the Renaming, Control flow, and String encryptions were set to their highest level. These were all the settings available to the trial version of the program.

5.4.1 Functionality Preservation and Usability

Functionality was preserved by Babel. The following user ratings were given regarding the usability of the tool.

UI: 5/5

- UI is well organized and professional.
- Obfuscation building includes statistics on what, and how much, was obfuscated of the given code.
- Building was very quick and verbose.
- Rules for each assembly can be set separate from others or settings can be set globally with exceptions in Rules.
- Features possible optimizations such as cleaning up attributes, removing dead code, etc.
- Tool required no assistance via documentation for full use.

Configuration: 4.5/5

- UI clearly defined what each configuration setting did, as well as allowing multiple different algorithms for certain obfuscation methods.
- Random seeds could be given to maintain some level of consistency in the obfuscation method throughout a system, or conversely set to random each time for increased security.
- All settings could be customized, including alternative algorithms to use for more control of the obfuscation.

- Documentation, while not typically needed, was relegated to a lengthy PDF.
- Some settings and algorithms may require usage of the documentation for understanding but were self-explanatory for the most part.

Integration: 5/5

- Easily available through command line, UI, or through Visual Studio Add-don.
- Several optimizations exist within the tool, some being used by default which may cut down on inconsistency of processes.
- Tool contains settings which assist with integration.

5.4.2 Obscurity and Resilience

Anti-ILDASM disassembly was prevalent and helped to protect the program from ILSpy. The full version of the program also features Anti-Reflection, Anti-Debugging, and Anti-tampering tools in addition to this.

Unfortunately, despite settings to include Control flow obfuscation, no opaque predicates were included. It simply introduced two new dummy functions which access and check the Date Time (Figure 16; appendix 4)

However, renaming obfuscation took place, doing the “side” to “num” swap seen in other obfuscation tools. Strings were also encrypted, as can be seen by WriteLine, which has also been introduced via a dummy code adding a bit extra intentional confusion to the string that was originally in place.

Overall, the original purpose of the call can be gleamed from the sample without too much effort, but obfuscation has taken place to make it less readable and to add extra steps which were not originally there and lead to potential confusion which could have greater effect in a larger code base.

No Obfuscation

```
using System;

namespace ExampleApp
{
    class Program
    {
        public int RectangularPrismArea(int length, int width, int height)
        {
            if (length > 0 && width > 0 && height > 0)
            {
                int sides1 = 2 * length * width;
                int sides2 = 2 * length * height;
                int sides3 = 2 * width * height;
                return sides1 + sides2 + sides3;
            }

            return 0;
        }

        static void Main(string[] args)
        {
            var area = new Program();
            var number = area.RectangularPrismArea(3, 5, 8);
            Console.WriteLine("The calculated area of a rectangular prism is: ");
            Console.WriteLine(number);
        }
    }
}
```

Obfuscated (Truncated)

```
using System;

internal class a
{
    public int a(int a, int b, int c)
    {
        if (a > 0 && b > 0 && c > 0)
        {
            int num = 2 * a * b;
            int num2 = 2 * a * c;
            int num3 = 2 * b * c;
            return num + num2 + num3;
        }
        return 0;
    }

    private static void Main(string[] a)
    {
        DateTime dateTime = new DateTime(*);
        if ((dateTime - DateTime.Now).TotalDays < 0.0)
        {
            int num = (-315578068 ^ -118937126) + -366575862;
            num = ((-(-180754128 + -185828599) + 340932503) ^ 0x2A2BD35F) / num;
        }
        DateTime dateTime2 = new DateTime(*);
        if (DateTime.Now > dateTime2 || 1 == 0)
        {
            throw new ArgumentException();
        }
        a a2 = new a();
        int value = a2.a(3, 5, 8);
        Console.WriteLine(b.a("\ue05d\ue061\ue06c", 57353));
        Console.WriteLine(value);
    }
}
```

Figure 16. Truncated code obfuscated by Babel

5.4.3 Performance Cost

The obfuscated sample code was performed ten times at around **104.6** ms on average, which is an estimated **0.58%** performance cost. Babel also had a range of **39** ms meaning that it possibly introduced more consistency in its calculations than the default performance possibly due to optimizations it makes.

