Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X Volume 3, Issue 8, 2025

A COMPARATIVE ANALYSIS OF CONTROL FLOW OBFUSCATION TECHNIQUES TO PROTECT SOFTWARES FROM REVERSE ENGINEERING

Muhammad Arham Khawar[1], Seemab Tariq[2], Zia ur Rehman[*3], Muhammad Haroon[4]

[1,2,4]*Department of Cyber Security, Air University, Islamabad, Pakistan;*
[*3]*Department of Computer Science, National University of Modern Languages (NUML), Islamabad, Pakistan;*

[1]arhamkhawar3331@gmail.com, [2]seemabtariq126@gmail.com, [*3]zrehman@numl.edu.pk
[4]muhammad.haroon@au.edu.pk

DOI:https://doi.org/10.5281/zenodo.17786380

Keywords
Obfuscation, Flattening, Splitting, Virtualization, Graph, Protection

Article History
Received 15 June 2025
Accepted 15 August 2025
Published 26 August 2025

Corresponding Author: *
Zia ur Rehman

Abstract
Code obfuscation software protection is an area of paramount importance, which dynamically changes its significance with regard to modern endeavors to ensure the protection of intellectual property against unauthorized access and reverse engineering. The main benefit of obfuscation, overall, is the massive increase in the effort needed to analyze proprietary binaries, which will aid in deterring bad actors and maintain the integrity of commercial software. However, it has been observed that clear and measurable metrics remain absent which can help developers settle on an optimal combination of obfuscation techniques which provides a reasonable level of protection at moderate performance costs. In particular, the research problem covered by the study is the systematic assessment of the effectiveness of the most prevalent methods of controlflow obfuscation, i.e., flattening, splitting, and virtualization, in terms of their correlation with the complexity of binary analysis. Our methodology is to apply the before mentioned techniques on a simple C program by using the Tigress obfuscation toolkit and then examine the resulting binary file using Cutter GUI. The parameters measured included McCabe Cyclomatic Complexity, Control Flow Graph Size and Instruction Count. The effectiveness of the analysis method is evaluated by looking at the change in complexity of the measurements on the obfuscated variants in comparison with the original and with each other, to achieving the best protection profiles. The findings give the developers a practical insight into the best control-flow obfuscation method that would be most effective in hindering reverse engineering with minimal performance costs.

INTRODUCTION

Software protection has indeed been one of the most important areas that keeps finding new developments to address contemporary needs and protect digital artifacts against unauthorized access and analysis. Where contemporary economies and organizations are becoming more and more dependent on proprietary software, intellectual property, competitive advantage, and user trust are still at stake. The most important advantage of strong mechanisms of protection is that they

increase the degree of effort and expense of analysis to such an extent that organizations with intent to do harm do not reverse engineer executable software to allow them to access the underlying source code, algorithms or cryptography keys. This deterrent effect is of particular relevance in view of the fact that reverse engineering methodologies have been known to expose vulnerabilities or allow unauthorized copying with serious compromises in software and business security [1]. In this regard, strong protection becomes relevant not only for considerations related to revenue but also to legal and regulatory requirements concerning data protection and intellectual property rights.

On the other hand, code obfuscation has emerged as one of the premier strategies against these advanced threats due to reverse engineering, which transforms a program's binary or source code into an equivalent version that is much more difficult to understand or analyze [5]. Among those, control flow obfuscation is important, since it directly attacks the program execution paths and complicates the reconstruction of the control flow graph for both human and automated analysis [8]. However, one of the most serious problems developers face in practice is the general lack of clearly defined and quantified metrics that guide the choice of an optimal protection strategy. This opens up room for trade-offs between the efficacy of protection and acceptable performance overheads and maintenance considerations [7]. The following paper directly investigates the problem of studying the effectiveness of dominant control flow obfuscation techniques, namely control flow flattening, control flow splitting, and control flow virtualization, on binary analysis complexity.

In this respect, the paper will introduce a structured experiment in which a representative basic C program will be subjected to the three different obfuscation techniques. The techniques will be implemented through the Tigress obfuscation toolkit-an agile influential source-to-source transformation platform, particularly for C programs [8]. Fundamentally, the central methodology will comprise the practical application of the techniques and consequent analysis of the resultant obfuscated binaries using a professional reverse engineering tool. More precisely, the analyses of obfuscated outputs will be done using Cutter-a graphical user interface atop the framework of Radare2 [12, 14]. Such a setup by Cutter is able to visualize the control flow graphs and decompiled code and can calculate some of the important software complexity metrics like McCabe Cyclomatic Complexity, Control Flow Graph Size, Control Flow Depth, Program Length, and Instruction Count. These metrics again would provide a robust measurable basis for the assessment of protection provided by each technique [11]. Thus, the effectiveness of each obfuscation technique will be evaluated by comparing the measured complexity of the obfuscated binaries with that of the original and with each other. This kind of comparative evaluation will indicate the degree to which each method hinders the readability of decompiled output and the reconstruction of the original control flow logic. Together with considerations concerning performance overhead, analysis of the complexity metrics will have technical consequences for the determination of methods that offer substantial protection while maintaining the system's usability. Considering that this study is addressed to developers, it is expected to give clear practical recommendations concerning optimal protection profiles for given scenarios. This paper aims to contribute to the software security community by providing measurable metrics of obfuscation practices which can then be implemented upon informed decisions on proper trade-off between security benefits, system performance, and maintenance considerations.

