

Obfuscation technologies of high-level source code using artificial intelligence

Igor Golovko^{1,†}, Oleg Savenko^{1,†}, Petro Vizhevskiy^{1,†}, Olexandr Klein^{1,†}, Abdel-Badeeh M. Salem^{2,†}

¹ Khmelnytskyi National University, 11 Institutaska Street, Khmelnytskyi, 29000, Ukraine

² Ain Shams University, Egypt

Abstract

This article provides an in-depth exploration of "Obfuscation Technologies of Source Code," focusing on the latest advancements in methodologies to safeguard intellectual property in software. It meticulously analyzes several key obfuscation techniques, including Identifier Renaming, Control Flow Obfuscation, and the strategic insertion of Dead or Junk Code. Each technique is detailed in terms of its implementation, benefits, and the specific aspects of software security it enhances. The research further introduces a significant innovation through the integration of Artificial Intelligence (AI) in the obfuscation process. AI is leveraged to dynamically optimize obfuscation patterns and predict the most effective techniques tailored to specific software environments, which marks a considerable improvement over traditional methods that often require manual intervention and are prone to errors. The article substantiates these advancements with a theoretical framework that models the effectiveness of obfuscation strategies using advanced machine learning algorithms. These models assess the resilience of obfuscated code against reverse engineering, providing a quantitative basis for the enhancements in security measures. This comprehensive discussion not only sheds light on current practices but also sets the stage for future research and application in software security, making it an essential resource for developers and cybersecurity experts dedicated to enhancing the robustness of software protection.

Keywords

Code Obfuscation, Software Security, Artificial Intelligence in Security, Source Code Protection Strategies.

1. Introduction


In the digital era, where software becomes an integral part of nearly every industry, the protection of intellectual property assumes a special significance. Code obfuscation, as a method of safeguarding software from reverse engineering, plays a pivotal role in ensuring the security and confidentiality of the developed product. This process involves transforming the primary

ICyberPhyS-2024: 1st International Workshop on Intelligent & CyberPhysical Systems, June 28, 2024, Khmelnytskyi, Ukraine

* Corresponding author.

† These authors contributed equally.

✉ <mailto:i85.golovko@gmail.com> (I. Golovko); mailto:savenko_oleg_st@ukr.net (O. Savenko); petro.vizhevskiy@gmail.com (P. Vizhevskiy); <mailto:olexandrkleyn@gmail.com> (O. Klein); <mailto:abmsalem@yahoo.com> (Abdel-Badeeh M. Salem);

 0009-0004-2173-5126 (I. Golovko); 0000-0002-4104-745X (O. Savenko); 0009-0009-4851-0839 (P. Vizhevskiy), 0000-0002-1896-943X (O. Klein); 0000-0003-0268-6539 (Abdel-Badeeh M. Salem)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

code of the program into a form [1] that makes reverse engineering difficult, while still preserving its functionality.

In contemporary programming, one of the greatest challenges is the protection of commercial secrets and innovations which are critical to a company's competitiveness. The leakage or unauthorized copying of software can lead to substantial financial losses and undermine the company's market position. The advancement of internet technologies and multimedia has heightened the need for research in the field of protection and security. Every organization, possessing its own intellectual property, faces the challenge of protecting its data, such as software from piracy or the injection of malicious code [2, 3].

Code obfuscation stands out as one of the advanced techniques in the domain of software protection, which involves transforming the initial code of the program in such a way that, while it becomes difficult to understand, it loses none of its functional properties [4, 5]. This complicates the process of extracting and utilizing important algorithms and procedures that are part of the software product, thus preserving the confidentiality of data processing through programs that is critically important for the business of software purchasers. Even if an obfuscated code can be deciphered by a persistent attacker, integrating obfuscation with other methods, such as code modification detection or protection updates, limits the time available to achieve malicious objectives.

Intellectual property protection can be secured both legally and technically. While legal protection involves obtaining copyrights and signing contracts against the creation of duplicates, technical protection requires developers to implement protective mechanisms directly into the software. Together, these strategies form a multi-layered approach to program protection, which is crucial for ensuring long-term security in the digital age.

2. Main Obfuscation Techniques

Before diving into the obfuscation process, let's briefly look at a very simplified model of the compilation process of high-level programming languages (C#/Java). First of all we need to understand basic definitions in this process.

Source code – Intermediate Language code "IL code" is a stack-based assembly language and serves as the output of the compilation of high-level .NET/Java languages.

JIT – Just-In-Time (JIT) compiler is a component of the runtime environment that compiles "IL code" to native machine code at run time.

