universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

A Modular Approach to Compile-Time Obfuscation Using Sequential Transform Passes

verfasst von | submitted by

Elizaveta Zaidenvarg

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von | Supervisor:	Univ.-Prof. Dipl.-Ing. Mag. Dr.techn. Edgar Weippl

Acknowledgements

I would like to express my sincere gratitude to Dipl.-Ing. Dr. Sebastian Schrittwieser for his invaluable support, ongoing guidance, and encouragement throughout the development of this thesis. His thoughtful advice and patient mentorship have shaped this study from an initial idea into its final form.

I would also like to thank Patrick Kochberger and Patrick Felbauer for granting access to OSAGE, a private profiling framework, and for their help in integrating it into this project.

Finally, I would like to express my heartfelt thanks to my family, whose unwavering support and belief in me have been a constant source of strength throughout my degree studies.

Abstract

Software products distributed in binary form and deployed on untrusted hosts may contain sensitive information that should not be revealed to third parties. The confidential data includes proprietary algorithms, encryption keys, credentials, dependencies, and the code architecture itself. Third parties can employ reverse engineering practices to uncover valuable data and discover security flaws, leading to intellectual property theft or exploitation.

Code obfuscation is a software security approach which aims to obscure programs behaviour and complicate the static analysis of a binary. During code compilation and optimisation, obfuscations transform the code while preserving semantics by applying various techniques, such as modifying control and data flow, renaming variables, and inserting pieces of irrelevant code. Currently, there does not exist a well-established freely available obfuscation tool for programs written in traditional languages such as C, C++, and Java.

In this thesis, several traditional static obfuscation techniques are implemented as modular LLVM transformation passes. These include Control Flow Flattening, extended with an independent Bogus Control Flow mechanism, Function Merging, and Instruction Substitution based on custom Mixed Boolean-Arithmetic (MBA) expressions. The passes leverage an annotation-based mechanism that enables selective obfuscation by targeting only functions explicitly marked in the source code. The implemented obfuscation techniques are validated for preserving semantics and evaluated by measuring execution time and collecting software metrics across a benchmark suite of 47 programs. Profiling results reveal that certain obfuscation techniques can double software complexity scores, while execution time remains stable for both original and obfuscated binaries.

Kurzfassung

Softwareprodukte, die in binärer Form verteilt und auf nicht vertrauenswürdigen Hosts ausgeführt werden, können sensible Informationen enthalten, die nicht an Dritte weitergegeben werden sollten. Dazu zählen proprietäre Algorithmen, kryptographische Schlüssel, Zugangsdaten, Abhängigkeiten, sowie die Programmarchitektur selbst. Angreifer können diverse Reverse Engineering-Techniken nutzen, um Daten offenzulegen und Sicherheitslücken zu identifizieren, was zu Diebstahl geistigen Eigentums oder missbräuchlicher Nutzung der Software führen kann.

Der Begriff Code-Obfuscation beschreibe Methoden, die darauf abzielen, das Verhalten von Programmen zu verschleiern und die statische Analyse von Binärdateien zu erschweren. Während der Code-Kompilierung und -Optimierung transformieren Obfuscation-Techniken den Code ohne dabei seine Semantik zu verändern, indem sie unter anderem den Kontroll- und Datenfluss verändern, Variablennamen umbenennen und irrelevante Codefragmente einfügen. Derzeit existiert kein gut etabliertes, frei verfügbares Obfuscationtool für Programme, die in traditionellen Programmiersprachen wie C, C++ oder Java geschrieben sind.

In dieser Arbeit werden mehrere traditionelle statische Obfuscation-Techniken als modulare LLVM-Transformation-Passes implementiert. Dazu gehören Control Flow Flattening, erweitert durch einen unabhängigen Mechanismus für Bogus Control Flow, Function Merging sowie Instruction Substitution auf Basis benutzerdefinierter Mixed Boolean-Arithmetic (MBA)-Ausdrücke. Die Passes nutzen ein annotierungsbasiertes System, das selektive Obfuscation ermöglicht, indem nur explizit im Quellcode markierte Funktionen betroffen sind. Die implementierten Obfuscation-Techniken werden hinsichtlich ihrer Semantikerhaltung validiert und durch Messung der Ausführungszeit sowie Erhebung von Softwaremetriken über eine Benchmark-Suite von 47 Programmen bewertet. Die Resultate der Evaluierung in dieser Arbeit zeigen, dass bestimmte Obfuscation-Techniken die Komplexitätskennzahlen der Software verdoppeln können, während sich die Ausführungszeit der geschützten Binärdateien nicht signifikant erhöht.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

1 Introduction

Software security is a broad and highly relevant field of study, while software products are continuously developing and become attractive targets for cybercriminals and unauthorized parties. Robust security measures safeguard sensitive data and prevent unauthorized access, system breaches, and intellectual property theft. Protection strategies are individually designed for each piece of software depending on the specific product domain and underlying system architecture [31, 46].

Software products are exposed to customers and the outside world from various sides, including network connections, physical hosting and cloud servers, and public application interfaces. In terms of the access model, cloud-based services, also known as Software-as-a-Service, are one of the most secure forms of software product distribution because the client operates only with the program interface and does not have access to the code executed on a remote machine. However, a significant part of software products is distributed in binary form and deployed in untrusted environments, such as in the form of desktop applications, and then a user acquires full access to the code which contains sensitive information, including proprietary algorithms, encryption keys, credentials, and other secret data.

Reverse engineering practices and tool suites like Ghidra [15] enable adversaries to reconstruct a higher-level representation of the initial code from the compiled binaries and expose the underlying logic that was not intended to be publicly accessible. As a result, it compromises the confidentiality and integrity of the system and allows a third party to modify program's behaviour, bypass security mechanisms such as authentication or licence checks, and extract sensitive information embedded within the binary.

Surreptitious software employs obfuscation techniques to obscure the system functionality and prevent the adversaries from inspecting the code, aiming to retrieve valuable information and reveal control flow and sensitive data. Obfuscation complicates static and dynamic analysis by transforming the code while preserving the intended functionality using various schemes, including altering control flow, breaking abstractions in the internal structure, and adding misleading instructions (e.g., [5, 2, 7]). Obfuscation techniques preserve program's semantics and behaviour while increasing the amount of effort required for an adversary to understand the internal logic of the code, distinguish certain patterns, and locate sensitive sections which can be targeted during an attack.

In practice, obfuscations are widely employed by hackers evading malware detection [11, 39] and by mobile and web developers aiming to protect highly vulnerable Android and JavaScript applications [11, 44]. Skype is a vivid example of an application extensively secured by a number of embedded code obfuscations, including indirect calls and fake conditional jumps, and network protocol obfuscations based on checksum approaches [10]. The Dropbox client also uses obfuscated Python bytecode, which is enhanced with file

encryption [23].

In a broad sense, obfuscating techniques can be performed on three stages - directly on the source code, during compilation, and on the binary level, while each approach has its advantages and downsides and poses certain limitations on the applicable transformations. This thesis highlights Intermediate Representation (IR) level obfuscations performed on LLVM compiler framework - arguably the most flexible approach allowing to implement a variety of sophisticated techniques conveniently operating medium-level abstractions of basic blocks, instructions, variables, and functions, while staying language-independent and being compatible with several programming languages (see Section 2.2.3).

Currently, there are no well-known open-source IR obfuscator tools, although there are a number of related works and researches. For example, LLVM-Obfuscator [20] is an open-source project which provides obfuscation and tamper-proofing techniques on the IR level, but it is based on an outdated LLVM version which is no longer supported due to breaking changes in the suite. Additionally, it is important to develop a way to mark obfuscation targets in the source code and annotate an exact obfuscation technique that should be applied to the target, whereas by default LLVM passes implementing IR transformations are automatically applied to every function or module. The annotation mechanism can be applied to any problem, for example, debugging only specific functions with a utility pass, but is especially relevant for obfuscations, which heavily differ from each other in terms of code coverage, level of stealth, performance overhead, flexibility, and other aspects. Moreover, often only a few parts of the software need to be obfuscated, such as a proprietary algorithm and a secret key.

In the scope of this thesis, multiple traditional static obfuscation techniques are implemented as modular LLVM transform passes, including Control Flow Flattening with an independent extension of Bogus Control Flow, Function Merging, and Instruction Substitution based on custom Mixed Boolean-Arithmetic (MBA) expressions. The passes are built with the annotation mechanism, which enables obfuscation passes to modify only functions specifically marked in the source code.

Implemented obfuscation techniques are validated for preserving program's semantics and profiled for execution time and software metrics, targeting control flow and data flow complexity, code volume, and maintainability of the obfuscated code in comparison to the original metric scores. Both validation and evaluation are deployed on a suite of 47 programs with diverse internal structure and size, realising both real-life mathematical and cryptographic algorithms, and synthetic logic involving complex data transformations. As a result, particular software complexity scores are doubled for Bogus Control Flow and Function Merging, and other obfuscations pose an average overhead of under 30%, while performance profiling reveals no correlation between obfuscating technique and code execution time, which remains in the same range for both obfuscated and original programs.

1.1 Contributions

The main contributions of this thesis are:

- Implementation of static obfuscation techniques in a form of independent LLVM transform passes. The obfuscation approaches include Control Flow Flattening, Bogus Control Flow as a modular extension to Flattening, Function Merging, and Instruction Substitution configured with novel, self-designed MBA expressions.
- Design and implementation of a universal method for annotating obfuscation targets with an arbitrary key string, allowing to specify LLVM passes which are executed exclusively on the marked functions and variables without additional boilerplate code. The mechanism is embedded for annotation of obfuscating targets.
- Validation and performance profiling of implemented obfuscation techniques on a suite of 47 benchmark programs, covering checks of preserved semantics, evaluation of software metrics, and execution time comparison.

2 Background

Software protection is an essential aspect of software security, aimed at safeguarding digital assets from unauthorized access, tampering, and exploitation. Protection mechanisms ensure confidentiality, integrity, and availability through techniques such as cryptography, obfuscation, and tamper-proofing. This thesis is focused on protecting software distributed in binary form, when end users have full access to the executable code, and program protection relies entirely on the mechanisms embedded in the code itself in advance. This type of attack is classified as Man-At-The-End (MATE) and occurs when adversaries directly access and manipulate software or hardware on the endpoint using methods such as reverse engineering and debugging.

2.1 Attacks scenarios

2.1.1 Attack Classification

In a broad sense, a MATE attack implies an all-powerful attacker who has full access to both software and hardware and can examine, modify, and execute the program, or gain physical access to a device and tamper with the hardware itself or the software it hosts [12].

Akhunzada et al. [1] discuss the taxonomy of Man-At-The-End (MATE) attacks using a four-dimensional model, covering a source, a perpetrator, an attack, and consequences. Thus, the source is mainly a proprietary software, hardware, or any other informational asset, and the perpetrator is always a human of arbitrary technical skills and access level to the target, while MATE classification also covers attacks performed using remote access tools over a network. MATE attacks are malicious, in a sense that an attacker always exploits the system vulnerabilities deliberately and intentionally pursuing various objectives, such as fraud, modification, denial-of-service, and espionage.

Nagra and Collberg [33] claim that the adversary's attack methodology is split into five attack phases:

1. The black box phase,
2. The dynamic analysis phase,
3. The static analysis phase,
4. The editing phase,
5. The automation phase.

The black box phase is a starting point of drawing assumptions about the internal logic of a system and is typically executed on machine code, acting as an oracle transforming inputs to outputs. It is followed by the dynamic analysis phase giving a first glimpse of the internal program's structure, using specialized tools which repeatedly execute the code on a set of inputs and record invoked code sections. Then, static analysis suites such as Ghidra [15] inspect the internal design in more detail without actually executing the code and reveal control and data flow and sensitive data. The first three attack phases provide sufficient information for an attacker attempting to extract confidential data such as encryption keys, or reveal a proprietary algorithm. An adversary would use the editing phase to remove existing parts of the code like licence checks or add custom logic, for instance, a component which logs encryption keys transmitted during authorization sessions with an external provider. After that, in the automation phase an attacker eliminates the need to perform certain actions manually and combines all gained knowledge into a script which can be used for future automated attacks without any interference.

2.1.2 Real-life Man-At-The-End Attacks

MATE attackers aim at both hardware and software using both local and remote tools, meaning that every physical device such as a card reader or an ATM machine, and every program distributed in binary form becomes a potential target.

Online games and gaming consoles hosting proprietary software have traditionally been a popular target among users wanting to invoke behaviour not provided by the vendor. A classic example is the PlayStation 3 security breach performed in 2010 by engineer George Hotz [19], who exploited an encryption security flaw and managed to extract a console's private key, install custom firmware and gain hypervisor access level on a CPU. This breach enabled gamers to install unauthorized software and run proprietary applications on the console. Another popular target among gaming industry hackers is Steam, a digital video games distribution service. Thus, attackers reverse-engineered Steam's VAC (Valve Anti-Cheat) system machine code [1] and based on that implemented a publicly available VAC bypass tool [2] which disables cheat detection mechanisms located in the Steam's source code.

MATE attacks can be particularly dangerous and even life-threatening when it comes to vehicle hacking. In 2015, two white-hat attackers, Dr. Charlie Miller and Chris Valasek, discovered [32] they could remotely connect to 2014 Jeep Cherokee's infotainment system via its Uconnect cellular connection, while the car was moving on the highway. They could take over steering, brakes, and transmission, shutting off the engine to bring the vehicle to a halt. Later, Fiat Chrysler had to recall 1.4 million cars to address the vulnerability. Another example of a vehicle attack was performed in 2018 by KU Leuven University researchers [43], who cracked Tesla's key fob encryption by reverse engineering its firmware. With physical access to the fob, they could clone it within seconds using

[1] `https://github.com/danielkrupinski/VAC` (Accessed: 2025/04/22)
[2] `https://github.com/danielkrupinski/VAC-Bypass` (Accessed: 2025/04/22)

radio signals, allowing to steal Tesla vehicles without triggering alarms. Tesla responded with a software update and new key fobs with better encryption, but some older models remained vulnerable.

2.2 Security measures

2.2.1 Security Measures Classification

The taxonomy of software attacks and corresponding defensive measures is visualised in Figure 2.1.

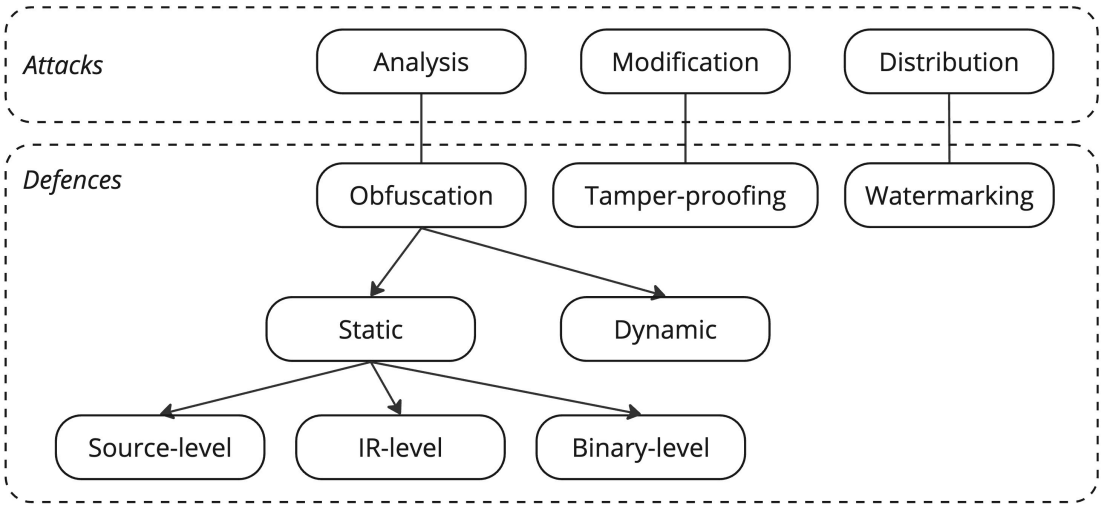


Figure 2.1: Software attacks and respective protection measures

In a broad sense, all software attacks are divided into three categories - analysis, modification, and distribution. Code analysis implies inspecting the code in order to extract valuable information about the internal structure of the program or the data it operates, such as proprietary algorithms, encryption keys, or other confidential assets. Modification technique involves editing the software to invoke the behaviour not provided or even blocked by the vendor, for instance bypassing licence checks or gaining advantage over other game players. Finally, the modified software can be illegally distributed for financial reasons or malicious intention to trick users into thinking they use the original software and then extract real user credentials.

The risks of mentioned attacks can be eliminated by the respective security measures.

Anti-analysis Measures

Static and dynamic obfuscation reduces the risk of sensitive information disclosure in code analysis attacks. Both obfuscation types involve modifying the code while preserving its semantics in order to obscure the internal structure and data, and the difference between

two types lies in the moment the modifications are applied. Static rewriters are similar to optimising compilers in a sense that they produce modified executable files, while dynamic obfuscators change the code in runtime, so that static representation always differs from the code which is actually executed.