2. Related Work

Source code obfuscation of software has proved to be an important research domain within the software protection domain. In a world where reverse-engineering and symbolic-execution attacks are engineered to increase in complexity, scholars have explored a number of methods to analyze, develop, and deploy various obfuscation approaches, especially those based on control-flow manipulations.

Various contributions try to synthesize an intelligent assessment of the quality and effectiveness of obfuscation techniques. Jin et al.

[5] proposed a formal framework for evaluating the quality of source-code obfuscation using such metrics as potency, resilience, stealth, and cost. This paper enables comparison of empirical methods across a wide range of different obfuscation techniques. Likewise, Viticchié et al. [6] presented a comprehensive taxonomy and analysis of numerous proposals for obfuscation, describing their advantages, disadvantages, especially in the context of source-level transformation.

The applicability of control flow obfuscation as a countermeasure has been studied in many papers. Ahmed et al. [8] analyzed how compiler optimization spaces may be searched for control flow obfuscation, with emphasis on transformations that obfuscate control flow graphs without disturbing program semantics. BinShamlan et al. [9] studied how these techniques influence resistance to human analysis, affirming the value of obfuscating execution paths.

Sophisticated obfuscation techniques have also been developed, including virtualization and metamorphic conversions. Caliandro et al. [11] proposed VMorph, a strong system integrating virtualization and metamorphism to obfuscate binary code and defend intellectual property. Their system increases the difficulty of static and dynamic analysis tools for trying to reverseengineer obfuscated programs.

To support technical obfuscation, creative and manual approaches have also been suggested. Hamadache and Elson [1] argued for creative manual obfuscation techniques on the grounds that human creativity can quite often do better than automated transformation in misleading opponents.

Researchers have also explored the application of obfuscation in particular programming scenarios. Ahire and Abraham [7] were interested in real-world mechanisms for obfuscating C source code. Their research showed how flattening control flow and opaque predicates could be used to great effect in actual codebases.

The danger of reverse engineering and automatic tools continues to motivate research on obfuscation. Zhao et al. [2] outlined automated methods of retrieving secrets from malware, highlighting the need for sophisticated obfuscation. Sharif et al. [3] demonstrated that malware behavior could be studied by reverse-engineering emulators. In one of the earliest studies, Ceccato et al. [10] pointed out the significance of sound experimental analysis of obfuscation techniques in the assessment of their practical performance.

Basile et al. [4] suggested a meta-model of software protection and reverse engineering attacks that offered a theoretical framework of classifying techniques and threats. It enables one to discover strong defenses and justify where the vulnerabilities were common in software systems.

Concerning software analysis techniques, the recent input is the work of Sun et al. [12], who presented the concept of the flattening of the control flow to enhance the vulnerability detection of the source-code and, therefore, proved the importance of code representation to analysis. Jang et al. [13] offered BitShred, a scalable semantic malware-analysis approach; this is useful when assessing the efficiency of obfuscation against such detection techniques.

The role of software-complexity metrics in measuring obfuscation has been pointed out by Kafura [14], who reassessed McCabe's Cyclomatic Complexity as a basic metric, particularly applicable in reverse engineering, since the higher the complexity, the higher the resistance there is to analysis.

Raubitzek et al. [15] show that layering obfuscation complicates the identification of single techniques and undermines the assumption that the application order significantly determines the final result. Their results show that the transformations embedded in a specific layer, instead of only the last applied transformation, are crucial for correct classification. Although some metrics, such as Branches (B), are able to discriminate with robustness between obfuscated and nonobfuscated code, the authors conclude that no structural metric can capture the complete complexity of all layered techniques.

Zhang et al. [16] believe there is an “arms race” between software obfuscation and binary diffing. Traditional obfuscation techniques, like instruction substitution, bogus control flow, and control-flow flattening, all work at the intra-procedural level and are becoming viewed as

ineffective against state-of-the-art binary diffing techniques able to correctly extract function semantics. This perception demonstrates a shift in focus toward using interprocedural code obfuscation, with the KHAOS approach proposed in this paper using fission and fusion to transform function semantics in a way that can help defeat such sophisticated analyses.

market viability of such methods for guiding organizations in protecting their IP against RE and related cyber threats.