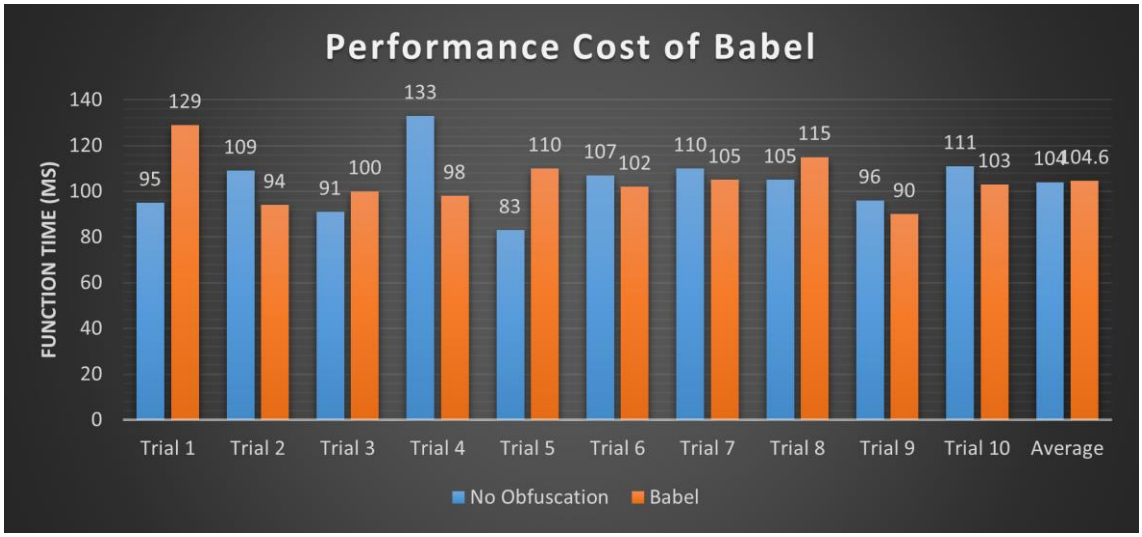

Figure 17. Performance cost of Babel vs normal performance

5.5 Comparison

Some tools performed clearly better than others due to any number of factors such as being intended for a different environment than the one used in testing or due to having more restricted free/trial versions of the program. It is best to perform these tests within one's own environment for a more exact fitting of what suits that system best. However, in the environment listed in the earlier Testing Plan section, the following information was found.

5.5.1 Functionality Preservation and Usability

All programs kept the base functionality after obfuscation. As said previously, this was a complete pass or fail test due to the necessity of any obfuscation tool never changing or removing necessary features from the code being obfuscated.

Regarding Usability, **Babel** performed the best overall, very closely followed by .Net Reactor. These are the rankings of the tested tools with their overall score out of 15:

- **Babel** – 14.5/15
- **.Net Reactor** – 13.5/15
- **ConfuserEx2** – 9.5/15
- **Eazfuscator** – 5/15

5.5.2 Obscurity and Resilience

For the obscurity and resilience analyses each tool had its obfuscated code de-obfuscated using the tool ILSpy. The level of obscurity, if it were to be ranked, would be as follows:

- **.Net Reactor** – Code was obfuscated, and unreadable characters were introduced in certain parts, which could not be said for any of the other tools.
- **ConfuserEx2** – The original intent of the code was challenging to decipher. However, not having string encryption left key details that gave away the purpose of the code.
- **Babel** – While still missing opaque predicates, it introduced dummy code which deceptively appeared to have a purpose while not doing anything to severely harm performance.

- **Eazfuscator** – Very little was done outside of renaming and string encryption to prevent access to the code or revise readability.