Let's assume that we have written the code on C# language and want to obfuscate it. Schema below describe this process.

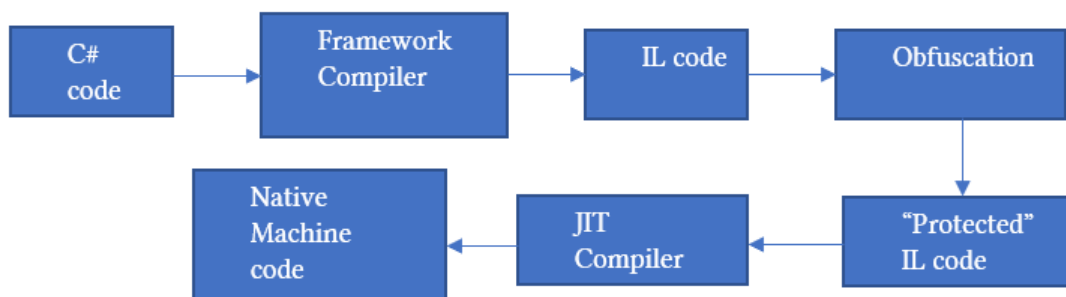


Figure 1: simplified compilation scheme with obfuscation.

On this schema we can see Post-Compilation Obfuscation process that helps us to protect our source code. Let's look at which obfuscation techniques can be used for that. Code obfuscation includes a series of techniques that make the software code more complex to analyze and understand while preserving its functionality [6]. Here are some of the principal obfuscation techniques commonly employed in software engineering.

Identifier Renaming (Renaming) involves changing the names of variables, classes, methods, and other identifiers to non-informative or random names. This complicates the understanding of the program since semantic information that could assist in decrypting the purpose of code components is lost.

Control Flow Obfuscation is a technique that modifies the logic of program execution in such a way that it retains functionality but makes the code less comprehensible. For example, the use of fake loops or unnecessary conditional statements makes the execution flow of the code less clear [7, 8].

Insertion of Dead or Junk Code adds code that does not affect the final behavior of the program but complicates its structure. This code can include non-executable instructions or functions that lead nowhere.

String Encryption enables the encrypting text strings in the code such as error messages, URLs, or other sensitive data. It prevents the easy extraction of information from executable files.

Resource Obfuscation: Protecting program resources such as images, audio files, and other assets by encrypting or modifying them.

Control Flow Obfuscation is a technique that introduces changes in the program's control flow to complicate code analysis [27-30]. Instead of direct and obvious execution, the program is reorganized in such a way that the logic of its execution becomes more complex and less predictable for observers or analytical tools. This may include the introduction of false loops, dead code blocks, changes in the execution order of instructions, or the use of conditional statements that appear illogical or mixing control flows (Interleaving Code Paths): Merging several functions or execution paths into one, complicating the separation and analysis of individual components. Transparent Branches is a conditional statements that always execute or never execute, which misleads analytical tools.

These methods can be used individually or in combinations to achieve a higher level of code security. The choice of specific obfuscation methods depends on the specific requirements and context of the program's use.

2.1. Identifier Renaming Method in Code Obfuscation

Identifier renaming is one of the most popular and effective code obfuscation techniques. This method involves changing the names of variables, functions, classes, and other identifiers to names that carry no semantic load. The goal of this method is to create confusion or mislead anyone trying to perform reverse engineering or unlawfully use the code.

Renaming identifiers is based on replacing semantically meaningful names with ones that are random or unintelligible. For example, a variable storing an intermediate result in calculations, originally named tempResult, might be renamed to a1 or x47[9]. This complicates understanding what the code does and reduces the possibility of its analysis on an intuitive level.

Manual and Automatic: Identifier renaming can be implemented both manually and

automatically using specialized obfuscation tools. Automatic renaming programs analyze the code and replace names using a generation of random character sequences or by using a predefined set of non-informative names [10].

Static Renaming: During static renaming, each unique identifier is assigned a new name that remains unchanged throughout the project. This is a simpler but less flexible approach.

$$S \rightarrow F \rightarrow P, \quad (1)$$

Where: S – Set of original code identifiers. P – Set of obfuscated code identifiers. F – Transformation function.

Thus, for each value "x" in the set S, there exists a unique value "y" in the set P satisfying the condition: $F(x)=y$

Dynamic Renaming: In dynamic renaming, new names can change depending on the context(C – Context function) in which the identifier is used, adding an additional layer of complexity for those attempting to understand the program logic[11].

$$S \rightarrow F \rightarrow C \rightarrow P, \quad C(F(x)) = y' \rightarrow C(y) = y' \quad (2)$$

Identifier renaming is an important obfuscation strategy used to protect software code from unauthorized access and analysis. Despite some limitations, this technique is one of the most common approaches in the software industry due to its effectiveness and ease of implementation.