Rodríguez-Veliz et al. [18] present a qualitative documentary review of the code obfuscation techniques for software component security. They define obfuscation as the process of code transformation to make it incomprehensible yet keeping its functionality intact, which is intended

Ref.	Author(s)	Main Focus / Contribution	Area of Research
[8]	Ahmed et al.	Compiler optimisation spaces for control flow obfuscation were examined in order to modify control flow graphs without altering program semantics.	Examines directly how control flow obfuscation affects program structure.
[9]	BinShamlan et al.	Confirmed the importance of hiding execution paths by examining how control flow strategies affect resistance to human analysis.	Directly relates to your research's objective, which is to measure resistance to human analysis and reverse engineering.
[14]	Kafura	Re-examined McCabe's Cyclomatic Complexity as a key indicator of software complexity (and resistance	Gives a crucial complexity metric that you use in your methodology its

Saleh et al. [17] present a general overview of hardware, software, and hybrid obfuscation methods, as well as corresponding deobfuscation and detection techniques. This paper reviews the methodologies, effectiveness, drawbacks, and concluded that obfuscation, through variable renaming and control-flow changes, among other transformations, is an important practice in protecting IP and preventing reverse engineering. They recommend
integrating obfuscation with complementary security techniques.

Al-Hakimi et al. [19] introduce a new Hybrid Obfuscation Technique to defeat RE attacks on software. In this approach, three techniques are merged: encryption of strings with mathematical equations, Unicode renaming of system keywords, and junk renaming of identifiers. Empirical testing

to protect intellectual property and hinder reverse engineering. In this review, they identified four major classes of transformations: lexical, control flow, data flow, and preventive. The authors have

against four RE tools on Java applications demonstrated the technique's effectiveness, causing obfuscation-induced chaos, errors, and an inability to read or analyze the code.

De la Torre et al. [20] present a new sourcecode obfuscation approach that makes use of GAs to find near-optimal sequences of LLVM passes. Addressed in both single- and multi-objective optimization, this method leverages complexity metrics along with runtime performance in order to maximize the efficacy of obfuscation while minimizing performance penalties with promising results on code protection.

		to analysis).	theoretical foundation.
[10]	Ceccato et al.	Highlighted how crucial it is to conduct thorough experimental testing of obfuscation techniques in order to fully understand their efficacy in practical situations.	Explains your experimental approach for conducting an empirical technique comparison.
[5]	Jin et al.	Established a systematic framework to assess the quality of source code obfuscation using metrics such as cost, potency, resilience, and stealth.	Offers a framework based on metrics for evaluating the efficacy and quality of obfuscation.
[7]	Ahire and Abraham	Examined practical methods for obfuscating C source code, emphasising the efficient application of opaque predicates and control flow flattening.	Focusses on using a technique you are testing (flattening) in practical C.
[6]	Viticchié et al.	Provided a thorough taxonomy and analysis of obfuscation proposals, outlining their benefits and drawbacks.	Provides a more comprehensive framework and categorisation for the particular methods (virtualisation, splitting, and flattening) that you are contrasting.

Table 1: Summary of the Literature

3. Objectives

Software protection against reverse engineering and unauthorized tampering has become increasingly important as the pace of the contemporary digital environment accelerates. Intellectual property protection, piracy deterrence, and maintenance of customer trust in the face of rising manual and automated attack vectors motivate software vendors to operate in such a manner. Among these many lines of defense, control-flow obfuscation remains one of the most successful methods against attackers. So far, however, the challenge has always been in choosing an obfuscation method that provides strong protection without excessively affecting performance and maintainability. This work is supported by several motivations:

Protecting valuable software assets by making reverse engineering difficult.

Filling the void of empirical comparisons of various control flow obfuscation techniques under real-world scenarios.

To utilize strong tools like Tigress and Cutter to thoroughly study the impact of these obfuscation techniques.

Providing practical advice for developers who need to balance security robustness against system performance.

The goals of this study are:

Comparing and assessing three prominent control-flow obfuscation methods i.e. flattening, splitting, and virtualization.

To determine their effect on reverse engineering complexity and runtime performance by:

Quantifying the impact that these approaches have on static and dynamic analysis activities and program maintainability.

Delivering empirical evidence and guidance that helps developers select obfuscation strategies suited to their security requirements without incurring performance costs.

4. Methodology

In this paper, the obfuscation methods are evaluated by the single parameter of the potency,

which is a measure to which an obfuscation approach is used to raise the complexity of the modified code relative to the original. In that regard, potency is a measure of a level of how difficult a program is to comprehend or analyze due to obfuscation, particularly from the reverse engineering viewpoint. Jin et al. [5] suggest that potency is an essential factor in measuring the obfuscation quality since it is intimately related to the structural rewriting and visual disruption of the code while keeping it functionally identical.

The authors suggest that measures like McCabe Cyclomatic Complexity, Control Flow Graph Size, and Instruction Count are good indications of potency since they measure the complexity growth introduced by obfuscation. This research employs the same principle, utilizing these measures to assess the efficacy of various control flow obfuscation methods used on a straightforward C program.

4.1. Description of the original C program

This C code is a basic calculator that does simple arithmetic calculations (+, -, *, /) on two floating point numbers depending on user choice. If we are particularly interested in hard-coded English strings i.e., just the user visible strings in natural language (not format specifiers such as "%lf %lf" or " %c"), then there are 4 hard-coded strings.

The source code had around 30 lines.