Anti-modification Measures

Another software protection technique is tamper-proofing, used to prevent modification attacks and ensure code integrity is not violated during execution. Tamper-proofing algorithms are composed of two parts - the first mechanism detects that a program has been modified, and the other performs a tamper response. Tamper detection mechanisms include result checking which validates the output, environment checking scanning for possible emulation or debuggers, code checking networks responsible for comparing the integrity of code sections using hash algorithms, and state inspection of control and data flow. Tamper response mechanisms follow different strategies depending on the type of software and can terminate the execution completely, restore the attacked code pieces, degrade the result or performance, or report an attack to a developer.

Anti-distribution Measures

Unauthorized program distribution is tracked through watermarking to detect and claim duplicates and fingerprinting to trace violators. Watermarking implies embedding a value which indicates the uniqueness and author rights, and fingerprinting stands for placing a unique identifier to track program instances. Watermarks must be robust and hidden well enough to remain after possible code transformations and not be noticed and removed by an adversary.

2.2.2 Static and Dynamic Obfuscation

Code obfuscation can be broadly classified into static and dynamic obfuscation, applied at different stages of a program's lifecycle. Static techniques transform the code during compilation or optimisation and produce a semantic-equivalent representation, such as a binary code, which is then distributed to end-users. In contrast, dynamic transformations alter the code at runtime and complicate the inspection of execution behaviour using dynamic tools like debuggers and tracers. Dynamic obfuscation includes self-modifying code, runtime encryption, anti-debugging mechanisms, and virtualization-based obfuscation, where code is translated into an intermediate representation executed by a custom virtual machine. Dynamic obfuscation is beyond the scope of this research.

2.2.3 Static Obfuscation Levels

Static obfuscation can be performed in three stages:

1. On the source code
2. On the Intermediate Representation (IR) level

3. On the binary level

Source-level obfuscators modify the source code directly before it is processed by the respective compiler's frontend, which makes them entirely dependent on the specific high-level programming language. These obfuscators cannot be unified to work with an arbitrary language and are likely to become obsolete when a newer language version is released. Additionally, there is always a chance that obfuscations like instruction equivalence will be automatically removed by a compiler optimiser. However, source-level obfuscators are highly popular for scripting languages such as Python and JavaScript, because in this case there is no other level to perform obfuscations on. A well-established example of source-to-source obfuscator is Tigress [40], which transforms a C program to another semantically-equivalent C program.

IR level obfuscations are applied at the compiler's IR stage, such as LLVM IR, Java bytecode, or .NET IL. From the perspective of compatibility, the most reasonable platform choice is LLVM compilation suite, which provides a language-agnostic IR for high-level languages, including C, C++, Ruby, and Scala. Well-known examples of IR level obfuscators are Obfuscator-LLVM [20], Hikari obfuscator [18], and OBFUS [21], each based on LLVM.

Binary-level obfuscators are built for specific instruction architectures (x86, ARM) and binary formats (ELF on Unix, Mach-O on MacOS), which makes them quite difficult to develop. However, this type of obfuscators is not tied to any compiler or high-level programming language, and allows to design fine-grained obfuscations that are not possible on source or IR level. A case in point of the freely available binary-level obfuscators is Themida [3], which protects a binary using partial decompilation and obfuscation.

This thesis is dedicated to static IR level obfuscations performed on LLVM compiler framework.

Obfuscator Tools

Tigress is a highly popular source-to-source obfuscator compatible with C programs. It provides anti-analysis techniques, tamper-proofing mechanisms such as checksum, and a variety of traditional control and data flow obfuscations, including flattening, opaque branches, function splitting and merging, data and literals encoding. However, Tigress lacks variance in some obfuscations and produces predictable outputs in instruction equivalence transformations, which is exploited by a number of publicly available Tigress deobfuscator tools [4]. Another well-known obfuscator suite is Obfuscator-LLVM, an open-source LLVM-based obfuscation and tamper-proofing tool, providing traditional obfuscations, such as control flow flattening and bogus control flow.

2.2.4 Open Problems of Obfuscation

Integrating obfuscation as a protection technique poses certain limitations referring to the software product characteristics.

[3] `https://www.oreans.com/Themida.php` (Accessed: 2025/03/15)

[4] `https://github.com/JonathanSalwan/Tigress_protection` (Accessed: 2025/03/16)

Validity is the most significant open problem of any code transformation, such as obfuscation and optimisation, regardless of the operating level and purpose of the modifications. Rice's theorem [35] states that non-trivial semantic properties of the program are undecidable, and in the context of obfuscation, the property defines if a program demonstrates behaviour identical to the other program. The validation issue also refers to the Halting problem, which is proved to be undecidable by Turing [41]. Thus, obfuscations should not be applied to security-oriented and highly important software, where an undefined program's behaviour would lead to unacceptable consequences.

Another limitation involves the performance overhead that obfuscation techniques introduce to varying degrees. For example, data obfuscations, such as array permutations and string encoding, and control flow modifications naturally lead to performance issues to a certain extent, depending on the specific implementation and targeted domains. Before integrating particular obfuscation approaches, developers should take into account potential response time penalties and assess whether the software is performance-critical, such as in online games and exchange markets.

2.2.5 Evaluation of Obfuscations

Collberg et al. [8] proposed a taxonomy of obfuscating transformations which evaluates effectiveness and robustness of a transformation based on three key metrics: potency, resilience, and cost.

Potency qualifies obfuscations from a human perspective and measures how much more obscure, complex or unreadable is the obfuscated version of a program. It is often evaluated with software complexity metrics like cyclomatic complexity, or by counting how many analysis methods it complicated.

Resilience, on the other hand, refers to the strength of a transformation against automatic deobfuscator programs. A resilient obfuscation is adaptive and remains effective even as attackers develop more sophisticated tools to bypass obfuscation layers. Resilience is composed of two factors - programmer's effort evaluating the amount of time required to design a deobfuscator to reduce a potency of a particular transformation, and computational effort, which refers to time and space complexity of the proposed deobfuscator.

The third metric, cost, does not describe the resulting quality of a transformation, but instead it is defined as an execution time and space overhead, incurred by an obfuscation. Cost is measured based on a number of additional resources required to run an obfuscated program compared to the original version.

Stealth is an additional quality of obfuscating transformations, addressed by a number of authors including Balachandran et al. [3], Kanzaki et al. [22], and Lackner et al. [24], and describing if the obfuscated code can be distinguished from the unprotected one. Local stealth describes particular transformed instructions isolated from their context, and steganographic stealth refers to a program as a whole.

2.3 LLVM Architecture

LLVM [28] was initially developed in 2000 at the University of Illinois as a research project led by Vikram Adve and Chris Lattner, and over the years it has become an umbrella project for compiler and toolchain technologies, including frontends, backends, linkers, and debugger. LLVM is designed around language-agnostic and machine-agnostic IR that can be analysed, optimised and transformed using built-in transform pass interface, and then be further compiled to a machine code. Originally designed for C and C++, LLVM currently supports a variety of high-level programming languages such as C, C++, Java bytecode, Ruby, Rust, and Scala.

LLVM serves as a foundation for various compilers, including Clang, Rust compiler (rustc) and Swift compiler, leveraging LLVM for compilation and optimisation. LLVM is also a basis of NVIDIA's CUDA Compiler (NVCC), which allows developers to create and extend programming languages with support for GPU acceleration.

LLVM IR is a well-established intermediate representation suite and remains a popular choice among both researchers and developers creating new optimisation or obfuscation techniques or extending programming languages. A possible alternative of LLVM IR is GCC IR (GIMPLE) [16], which provides a structured representation for optimisations, but is tightly bound to compiler implementation and lacks modularity and reusability. In contrast, LLVM IR uses a modular approach, is well-documented and provides a rich API for user-defined analysis and transform plugins.

A high-level overview of LLVM tool suite is depicted in Figure 2.2. The primary components of LLVM revolve around LLVM IR and include:

- Frontend, responsible for parsing source code in high-level language and generating LLVM IR;
- Middle-end, focused on optimising and transforming LLVM IR with built-in or user-defined plugins;
- Backend, which converts LLVM IR into machine code for target architecture;
- Interpreter, which allows to execute IR directly avoiding machine code.

2.3.1 LLVM IR

LLVM IR is a low-level, platform-independent and language-agnostic representation of code that serves as the common interface between the frontend and backend. LLVM IR is designed to be both human-readable and machine-processable, and it can be directly executed by an interpreter without prior translation to machine code. It is a static single assignment (SSA) based representation, meaning that each variable is assigned exactly once, and every use of the variable is dominated by its definition. SSA property simplifies data flow analysis and enables powerful optimisations, such as dead code elimination, constant propagation, and loop-invariant code motion, by providing a clear and unambiguous representation of def-use chains.

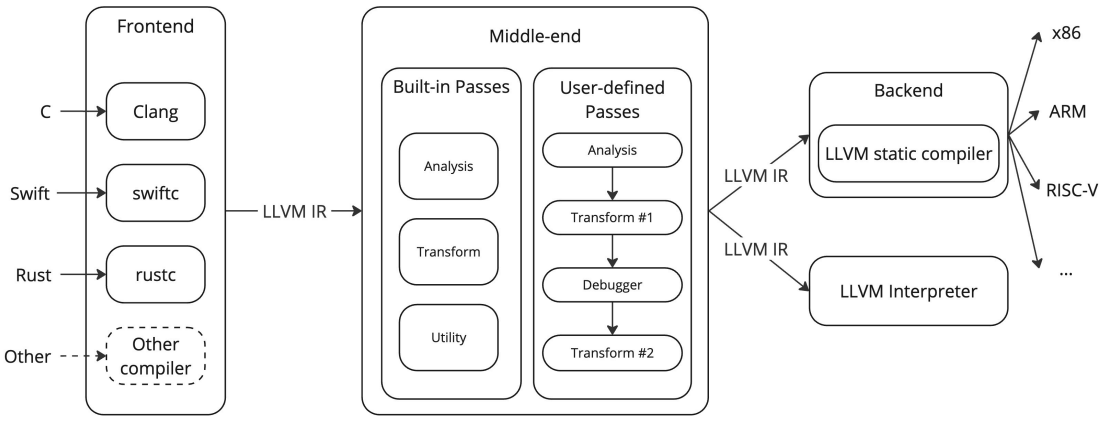


Figure 2.2: High-level overview of LLVM architecture

LLVM IR has three main forms - textual, bitcode, and in-memory.

Textual IR is a human-readable form of IR with file name extension *.ll* that can be written and edited by developers, which significantly simplifies transform passes debugging and helps to trace the impact of optimisations step-by-step, comparing different versions of an IR text file.

A binary-encoded form of IR with *.bc* extension that is more compact and efficient for storage and transmission, but lacks readability. This format is preferable for usage in production environment, when no debugging and tracing are required.

In-memory IR stands for the internal representation used by LLVM libraries during compilation, and by optimisation passes. A sequence of LLVM transform or analysis passes implementing a pass manager interface can either update a textual or bitcode IR file after each transformation, or use an in-memory representation to transmit changes directly to the next scheduled pass. The last option is more efficient because it reads the input IR file only once during the first pass, and updates the file only when all transformations are processed.

Figure 2.3 demonstrates a sample C source code and its resulting LLVM IR in a textual format. It visualises that IR code can be interpreted by a human analyst with little effort, while additionally providing information about the program's basic blocks and their predecessors to reveal control flow details.

2.3.2 Frontend

Frontend is the first stage in the LLVM compilation pipeline. It is responsible for parsing the source code written in a high-level programming language and translating it into IR. LLVM supports a number of frontends built for C, C++, Rust and other high-level languages, and additionally provides tools which developers can use for building frontends. Thus, Clang, a compiler for C, C++ and Objective-C, was originally based on LLVM, which played a crucial role in increasing the suite's popularity among researchers and developers.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Invalid input");
    return 0;
  }

  printf("Hello, %s!", argv[1]);
  return 1;
}
```

Listing (2.1) C source code

```
@.str = private unnamed_addr constant [14 x i8] c"Invalid input\00",
    align 1
@.str.1 = private unnamed_addr constant [11 x i8] c"Hello, %s!\00", align
     1

; Function Attrs: nofree nounwind ssp uwtable(sync)
define range(i32 0, 2) i32 @main(i32 noundef %0, ptr nocapture noundef
    readonly %1) local_unnamed_addr #0 {
  %3 = icmp eq i32 %0, 2
  br i1 %3, label %6, label %4

4:                                                ; preds = %2
  %5 = tail call i32 (ptr, ...) @printf(ptr noundef nonnull
    dereferenceable(1) @.str)
  br label %10

6:                                                ; preds = %2
  %7 = getelementptr inbounds i8, ptr %1, i64 8
  %8 = load ptr, ptr %7, align 8, !tbaa !6
  %9 = tail call i32 (ptr, ...) @printf(ptr noundef nonnull
    dereferenceable(1) @.str.1, ptr noundef %8)
  br label %10

10:                                               ; preds = %6, %4
  %11 = phi i32 [ 0, %4 ], [ 1, %6 ]
  ret i32 %11
}
```

Listing (2.2) LLVM IR code

Figure 2.3: LLVM IR example: (1) C source code and (2) corresponding LLVM IR code

2.3.3 Middle-end

LLVM middle-end is primarily concerned with optimising the IR using modular and reusable optimisation passes that transform the IR to improve performance, set up protection mechanisms, or achieve other goals.

Optimisation passes can be used for analysis, transformation, and utility purposes. Analysis passes do not modify the code and serve for visualising or collecting the information useful for the further passes. Transform passes mutate the program, and their functionality is not limited by optimisation purposes only - these passes provide all the necessary tools to obfuscate the program. Utility passes provide infrastructural functions for other passes, such as debugging the current state of IR or verifying its correctness.

The optimiser performs built-in optimisations of the program's IR by default, but is also configurable to run user-defined passes implemented as plugins based on LLVM Pass Manager interface.

2.3.4 Middle-end: Pass Manager

Developers can analyse and transform IR code by defining custom C++ plugins implementing LLVM Pass Manager interface and passing them to the optimiser. Pass Manager is a critical component of LLVM infrastructure, which orchestrates the execution of transform passes and also provides mechanisms for controlling the granularity of optimisations, allowing developers to apply passes at the function, module, or whole-program level. Pass Manager supports the concept of pass pipelines, allowing developers to define sequences of passes executed one-by-one, each operating with the program's IR modified by the previous pass. Thus, LLVM ensures the modularity of user-defined transform passes and allows developers to implement complex and maintainable transform and analysis suites.

Pass Manager grants all necessary instruments to analyse and modify the program and gives access to higher-level abstractions, such as control flow and data flow. It provides rich interface containing program's data, including variables, functions, modules, metadata, and basic blocks with control flow information, such as parents and successors. It also allows to analyse and insert complex instructions, alter function parameters, allocate memory, reorder basic blocks, and modify metadata. For example, Listing 2.3 demonstrates a snippet of Control Flow Flattening obfuscation pass, which assigns an index of the basic block's successor to the switch condition variable.