2.2. Insertion of Dead or Junk Code

The technique of inserting "dead" or "junk" code forms an integral part of the strategies used to complicate the reverse engineering process of software. This method incorporates code segments that, while non-functional concerning the program's outcome, enhance the complexity of the software structure, making it arduous for unauthorized interpretation or analysis. Such code may include inert instructions or purposeless functions that do not impact to the primary functionality of the program.

Examples of "Dead" Code:

1. Superfluous variables and computations.

```
int a = 10;
int b = a * 2; // An extraneous variable and computation that remain unused
```

2. Non-functional loops:

```
for (int i = 0; i < 10; i++) {
    // A loop that performs no meaningful action within the program
}
```

3. Always-true conditional statements:

```
if (true) {
    // This block of code will invariably execute
}
```

Developers may employ automated obfuscation tools to randomly intersperse such code within the source code, thereby mitigating pattern recognition strategies that could potentially

identify and excise the redundant code. These tools also ensure that the integration of additional code does not disrupt the core logic or performance of the application. With this approach, we can improve the following indicators:

- Increased Analytical Complexity. The insertion of "dead" code significantly muddles the structural clarity of the software, thus thwarting straightforward analytical efforts by potential attackers.
- Versatility. This method is universally applicable across various programming languages and software architectures, enhancing its utility in diverse developmental contexts.

2.3. Control Flow Obfuscation

Control Flow Obfuscation is a sophisticated technique aimed at complicating the understanding of a program's logic by altering the order of operations and instructions, as well as by introducing additional conditional transitions and loops[12, 13]. This method seeks to obscure the true execution path of a program, thereby hindering analysis and reverse engineering efforts [14,15]. One of the options of the Control Flow is interleaving Code Paths.

Interleaving code paths is an advanced obfuscation technique that modifies the execution structure of a program such that logically independent blocks of code are interwoven. This significantly complicates the understanding of the program, as both analytical tools and humans struggle to easily separate individual execution streams. This method involves intertwining several functional parts of the code together, creating a single entangled execution flow that is difficult to separate into primary components. This can be achieved by crossing conditional operators, loops, and functions across different parts of the program.

For example, consider the interleaving of conditional operators and loops:

```
if (conditionA) {
    // Block A1
    if (conditionB) {
        // Block B1
    }
    // Block A2
} else {
    if (conditionB) {
        // Block B2
    }
    // Block A3
}
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        if (i == j) {
            // Mixed operation
        }
    }
}
```

In these examples, the blocks of code and conditions are intertwined in such a way that the logical and execution flows of conditions A and B, as well as the loops i and j, interact with each other in a complex manner, making the code analysis more challenging. Implementing this method can be challenging as it requires a deep understanding of the program's logic and potential impact on performance. Developers must ensure that changes in control flows do not violate the business logic of the application or affect its performance. Automated obfuscation tools can aid in the implementation of this method, but it is crucial to conduct thorough testing. Constructing a mathematical model for this method is not straightforward. Such a model would need to utilize concepts from graph theory[17] and complexity theory to analyze and evaluate the impact of obfuscation on code comprehension.

The model includes the following aspects:

1. Definition of Control Flow Graph (CFG)[17]. The basis for analyzing any program code is its Control Flow Graph (CFG), where nodes represent blocks of instructions (such as functions or basic instruction blocks) and edges show the flow of control between those blocks. CFG allows you to visualize and analyze the structure of the program [16].

Let $G = (N,E)$, where each node $n \in N$ corresponds to a base node. Each edge $e=(ni,nj) \in E$ corresponds to a possible transfer of control from block ni to block nj .

CFG provides a graphical representation of the possible paths to control the flow of execution. It differs from syntax-oriented IRS such as AST, which show grammatical structure. Consider the while loop shown below.

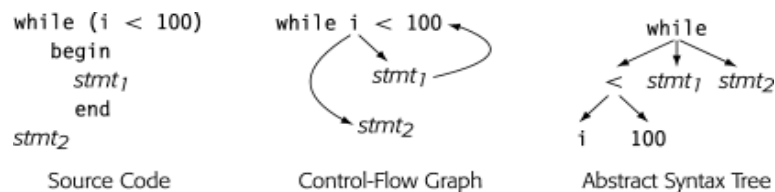


Figure 2: The loop operator in different representations.