4.2. Obfuscation Techniques

4.2.1. Control Flow Flattening

Control Flow Flattening is an obfuscation technique to make the logic of a program more difficult to reverse engineer or understand. Rather than having readable and understandable control flow (such as if-else, loops, switch cases), it converts the control flow into one flat loop that relies on a dispatcher variable to determine what to run next.

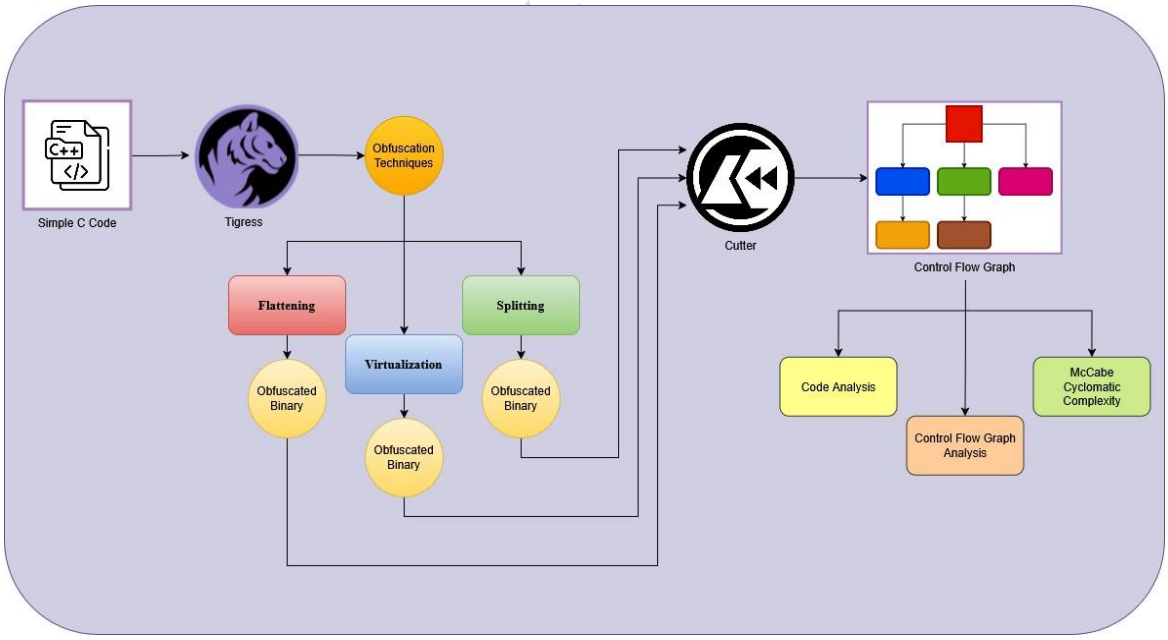


Figure 1: Architecture Diagram

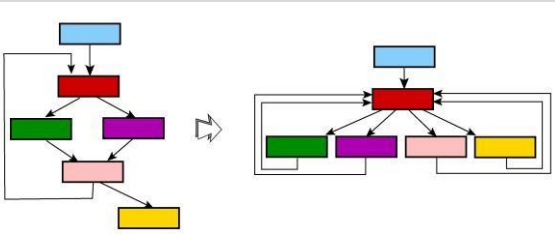


Figure 2: Control Flow Flattening tigress.cs.arizona.edu

4.2.2. Control Flow Splitting

In contrast to Control Flow Flattening, which condenses the control flow into one loop, splitting disintegrates the flow into several disconnected or non-consecutive pieces, so it is more difficult to predict and read. Splitting the control-flow involves one logical block of code and dividing it into a number of smaller code blocks or functions, most of which are connected in a non-linear or indirect manner (e.g., via function calls, gotos, or computed jumps). This shatters the clear top-down execution structure.

4.2.3. Control Flow Virtualization

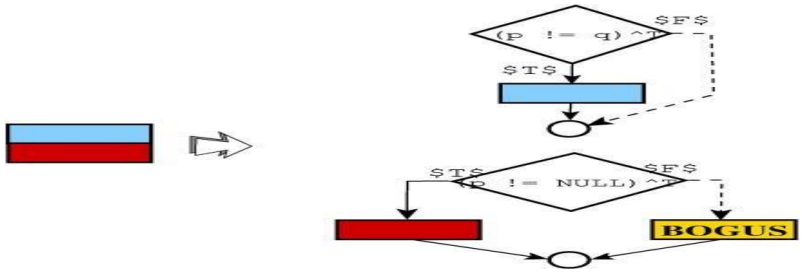
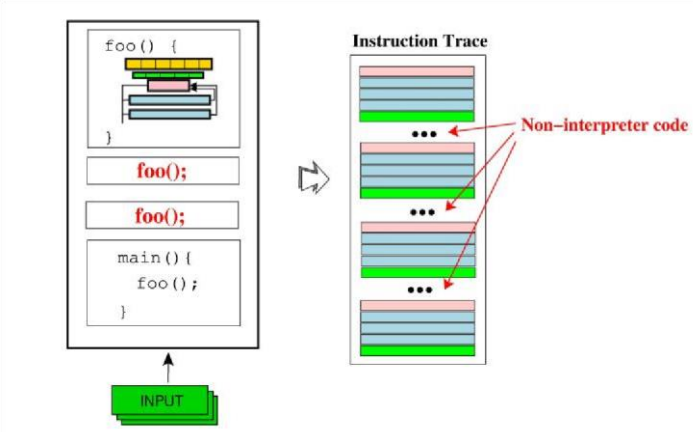