5.5.3 Performance Cost

Overall, the best performance cost is **Babel**, having both the best Range and performance cost, meaning it introduces optimizations which cut down on performance inconsistency while also not highly increasing the cost. In order of performance cost, the tools are:

- **Babel** – 0.58% performance cost, 39ms range
- **Eazfuscator** – 2.31% performance cost, 83ms range
- **.Net Reactor** – 2.5% performance cost, 42ms range
- **ConfuserEx2** – 10% performance cost, 59ms range

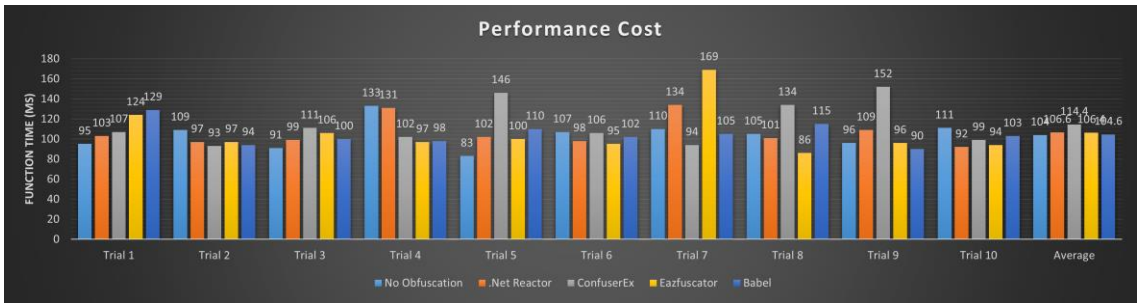

Figure 18. Performance cost of all tools

Table 1. Performance cost statistics

Tool	*Max*	*Min*	*Average*	*Cost %*	*Range*
No obfuscation	133	83	104	-	50
ConfuserEx	152	93	114.4	10%	59
.Net Reactor	134	92	106.6	2.5%	42
Eazfuscator	169	86	106.4	2.31%	83
Babel	129	90	104.6	0.58%	39

6 DISCUSSION

In this thesis, obfuscation methods and other notable protections were outlined and described. Numerous examples outlined how each method worked using both theoretical and real examples of obfuscated code. Considerations to bear in mind when implementing each method were discussed individually, as well as how each method can pair or clash alongside one another.

Tamper-Proofing and Watermarking were briefly discussed, and the general idea of protection outside of obfuscation was mentioned and analysed. A testing plan was drafted, and the qualities sought after in an obfuscation tool were outlined and tests were planned around those given qualities.

Four tools were chosen (ConfuserEx2, .Net Reactor, Eazfuscator and Babel) which fit the environment present within an automation company. These tools were tested for: Functionality preservation, usability, obscurity/resilience, and performance cost.

When considering all aspects of the tests, Eziriz's .Net Reactor performed the best out of the four tested. While Babel's performance cost was lower and overall usability was marginally better, the obfuscation methods it employs lead to only a slight detriment to the overall readability of the code (in the trial version). Meanwhile, .Net Reactor performed well with performance, exceptionally well with Usability, and was best in terms of obscurity.

Many people still fervently discuss the validity of obfuscation as a form of security, many opponents of which say that obscurity can be eventually overcome through analysis and studying of the obfuscated code. Proponents of obfuscation argue that, like locking the front door of our homes, the basic level of entry has been raised and since ultimately any security method can be defeated, it is as valid as any other.

This thesis proposes that obfuscation can be achieved for little cost while presenting a worth-while deterrent. Peace-of-mind of one's livelihood is particularly important, especially so in competitive fields such as automation.

REFERENCES

Collberg, C. & Thomborson, C. 1999. Software watermarking: models and dynamic embeddings. Association for Computing Machinery

Collberg, C. & Thomborson, C. 2000. Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection. University of Arizona & University of Auckland. Read on 27.9.2022. https://www.cs.auckland.ac.nz/~cthombor/Pubs/tsewmtpobf.pdf

Kirovski, D. & Malvar, H. 2003. Spread-Spectrum Watermarking of Audio Signals. Institute of Electrical and Electronics Engineers, Inc.

Merriam-Webster. N.d. Obfuscate. Dictionary. Read 28.8.2022. https://www.merriam-webster.com/dictionary/obfuscate

Niagra, J. & Collberg, C. 2009. Surreptitious Software: Obfuscation, Watermarking, and Tamper-proofing for Software Protection. Pearson Education.

PreEmptive Solutions, LLC. N.d. Control Flow. Website. Read 23.9.2022. https://www.preemptive.com/dotfuscator/pro/userguide/en/protection_obfuscation_control_flow.html

PreEmptive Solutions, LLC. N.d. Renaming. Website. Read 7.11.2022. https://www.preemptive.com/dotfuscator/pro/userguide/en/protection_obfuscation_renaming.html

Ross, R., Pillitteri, V., Graubart, R., Bodeau, D., & McQuaid, R. 2021. Developing Cyber-Resilient Systems: A Systems Security Engineering Approach. Volume 2. U.S. Department of Commerce.