The CFG reflects the essence of the loop: it is a control flow construct. The cyclic edge goes from $stm1$ to the condition at the beginning of the cycle. Ast, on the other hand, fixes the syntax; it is acyclic, but puts all the pieces in place to restore the source code for the loop. For conditional statements, the CFG will look like presented in Figure 3.

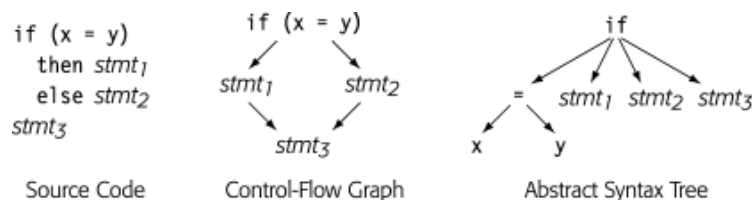


Figure 3: The condition operator in different representations.

In this example, the CFG displays the control flow construction for the conditional statement. Either $stm1$ or $stm2$ will be executed, but not both.

2. Functional mixing of streams. For each node in the CFG, a function can be defined that describes the mixing of control flows. This function can take into account variable factors such as the nesting depth of conditional statements or the number of dependencies between different parts of the code.

Let's assume D_{ij} - represents the dependency between node i and node j . This can be expressed as the weight of an edge in a graph, where the weight indicates the strength of the dependency (for example, due to the number of variables shared between blocks). N_i (Nesting of conditional operators): evaluates the nesting depth of conditional operators in node i . This can be expressed as the number of conditional statements that directly or indirectly affect the execution of a block of code.

Then for estimating the complexity of mixing flows can be presented as follows:

$$S = \sum_{i=1}^n \sum_{j=1, j \neq i}^n D_{ij} \times N_i \quad (3)$$

where: S - is the total complexity of mixing control flows in the program. n - is the number of nodes in the CFG.

This function attempts to quantify the complexity of a program in terms of obfuscation, taking into account dependencies and nesting of conditional statements.

It can be supplemented by other factors, such as:

- Frequency of use of variables: Consider how often variables affecting node i occur in other nodes. For each node i in the CFG, we define F_i , which indicates the frequency of use of variables in this node. This can take into account both local and global variables used in the block.
- Function side effects: evaluating the impact of functions called in a node on other parts of the program. E_i evaluates the impact of functions called at node i on other parts of the program. This can include state changes that are not obvious from the local context of the node, such as changes to global variables, calls to other functions, etc.

Therefore, the final model can be represented as:

$$S = \sum_{i=1}^n \left(\sum_{j=1, j \neq i}^n (D_{ij} \times N_i) + \alpha \times F_i + \beta \times E_i \right) \quad (4)$$

Where: α and β are weighting factors that regulate the influence of the frequency of use of variables and side effects of functions, respectively; n is the number of nodes in the CFG.

Variable usage frequency F_i shows how heavily a node depends on certain variables, which can make it difficult to understand the data flows in the program. The side effects of functions E_i allow you to evaluate how much the changes made by the functions affect the global state of the program, which also increases the overall complexity of the code.

This model can be used to evaluate the effectiveness of obfuscation in terms of its ability to complicate code analysis. It allows you to quantify how changes in the structure of the application affect the ability of analysts or attackers to understand the logic of the application and detect vulnerabilities.

3. Quantification of complexity. Using metrics to quantify the complexity of mixed control flows, such as: McCabe cyclomatic complexity [18,19], which measures the number of linearly independent paths through a CFG. Proposed by Thomas McCabe in 1976, is a metric that measures the number of linearly independent routes through a program's control flow graph (CFG). This is one of the key indicators that helps to understand the complexity of the application from the point of view of its testing and maintenance.

Formula for calculating cyclomatic complexity:

$$V(G)=E-N+2P \quad (5)$$

Where: E - is the number of edges in the graph; N - is the number of nodes in the graph; P - is the number of connectivity components (usually $P=1$ for most programs with a single entry point).

Therefore, the cyclomatic complexity due to the introduction of obfuscation can be given by the function:

$$\Delta V(G)=V(G')-V(G) \quad (6)$$

Where $V(G)$ i $V(G')$ – cyclomatic complexities of the original and obfuscated graphs, respectively.

Such a model helps to evaluate how effective obfuscation is in terms of increasing the complexity of the program. If $\Delta V(G)$ is significant, it can be assumed that obfuscation makes a significant contribution to protecting the program from unauthorized analysis and modifications. This model can serve as an important tool when selecting and configuring obfuscation techniques, as well as when evaluating their impact on the overall security of a software product.