Figure 3: Control Flow Splitting tigress.cs.arizona.edu



Control Flow Virtualization is a sophisticated code obfuscation strategy in which a program's native control flow is converted into a personal virtual instruction set that is subsequently interpreted by an embedded virtual machine (VM) in the program. Rather than directly executing native machine code, the program translates its logic into personal bytecode and

executes this bytecode utilizing a personally developed interpreter (VM). The native control-flow is concealed within the VM's interpreter logic. These include converting a function to a custom bytecode interpreter by automatically generating a dedicated virtual instruction set and a dispatch method specific to the function. This transformation adds significant diversity through randomization of the opcodes of the instructions, the methods of communication between instructions, and the algorithms for dispatch; this creates both static and dynamic variability in the code and pattern of execution of the interpreter. In order to improve stealth and withstand analysis, sophisticated methods are utilized such as superoperators that can take many operations and aggregate them into intricate commands, fake functions and loops that hide the flow of execution, reentrant interpreters that alternate instruction execution with the application around it, and encoded bytecode arrays that cannot be easily inspected. Collectively, these techniques form an interpreter that is both structurally and behaviorally varied, making it very hard to reverse engineer or analyze the obfuscated code.

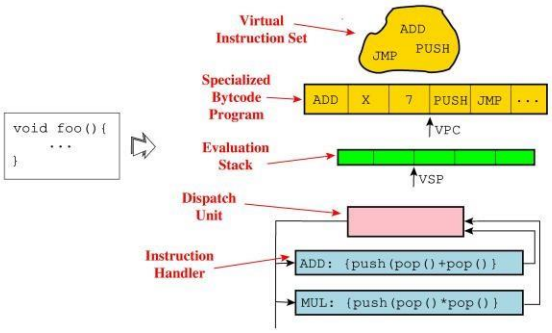


Figure 4: Control Flow Virtualization (zoomed in) tigress.cs.arizona.edu

Figure 5: Control Flow Virtualization (zoomed out) tigress.cs.arizona.edu

4.3. Code Obfuscation using Tigress

Tigress is a source-to-source compiler that accepts an input C program and generates an output obfuscated C program that has the same functionality but is much more robust against reverse engineering. For instance, to apply control flow virtualization to a program code called foo, you can execute the following command:

tigress foo.c–Transform==Virtualize–Functions=main–out=obfuscated.c

This command tells Tigress to virtualize the function foo's control flow in foo.c to produce an obfuscated output file named obfuscated.c.

4.4. Reverse engineering process using Cutter

Through Cutter, analysts are able to properly analyze complex binaries, comprehend malware or proprietary code, and develop good security solution. The procedure generally begins with loading the target binary into Cutter, where the analyst may navigate the program structure, examine assembly instructions, and comment on code. Cutter also includes features such as function graph visualization, string analysis, and interactive debugging, which assist reverse engineers in following program logic and identifying stealthy behaviors.

5. Evaluation Criteria

The analysis of obfuscation methods in this research is based only on the potency notion, which indicates how much more complex the altered code is compared to the original code. Potency is a measurement of how much harder it gets to understand or analyze a program as a result of an obfuscation method, especially from the perspective of reverse engineering. Jin et al. [5] suggest that potency is an essential fac tor in measuring the obfuscation quality since it is intimately related to the structural rewriting and visual perturbation of the code while keeping it functionally identical. The authors suggest that measures like McCabe Cyclomatic Complexity, Control Flow Graph Size, and Instruction Count

are good indications of potency since they measure the complexity growth introduced by obfuscation. This research employs the same principle, utilizing these measures to assess the efficacy of various control flow obfuscation methods used on a straightforward C program.

5.1. McCabe Cyclomatic Complexity:

McCabe Cyclomatic Complexity is a widely accepted software metric that provides a quantitative measure of program complexity, based on the number of linearly independent paths through a program's source code. It is defined by the relation:

(Number of Edges) × (Number of Nodes) + 2

where edges represent flow transitions, and nodes represent decision points or statements of the control-flow graph. A higher cyclomatic complexity means that the code is more complex; it therefore complicates comprehension, testing, and maintenance. These metric forms one of the base measures in complexity analysis; hence, this is usually cited in most literature in software engineering as Kafura, 14 illustrates.