```
1 builder.SetInsertPoint(block->getTerminator());
2
3 if (auto branch = dyn_cast<BranchInst>(block->getTerminator())) {
4     if (branch->isUnconditional()) {
5         BasicBlock *successor = branch->getSuccessor(0);
6         auto int32Type = Type::getInt32Ty(context);
7
8         builder.CreateStore(ConstantInt::get(int32Type, idxs[successor]),
      caseVar);
9     }
```

```
10 }
```

Listing 2.3: LLVM transform Control Flow Flattening pass snippet: update *caseVar* variable depending on the block terminator

2.3.5 Backend

Backend is the final stage in the LLVM compilation pipeline, responsible for converting the optimised LLVM IR into machine-specific code that can be executed on a target architecture. LLVM provides a highly configurable backend framework that supports a wide range of target architectures, including x86, ARM, PowerPC, and others. Backend also follows modular approach, allowing developers to add support for new architectures or customize existing ones.

2.3.6 Interpreter

In addition to backend, LLVM also includes an interpreter that executes LLVM IR directly without generating machine-specific code by executing IR in a virtual machine environment, providing a way to run code without the overhead of compiling it to binary code. The interpreter is useful for debugging, testing, and cross-platform evaluation.

2.4 Obfuscations

2.4.1 Annotations

Obfuscations can be applied to different levels of abstraction, depending on a specific technique. For instance, name scrambling is applicable to both variable and class names, replacing instructions and loop transformations operate on instruction level, data modifications, such as array reordering and changing encodings, work with primitive or nested data structures, and control flow modifications operate mostly on basic blocks. Obfuscations also differ from each other in terms of code coverage, level of stealth, performance overhead, and often target different objectives, such as improving resistance against human analysts, static or dynamic automated analysers.

Thus, it is essential to allow developers to choose and configure which obfuscation techniques should be applied to the specific pieces of code, depending on the type of software, sensitive sections, existing vulnerabilities, or other preferences. For example, performance overhead is the core issue of the majority of obfuscations, and developers should consider the acceptable response or execution time on the user's device. Thus, it is important to develop a way to mark obfuscation targets in the source code, such as modules or functions, and annotate an obfuscation technique that should be applied to the target. Additionally, it should be possible to seamlessly integrate annotation parsing in the obfuscation passes and avoid repeating boilerplate code.

2.4.2 Control Flow Flattening

Control flow flattening obfuscation is used to complicate both static and dynamic analysis by transforming the original control flow into a flattened structure and obscuring the original sequential execution flow. This is achieved by replacing control constructs, such as loops and conditionals, with a single dispatcher loop that uses a state variable to determine the next code block to execute.

A program's control flow graph is a representation of all possible paths that can be traversed during execution, considering all conditional and unconditional branches and return instructions. Control flow graph is composed of basic blocks - sequences of instructions with no branches except for the last, terminating instruction, which leads to the next basic block or marks the end of the execution path. Basic blocks are usually generated by compilers as the first step in analysis process.

Control flow flattening removes relations between basic blocks by creating a centralised dispatcher mechanism, typically implemented as a loop that uses a state variable to determine which basic block to execute next. Each basic block is assigned a unique state value, and transitions between blocks are controlled by updating the state variable within the dispatcher loop.

The dispatcher can be implemented as a switch statement depending on a state variable and wrapped inside an infinite loop, referencing all basic blocks as switch cases. In the end of each basic block, the state variable is assigned to the successor's index based on a terminating instruction. Figure 2.4 demonstrates an example of how control flow flattening can transform the function's control flow graph.

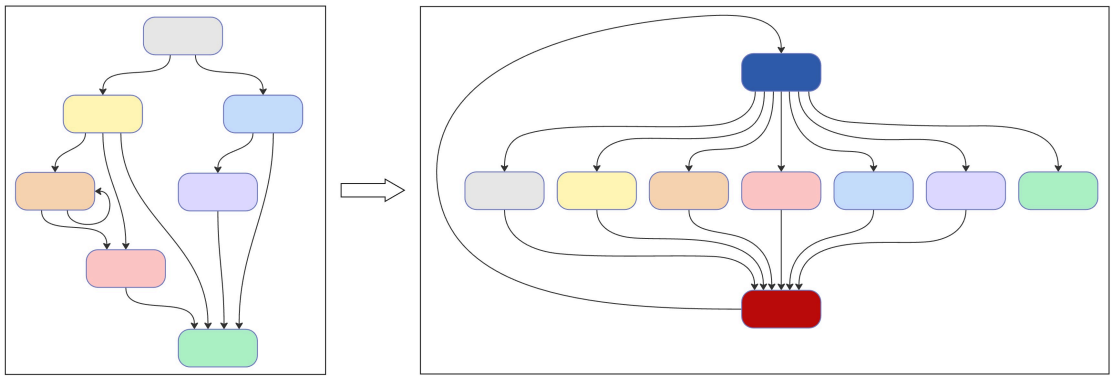


Figure 2.4: Control flow flattening visualisation: transformation of a control flow graph

The primary advantage of control flow flattening is the ability to complicate reverse engineering for both automated tools and a human analyst. By changing the internal structure of a program, the obfuscation makes it difficult to reconstruct the original logic, identify key functionality, and locate sensitive code segments. This technique is particularly effective against automated analysis tools that rely on pattern matching or control flow analysis to reconstruct program behaviour. The approach can be further enhanced in combination with other obfuscation techniques, such as bogus control flow,

dead code insertion, or opaque predicates, which can significantly increase the number of basic blocks and create the appearance of additional execution paths that need to be analysed.

The main drawback of control flow flattening is the performance overhead introduced by a dispatcher mechanism. The additional instructions required to manage a state variable and execute the dispatcher loop can lead to increased execution time and resource consumption. The exact performance penalty depends on implementation details, such as if entry block becomes a part of the loop, but in a general case, the number of edges between basic blocks is doubled because every link is mediated by the dispatcher block. As an example, László et al. [25] implemented control flow flattening for C++ and demonstrated that it results in a 2-fold to 5-fold increase of McCabe complexity. Thus, CPU time and memory overhead of control flow flattening may be unacceptable in performance-critical applications, such as online games or trading tools.

Another disadvantage of control flow flattening is its vulnerability to deobfuscation. Advanced reverse engineering techniques, such as symbolic execution, dynamic analysis, or pattern-based deobfuscation, can potentially recover the original control flow. For instance, symbolic analyser can model execution paths and monitor the value of dispatcher state variable to reconstruct control flow or detect which values correspond to certain behaviour.

Control flow flattening is an effective protection mechanism for algorithm-based software and programs with sensitive segments, such as anti-virus checks or anti-cheat mechanisms in gaming platforms. The obfuscation can obscure core logic of proprietary algorithms and make apart closely related sections in order to complicate their detection by an analyser.

2.4.3 Bogus Control Flow

Control flow flattening complicates static analysis by obscuring the internal program structure to make a control flow graph flattened and completely uninformative. On the other hand, bogus control flow insertion generates irrelevant code sections and adds them to the control flow under various conditions to create an appearance of additional execution paths.

Bogus control flow can be used to imitate non-existing logic, hide the existing sensitive functionality, or simply increase the number of instructions so that analyser requires a significant amount of resources only to clear out the irrelevant code sections. Inserting non-existing logic such as large complicated algorithms or external dependencies may throw off an adversary and provoke them to analyse unrelated pieces of code, wasting time and expensive resources. Alternatively, confidential functionality can be obscured among multiple similar-looking code sections, which for example can be applicable to anti-cheat mechanisms or licence checks.

Similar to control flow flattening, bogus control flow obfuscation typically operates with basic blocks in the control flow graph of a program. There are two closely related aspects of this obfuscation: the method of creating a new block and the predicate connecting it to the control flow. The block can be an arbitrary code or duplicate the functionality of the other piece of code, and be preceded by a condition that is always false. Another idea

is to copy an existing basic block, obfuscate it and connect it to the initial block so that only one or any of the two blocks is executed. This approach preserves the semantics while creating an appearance of a new functionality which may or may not be called.

Figure 2.5 illustrates both of these approaches - the obfuscation duplicates a violet block and creates a conditional branch under the same condition as for the original violet block, so that only one of them is executed. Additionally, it forwards a spurious branch to an orange block, leading to the path with arbitrary code, which in fact is not executed under any conditions.

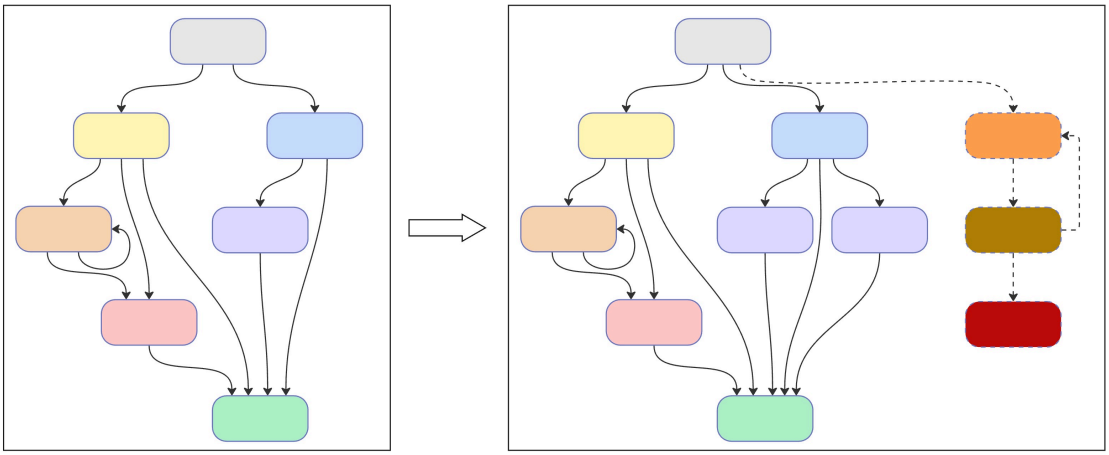


Figure 2.5: Bogus control flow example: duplicating a violet basic block and generating a spurious branch starting with an orange block

The combination of a block generation method and a connecting predicate produces a variety of possible implementations. For example, bogus control flow can be applied after control flow flattening to copy and obfuscate dispatcher case blocks and update some of original block references to point to the obfuscated blocks. Bogus control flow is also often used in conjunction with opaque predicates - arbitrary complicated expressions whose outcomes are predetermined but difficult for a static analyser to deduce.

Bogus control flow is a flexible instrument to significantly increase the difficulty of code analysis and detect the original execution paths by introducing redundant or misleading control flow structures using spurious branches. This technique is particularly effective against static analysis tools, which struggle to differentiate between initial and artificially generated constructs. Additionally, bogus control flow obfuscation can be selectively applied only to critical code segments and with an arbitrary number of bogus instructions, allowing developers to balance security with performance.

One of the key challenges of designing bogus control flow is balancing complexity with performance overhead. However, the obfuscation has a variety of implementations and complexity can be carefully adjusted for the specific problem, meaning it can only be applied to sensitive sections of code under certain conditions and consist of a predefined amount of code. Another significant downside of bogus control flow is its vulnerability to

dynamic and symbolic execution analysers, which monitor code sections executed under various inputs and can efficiently detect unreachable or rarely used code.

Bogus control flow is a universal tool to obscure program's internal logic or hide the location of certain mechanisms, such as credentials and licence checks. It is a flexible and yet effective tool, which has a wide range of applications and can be adjusted in terms of security and performance to meet the needs of developers.

2.4.4 Function Merging

Separating large pieces of code into independent functions, classes and modules is considered a good programming practice which promotes system readability and facilitates maintenance. Breaking abstractions into smaller pieces and merging unrelated pieces together can severely obscure the code structure and dependencies.

Function merging combines multiple functions into a single, unified function and optionally removes the original functions. This transformation is commonly used both in obfuscation to break abstractions associated with functions, and in compiler optimisations to remove unnecessary references and speed up the execution. Merging functions can eliminate redundant code, reduce the overhead associated with function calls, and create a more compact executable. In the context of obfuscation, function merging complicates reverse engineering by blending the boundaries between distinct functionalities, making it harder for human analysts and automated tools to isolate and recover individual components of a program.

Function merging can be implemented using various techniques. A one effective way to break the bounds and obscure commonly used logic is to remove a function and inline its content into every piece of code where it is referenced. Alternatively, multiple unrelated functions can be merged together to appear interdependent, often using additional parameters or control variables to differentiate between the original behaviours. Merging void functions is trivial - first, function parameter lists are merged together, and a switch parameter is added to mark which original function should be called. Then, function bodies are merged together under a switch statement or a number of conditional branches, which isolate the bodies from each other. Any call to either of these functions is replaced with a call of the merged function with a corresponding function index as a parameter. Merging non-void functions with different return types is more challenging and involves creating a unified structure to use as a return type.

Figure 2.6 demonstrates a sample control flow graph of a unified function, containing execution paths of two merged functions and a default behaviour, typically a simple return statement.

Function merging increases the code complexity, challenging reverse engineers to disentangle the merged functionalities and recreate the original program's structure. By merging functions that perform unrelated or loosely related tasks, the obfuscation breaks down domain separation and obscures the logic from the perspective of a human analyst. Additionally, function merging can reduce the effectiveness of automated deobfuscation tools, while they often rely on clear function boundaries and predictable control flow patterns.

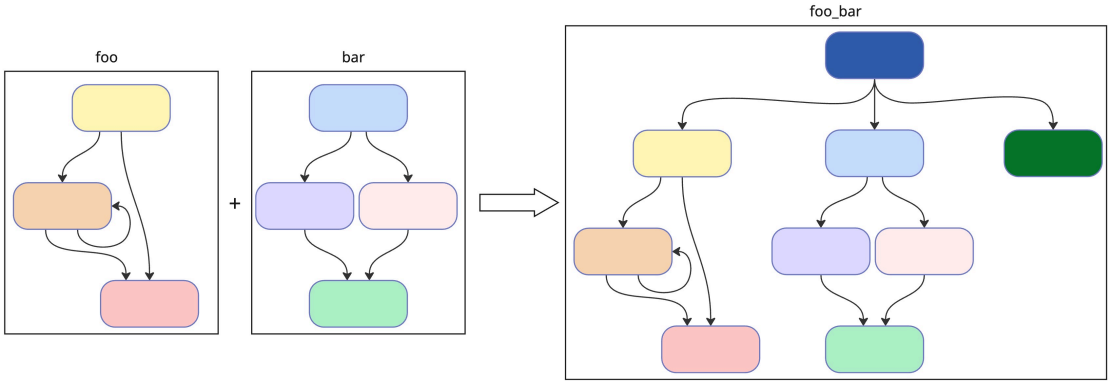


Figure 2.6: Function merging example: merging functions *foo* and *bar*. A dark green block corresponds to a default behaviour in case of invalid arguments

Increase in code complexity may lead to a performance overhead because of conditional branching inside the unified function and potentially long list of parameters in case of a large number of merged functions. One of the main issues with function merging is a lack of stealth, while unified functions follow a specific structure consisting of a single switch or a sequence of conditional branches that depend on a function parameter. This pattern can be visually detected by a human analyst inspecting the code, or automated tools configured to detect such functions. Although function merging is not a relatively robust stand-alone obfuscation, its stealth can be significantly improved by embedding additional obfuscations, such as bogus control flow and control flow flattening, to obscure a high-level dispatcher in the unified function's body, and opaque predicates to complicate the parameter indicating the original function.

Function merging can be especially effective when applied to proprietary algorithms and functions that involve mathematical computations or data transformations, meaning that their initial purpose can hardly be understood out of context. For instance, components of a cryptographic algorithm used for blockchain can be merged with hash-based password verification used in the application, and further complicated by a bogus control flow containing mathematical algorithms. A unified function aggregating multiple functions of a similar type is likely to appear natural and blend in with the rest of the code.

2.4.5 Instruction Substitution with MBA expressions

Instruction substitution based on Mixed Boolean-Arithmetic (MBA) expressions obscures commonly used algebraic or boolean operations by replacing them with semantically equivalent but more complex instruction sequences. This approach leverages the fact that all boolean and arithmetic operations can be transformed into various combinations of instructions using identities and properties from algebra and logic using both standard operations, such as addition and multiplication, and also more sophisticated instructions including bitwise shifts and XOR operator, which are not intuitive for a human analyst.

MBA expressions are a form of a lightweight, robust, and highly flexible obfuscation

technique. Instruction substitution helps to obscure branching conditions or argument validation checks spread across the entire code, and is especially effective on proprietary mathematical and cryptographic algorithms, such as hash functions. Compact MBA expressions consisting of several operations are highly performant and natively processed by CPU as any other arithmetic expressions, but are hardly understood by an attacker performing static analysis, recreating code patterns, and, for example, searching in which part of the code a variable is compared to a particular numeric constant. One of the main MBA advantages is flexibility, allowing a developer to configure multiple expression options for a particular operation, so that each entry is substituted by a different sequence of instructions to enhance code variability. The other flexibility aspect is the ability to use arbitrarily complex MBA expressions to balance obscurity and performance [27].

The core challenge of MBA expressions is a possible performance degradation in case of especially long and nested sequences of dozens of operations. It should be carefully applied to frequently used code sections, such as loop break conditions expected to be checked a significant number of times. However, instruction substitution is rather a conceptual obfuscation than a very specific guideline, such as control flow flattening, and a developer is free to implement it for their specific needs depending on the acceptable performance overhead and expected user computational resources.

MBA expressions are applicable to virtually any software due to their reliance on basic boolean operations, but they are especially powerful in algebra-focused domains like cryptographic algorithms. This approach also effectively complements other obfuscation techniques involving code generation, such as bogus control flow, where the duplicated sections can be transformed by different MBA expressions to appear unrelated.

3 Related work

3.1 Obfuscators

Obfuscator-LLVM [20] is an open-source project which provides obfuscation and tamper-proofing techniques for LLVM compilation suite. The obfuscation techniques include instruction substitution, control flow modifications, and procedures merging, and tamper-proofing mechanism is integrated with code flattening capabilities. The project maintenance ended with LLVM-4.0 and thus the work is incompatible with the most relevant LLVM-19.1 version at the time, due to the breaking LLVM changes.

Hikari Obfuscator [18] is an improvement over Obfuscator-LLVM with a few extra custom built passes. The project maintenance stopped in October 2024.

Schrittwieser et al. [37] present an obfuscation tool which significantly complicates dynamic reverse engineering by diversifying control flow graph of the software. The obfuscation technique ensures that the information about the program gained on a single run with a certain input cannot be used to predict the behaviour on other inputs. Evaluation section provides versatile insights into the algorithm's security, classified with well-established Collberg's metric, which measures potency (strength against humans), nesting complexity, data flow complexity, and resilience.

Kang et al. [21] suggest a convenient lightweight web-based software protection tool OBFUS, split into two parts. The first part is a high-level obfuscator operating on the source code of the programs written in C11 standard. The second part is a low-level language-agnostic obfuscator which decompiles binary programs to LLVM-4.0 IR and obfuscates them at IR level. The low-level obfuscator performs only MBA expressions, while high-level version operates MBA expressions and opaque predicates, renames variables, and removes indentation of the code.

Sharif et al. [39] present a compile-time malware obfuscation technique which conceals trigger-based behaviour from both static and dynamic input-oblivious analysers. The novel approach is based on conditional code obfuscation and aims to hide malware sections triggered by specific user inputs. First, the obfuscator identifies input-based branch conditions and uses hash functions in order to conceal which values satisfy the condition. Then, the code executed under the particular condition is encrypted with a key derived from the value which satisfies the condition. Thus, this work reveals flaws in existing malware analysers.

de la Torre et al. [9] employ genetic algorithms to find the most robust combination of obfuscation techniques out of built-in LLVM obfuscations. The defined multi-objective problem consists of decision variables acting as sequences of transformation passes to apply, while the objectives are program runtime and an obfuscation metric, namely the

cyclomatic complexity, the nesting complexity and the number of code lines. The resulting transformations can produce highly obfuscated code without penalising its performance, and even generating a more effective code version, while meeting obfuscation quality objectives.

Linn et al. [26] instead of employing obfuscation to obscure code analysis, employ it to complicate static disassembly process to protect the binary in the first step of reverse engineering. The authors inspect two state-of-the-art static disassembler strategies - linear sweep and recursive traversal - and introduce a way to invoke disassembler errors by injecting "junk bytes" into the instruction stream. The obfuscating technique is evaluated on widely used commercial disassembler suites and as a result, the majority of instructions from benchmark suite programs had disassembly errors.

3.2 Surveys

Schrittwieser et al. [38] provide a comprehensive view on existing obfuscation techniques and their robustness against various analysis tools. The research reveals that obfuscation strength heavily depends on a type of analysis tools and resources available to the analyst. Thus, simple obfuscations can be effective against lightweight static analysis tools, as it is often exploited by malware writers, while dynamic, manual and hybrid analysis tools demonstrate a significantly higher deobfuscation rate. Additionally, another finding is the increasing number of papers dedicated to malware obfuscations, namely revealing malware-specific patterns and obfuscation techniques which may be used to identify malicious software.

Ceccato et al. [6] is an empirical study evaluating the robustness of code obfuscation from the point of view of a human attacker, targeting two obfuscation techniques - opaque predicates and identifier renaming. The survey reveals that renaming variables is more effective than opaque predicates in terms of understanding the code structure and locating core logic. It is also found that learning the obfuscated code is limited due to the limited amount of knowledge which can be reused between attacks.

Banescu et al. [4] evaluate obfuscation resilience against symbolic execution attacks, complimenting existing researches focused measuring the potency of protection techniques against human-assisted attacks. A set of C programs is obfuscated by state-of-the-art Tigress C Obfuscator and Obfuscator-LLVM, operating on source code level and LLVM intermediate representation level respectively, and configured to an large set of techniques, including virtualisation, control flow modifications, instruction substitution, and opaque predicates. The results reveal that all obfuscations can be broken by a symbolic execution attack with different computational effort, depending on code characteristics of the original program and the applied obfuscation technique. In particular, instruction substitution appears useless against symbolic execution, while control flow flattening and virtualisation significantly affected the performance due to the larger number of queries to solve.

A number of papers address obfuscations of Android applications, motivated by the fact that mobile applications are especially vulnerable to disassembling and decompilation compared to an average binary code, such as X86 executable.

Dong et al. [11] investigate the usage of obfuscation techniques among real-life Android Java applications collected from widely used markets and malware databases. The targeted obfuscations, namely identifier renaming, string encryption and Java reflection, are detected in real applications and statistically classified using machine-learning algorithms. The study reveals a number of findings - for instance, identifier renaming is more common among third-party applications than in malware, and renaming in general is more popular among Chinese third-party developers. Also, although lightweight obfuscations are widespread among benign applications, more sophisticated techniques are generally more in demand in malware programs.

Xu et al. [44] conduct a measurement study addressing the popularity and classification of obfuscation techniques in the real-world malicious JavaScript software. The authors manually inspect target programs and reveal that the majority of them use at least one obfuscation technique. The most popular method appears to be string computation, such as string splitting and keyword substitution, while other common approaches are ASCII, Unicode, Hex code and customized encoding functions. The study also covers effectiveness of a number of widely used anti-virus software samples, and it demonstrates a low detection rate for malware protected by even simple obfuscation techniques. To sum up, malicious JavaScript programs are commonly enhanced with multiple levels of obfuscations which obscures their nature and allows to evade detection by popular anti-virus software.

Sahin et al. [36] investigate performance costs of obfuscations in the context of energy consumption of real-life mobile applications. The empirical study tracks a set of Android applications obfuscated using traditional techniques such as control flow modifications, optimising, and string encryption, and executed under different scenarios which imitate real user experience. The authors demonstrate that obfuscated programs generally lead to higher energy usage, with the extent comparable to the magnitude of impacts of other code changes, such as refactoring. However, the impact on battery life is unlikely to be noticeable to mobile users.

3.3 Deobfuscators

Udupa et al. [42] examine the strength of control flow flattening obfuscation relying on a hybrid of static and dynamic analysis. The research proves that the impact of control flow flattening can be removed by purely static techniques, while dynamic deobfuscation techniques are useful for enhancements such as code duplication and inter-procedural data flow. While the presented algorithm is successfully evaluated on a set of benchmark C programs, it is difficult to reason about the performance on real-life software.

Yadegari et al. [45] propose a universal deobfuscator which does not rely on assumptions about the nature of applied obfuscations. The authors focus on emulation-based obfuscation in the context of return-oriented programming. The deobfuscator is designed to detect and simplify the code sections which modify input values and affect output values, using taint propagation to track the flow of values and semantics-preserving transformations to modify the instructions. The described approach allows to extract

core logic of the original code without relying on specific obfuscation techniques.

Garba et al. [14] propose a generic approach to lifting a binary code back to LLVM Intermediate Representation level language and reconstructing a control flow graph of an obfuscated function. Experiments are not based on assumptions regarding a type of applied obfuscations - instead, the authors use compiler optimisations and satisfiability modulo theories solving. The proposed technique lifts the reverse engineering attack surface up to the intermediate representation level and removes binary-specific deobfuscation problems, while also allowing to build more advanced deobfuscation and analysis schemes using LLVM compilation suite tools.

4 Implementation

Obfuscation techniques are implemented in a form of independent LLVM transform passes, which are applied to LLVM IR by a built-in optimiser *opt*. The produced obfuscated IR file can further be compiled into a binary file and executed on a target architecture. All transform passes follow an in-memory approach, meaning that they do not modify the IR file in the intermediate stages, but instead modify the IR code virtually shared between passes, and only produce the final obfuscated IR file after all transformations are executed.

The obfuscations are sequentially applied to the IR of a program, and each obfuscation pass forwards the transformed IR to the next pass. Thus, the order of transform passes matters. LLVM optimiser is configured to execute the ordered sequence of transform passes on either all functions or all modules of a program, but the implemented annotation mechanism allows to add annotations to source code that define if a specific obfuscation should actually transform the corresponding function. By default, an obfuscation pass is executed on every function, but as the first step, it checks if the function has a respective annotation marking that it should be a target of this obfuscation.

The source code is available at `https://github.com/eliz-zay/masters-thesis`.

Technical Details

The project is implemented on LLVM v-20, the latest version at the time of writing. All transform passes are written in C++17 standard and orchestrated using CMake. To support cross-platform execution and provide a portable runtime environment, the project is encapsulated within a Docker image, which manages all dependencies and configurations. Docker container acts as a black box, receiving source C file and obfuscation configurations as an input and producing an obfuscated executable as an output. However, it is also possible to run the obfuscator outside a Docker container and gain access to intermediate steps, such as original and obfuscated IR files.

4.1 Annotations

In C language, the `__attribute__` keyword is an extension provided by GCC and Clang to specify additional properties of functions, variables, and types. The keyword can be used to state compiler-related preferences and configurations, including `__attribute__((noinline))` marking that function's body should not be automatically copied instead of a function's reference for optimisation reasons, and
`__attribute__((noreturn))` which informs a compiler that a function does not return

control flow after it finishes. A developer can also specify arbitrary information using `__attribute__((annotate("custom_name")))` syntax, then such attributes will be ignored by a compiler but kept in the IR.

As a prerequisite, every target function should first be annotated with `__attribute__((noinline))` instruction to prevent compiler from automatically inlining the function calls. This annotation ensures that the obfuscated function will actually be used - otherwise, obfuscations will be successfully applied but pose no effect in the runtime.

To mark which obfuscation techniques should be applied to a function, a developer should add an annotation with the corresponding name:

- `__attribute__((annotate("flatten")))` for Control Flow Flattening
- `__attribute__((annotate("bogus-switch")))` for Bogus Control Flow
- `__attribute__((annotate("function-merge")))` for Function Merging
- `__attribute__((annotate("mba")))` for MBA expressions

It is also possible to specify multiple obfuscations for a single function, as shown on Listing 4.1.