The number of intersections in the CFG, where a higher number of intersections may indicate a more complex obfuscation structure.

4. Predicting the impact of obfuscation. Using statistical methods to predict the effectiveness of obfuscation:

- Building regression models to predict the effort required to understand obfuscated code based on the aforementioned complexity metrics.
- Simulation of different attack scenarios on obfuscated code to evaluate its resistance to reverse engineering.

5. Risk assessment. Analysis of the possible risks associated with obfuscation, including the probability of successful reverse engineering or obfuscation detection [20]. This may involve using probability theory and statistics to assess risks.

Let V be the set of nodes in the CFG, and E be the set of edges.

The function $f: V \rightarrow R$ evaluates the "weight" of each node in terms of its impact on the overall complexity.

Then the complexity of the code C can be expressed as:

$$C = \sum_{v \in V} f(v) + \lambda \cdot |E| \quad (7)$$

where: λ – a parameter that controls the effect of the number of intersections.

Such a model allows you to evaluate, analyze and optimize code obfuscation, providing a science-based approach to software protection.

3. Improvement of the obfuscation process with AI

As we can discern from the previous section, many processes require the engineer to independently decide on the obfuscation method, conduct performance testing, etc., which is not always the most efficient or error-free approach to obfuscation, particularly for engineers with limited experience in code obfuscation. In such cases, utilizing artificial intelligence (AI) can significantly enhance the effectiveness of obfuscation techniques, even for engineers with minimal experience [21]. The idea of employing obfuscation mechanisms based on machine learning can be applied in the .NET obfuscation sphere to model obfuscation strategies, i.e., using machine learning algorithms to generate and optimize obfuscation rules that can be applied to .NET code. The model can learn from existing examples of obfuscated code to identify the most effective techniques.

The machine learning model can predict the effectiveness of various obfuscation methods using the following process:

1. Model Training. The model trains on examples of code (using machine learning algorithms such as random forest [22,23] or gradient boosting [24,25]) that have been obfuscated using different methods. It learns the characteristics of the code (e.g., structure, execution flows, variable usage) that change as a result of each obfuscation method.
2. Obfuscation Assessment. Using a set of metrics such as resistance to reverse engineering, impact on performance, or effects on automated code analysis tools, the model evaluates the effectiveness of the obfuscation [28-31].
3. Prediction. After analyzing the input code, the model can use the learned relationships between code features and obfuscation effectiveness to predict which methods will be most effective for new code.

Thus, the model allows for the identification of optimal obfuscation strategies for specific use cases, providing better protection and minimizing negative impacts on software functionality.

The mathematical model for assessing the effectiveness of obfuscation using machine learning can be constructed as follows:

1. Data: - X : A set of code features (e.g., number of operators, depth of nesting, types of operators). - Y : The target (dependent variable) which determines the effectiveness of obfuscation (e.g., time required for reverse engineering).
2. Loss Function. Can be defined to minimize the difference between the predicted effectiveness of obfuscation and the actual effectiveness.
3. Model. Uses a machine learning model $f(X)$ to learn the relationship between code features and obfuscation effectiveness.
4. Optimization. Uses optimization methods to adjust the model parameters that best explain the effectiveness of obfuscation.

$$\min_{\theta} \sum (y_i - f(X_i; \theta))^2 \quad (8)$$

where: θ are the parameters of the model that we aim to optimize [26,27], X_i , are the features of the i -th code example, y_i , is the effectiveness of obfuscation for the i -th example, and the sum is calculated over all examples in the training set.

The term \min_{θ} signifies an optimization process where the goal is to find the parameter values θ that minimize the sum of the squared differences between observed values y_i and the values predicted by. The aim of \min_{θ} is to adjust the parameters θ to achieve the lowest possible value of the sum of squared errors, indicating the best fit of the model to the data. This process is central to regression analysis, where you want to fit a model so that the predicted values are as close as possible to the actual data values.

This modeling also enhances the automation process - integrating machine learning will allow for the automation of the obfuscation process, adapting it to specific needs and characteristics of the software as well as potential threats. Additionally, it can create dynamic code obfuscation processes - machine learning methods can help develop systems that dynamically adapt obfuscation depending on the context of software usage and changes in the external environment.

4. Experiments

To verify the effectiveness of the model, we will use the following metrics:

- Resistance to Analysis - an assessment of the code's ability to resist reverse engineering attempts.

$$S = 1 - \frac{K}{M} \quad (9)$$

where: K – number of successful analyses, M -total number of attempts

- Change in Performance - the impact of obfuscation on the speed of the program.

$$P = \frac{R - L}{L} \quad (10)$$

Where: R – execution time after obfuscation, L – execution time before obfuscation

- Preservation of Functionality:

$$F = \frac{N}{M} \quad (11)$$

where: N – number of dysfunctional functions, M – total number of functions.