5.2. Control Flow Graph (CFG) Size:

The CFG Size is the number of nodes included in a program's control-flow graph. Each node represents a basic block of statements or instructions, while edges illustrate the flow of control between blocks. The CFG size provides a metric regarding structural complexity and the granularity of the program's control logic. Larger graphs with more vertices suggest that the complexity or the fragmentation of program behavior is higher, which may increase the maintenance effort or chances of bug introduction. It has been used in reverse engineering research, such as in the work of Sun et al. [12] and Viticchié et al. [6].

5.3. Control Flow Depth:

Control Flow Depth measures the distance between two points in the control flow graph (CFG) based on the number of control steps taken to move from Point A to Point B. More specifically, this metric corresponds to the execution of multiple levels of control (nested) between the two points. Therefore, a deeper control flow will result in more levels of nesting due to conditions, loops, and function calls, increasing the cognitive load required for a programmer to understand the control structure, and can produce competing behaviours during execution. Sun et al. [12] and Caliandro et al. [11] have used this metric to assess binary complexity based on the level of obfuscation applied.

.

5.4. Program Length:

Length of a Program is simply a count of the number of lines of source code for a given source code file. Many software developers will refer to this metric as an indicator of program size. The total Lines of Code (LOC) in a source code file correlates with the amount of development effort that goes into creating the software and the complexity of the source code as well as its potential for defects. While LOC does not assess Code Quality as a standalone metric, it is a useful metric when comparing program size, and tracking development progress. Obfuscation has often been linked with increased program length by Viticchié et al. [6] and Ceccato et al. [10].

5.5. Instruction Count:

Instruction count can be explained as the number of all the low-level instructions which the program has executed. It represents how much computational processes took place, which is a better metric than the number of source-code line. A higher instruction count might indicate higher resource consumption and a lower count might indicate lower efficiency, it might also mean that the program might be too simple with less functionality. This metric has been explored by Jin et al. [5] in their quality model and by Jang et al. [13] in their work on semantic analysis tools.

6. Results and Discussion

Control flow flattening has moderate impact on complexity and structure size; hides the logic flow well but still not very analyzable. Control flow splitting on the other hand lowers local complexity but has great instruction and length overhead. It lengthens the code and makes it more difficult to follow, though not very nested. Control flow virtualization is the best for obfuscation. It transforms the code to be executed by a proprietary

VM, reverse engineering is nearly impossible, but has great size and overhead cost.

6.1. Discussion

The strength of any obfuscation technique is its ability to hamper reverse engineering by complicating programs and concealing logical flow. In this regard, Control Flow Virtualization is the most effective method among. It increases the cyclomatic complexity, control flow graph (CFG) size, and instructions significantly, effectively implementing the program logic in a custom interpreter. This makes both the process of conducting a static and dynamic analysis a lot harder as a reverse engineer must first reverse the behavior of the virtual machine before understanding the logic beneath it. These advantages occasionally cost the price of a large code size and execution overhead, and are therefore prohibitive to performance-critical programs. The Control Flow Flattening, on the contrary, provides a balanced tradeoff between the effectiveness/resource expenditure ratio. Replacement of nested control structures by a flat state-driven dispatcher mechanism dramatically raises the complexity of this technique and obscures the logical flow of the program, making the analysis of the problem difficult without either prohibitive overhead, in terms of program size or program execution time. It is a suitable choice in those cases when moderate security is necessary and the performance cost is significant, e.g., desktop or embedded systems, which need to mitigate against casual reverse engineering. Control Flow Splitting is the most incapable of introducing structural complexity because there are some in stances where it actually reduces cyclomatic complexity. Its primary trade-off is that it adds code size and instruction count while having very little impact on overall control flow depth. This method may have application in combination with others to add redundancy and make static analysis tools confused, but by itself, it provides little protection. In general, the selection of an obfuscation technique is a trade-off between security requirements, performance needs, and maintainability, and typically a combination method offers optimal protection with reasonable overhead.

Measurement Indicator	Description
Mccabe Cyclomatic Complexity	Formula: (Number of Edges)- (Number of Nodes) + 2. Measures the number of linearly independent paths through the program's source code. Higher values mean more complex code.
Control Flow Graph Size	Number of Nodes in the control flow graph. Reflects the size of the graph representing program flow.
Control Flow Depth	Maximum number of Edges to traverse from one Node to another in the control flow graph. Indicates the longest path through the code flow.
Program Length	Lines of Code (LoC) in the source code. Simple size metric indicating code volume.
Instruction Count	Number of Instructions in the program. Measures the actual low-level instructions executed.