APPENDICES

Appendix 1. Usability Test Criteria

Usability Criteria

Assessment Target	None (0)	Poor (1)	Satisfactory (2)	Good (3)	Excellent (4)	Perfect (5)
User Interface (UI)	• There is no user interface available	• Settings and tools are disorganized • Editing of configuration files is still necessary to achieve desired results	• UI is professional in appearance • It is possible to modify most configurations within the UI • Documentation is needed to achieve desired results • Build logging exists but may be unclear on what issues may remain	• UI is professional and organized • Configuration settings are clearly named but documentation may still be necessary • Build logging shows clear errors when they occur	• UI is well organized • Configuration settings have clear names • Documentation is not needed but may aid in achieving desired results • Build logging shows file being worked on and with what protection	• UI is very well organized • Configuration options show descriptions • Tool can be used with absolutely no assistance of documentation • Statistics of obfuscation are visible • Clear and verbose build logging
Configuration	• There are no configuration options available • No documentation is available • There are no settings which can be modified	• Most configuration options do not work by default • Documentation is hard to find • The code itself must be modified to fit the tool	• All configuration options work by default or potential incompatibilities are outlined • Documentation is readily available • Some settings can be customized	• Tool introduces some level of variation with repeated use • Most settings can be customized • System comes with preset options for settings	• The tool attempts to correct incompatibility issues itself • Tool introduces toggle-able variation between repeated use • All settings can be customized	• Configuration is simple and straight-forward • Documentation is not necessary but readily available. • All settings customized and well explained
Integration	• Integration is not possible or is extremely difficult • Usage of the tool requires large modifications to the existing code	• Heavy modifications necessary for deployment of obfuscated code • Errors may be produced, even with sample code • Only one method of implementation is available	• Build process produced no errors in basic usage • Configurations are not saved • More than one method of implementation is available	• Configurations are saved and easily passed between systems • Build process produces no errors in full use	• Visual Studio Add-on is available • Support is available from provider • Stack tracing deobfuscation tool is available for use	• Tools exist in program to assist with integration • Optimizations are available for use

Appendix 2. Obfuscated ConfuserEx Sample

Obfuscated (ConfuserEx)

```
using System;

internal class a
{
    public int b(int P_0, int P_1, int P_2)
    {
        if (P_0 > 0)
        {
            goto IL_0008;
        }
        goto IL_00eb;
        IL_00eb:
        int num = 0;
        goto IL_00ec;
        IL_00ec:
        bool flag = (byte)num != 0;
        int num2 = 1602884422;
        goto IL_000d;
        IL_000d:
        int result = default(int);
        int num4 = default(int);
        int num8 = default(int);
        int num5 = default(int);
        while (true)
        {
            uint num3;
            switch ((num3 = (uint)num2 ^ 0x54BC4B7Au) % 13u)
            {
            case 4u:
                break;
            case 8u:
                goto IL_0057;
            case 1u:
                result = num4 + num8 + num5;
                num2 = ((int)num3 * -2120314644) ^ 0x22649DE4;
                continue;
            case 9u:
                num2 = (int)((num3 * 1506059200) ^ 0x7364912A);
                continue;
            case 7u:
                num2 = ((int)num3 * -252602041) ^ -1692560988;
                continue;
            case 12u:
                num8 = 2 * P_0 * P_2;
                num2 = (int)(num3 * 559680490) ^ -2004583213;
                continue;
            case 3u:
                result = 0;
                num2 = 744354000;
                continue;
            case 0u:
                num2 = (int)((num3 * 2018347205) ^ 0x718AD90F);
                continue;
            case 2u:
                goto IL_00e5;
            case 11u:
            {
                int num6;
                int num7;
                if (!flag)
                {
                    num6 = 2140987606;
                    num7 = num6;
                }
                else
                {
                    num6 = 1304434495;
                    num7 = num6;
                }
                num2 = num6 ^ ((int)num3 - -286719267);
                continue;
            }
            case 6u:
                num5 = 2 * P_1 * P_2;
                num2 = ((int)num3 * -1518019195) ^ 0x4F0FEC4A;
                continue;
            case 10u:
                num4 = 2 * P_0 * P_1;
                num2 = ((int)num3 * -1370689115) ^ 0x2146161F;
                continue;
            default:
                return result;
            }
            break;
            IL_0057:
            if (P_1 > 0)
            {
                num2 = ((int)num3 * -170045661) ^ 0x21BB4BB9;
                continue;
            }
            goto IL_00eb;
        }
        goto IL_0008;
        IL_00e5:
        num = ((P_2 > 0) ? 1 : 0);
        goto IL_00ec;
        IL_0008:
        num2 = 1533892135;
        goto IL_000d;
    }
```

```
    private static void c(string[] P_0)
    {
        a obj = new a();
        int value = obj.b(3, 5, 8);
        while (true)
        {
            int num = 1904030271;
            while (true)
            {
                uint num2;
                switch ((num2 = (uint)num ^ 0x1EB267Eu) % 4u)
                {
                case 3u:
                    break;
                default:
                    return;
                case 1u:
                    Console.WriteLine("The calculated area of a rectangular
prism is: ");
                    num = ((int)num2 * -1998630717) ^ -1962245381;
                    continue;
                case 2u:
                    Console.WriteLine(value);
                    num = ((int)num2 * -1075282255) ^ -136791212;
                    continue;
                case 0u:
                    return;
                }
                break;
            }
        }
    }

    public a()
    {
        while (true)
        {
            int num = -929268207;
            while (true)
            {
                uint num2;
                switch ((num2 = (uint)num ^ 0xC1A89880u) % 3u)
                {
                case 2u:
                    break;
                default:
                    return;
                case 1u:
                    goto IL_0028;
                case 0u:
                    return;
                }
                break;
                IL_0028:
                num = (int)(num2 * 1976432474) ^ -2039587032;
            }
        }
    }
}
```