```
1 __attribute__((noinline))
2 __attribute__((annotate("flatten")))
3 __attribute__((annotate("bogus-switch")))
4 __attribute__((annotate("function-merge")))
5 __attribute__((annotate("mba")))
6 static void foo(const unsigned int N);
```

Listing 4.1: Annotation example: annotating a target function with multiple obfuscation techniques

4.1.1 Annotation Pass

LLVM IR does not directly associate functions and variables with their annotations - instead, the annotations are kept in a global section and cannot be immediately accessed from the objects. To solve this, the Annotation pass has been implemented - it iterates over all annotations in a module and modifies the IR by attaching metadata directly to the annotated values. The subsequent function passes can access this property straight from this metadata and do not need to parse the global section for every function individually.

The Annotation utility pass is executed once before any other obfuscation passes to attach annotation metadata directly to functions.

Before performing transformations, every obfuscation pass parses function's annotation and checks if it equals the tag string of the current obfuscation. To avoid code duplication, this boilerplate code is encapsulated in a template pass, and obfuscation passes extend this base class to inherit the annotation logic.

4.2 Control Flow Flattening

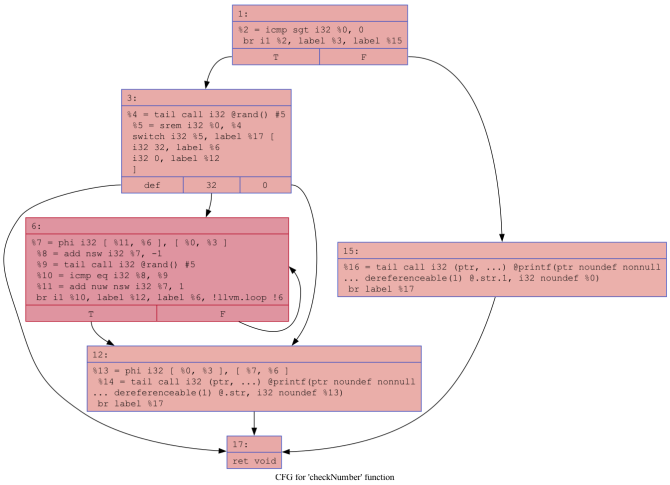
Control Flow Flattening pass is applied independently to each function which has a corresponding annotation. In LLVM IR, the functions are decomposed into basic blocks representing the units of control flow graph, in which every block references its successor using a single terminating instruction - a switch, a conditional or unconditional branch, a return statement, or other instruction. Control Flow Flattening algorithm mostly rearranges basic blocks based on their terminating instructions, and creates several new blocks representing an infinite loop with a single switch statement.

Algorithm 1 describes the core steps of a Control Flow Flattening algorithm. From a broad perspective, it generates an infinite loop with a single switch statement inside, acting as a dispatcher controlling which basic block should be executed next based on a dispatcher variable. At the end of each block, the variable is updated to mark a subsequent block, and then the control is passed to the end of the loop and the cycle repeats. The algorithm does not generate additional exit instructions, and the infinite cycle ends when an execution path reaches a block containing a return statement.

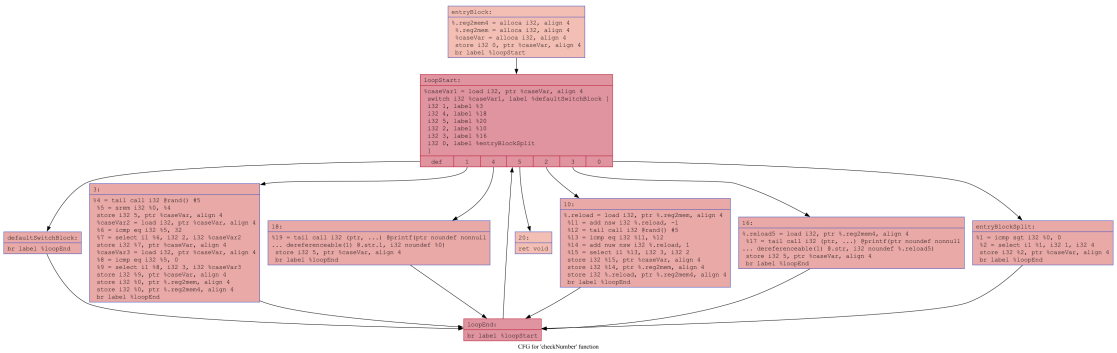
There are different ways to embed a switch loop into the control flow, and in this implementation it is placed right after the first, entry block of a function, and switch cases cover all basic blocks except for the entry block. Current approach allows to avoid additional steps required for overriding the function's entry block.

```
Data: Function F split into basic blocks
Result: Function F with flattened structure
foreach block ∈ F do
    if block.isLandingPad() || block.getTerminator().isInvokeInst() then
        return; // F cannot be flattened

if entryBlock.getTerminator() is a switch or a conditional branch then
    /* Make the 1st block end with unconditional branch                    */
    splitBlockByConditionalBranch(entryBlock);

caseVar ← allocateVar();
switchLoop ← generateLoopWithSwitch(caseVar);

/* Map blocks to unique integers, skip entryBlock                          */
blockCaseIdxs ← assignEachBlockAUniqueInteger();

/* Initially, a switch executes entryBlock's successor (the 2nd block)     */
caseVar ← blockCaseIdxs[entryBlock.getSuccessor()];

/* Update caseVar in the end of every block based on terminating instruction */
foreach (block) ∈ blockCaseIdxs do
    storeBlockSuccessorInCaseVar(caseVar, block, blockCaseIdxs);

/* Add basic block as a switch case under the respective index             */
foreach (block, index) ∈ blockCaseIdxs do
    addBlockCase(switchLoop, block, index);

foreach inst ∈ getInstructionsReferencedInMultipleBlocks() do
    DemoteRegToStack(inst);

foreach phiNode ∈ getPHINodes() do
    DemotePHIToStack(phiNode);