Table 2: Summary of Evaluation Criteria

Obfuscation Techniques	McCabe Cyclomatic Complexity	Control Flow Graph Size	Control Flow Depth	Program Length	Instruction Count
Original Code	10	14	8	111	43
Control Flow Flattening	18	32	13	176	76
Control Flow Splitting	7	17	8	194	104
Control Flow Virtualization	48	60	13	591	125

Table 3: Comparison of Techniques

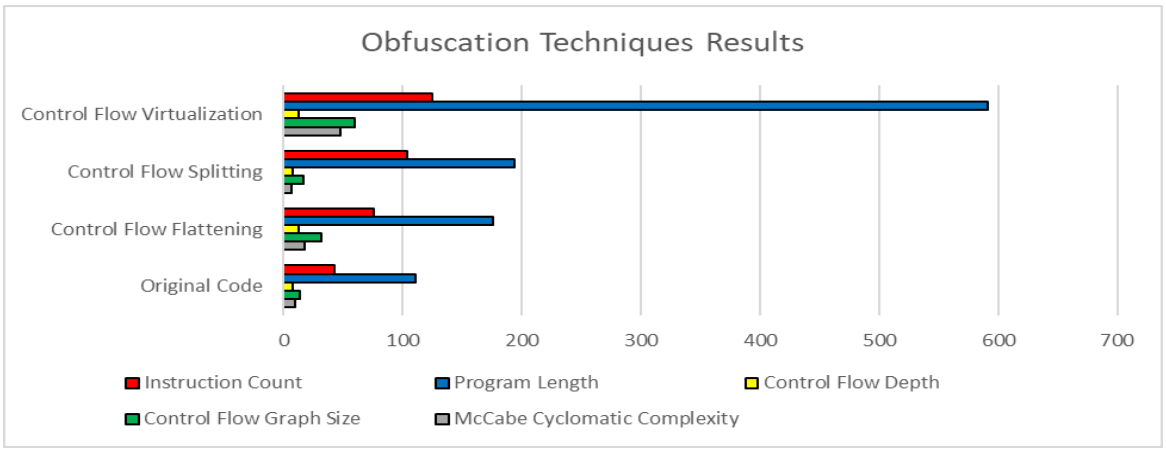


Figure 10: Measurement Results

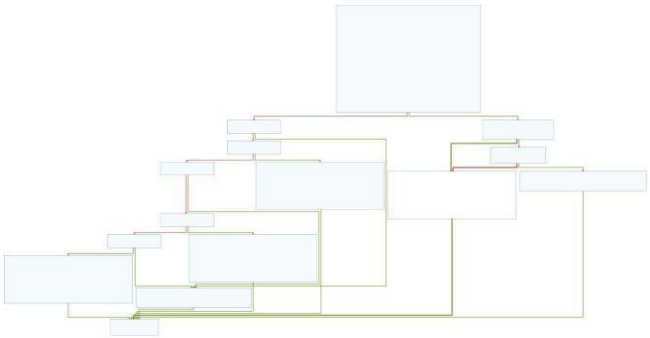


Figure 6: Original Code Control Flow Graph

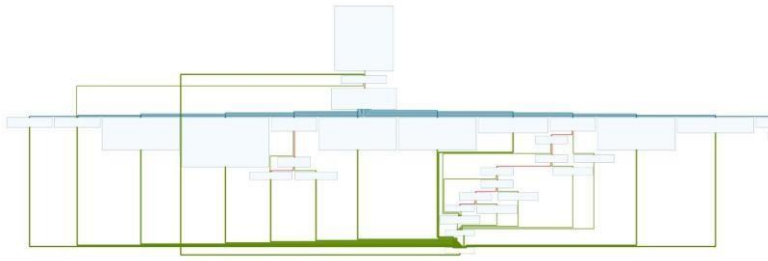
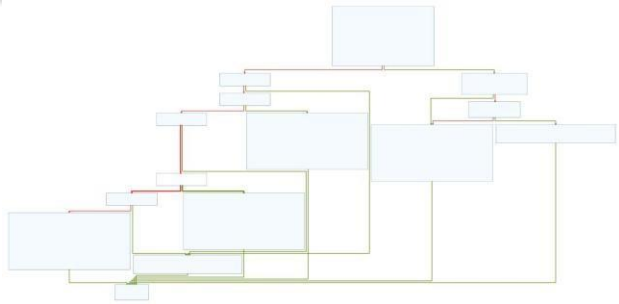