- Pattern Detection:

$$D = 1 - \frac{N}{M} \quad (12)$$

where: N – number of detected patterns, M – total number of patterns

This metric is important because one of the main aspects of effective obfuscation is complicating or masking the logic or structure of the code so that it cannot be easily analyzed or recognized by static analysis tools, which often use patterns to identify typical constructions in program code. This expression shows the percentage of patterns that were not detected during the analysis, and therefore, the higher the value of D, the more effective the obfuscation in terms of avoiding pattern detection.

- Code Complexity:

$$C = R - L \quad (13)$$

where: R – cyclomatic complexity after obfuscation, L - cyclomatic complexity before obfuscation.

This metric helps assess the complexity of understanding and testing the code. High cyclomatic complexity indicates a high level of code complexity, which can increase the risk of errors and complicate understanding of the code. In the context of code obfuscation, the goal is to increase this complexity, thereby making the code less understandable for analysis or reverse engineering. This initial complexity indicator is important for assessing the effectiveness of obfuscation. An increase in cyclomatic complexity after obfuscation typically indicates that the obfuscation has added additional control paths, thereby potentially increasing the security of the program by complicating reverse engineering attempts. For reverse engineering and code analysis, we will use two tools:

1. Ildasm.exe [32].
2. dotPeek [33].

To verify the effectiveness of the model, 100 dll/exe files compiled using MSBuild with .NET 8 programming language C# were used. Divide these DLLs into two groups (50/50): control (without AI) and experimental (with AI). Apply standard obfuscation methods to the control group without using AI. Calculate the average values for each metric for 50 iterations of the control group (without AI) and the experimental group (with AI). We are going to analyze such parameters:

Resistance to Analysis (S):

- Total number of reverse engineering attempts: 100.
- Number of successful analyses: 20 – By successful analyses is meant the full reproduction of the program's behavior after decompiling IL code using Ildasm/dotPeek and transferring it to a new program that fully retains the behavior of the original program, and reproduces the same results as the original program.
- Percentage of unsuccessful attempts: $S = 1 - 20/100 = 0.80$ or 80%.

Change in Performance (P): average program execution time: 200 ms.

Preservation of Functionality (F):

- Total number of functions: 1000.
- Number of dysfunctional functions: 0.

Pattern Detection (D):

- Total number of patterns: 50.
- Number of detected patterns: 30.
- Percentage of undetected patterns: $D=1 - 30/50 = 0.40$ or 40%.

Code Complexity (C). Cyclomatic complexity: 150 – means 150 different paths that potentially need to be checked to ensure full coverage during testing, making the code more complex to fully understand and support. The results of the experiment shown in Table 1.

Table 1

The results of the experiment

Metric	Description	Control group (without AI)	Experimental group (with AI)	Comment
Resistance to Analysis	An assessment of the code's ability to resist reverse engineering attempts.	Number of successful analyses: 20 Percentage of unsuccessful attempts: $S = 1 - 20/100 = 0.80$ or 80%	Number of successful analyses: 5 Percentage of unsuccessful attempts: $S = 1 - 5/100 = 0.95$ or 95%	As we can see resistance to analysis is increased when using AI.
Change in Performance	The impact of obfuscation on the speed of the program.	Average program execution time: 210 ms Change in Performance: $P = (210 - 200)/200 = 0.05$ or 5% increase	Average program execution time: 210 ms Change in Performance: $P = (210 - 200)/200 = 0.05$ or 5% increase	The execution time of the program has changed compared to the original program without obfuscation. But this is also true for the control group (without AI)
Preservation of Functionality	A measure of the preservation of the original functionality of the code	Number of dysfunctional functions: 10 Percentage of dysfunctional	Number of dysfunctional functions: 12 Percentage of dysfunctional	As we can see, the number of dysfunctional functions is slightly higher compared to the control group (without AI). This

	after obfuscation.	functions: $F=10 / 1000$ = 0.01 or 1%	functions: $F = 12 / 1000$ = 0.012 or 1.2%	percentage can be reduced if the AI model is allowed to learn on its own or if the training period is extended.
Pattern Detection	The ability of obfuscation tools to avoid pattern detection.	Number of detected patterns: 30 Percentage of undetected patterns: $D = 1 - 30/50$ =0.40 or 40%	Number of detected patterns: 5 Percentage of undetected patterns: $D = 1 - 5/50$ =0.90 or 90%	

5. Conclusions

The experimental results provide compelling evidence supporting the integration of AI in the obfuscation process, underscoring its potential to significantly enhance software security.