/* Annotate the switch to be later recognised by Bogus Control Flow pass  */
annotateSwitchForBogus(switchLoop);
```

Algorithm 1: Control Flow Flattening



(a) The original control flow graph



(b) The flattened control flow graph

Figure 4.1: Control Flow Flattening real-life example on a sample function

Control Flow Flattening obfuscation is visualised on Figure 4.1, illustrating the original and resulting control flow graphs.

4.2.1 Source Code Limitations

Control Flow Flattening algorithm imposes certain limitations on the source code.

First of all, a function must contain at least two basic blocks so that flattening is not trivial.

Then, a function must not have *invoke* instructions, which are similar to simple function calls but additionally provide an exception handling mechanism. The *invoke* instruction transfers the control to another function and defines both normal and exceptional execution paths. The normal path is taken if the function is successfully processed, and execution jumps to a landing pad block if the function throws an exception. Exception handling

mechanism is embedded into *invoke* instruction and thus cannot be directly translated into a combination of function call and conditional branch defining normal and exceptional successor blocks, which could be used to update a dispatcher variable. For the same reason, a function must not contain landing pad blocks because it can provoke an undefined behaviour in flattened structure.

4.2.2 Challenges

The algorithm reorders basic blocks and changes edges of control flow graph, introducing certain challenges related to the order of instructions in a function.

One concern is that some instructions can be referenced outside the block where the instruction is located. After reordering, the blocks referencing the instruction can appear before the block with initialisation and it will lead to SSA violation in IR, indicating that some uses of a variable are not dominated by its definition. This issue is solved by first locating instruction usage outside of a block, and then demoting virtual register computed by the instruction to a slot allocated in the stack frame. This precautionary measure allows to modify control flow graph while preserving the correct SSA form of the IR code.

Another issue with IR code is phi nodes, representing a special type of instruction which calculates a value depending on the predecessors of the current block. It is used for blocks with multiple predecessors, where phi instruction tracks which predecessor was actually executed and then returns the corresponding value. Control Flow Flattening reorders basic blocks and invalidates all phi nodes, detaching expected predecessors. To resolve it, all phi nodes are replaced with instructions operating with slots on the stack frame.

4.3 Bogus Control Flow

Bogus Control Flow pass creates an appearance of additional execution paths by duplicating the existing functionality and embedding it into control flow under plausible conditions. Current implementation copies case blocks of switch statements located in a function, embeds them into the switch, and then forwards a certain part of execution paths to the new blocks to avoid unreachable sections if possible. The pass is designed to be applied after Control Flow Flattening obfuscation to complement flattening technique by creating additional reachable basic blocks as if they represented particular logic in the original source code and were flattened with the rest of the code. Partial elimination of dead code is achieved by modifying the dispatcher variable generated by Control Flow Flattening and used in a unified switch statement.

Additionally, static code rewriting transformations implemented by third-party tools can increase obscurity when applied after the Bogus Control Flow pass. Transformations employing instruction equivalence, opaque predicates, aliasing, reordering and various data modifications can alter both original and duplicated blocks to obscure their equivalence and make them look unrelated.

MBA expression obfuscation effectively complements Bogus Control Flow by substituting duplicated arithmetic and boolean operations with different semantically equivalent versions. The combination of these obfuscations increases stealth and complicates pattern detection by making the duplicated blocks appear less related.

Algorithm 2 demonstrates core steps of bogus control flow obfuscation. The algorithm is configured by two parameters - $caseTargetPart$ and $storeInstRemappingPart$, which are statically assigned values of 0.7 and 0.5 respectively during runtime profiling to maintain a balanced setup:

- $caseTargetPart$ indicates a fraction of switch case blocks to duplicate, so that a value of 0.7 means that 70% of switch blocks will be duplicated. This parameter specifies the number of basic blocks to be traversed and processed in the transformation pass, and therefore reflects the obfuscation coverage and is proportional to overhead costs of the pass.

- $storeInstRemappingPart$ stands for the fraction of store instructions assigning a switch variable to the case values of original blocks, which must be replaced by similar instructions with case values of duplicated blocks. Thus, $storeInstRemappingPart$ indicates a proportion of execution paths to be forwarded from original to duplicated blocks, and any value close to 0 or to 1 may lead to unreachable code in either duplicated or original blocks, respectively.

The obfuscation performs three core steps for each detected switch statement and for each target basic block - duplicating the block, adding it as a new switch case, and remapping its references. In order to embed a duplicated block into the switch, it needs to be assigned a unique value of the dispatcher variable, which is not yet reserved by existing cases. As a first guess, a value generator takes a number of existing cases as the most plausible value, which is likely to blend in with other case values and, moreover, is

expected to be the next in order if the switch was generated by Control Flow Flattening obfuscation. If the value is already used by another case, then a random value is generated until a unique index is found.

Control Flow Flattening obfuscation is not deterministic in a sense that it adds blocks as switch cases in an arbitrary order, which in fact does not degrade runtime performance. However, in combination with Bogus Control Flow transformation, it leads to duplicating a certain number of arbitrary case blocks, meaning that every execution of these passes together on the same program may lead to different basic blocks being duplicated. Technically, it would be possible to ensure a strict ordering of basic blocks in Control Flow Flattening, but it cannot contribute to the performance and, moreover, can create a specific recognisable pattern which can be exploited by the adversaries, for example if blocks would be strictly ordered based on their initial occurrence in the code.

4.3.1 Source Code Limitations

The obfuscation only transforms switch statements generated by Control Flow Flattening pass, because adding cases to arbitrary switch statements may interfere with execution paths, in case there exists a flow in which a switch variable equals the case value for the duplicated case block. It cannot be trivially checked as it would require tracking down all uses of the variable up to its definition, including the usage of external modules, library calls, and input mechanisms. This restriction is implemented using an explicit annotation, which the Control Flow Flattening pass inserts for each generated switch, and the bogus pass parses to check if it is safe to generate new cases. Thus, it is ensured that Bogus Control Flow does not produce any side effects.

```
Data: Function F
Result: Function F with Bogus Control Flow
foreach block ∈ F do
    switchInst ← block.getTerminator();
    if !switchInst.isSwitchInst() then
        continue; // block does not have a switch instruction

    /* Check if the switch was generated by Control Flow Flattening pass */
    if !checkIfSwitchFlattened(switchInst) then
        continue;

    /* Parse a condition variable: caseVar from switch(caseVar) */
    caseVar ← getSwitchCaseVar(block, switchInst);

    /* Define the number of blocks to be duplicated */
    targetCount ← round(caseTargetPart * switchInst.getNumCases());

    for i ← 1 to targetCount do
        switchCase ← switchInst.cases()[i];
        block ← switchCase.getCaseSuccessor();

        /* Duplicate the block */
        dupBlock, valueMap ← CloneBasicBlock(block);
        foreach inst ∈ dupBlock do
            RemapInstruction(inst, valueMap);

        /* Generate a plausible case index for the new block */
        dupCaseValue ← generateCaseValue(switchInst);

        /* Add duplicated block as a switch case */
        addBlockCase(switchInst, dupBlock, dupCaseValue);

        /* Forward storeInstRemappingPart part of instructions to the duplicated
           block, instead of the original one */
        remapCaseVarStoreInstructions(caseVar, switchCase.getCaseValue(),
         dupCaseValue, storeInstRemappingPart);
```

Algorithm 2: Bogus Control Flow

4.4 Function Merging

Function Merging pass combines an arbitrary number of functions into one unified function and replaces the calls to the original functions with the new version.

Core steps of the algorithm are demonstrated in Algorithm 3. Inside a merged function, a switch statement is used as a dispatcher which forwards the control flow to one of the cases representing function bodies, while a specially allocated function argument indicates a dispatcher value. The argument list begins with the dispatcher value and is followed by concatenated arguments of target functions, together with pointers to their return values. Thus, the unified function has a void return type and supports an arbitrary number of target functions, with different arguments and return types.

References of target functions are replaced by the calls of the unified function with the corresponding index of dispatcher value, indicating the original function. Unused arguments standing for other merged functions are mocked with default values, such as zero or void.

Due to the nature of implementation, the unified merged function has a recognisable control flow graph consisting of a large switch statement with multiple sub-graphs as cases, which is not quite stealthy as a stand-alone obfuscation. However, both stealth and obscurity can be significantly improved if this pass is followed by Control Flow Flattening, which treats the artificially generated switch statement as if it is usual source code and flattens the switch together with basic blocks of the merged functions. The combination of Function Merging and Control Flow Flattening substitutes multiple functions with a unified obscure function with an entirely flattened structure, which does not reveal that it includes unrelated pieces of code and independent execution paths.

A sample control flow graph of a unified function generated by merging two functions is visualised in Figure 4.2.

4.4.1 Source Code Limitations

All target functions must have internal linkage, meaning that they are only accessible in the module scope rather than the whole program. If a function is exported outside the module, it is not possible to retrieve and replace all its uses by a unified function, because the module hosting the function can be distributed as a library and imported by any other module.

Another natural restriction is that it is only possible to merge functions with a fixed number of arguments, because argument lists are concatenated together.

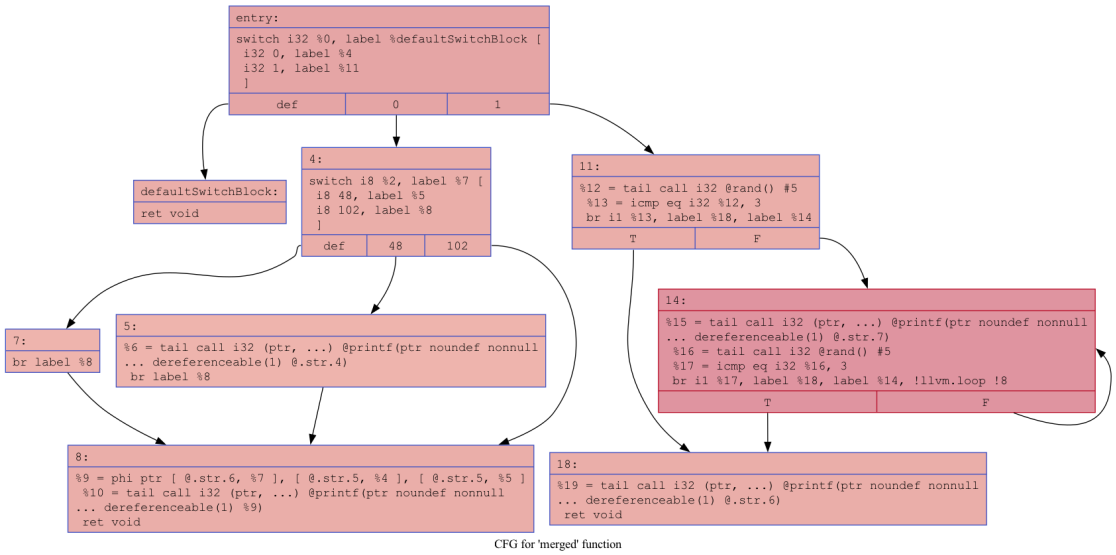


Figure 4.2: Function Merging example: control flow graph of a unified function generated by two functions

```
Data: Function F
Result: Function F with Bogus Control Flow
/* Function's metadata. Arguments are positioned at                    */
/* (argOffset; argOffset + argNum) in the unified function's argument list */
struct FuncInfo {
    int caseIdx; // Index in the switch
    int argOffset;
    int argNum;
    Type *returnType;
};

funcs ← getTargetFunctions();
if funcs.size() < 2 then
    return;

argTypesMap ← generateArgTypes(funcs);
funcInfoMap ← generateFuncInfo(funcs) as Map <Func → FuncInfo >;

/* Create a unified void function with a switch dispatcher             */
mergedFunc ← createVoidFunction(argTypesMap);
switchInst ← createSwitchCase(mergedFunc);

/* Merge functions and their arguments                                 */
foreach [f, info] ∈ funcInfoMap do
    addCase(mergedFunc, switchInst, f, info);