Figure 7: Control Flow Flattening



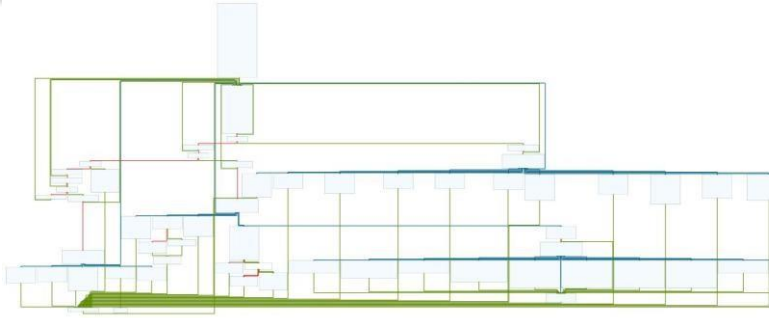


Figure 8: Control Flow Splitting
Figure 9: Control Flow Virtualization

7. Conclusion

This research presents a comparative analysis of three of the most well-known control flow obfuscation methods i.e., control flow flattening, splitting, and virtualization on a simple C program with the Tigress toolkit. With careful analysis with Cutter and various complexity measures including McCabe Cyclomatic Complexity, Control Flow Graph Size, and Instruction Count, the work illustrates that each technique provides unique trade-offs in protection strength versus performance overhead. Control flow virtualization is the strongest method of dramatically making reverse engineering more challenging by encapsulating program semantics inside a private virtual machine. This great level of security is achieved at the expense of large increases in the size of code and execution time, which can drastically restrict applicability to performance-sensitive situations. Control flow flattening is a viable tradeoff, as it greatly reduces the amount of code that can be analyzed in a reasonable amount of time with a relatively small code size and a relatively small runtime footprint. Control flow splitting, on the other side, has limited obfuscation value, and is more of an adjunct technique, and not a solution by itself. Lastly, the choice of control flow obfuscation has to rely on the special security requirements and the performance recommendations of the software. The study gives developers empirical information to choose and optimize obfuscation methods whose protection of intellectual property best respects maintainability and usability.

REFERENCES

S. Hamadache, M. Elson, Creative manual code obfuscation as a countermeasure against software reverse engineering, in: Advances in Human Factors in Cybersecurity: AHFE 2020 Virtual Conference on Human Factors in Cybersecurity, July 16–20, 2020, USA, Springer, 2020, pp. 3–8.

Z. Zhao, G.-J. Ahn, H. Hu, Automatic extraction of secrets from malware, in: 2011 18th Working Conference on Reverse Engineering, IEEE, 2011, pp. 159–168.

M. Sharif, A. Lanzi, J. Giffin, W. Lee, Automatic reverse engineering of malware emulators, in: 2009 30th IEEE Symposium on Security and Privacy, IEEE, 2009, pp. 94–109.

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.

H.Jin, J.Lee, S.Yang, K.Kim, D.H.Lee, A framework to quantify the quality of source code obfuscation, Applied Sciences 14 (12) (2024) 5056.

A. Viticchié, L. Regano, M. Torchiano, C. Basile, M. Ceccato, P. Tonella, R. Tiella, Assessment of source code obfuscation tech niques, in: 2016 IEEE 16th international working conference on source code analysis and manipulation (SCAM), IEEE, 2016, pp. 11 20.

P. Ahire, J. Abraham, Mechanisms for source code obfuscation in c: novel techniques and implementation, in: 2020 International Confer ence on Emerging Smart Computing and Informatics (ESCI), IEEE, 2020, pp. 52–59.

H. Ahmed, M. F. Hyder, M. F. ul Haque, P. C. Santos, Exploring compiler optimization space for control flow obfuscation, Computers &Security 139 (2024) 103704.

M. H. BinShamlan, M. A. Bamatraf,

A. A. Zain, The impact of control

flow obfuscation technique on software protection against human attacks, in: 2019 First International Conference of Intelligent Computing and Engineering (ICOICE), IEEE, 2019, pp. 1–5.

M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchi ano, P. Tonella, Towards experimental evaluation of code obfuscation techniques, in: Proceedings of the 4th ACM Workshop on Quality of

Protection, 2008, pp. 39–46.

P. Caliandro, M. Ciccaglione, A. Pepe, G. Bianchi, A. Pellegrini,

Vmorph: A

virtualization/metamorphic framework for binary obfuscation and intellectual property protection (2025).

Z. Sun, K. Liu, Y. Yang, Enhancing source code vulnerability detec tion using flattened code graph structures, in: 2024 6th International Conference on Frontier Technologies of Information and Computer (ICFTIC), IEEE, 2024, pp. 209–213.

J. Jang, D. Brumley, S.

Venkataraman, Bitshred: feature hashing malware for scalable triage and semantic analysis, in: Proceedings of the

18thACMconferenceonComputerand communicationssecurity, 2011, pp. 309–320.

D. Kafura, Reflections on mccabe's cyclomatic complexity, IEEE Transactions on Software

Engineering (2025).

Raubitzek, Sebastian, et al. "Obfuscation undercover: Unraveling the impact of obfuscation layering on structural code patterns." Journal of Information Security and Applications 85 (2024): 103850.

Zhang, Peihua, et al. "Khaos: The impact of inter-procedural code obfuscation on binary diffing techniques." Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and

Optimization. 2023.

Saleh, Khaled, et al. "Hardware and Software Methods for Secure Obfuscation and Deobfuscation: An In-Depth Analysis." Computers 14.7 (2025): 251.

Rodríguez-Veliz, M. ., Nuñez-Musa, Y. ., & Sepúlveda-Lima, R. . (2023).

Study of Code Obfuscation Techniques for the Security of Software Components. International Journal of Intelligent Systems and Applications in Engineering, 11(10s), 913–922. Retrieved from

https://ijisae.org/index.php/IJISAE/art icle/view/3385

Al-Hakimi, Asma'A. Mahfoud

Hezam, et al. "Hybrid obfuscation technique to protect source code from prohibited software reverse engineering." IEEE Access 8 (2020): 187326-187342.

de la Torre, Juan Carlos, et al. "Source code obfuscation with genetic algorithms using LLVM code optimizations." Logic Journal of the IGPL 33.5 (2025): jzae069.