Each of these metrics helps assess specific aspects of obfuscation, and their comparison before and after the application of AI allows measuring the real use impact of the of artificial intelligence on obfuscation. This also provides an opportunity to identify potential issues, such as increased execution time or loss of functionality, requiring additional attention and optimization. This approach allows for more precise adjustment of the use of AI for optimization of obfuscation in real conditions, ensuring a higher level of security of software. AI can analyze large volumes of data and choose optimal places and ways to apply obfuscation to maximize code complexity. After analyzing the metrics of the experiment, it is possible to distinguish:

1. Enhanced Efficacy of AI-Driven Obfuscation. The experiments demonstrated a notable improvement in resistance to reverse engineering attempts when AI-driven obfuscation techniques were employed compared to traditional methods. This indicates that AI can effectively increase the complexity and security of obfuscated code, making it more challenging for unauthorized analysis.
2. Performance and Functionality Consideration. While the use of AI in obfuscation shows promising results in enhancing security, it's important to also consider its impact on software performance and functionality. The experiments highlighted minimal impact on execution times and functionality, suggesting that AI-driven obfuscation can be implemented without significantly compromising the software's operational efficiency.

This approach to security can significantly reduce the costs and resources associated with resolving security issues after a product is released. Future research should explore additional AI models and techniques that could further enhance this aspect of software security. As these technologies become more sophisticated and available, we can expect changes in how companies approach the security of their software products. This change could encourage more

industries to adopt obfuscation best practices, thereby increasing the overall level of security across sectors.

References

- [1] K. D. Cooper, L. Torczon. *Engineering a Compiler*, Morgan Kaufmann; 3rd edition, 2023, 848 p.
- [2] S. A. Ebad, A. A. Darem, J. H. Abawajy. Measuring software obfuscation quality – A systematic literature review, *IEEE Access* 9 (2021) 99024-99038.
- [3] P. Ahire, J. Abraham. Mechanisms for source code obfuscation in C: Novel techniques and implementation. In *Proceedings of the 2020 International Conference on Emerging Smart Computing and Informatics (ESCI)*, Pune, India, 12-14 March 2020, IEEE: New York, NY, USA, 2020, pp. 52–59.
- [4] A. B. M. Sultan, A. A. Ghani, N. M. Ali, N. I. Admodisastro. Hybrid obfuscation technique to protect source code from prohibited software reverse engineering, *IEEE Access* 8 (2020) 187326–187342.
- [5] S. Bhansali, A. Aris, A. Acar, H. Oz, A. S. Uluagac. A first look at code obfuscation for webassembly. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, San Antonio, Texas, USA, 16-19 May 2022, pp. 140–145.
- [6] Y. Li, Z. Sha, X. Xiong, Y. Zhao. Code Obfuscation Based on Inline Split of Control Flow Graph. In *Proceedings of the 2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, Dalian, China, 28–30 June 2021, IEEE: New York, NY, USA, 2021, pp. 632–638.
- [7] C. K. Behera, G. Sanjog and D. L. Bhaskari. Control Flow Graph Matching for Detecting Obfuscated Programs, *Software Engineering* (2019) 267–275.
- [8] Y.-C. Chen, H.-Y. Chen, T. Takahashi, B. Sun and T.-N. Lin, Impact of Code Deobfuscation and Feature Interaction in Android Malware Detection, *IEEE Access*, 9 (2021) 123208-123219
- [9] B. Liu, W. Feng, Q. Zheng, J. Li, D. Xu. Software obfuscation with non-linear mixed boolean-arithmetic expressions. In *Proceedings of the Information and Communications Security: 23rd International Conference, ICICS 2021, Chongqing, China, 19–21 November 2021*, pp. 276–292.
- [10] C. Catalano, P. Afrune P., et al. Security Testing Reuse Enhancing Active Cyber Defence in Public Administration. In *Proceedings of the 2021 Italian Conference on Cybersecurity 2021, April 7-9, 2021, Salerno, Italy*, pp.120–132 (2021).
- [11] C. Catalano, A. Chezzi, M. Angelelli, F. Tommasi. Deceiving AI-based malware detection through polymorphic attacks, *Computers in Industry* 143 (2022) 103751.
- [12] H. Ahmed, M. F. Hyder, M. F. Haque, P. C. Santos, Exploring compiler optimization space for control flow obfuscation, 139 (2024) 103704.
- [13] M. Gervasi, N. G. Totaro, A. Fornaio, D. Caivano, Big Data Value Graph: enhancing security and generating new Value from Big Data, In *Proceedings of the 2023 Italian Conference on Cybersecurity 2023, May 03-05, 2023, Bari, Italy, 2023*.
- [14] M. Schloegel, T. Blazytko, M. Contag, C. Aschermann, J. Basler, T. Holz, A. Abbasi. A Technical Report: Hardening Code Obfuscation Against Automated Attacks. *arXiv* (2021), arXiv:2106.08913.
- [15] P. Rajba, W. Mazurczyk, Data hiding using code obfuscation. In *Proceedings of the 16th International Conference on Availability, Reliability and Security, Vienna, Austria, 17-20 August 2021*, pp. 1–10.