/* Replace function uses to the calls of a unified function            */
foreach [f, info] ∈ funcInfoMap do
    replaceFunctionUses(mergedFunc, f, info);
    deleteFunction(f);
```

Algorithm 3: Function Merging

4.5 Instruction Substitution with MBA expressions

Instruction Substitution obfuscation employs the fact that boolean and arithmetic instructions can be formulated in various forms while having the same semantics.

As represented on Table 4.1, Mixed Boolean-Arithmetic (MBA) expressions are implemented in multiple realisations for commonly used operations: signed comparison to zero, equivalence to zero, and addition of an integer number. For each target operation detected in the source code, a corresponding MBA expression is chosen randomly so that it appears like substitutions of the same operations are unrelated and correspond to different logic.

$\boldsymbol{x > 0}$
$(3 - ((x \gg 31) \oplus 1) \oplus 2 = 0) \wedge x \neq 0$
$(((x \gg 16) \oplus 0xCFD00FAA \gg 14)\&(1 \ll 1)) = 0) \wedge x \neq 0$
$\boldsymbol{x = 0}$
$56 \oplus x \oplus 72 = 112$
$76 \oplus \sim (x \oplus \sim x) \oplus 40 \oplus x = 100$
$((x \gg 6) < 5001) \wedge (x \geq 0) \wedge (((x \ll 2) \oplus 3) - 3 = 0)$
$(x \ll 1) \oplus x = 0$
$\boldsymbol{x + y}$
$(x\&y) + (y \mid x)$
$((y \mid x)\&(y \mid y)) + x$
$y + ((y\&x \oplus \sim y)\&(x \oplus y \oplus y))$
$(\sim (y \mid y) \oplus y \oplus \sim x) + y$
$(\sim y \oplus x \oplus y\&y \oplus (x \mid x) \oplus \sim (y\&x \mid x \oplus x)) + (\sim x \oplus \sim x \mid y \mid x \mid x \mid x)$
$(x \mid (\sim x \mid \sim x) \wedge (y \oplus y \mid y \oplus y) \oplus x \wedge (\sim x \oplus (y \mid x))) + y$

Table 4.1: MBA expressions

4.5.1 Integer comparison to zero: $x > 0$

MBA expressions for $x > 0$ are compatible with *SGT* (Signed Greater Than) operator based on signed integer types of $i32$ and $i64$. Comparison to zero is based on the binary structure of signed integer numbers, with the leftmost bit indicating if the number is negative. Expressions are aimed to extract the sign bit and check if it equals zero, while explicitly ensuring that the number itself does not equal zero.

All implementations have an explicit $x \neq 0$ check, because a sign bit does not differentiate between zero and numbers above zero.

Version 1

$$(3 - ((x \gg 31) \oplus 1) \oplus 2 = 0) \wedge x \neq 0$$

1. $x \gg 31$ shifts a sign bit of $i32$ to the right. For $i64$ values, this component is substituted by $x \gg 63$.
2. $\oplus 1$ extracts and flips the first bit, and the resulting value is either 1 if $x \geq 0$ and 0 otherwise.
3. $\oplus 2$ adds 2, thus the result is either 3 or 2.
4. $3 - (...) = 0$ is true if and only if $x \geq 0$.
5. $\wedge x \neq 0$ explicitly checks that other bits, except for leftmost sign bit, are not zero.

Version 2

$$(((x \gg 16) \oplus 0xCFD00FAA \gg 14)\&(1 \ll 1)) = 0) \wedge x \neq 0$$

1. $x \gg 16$ shifts sign bit to 16th position for $i32$ type. For $i64$, it is substituted by $x \gg 48$.
2. $\oplus 0xCFD00FAA$ flips a number of bits but does not affect the sign bit, located on the 15th position.
3. $\gg 14$ shifts the sign bit to the 2nd position from the right.
4. $\&(1 \ll 1)$ extracts the sign bit and makes all other bits zero.
5. $(...) = 0$ is true if and only if the sign bit is zero.
6. $\wedge x \neq 0$ explicitly ensures that the value is strictly above zero.

4.5.2 Equivalence to zero: $x = 0$

Version 1

$$56 \oplus x \oplus 72 = 112$$

1. XOR operation is commutative, meaning that operands order does not affect the result. Thus, the MBA is equivalent to $x \oplus 56 \oplus 72 = 112$.
2. $56 \oplus 72 \equiv 112$ - this equivalence simplifies the equation to $x = 0$.

Version 2

$$76\oplus \sim (x\oplus \sim x) \oplus 40 \oplus x = 100$$

1. $x\oplus \sim x$ produces a number where all bits are set to 1, and $\sim$ operation flips all bits and this component into a zero, which has no impact in XOR addition.
2. $76 \oplus 40 \equiv 100$, and the MBA is simplified to $x = 0$.

Version 3

$$((x \gg 6) < 5001) \wedge (x \geq 0) \wedge (((x \ll 2) \oplus 3) - 3 = 0)$$

1. $(x \ll 2) \oplus 3$ shifts x by 2 bits to the left and sets 2 right bits to 1.
2. $((x \ll 2) \oplus 3) - 3 = 0$ sets 2 right bits back to 0 and thus checks if every bit of x is zero, except for 2 left bits that were shifted out.
3. $x \geq 0$ checks if leftmost bit of x is zero.
4. $(x \gg 6) < 5001$ checks that second left bit is zero, otherwise $x >> 6$ is much larger than 5001, considering the maximum capacity of $i32$ and $i64$.

Version 4

$$(x \ll 1) \oplus x = 0$$

1. $x \ll 1$ shifts x one bit to the left.
2. $(x \ll 1) \oplus x$ applies XOR operation to neighbouring bits, and it is guaranteed to result in 0 if x is zero. However, if x does not equal zero, then a different non-zero value will be produced, because the rightmost bit of x set to 1 will be added with XOR to a zero bit and always result in 1.

4.5.3 Addition: $x + y$

MBA expressions for addition are generated using a commonly available obfuscator tool implemented by Justus Polzin [34]. The presented substitutions do not use obscure numbers to support native compatibility with different value types, such as floating point numbers and integers, including $i8$, $i32$ and $i64$.

5 Evaluation

Evaluation of obfuscation passes targets two essential aspects of any code transformation - validity and performance. An obfuscation is considered valid if it preserves the program's semantics, implying that even though transformations modify execution paths and data flow, both programs demonstrate identical behaviour and produce identical results on equal inputs. Performance aspect covers runtime characteristics and software metrics, such as Cyclomatic Complexity.

Profiling Framework

Profiling and performance evaluation are realised using OSAGE - a private profiling framework, which allows to configure compilation modes and specify a target set of programs, and then automates compilation and further analysis of target source code files with preconfigured compilation options, in this case representing different obfuscation techniques. The generated executables are then disassembled and analysed to extract software metrics which represent complexity, volume and internal characteristics of code which are directly related to performance and the amount of effort required to deploy reverse engineering practices against the binary code.

5.1 Benchmark Program Suite

Obfuscation passes are evaluated using multiple sets of sample programs, written in C language. Each obfuscation is independently applied to each benchmark program, and additionally a more complex set of programs, the synthetic set, is compiled using different combinations of obfuscating techniques. The resulting binaries are profiled and compared to the corresponding measurements of the original unmodified executables.

The targets are split into the following groups, summarised in Table 5.1:

- Open-source implementations of commonly used mathematical and cryptographic algorithms. This group is composed of 4 particularly large and complex programs having hundreds to thousands lines of code, and follow a black box principle, processing input data and producing the output based on it. The group is used exclusively for validation but not for profiling in OSAGE, due to specifics of internal implementation, such as rare C constructs which are heavily dependent on native libraries and host architecture.

- OSAGE built-in mathematical formulas and algorithms, a total of 26, including area and volume calculations and generation of Fibonacci sequence. This set is used

for both validation and performance profiling. However, programs in this set have only one function each and therefore cannot be used for Function Merging testing.

- Programs performing synthetic operations, rather than modeling real-life logic, and designed to have a complex control flow and nested internal structure. The set consists of 17 algorithms which generally transform input data or generate common data structures, such as graphs or matrices, based on various rules. This target group is employed for both validation and performance profiling of all implemented obfuscations, including Function Merging.

	Target group	Number of programs	Used for validation?	Used for profiling?
#1	Open-source algorithms	4	Yes	No
#2	OSAGE math algorithms	26	Yes	Yes, excluding Function Merging
#3	Artificial algorithms	17	Yes	Yes

Table 5.1: Benchmark program suites

5.1.1 Target Group: Open-Source Algorithms

Open-source programs in this benchmark group are used for extensive validation of obfuscation techniques. The chosen algorithms are complex, split into multiple functions and contain hundreds to thousands lines of code, aimed to ensure that obfuscations are applicable to diverse language constructions and preserve validity of the code with large data structures and control flow graphs, which is especially relevant for Control Flow Flattening.

Fast Fourier Transform (FFT)

FFT [1] is an optimised algorithm for computing the Discrete Fourier Transform (DFT), which decomposes a discrete-time signal into its constituent frequencies. The input is a sequence of N complex numbers representing time-domain samples, while the output is a complex-valued spectrum of the same length, encoding magnitude and phase information for each frequency component. The core innovation of the FFT lies in its divide-and-conquer approach, recursively breaking down the DFT into smaller DFTs, thereby reducing the computational complexity. This efficiency makes the FFT indispensable in signal processing, spectral analysis, and compression algorithms.

[1] `https://lloydrochester.com/post/c/example-fft/` (Accessed: 2025/04/10)

Advanced Encryption Standard (AES)

AES [2] is a symmetric block cipher that processes fixed 128-bit plaintext blocks using a secret key of 128, 192, or 256 bits to produce corresponding 128-bit ciphertext outputs. In other words, it transforms the plaintext into ciphertext of the same size through multiple rounds of processing, with the number of rounds determined by the length of the key. In its basic Electronic Codebook (ECB) mode, each plaintext block is encrypted independently using the same key, resulting in identical plaintext blocks producing identical ciphertext blocks. The algorithm operates through multiple rounds of transformation (10, 12, or 14 rounds depending on key size), each consisting of four core operations: SubBytes performs nonlinear byte substitution using a predefined S-box, ShiftRows permutes bytes through cyclic row shifting, MixColumns applies linear transformation to diffuse bytes across columns, and AddRoundKey combines the state with a round-specific subkey. The algorithm is parametrised by two samples of 16-byte sequences, representing key and plaintext, and produces two outputs - encrypted and decrypted plaintext. AES intentionally exhibits an avalanche effect, ensuring that even small changes in the input, such as flipping a single bit, lead to significant and unpredictable changes in the output, which makes it a perfect target for evaluating the equivalence of outputs between original and obfuscated versions of a program.

SHA256 hashing algorithm

The Secure Hash Algorithm 256-bit (SHA-256) [3] is a cryptographic hash function that produces a fixed-size output of 256 bits, regardless of the input size. The input to SHA-256 is a message of arbitrary length, and the output is a 256-bit hash value, typically represented as a hexadecimal string. The core logic of SHA-256 involves processing the input data in blocks of 512 bits, applying a series of bitwise operations, modular additions, and non-linear transformations across 64 rounds. The algorithm is designed to be collision-resistant, meaning that it is computationally infeasible to find two distinct inputs that produce the same hash output. Due to its high effectiveness, SHA-256 is widely used for data integrity, digital signatures, and authentication.

RSA keys generator

RSA [4] is an asymmetric cryptographic algorithm used for secure data transmission and digital signatures. The input to RSA consists of a message, typically represented as an integer, and a pair of keys: a public key for encryption and a private key for decryption. The algorithm is based on the mathematical principles of prime factorization and modular arithmetic. In RSA, the encryption process involves raising the message to an exponent corresponding to the public key and then taking the result modulo a large composite number. Decryption is performed by raising the ciphertext to the exponent corresponding

[2] `https://github.com/kokke/tiny-AES-c` (Accessed: 2025/04/10)