Appendix 3. Obfuscated .Net Reactor Sample

Obfuscated (.Net Reactor)

```
using System;
using hJ3;
using HQ;
using Q5;
using RwZ;

internal class N0
{
    internal static object Ve;

    internal static N0 tJy;

    public int X2(int  , int  , int  )
    {
        int num = 12;
        int num3 = default(int);
        int num8 = default(int);
        bool flag = default(bool);
        int num5 = default(int);
        int result = default(int);
        int num7 = default(int);
        while (true)
        {
            int num2 = num;
            while (true)
            {
                IL_0012:
                int num4;
                switch (num2)
                {
                case 10:
                    num4 = 3;
                    goto IL_0315;
                case 9:
                    num4 = 4;
                    goto IL_0315;
                case 1:
                    num4 = 0;
                    if (<Module>{a}.b.c == 0)
                    {
                        num4 = 0;
                    }
                    goto IL_0315;
                case 7:
                    num4 = 9;
                    goto IL_0315;
                case 2:
                    num4 = 0;
                    if (<Module>{a}.b.g == 0)
                    {
                        num2 = 0;
                        if (<Module>{d}.e.f == 0)
                        {
                            num2 = 10;
                        }
                        continue;
                    }
                    goto IL_0315;
                case 3:
                    num4 = 0;
                    if (<Module>{a}.b.h == 0)
                    {
                        num2 = 9;
                        continue;
                    }
                    goto IL_0315;
                default:
                    num4 = 5;
                    goto IL_0315;
                case 4:
                    num4 = 7;
                    goto IL_0315;
                case 12:
                    num3 = 6;
                    num2 = 11;
                    continue;
                case 5:
                    num8 = 2 *  * ;
                    num4 = 1;
                    if (<Module>{a}.b.i == 0)
                    {
                        num4 = 1;
                    }
                    goto IL_0315;
                case 8:
                    num4 = 12;
                    goto IL_0315;
                case 11:
                    num4 = num3;
                    goto IL_0315;
                case 6:
                    break;
                IL_0315:
                    while (true)
                    {
                        int num6;
                        switch (num4)
                        {
                        case 3:
                            if (flag)
                            {
                                num4 = 11;
                                continue;
                            }
                            goto IL_018f;
                        case 4:
                            break;
                        case 8:
                            goto IL_018f;
                        case 9:
                            num6 = ((  > 0) ? 1 : 0);
                            goto IL_035c;
                        case 1:
                            num5 = 2 *  * ;
                            num2 = 3;
                            goto IL_0012;
                        case 6:
                            if (  <= 0)
                            {
                                num2 = 0;
                                if (<Module>{d}.e.j == 0)
                                {
                                    num2 = 0;
                                }
                                goto IL_0012;
                            }
                            goto case 2;
                        case 2:
                            if (  > 0)
                            {
                                num2 = 7;
                                if (<Module>{d}.e.k != 0)
                                {
                                    num2 = 2;
                                }
                                goto IL_0012;
                            }
                            goto case 5;
                        case 0:
                            goto end_IL_0315;
                        default:
                            num2 = 5;
                            goto IL_0012;
                        case 7:
                        case 10:
                        case 12:
                            goto end_IL_0012;
                        case 5:
                            {
                                num6 = 0;
                                goto IL_035c;
                            }
                            IL_035c:
                            flag = (byte)num6 != 0;
                            num2 = 2;
                            if (<Module>{d}.e.l == 0)
                            {
                                num2 = 1;
                            }
                            goto IL_0012;
                        }
                        result = num7 + num8 + num5;
                        num4 = 1;
                        if (<Module>{a}.b.m != 0)
                        {
                            goto end_IL_0012_2;
                        }
                        continue;
                        IL_00d4:
                        num7 = 2 *  * ;
                        num2 = 1;
                        if (<Module>{d}.e.n == 0)
                        {
                            num2 = 0;
                        }
                        goto IL_0012;
                        IL_018f:
                        result = 0;
                        num2 = 8;
                        if (<Module>{d}.e.o != 0)
                        {
                            num2 = 3;
                        }
                        goto IL_0012;
                        continue;
                        end_IL_0315:
                        break;
                    }
                    goto case 5;
                    end_IL_0012:
                    break;
                }
                return result;
                continue;
                end_IL_0012_2:
                break;
            }
            num = 4;
        }
    }
}
```