- [16] H. Yao, S. Zhang, R. Hong, Y. Zhang, C. Xu and Q. Tian, Deep representation learning with part loss for person re-identification, *IEEE Trans. Image Process.*, 28 6 (2019) 2860-2871.
- [17] Q. Liu, S. Ji, C. Liu and C. Wu, A Practical Black-Box Attack on Source Code Authorship Identification Classifiers, In *Proceedings of the IEEE Transactions on Information Forensics and Security*, 15 June 2021, vol. 16, pp. 3620-3633.
- [18] J. Mayaka, J. C. Jung, Complexity reduction of the Engineered Safety Features Component Control System, 331 (2018) 194-203.
- [19] M. A. Subandri, R. Sarno, Cyclomatic Complexity for Determining Product Complexity Level in COCOMO II, 124 (2017) 478-486.
- [20] C. Basile, D. Canavese, L. Regano, P. Falcarin, B. De Sutter, A meta-model for software protections and reverse engineering attacks, *Journal of Systems and Software*, 150 (2019) 3-21.
- [21] I. Obeidat, M. AlZubi, Developing a faster pattern matching algorithms for intrusion detection system. *International Journal of Computing*, 18(3), 2019, 278-284. doi:10.47839/ijc.18.3.1520
- [22] S. Kang, S. Lee, Y. Kim, S. K. Mok, E. S. Cho. Obfus: An obfuscation tool for software copyright and vulnerability protection. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, Virtual, 26–28 April 2021, pp. 309–311.
- [23] H. Chen, M. Pendleton, L. Njilla and S. Xu. A survey on Ethereum systems security: Vulnerabilities attacks and defenses, *ACM Comput. Surv. (CSUR)*, 53 3 (2020) 1-43.
- [24] F. Feyzi and S. Parsa, A program slicing-based method for effective detection of coincidentally correct test cases, *Computing*, 100 9 (2018) 927-969.
- [25] M. Zhang, P. Zhang, X. Luo and X. Feng, Source code obfuscation for smart contracts, In *Proceedings of the 2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, 01-04 December 2020, Singapore, Singapore, pp. 513-514.
- [26] G. James, et al. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2nd edition, 2021.
- [27] K. Hajarnis, J. Dalal, R. Bawale, J. Abraham and A. Matange, A Comprehensive Solution for Obfuscation Detection and Removal Based on Comparative Analysis of Deobfuscation Tools. In *Proceedings of the 2021 International Conference on Smart Generation Computing, Communication and Networking*, Pune, India, 2021, pp. 1-7.
- [28] O. Savenko, A. Sachenko, S. Lysenko, G. Markowsky, N. Vasylykiv. Botnet detection approach based on the distributed systems. *International Journal of Computing*, 19, 2 (2020) 190-198.
- [29] A. Kashtalian, S. Lysenko, O. Savenko, A. Nicheporuk, T. Sochor, V. Avsiyevych. Multi-computer malware detection systems with metamorphic functionality. *Radioelectronic and Computer Systems*, 1 (2024) 152-175. doi: 10.32620/reks.2024.1.13
- [30] G. Markowsky, O. Savenko, S. Lysenko, A. Nicheporuk. The technique for metamorphic viruses' detection based on its obfuscation features analysis. *CEUR-WS*, 2104 (2018) 680–687.
- [31] O. Savenko, S. Lysenko, A. Nicheporuk, B. Savenko, Approach for the Unknown Metamorphic Virus Detection, *Proceedings of the 8-th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, Bucharest (Romania), September 21–23, 2017. Bucharest, 2017. pp. 71–76.
- [32] Ildasm.exe (IL Disassembler) tool. URL: <https://learn.microsoft.com/en-us/dotnet/framework/tools/ildasm-exe-il-disassembler>.
- [33] dotPeek. Free .NET Decompiler and Assembly Browser. URL: <https://www.jetbrains.com/decompiler>.