[3] `https://github.com/EddieEldridge/SHA256-in-C` (Accessed: 2025/04/10)
[4] `https://github.com/PascalLG/cinnabar-c` (Accessed: 2025/04/10)

to the private key, modulo the same composite number. RSA's security relies on the difficulty of factoring large numbers, and it is extensively used in secure communications, digital signatures, and key exchange protocols.

5.2 Validation

An obfuscation is considered valid if it preserves the program's behaviour, and for a given set of inputs the obfuscated version produces the same result as the original, unmodified program. The targeted algorithms have a certain output entropy and ensure that outputs are diverse, deterministic and highly dependent on the input.

A total of 47 benchmark programs are used to prove that all implemented obfuscation techniques preserve program's behaviour.

Strictly, the validation approach based on output consistency does not prove that transformations preserve program's semantics. Formally, it is not feasible to check that two programs have the identical semantics as stated in Section 2.2.4, and possible improvements of obfuscation validation are discussed in Section 6.1.

5.3 Software Metrics

ABC

ABC metric is a complexity measurement technique that quantifies code size and structure based on three primary dimensions: assignments, branches, and conditions [13]. A metric stands for assignments and refers to variable initializations and updates, reflecting the number of data manipulation instructions. B metric stands for branches and covers function calls, loops, and conditional jumps, capturing control flow complexity. C metric, conditions, involves logical predicates in decision-making constructs, such as if-statements and loops, measuring logical intricacy. The ABC score is defined as a composite of three metrics and derived as

$$ABC = \sqrt{A^2 + B^2 + C^2}$$

where A, B, C refer to the corresponding metric scores. ABC metric offers a more balanced assessment than unidimensional metrics like Cyclomatic Complexity, as it accounts for both data flow and control flow. For instance, elevated A score can highlight excessive state mutations, while B and C metrics respond to overly complicated control flow logic.

Cyclomatic Complexity

Cyclomatic Complexity is a fundamental software metric that quantifies the complexity of a program's control flow [30]. The metric is calculated based on the number of linearly independent paths through a program's source code and provides an objective measure of its structural complexity. Cyclomatic Complexity ($V(G)$) is derived from the control flow graph of a program as

$$V(G) = E - N + 2 * P$$

where E is a number of edges in the graph, N is a number of nodes, and P is a number of connected components, typically, 1 for a function.

Lines of Code

Lines of Code metric is defined as a number of assembler instructions in the executable and provides a straightforward estimation of the code volume. Instead of evaluating control or data flow structure, it refers to the size of the program in general and does not take into account which code sections are actually reachable. Although Lines of Code is not a solid stand-alone metric for measuring the complexity of the program's internal structure, it can provide valuable insights into characteristics of the source code in conjunction with other metrics, such as ABC or Cyclomatic Complexity. Thus, low control flow complexity score and a high Lines of Code score can point to mostly sequential internal logic, typical for mathematical functions or physical equations.

Halstead Metrics

Halstead complexity measures describe characteristics of the code by focusing on the syntactic elements of a program, such as operators and operands [17]. Metrics are calculated based on the following components: n_1 - number of distinct operators, n_2 - number of distinct operands, N_1 - total number of operators, and N_2 - total number of operands.

Halstead Difficulty Difficulty D quantifies how challenging it is to understand or modify the program.

$$D = (n_1/2) * (N_2/n_2)$$

Halstead Volume Volume V represents the size of the implementation.

$$V = (N_1 + N_2) * log_2(n_1 + n_2)$$

Halstead Effort Effort E accumulates both volume and difficulty and estimates the amount of work required to write, manage, or understand the program.

$$E = V * D$$

.

5.4 Profiling Results

Software metrics are evaluated on two sets of benchmark programs:

- Group 1: OSAGE built-in mathematical algorithms. This set is not used to evaluate Function Merging due to internal implementation, because algorithms have only one distinct function per module.

- Group 2: programs performing synthetic transformations using complex rules, designed exclusively for performance profiling in the score of this implementation.

The second artificial group consists of significantly larger code samples with nested logic, and they expectedly obtain metric scores up to 10 times larger than the corresponding scores of the OSAGE group. For illustration purposes, software metric scores are visualised using two charts for each metric, each one representing a corresponding benchmark group.

5.4.1 General trends

Overall, software metrics target characteristics related to the amount of effort required to understand and maintain the code, and cover control flow complexity, data manipulations, conditional branches, and a total number of instructions. Profiling results show that all obfuscation techniques either lead to an increase in metric scores, or do not affect them at all. This behaviour is expected and aligned with the idea that all code obfuscation techniques are essentially aimed to complicate a program and make it more difficult to analyse.

Bogus Control Flow

By design, Bogus Control Flow extends Control Flow Flattening by duplicating a certain number of basic blocks connected to a dispatcher. This setup is reflected in a majority of metrics, including ABC metric and Cyclomatic Complexity, where Bogus Control Flow score varies from the Flattening score up to twice as much, depending on the fraction of flattened code and characteristics of the particular metric.

5.4.2 ABC metric

Figure 5.1 illustrates ABC metric scores for two benchmark groups. For both sets of programs and all obfuscations, ABC scores are almost identical to A scores, being around 5% to 10% larger than A scores, visualised in Figure 5.2. This relation is explained by the order of magnitude of absolute values of A, B, and C scores, where B and C scores are down to 10 times smaller than corresponding values of A metric (see Figure 5.3 and Figure 5.4). This pattern reveals that in tested programs, the number of assignment instructions prevails over branches and conditional instructions.

In terms of A and ABC metrics, Bogus Control Flow, Function Merging and MBA obfuscations produce the largest complexity increase of up to twice as of the original program. It aligns with the fact that Flattening creates only several new instructions and aims to rearrange the existing control flow.

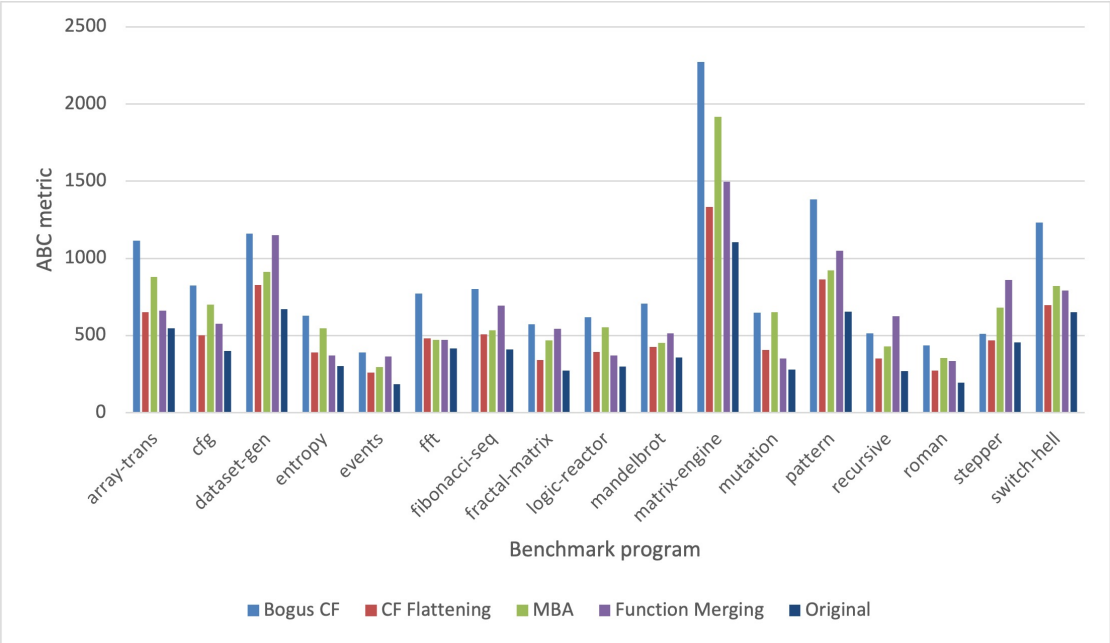
Control Flow Flattening introduces up to 10% of additional assignment instructions compared to the original code, which are necessary to implement a dispatcher mechanism.

Bogus Control Flow leads to the largest increase of B metric scores, whereas other obfuscation techniques produce only a slight increase of up to 5%. The Bogus Flow has a stronger effect on programs from the artificial group rather than OSAGE real-life benchmark group - the majority of synthetic code samples have doubled B metric scores,

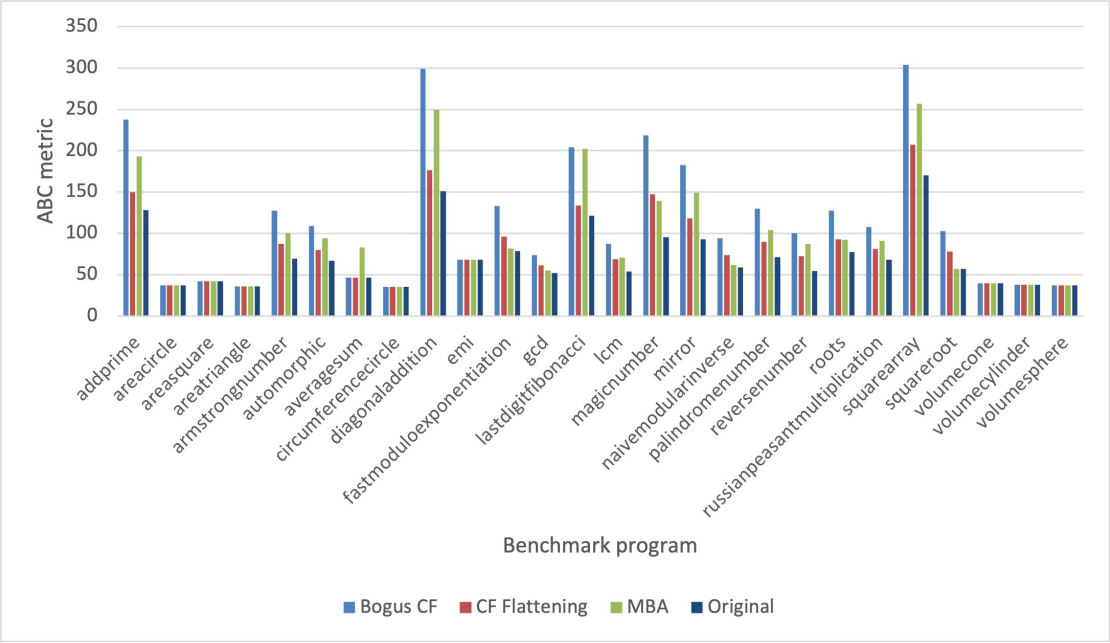
while for mathematical algorithms the score increased by around 25% from the original version. Such behaviour can be explained by the nature of Bogus Control Flow obfuscation, which inserts additional case blocks into a dispatcher, essentially creating new control flow branches, whereas other obfuscations are not aimed to generate new control paths.

Obfuscation Combinations

Figure 5.5 visualises ABC metric for artificial benchmark programs obfuscated by a sequence of several techniques. The results are consistent with independent obfuscation measurements listed in Figure 5.1a, indicating that the most significant complexity overhead is reached for versions which include Bogus Control Flow. Thus, the combination of all four obfuscations increases the original score by up to 3 times, and a combination of Bogus Flow with MBA expressions or Function Merging leads to a slightly smaller gain of between 2 and 3 times.

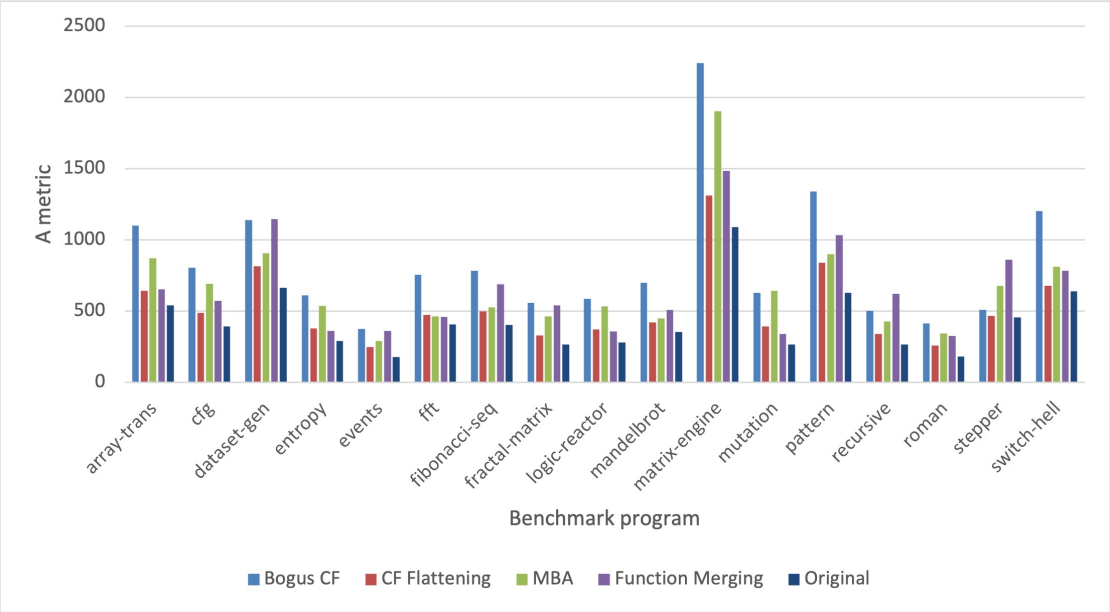


(a) Artificial benchmark group

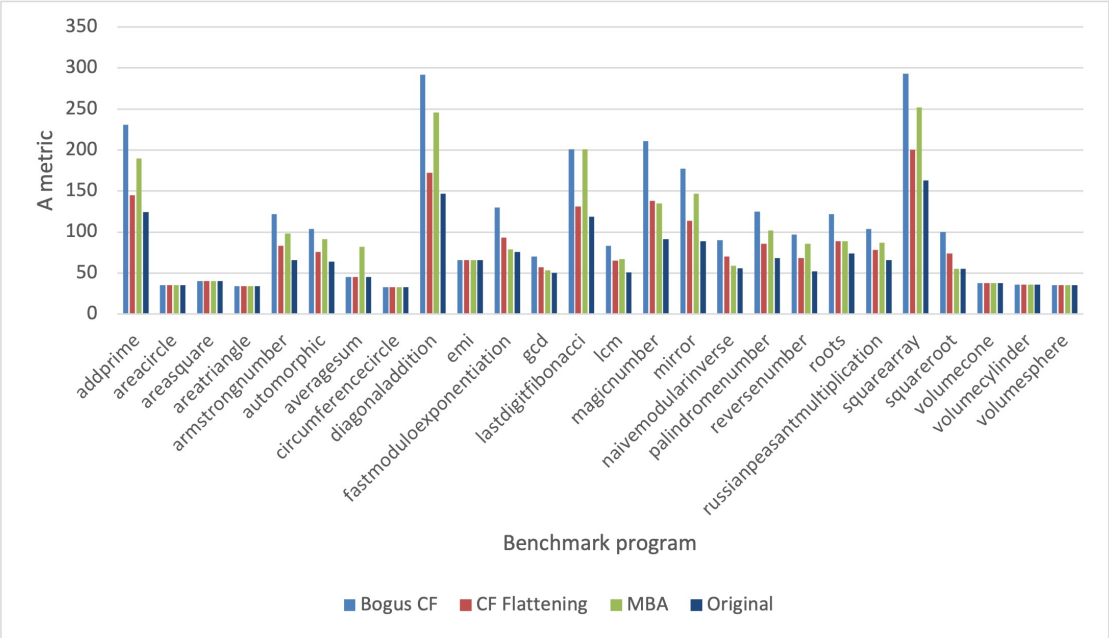


(b) OSAGE mathematical benchmark group

Figure 5.1: ABC metric score

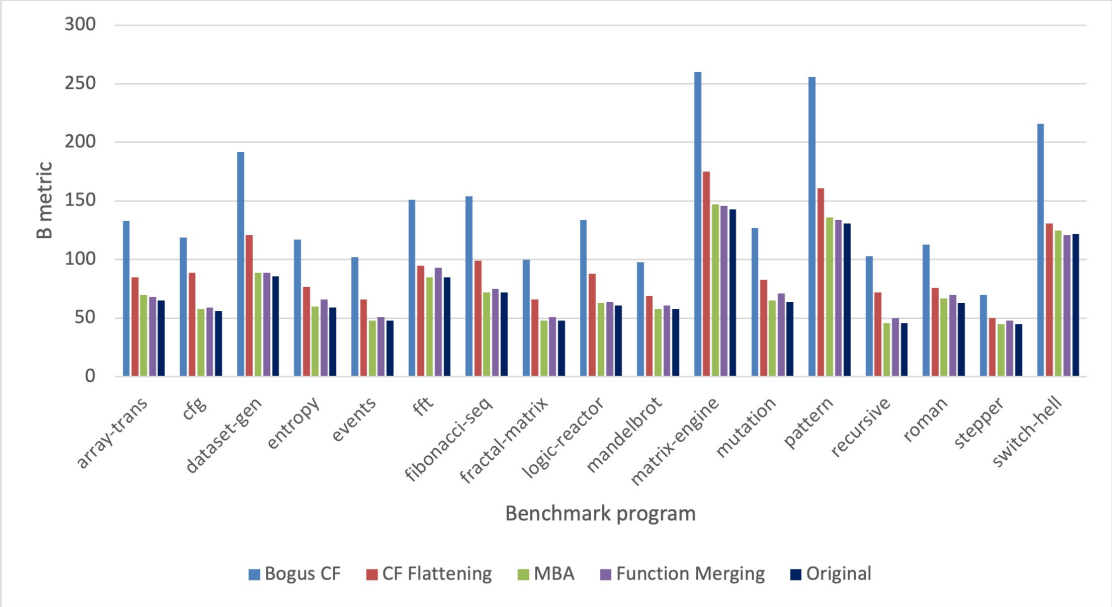


(a) Artificial benchmark group

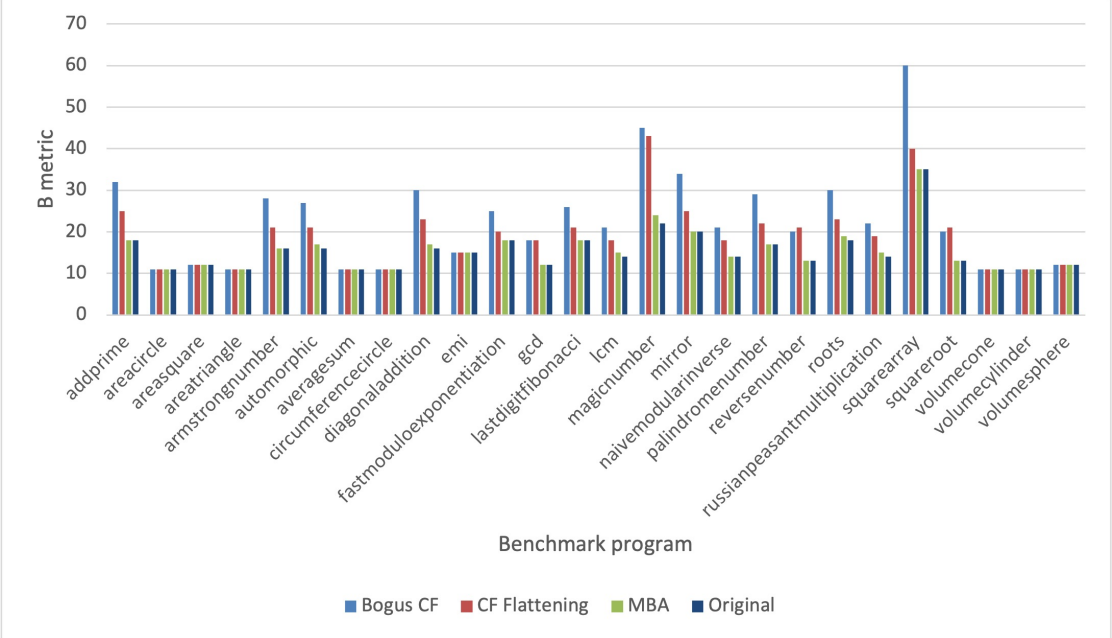


(b) OSAGE mathematical benchmark group

Figure 5.2: A metric score

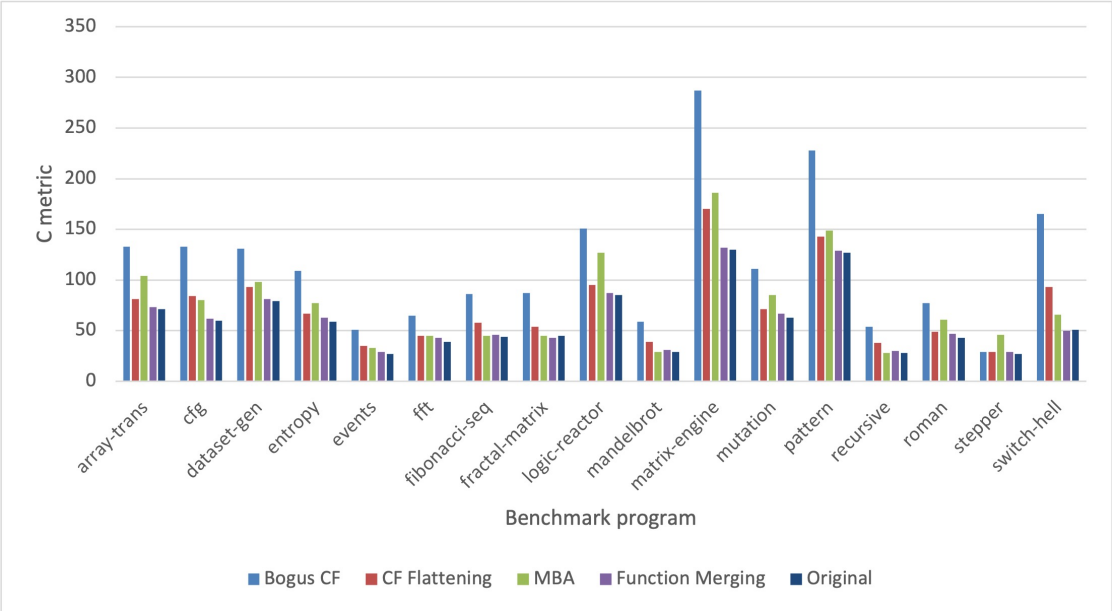


(a) Artificial benchmark group

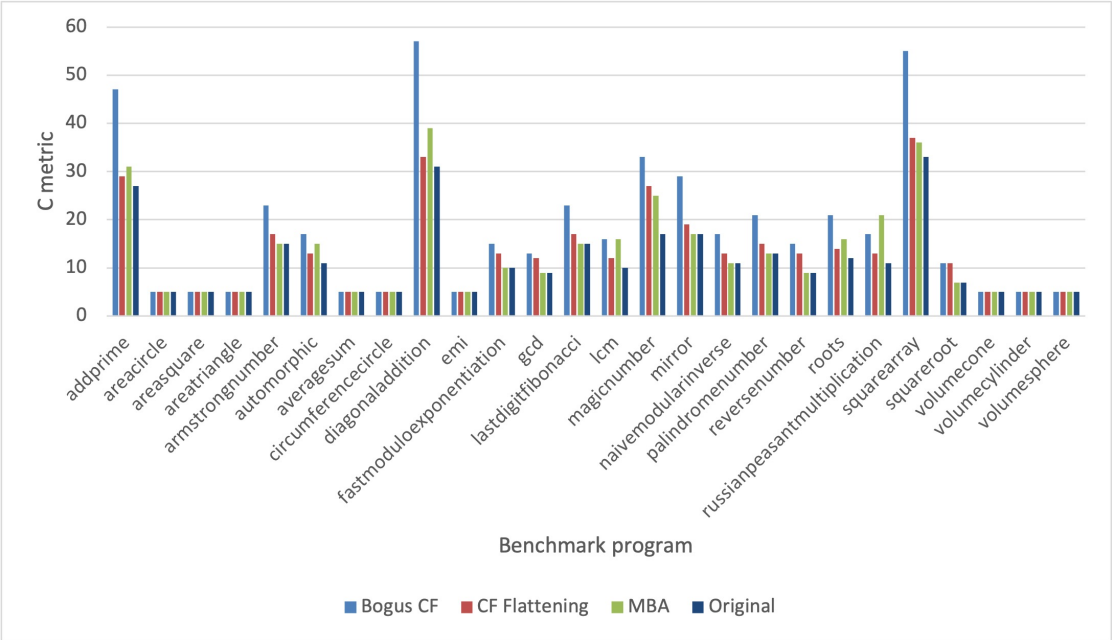


(b) OSAGE mathematical benchmark group

Figure 5.3: B metric score



(a) Artificial benchmark group



(b) OSAGE mathematical benchmark group

Figure 5.4: C metric score

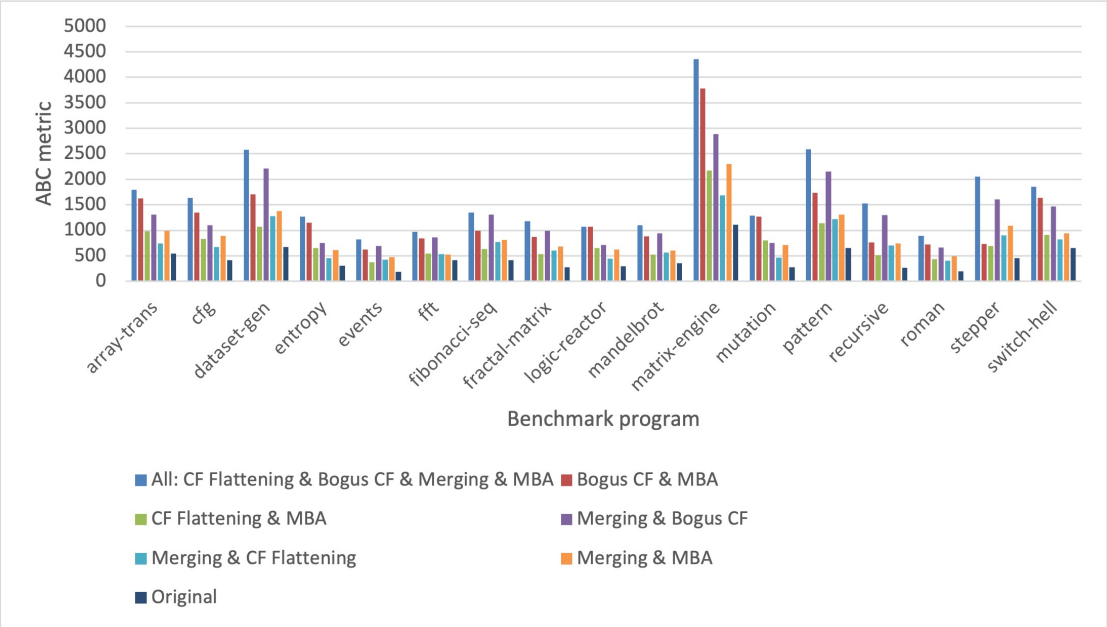


Figure 5.5: ABC metric score of artificial benchmark group for combinations of obfuscation techniques

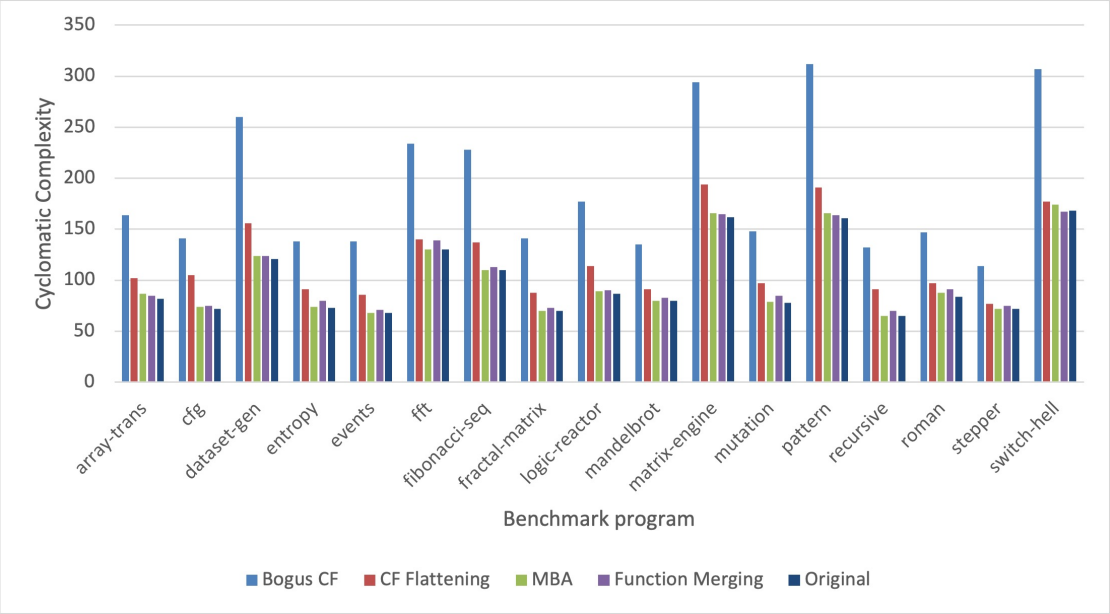
5.4.3 Cyclomatic Complexity

Cyclomatic Complexity evaluates a number of unique control flow paths in the code, and directly reflects structural complexity, covering loops and conditional branches.

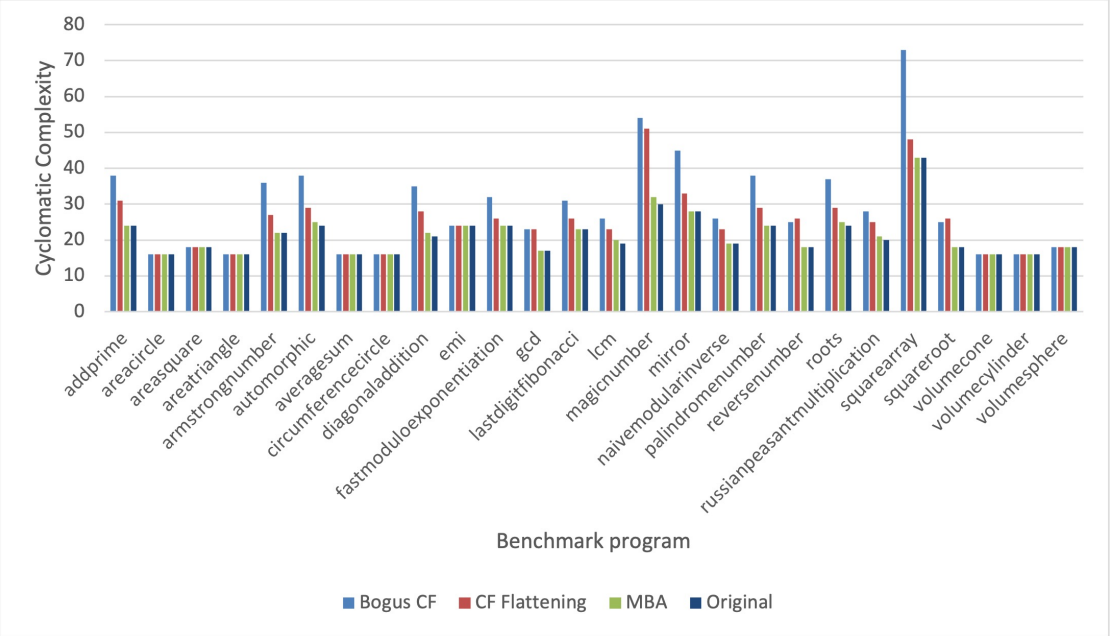
As visualised in Figure 5.6, similarly to ABC metric, Bogus Control Flow almost doubles Cyclomatic Complexity in the artificial benchmark group, but has a lighter influence on the OSAGE group, which is explained by the fact that the second set is composed of simpler and smaller programs.

MBA and Function Merging both have almost no impact on Cyclomatic Complexity, of under 1%. This is an expected outcome, because MBA simply substitutes an instruction by a consistent sequence of other instructions and does not create any conditional branches, while Function Merging generates a single dispatcher for the merged functions, acting as an alternative to previously used separate functions.

In terms of unique flow paths, Flattening obfuscation is similar to Function Merging, while both create a dispatcher mechanism to orchestrate existing basic blocks. For this reason, Control Flow Flattening similarly has no significant effect on Cyclomatic Complexity, increasing it by an average of 5%, because of additional preparation steps required to initialize the dispatcher.



(a) Artificial benchmark group



(b) OSAGE mathematical benchmark group

Figure 5.6: Cyclomatic Complexity score

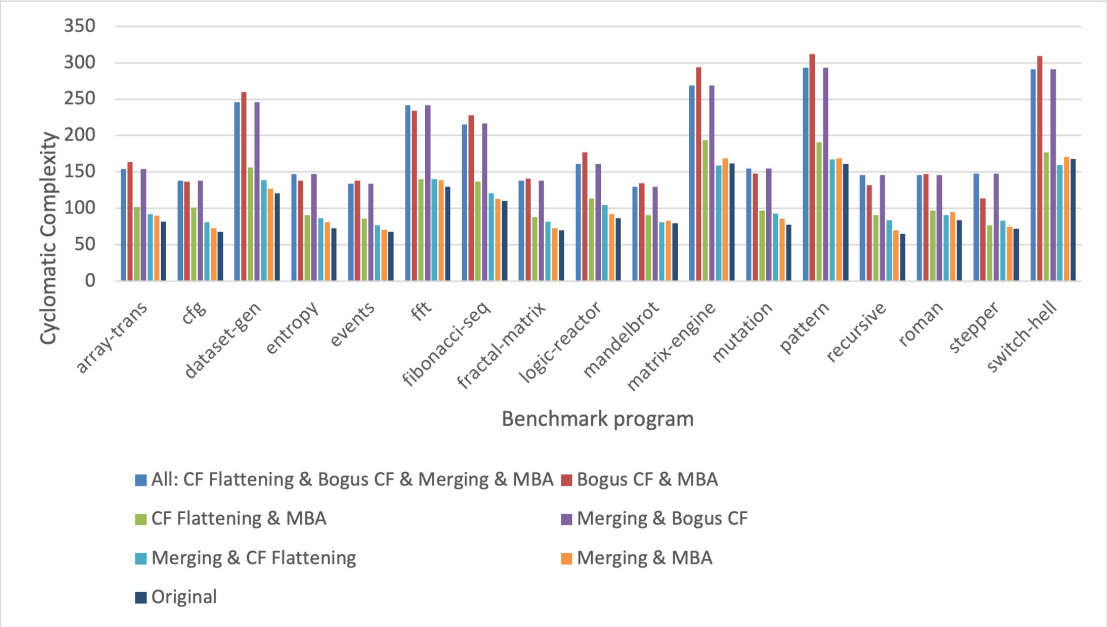


Figure 5.7: Cyclomatic Complexity score of artificial benchmark group for combinations of obfuscation techniques

Obfuscation Combinations

Figure 5.7 represents an effect of combinations of obfuscations on the artificial set of programs. Similarly to ABC metric, the combinations involving Bogus Control Flow produce the highest Cyclomatic Complexity scores, with an overhead reaching up to 50% of the original scores. All programs transformed by Bogus Control Flow with MBA, Bogus Control Flow with Function Merging, and a combination of all 4 techniques demonstrate approximately identical effect, except for a *stepper* entry, where Bogus Control Flow with MBA score equals 100, while two other combinations each reached a complexity of 150. This change is explained by the internal characteristics of the *stepper* program.

Listing 5.1 presents a snippet of the source code of *stepper* benchmark program, located in the artificial test group. The code is composed of 12 simple functions, and for each of them a control flow graph is constructed out of 2 to about 5 basic blocks, depending on the function body. Due to this structure, the *stepper* entry is hardly influenced by obfuscations involving control flow transformations, namely Control Flow Flattening and Bogus Control Flow, because graph transformations cannot produce significant effects for graphs composed of only several nodes. Meanwhile, Function Merging produces a drastic change in the internal structure of a program composed of a significant number of functions, which is enhanced by Control Flow Flattening extended by Bogus Control Flow, applied on a single unified function generated by Merging technique. For this reason, two combinations involving Function Merging together with Bogus Control Flow provide the highest scores for the *stepper* program.