Obfuscated (.Net Reactor)

```
private static void tK(string[] )
{
    int num = 4;
    int num2 = num;
    int num3 = default(int);
    N0 n = default(N0);
    int value = default(int);
    while (true)
    {
        int num4;
        switch (num2)
        {
        case 1:
            num4 = 0;
            if (<Module>{a}.b.p == 0)
            {
                num2 = 2;
                continue;
            }
            break;
        case 2:
            num4 = 0;
            break;
        default:
            num4 = 2;
            break;
        case 4:
            num3 = 3;
            num2 = 1;
            if (<Module>{d}.e.q == 0)
            {
                num2 = 3;
            }
            continue;
        case 3:
            num4 = num3;
            break;
        }
        while (true)
        {
            switch (num4)
            {
            case 4:
                Console.WriteLine(MV.FF(0x24096C91 ^ <Module>{a}.b.r));
                num4 = 1;
                if (<Module>{a}.b.s != 0)
                {
                    num4 = 1;
                }
                continue;
            default:
                return;
            case 0:
                return;
            case 3:
                n = new N0();
                num2 = 0;
                if (<Module>{d}.e.t != 0)
                {
                    num2 = 0;
                }
                break;
            case 1:
                Console.WriteLine(value);
                num2 = 0;
                if (<Module>{d}.e.u == 0)
                {
                    num2 = 1;
                }
                break;
            case 2:
                value = n.X2(3, 5, 8);
                num4 = 4;
                continue;
            }
            break;
        }
    }
}
```

```
public N0()
{
    OJ1();
    ia();
    base..ctor();
    int num = 2;
    if (<Module>{d}.e.v != 0)
    {
        num = 0;
    }
    int num2 = default(int);
    while (true)
    {
        int num3;
        switch (num)
        {
        case 2:
            num3 = 0;
            if (<Module>{a}.b.w == 0)
            {
                num3 = 0;
            }
            break;
        case 1:
            return;
        default:
            num3 = num2;
            break;
        }
        switch (num3)
        {
        case 0:
            return;
        }
        num = 1;
        if (<Module>{d}.e.x == 0)
        {
            num = 1;
        }
    }
}

internal static bool tN()
{
    return Ve == null;
}

internal static N0 dk()
{
    return (N0)Ve;
}

internal static void ia()
{
    Kwf.owj();
}

internal static bool SJC()
{
    return tJy == null;
}

internal static N0 RJF()
{
    return tJy;
}

internal static void OJ1()
{
    tJn.Irm();
}
}
```

Appendix 4. Obfuscated Babel Sample

Obfuscated (Babel)

```
using System;

internal class a
{
    public int a(int a, int b, int c)
    {
        if (a > 0 && b > 0 && c > 0)
        {
            int num = 2 * a * b;
            int num2 = 2 * a * c;
            int num3 = 2 * b * c;
            return num + num2 + num3;
        }
        return 0;
    }

    private static void Main(string[] a)
    {
        DateTime dateTime = new DateTime(--507320940 - 507318917,
        ((0x2646C78C ^ 0x7CD46BC) >> 4) - 35174418, -(~(112 >> 4)), (--116131872 >> 1) + -58065927, ~(-524239247 - -524161391 >> 5) >> 7,
        (-235817087 ^ 0x1DE1E9F5) + 334471828 >> 3);

        if ((dateTime - DateTime.Now).TotalDays < 0.0)
        {
            int num = (-315578068 ^ -118937126) + -366575862;
            num = ((-(-180754128 + -185828599) + 340932503) ^ 0x2A2BD35F) / num;
        }
        DateTime dateTime2 = new DateTime(--129472 >> 6, ((0xD71136F ^ 0x11B32877) + -356699064 >> 3) - 15724011, (-418627933 ^ 0x17AC9A40) + 257907492);
        if (DateTime.Now > dateTime2 || 1 == 0)
        {
            throw new ArgumentException();
        }
        a a2 = new a();
        int value = a2.a(3, 5, 8);
        Console.WriteLine(b.a("\ue06e\ue07c\ue065\ue068\ue07b\ue029\ue079\ue07b\ue060\ue07a\ue064\ue029\ue060\ue07a\ue033\ue029", 57353));
        Console.WriteLine(value);
    }
}
```

Obfuscated (Babel)

```
using System;
using System.IO;
using System.Reflection;

internal sealed class b
{
    private delegate string a();

    private sealed class b
    {
        private static readonly a a;

        public static readonly b b;

        private byte[] m_c;

        static b()
        {
            a = global::b.b;
            b = new b();
        }

        private b()
        {
            Stream manifestResourceStream = Assembly.GetExecutingAssembly().GetManifestResourceStream(a());
            if (((manifestResourceStream == null) ? (-((-925211434 >> 1) - -335542695) + -127063021) : ((~((-(1913667842 + 104474114) >> 1) ^ 0x1C602C9B) ^ -612326228) - -70927434)) == 0)
            {
                this.m_c = new byte[~(-(~((~(1289357402 - 74953515) >> 2) - -303600954)))];
                manifestResourceStream.Read(this.m_c, ~(-(0x296DC6D5 ^ 0x296DC6D4)), this.m_c.Length);
            }
        }

        public string c(string a, int b)
        {
            int num = a.Length;
            char[] array = a.ToCharArray();
            while ((num -= (~(-(1556579721 - 170843424)) >> 2) + -346434073) >= ~(~(-422567354 + -187584031) + -309686959 << 1) + 600928851)
            {
                array[num] = (char)(array[num] ^ (this.m_c[b & -(~((-396589203 ^ 0x66622BB) - -298146360))] | b));
            }
            return new string(array);
        }
    }

    public static string a(string a, int b)
    {
        DateTime dateTime = default(DateTime).AddYears(~(~-639533677) + 639535699).AddMonths(-(-396042765 + 396042765 << 3)).AddDays(7.37184027777778);
        if ((dateTime - DateTime.Now).TotalDays < 0.0)
        {
            throw new ArgumentOutOfRangeException();
        }
        return global::b.b.b.c(a, b);
    }

    public static string b()
    {
        char[] array = "';\b:\n*".ToCharArray();
        int num = array.Length;
        while ((num -= -(-(~-1107079082 + -411495917) >> 2) ^ 0xA5D706E) >= 0 << 5 >> 7)
        {
            array[num] = (char)(array[num] ^ (-(~(0x219C39B4 ^ -108090644)) + 669871378));
        }
        return new string(array);
    }
}
```