```c
#define N 8

static void step1(int *data, int val) {
  for (int i = 0; i < N; i++)
    data[i] = (val * (i + 1)) % 91;
}

static void step2(int *data) {
  for (int i = 0; i < N; i++)
    if (data[i] % 2 == 0)
      data[i] += 5;
    else
      data[i] -= 3;
}

static void step3(int *data) {
  for (int i = 0; i < N; i++)
    data[i] = (data[i] * data[i]) % 127;
}

static void step4(int *data) {
  for (int i = 1; i < N; i++)
    data[i] += data[i - 1];
}

static void step5(int *data, int mod) {
  for (int i = 0; i < N; i++)
    data[i] = (data[i] + mod * i) % 100;
}

// step6, step7, step8, step9, step10, step11 follow a similar structure

static void step12(int *data) {
  for (int i = 0; i < N; i++)
    printf("%d ", data[i]);
  printf("\n");
}

int main(int argc, char *argv[]) {
  int args[4];
  for (int i = 1; i <= 4; i++)
    args[i - 1] = atoi(argv[i]);

  int data[N];

  step1(data, args[0]);
  step2(data);
  step3(data);
  step4(data);
  step5(data, args[1]);
  // step6, step7, step8, step9, step10, step11 calls
  step12(data);
  return 0;
```

```
54 }
```

Listing 5.1: Source code of the *stepper* program from the artificial benchmark group consists of a number of simple functions

5.4.4 Lines of Code

Lines of Code metric directly reflects the number of additional instructions introduced by obfuscation techniques.

Based on artificial benchmark group metrics depicted in Figure 5.8, Function Merging and MBA each add more lines of code than the other in approximately half of the cases, meaning that it heavily depends on the fraction of possible expression substitutions and the total number of functions in the source code.

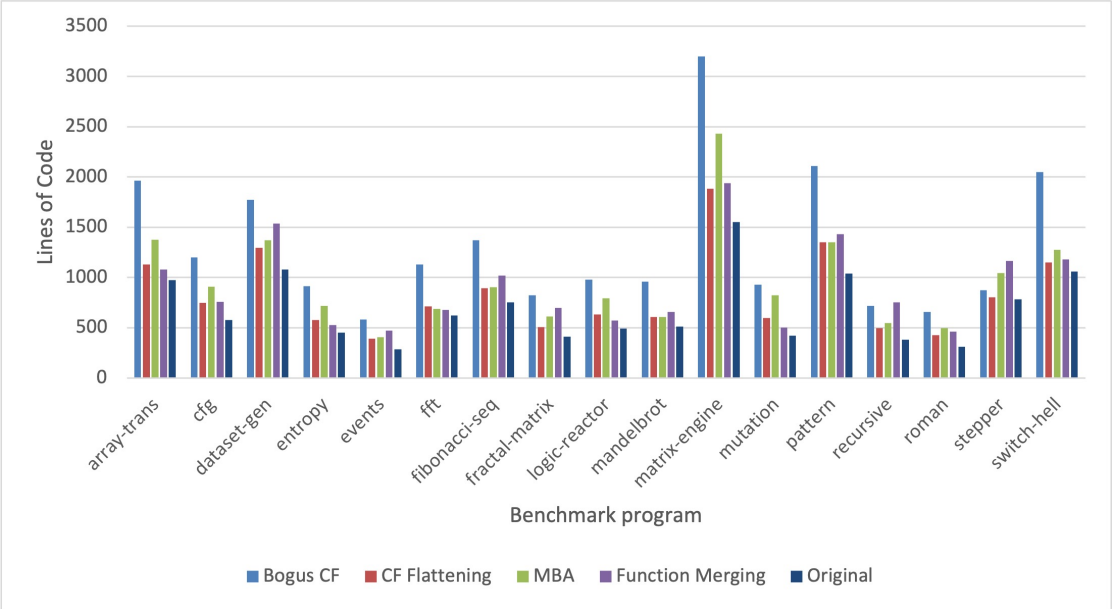
For the majority of code samples, Control Flow Flattening adds around 10% of additional lines of code, and only for a few programs, such as *squarearray*, *events* and *magicnumber*, it raises the score up to 40% over the initial value.

Similarly to other metrics, Bogus Control Flow obfuscation leads to the most significant overhead in terms of Lines of Code, almost doubling the score for $switch - hell$, *pattern*, and $matrix - engine$ entries, while MBA also adds up to 50% of additional instructions. Table 5.2 visualises the correlation between Lines of Code and other complexity-related software metrics. To allow comparison, metric values are transformed into relative scores indicating the effect of particular obfuscation in terms of how much it transformed the score of the original version of a program.

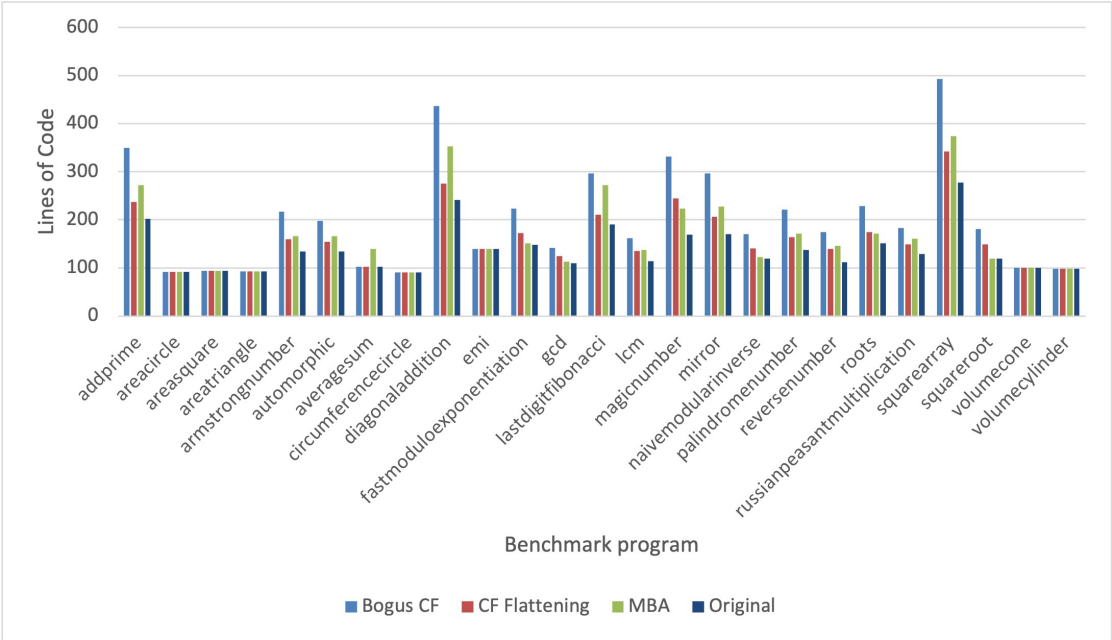
Comparison reveals that Bogus Control Flow scores of ABC metric, Cyclomatic Complexity, and Lines of Code for the chosen programs differ by no more than 25%, which shows that for this obfuscation, additional lines of code are accounted in complexity metrics, covering independent execution paths and various types of instructions. On the other hand, a large increase of Lines of Code metric for MBA obfuscation is reflected only in ABC metric, but not in Cyclomatic Complexity, which is only increased by MBA up to several percent. It is perfectly aligned with the nature of MBA obfuscation, which produces only additional assignment instructions (A component in ABC) but not branches, accounted by Cyclomatic Complexity.

Target	Metric	Bogus CF	CF Flattening	MBA	Function Merging
$switch - hell$	ABC	1.89	1.07	1.26	1.22
	Cyclomatic Complexity	1.83	1.05	1.04	0.99
	LOC	1.93	1.09	1.20	1.12
pattern	ABC	2.12	1.32	1.41	1.60
	Cyclomatic Complexity	1.94	1.19	1.03	1.02
	LOC	2.03	1.30	1.30	1.38
$matrix - engine$	ABC	2.06	1.20	1.73	1.35
	Cyclomatic Complexity	1.81	1.20	1.02	1.02
	LOC	2.06	1.21	1.57	1.25

Table 5.2: Correlation between largest Lines of Code overhead and complexity metric scores



(a) Artificial benchmark group



(b) OSAGE mathematical benchmark group

Figure 5.8: Lines of Code score

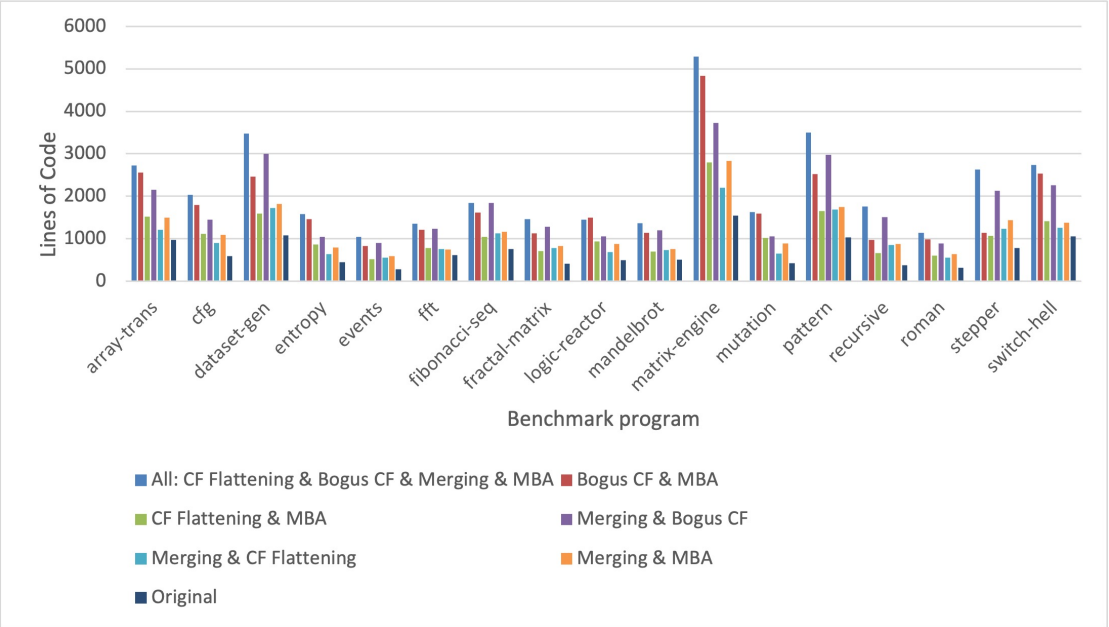
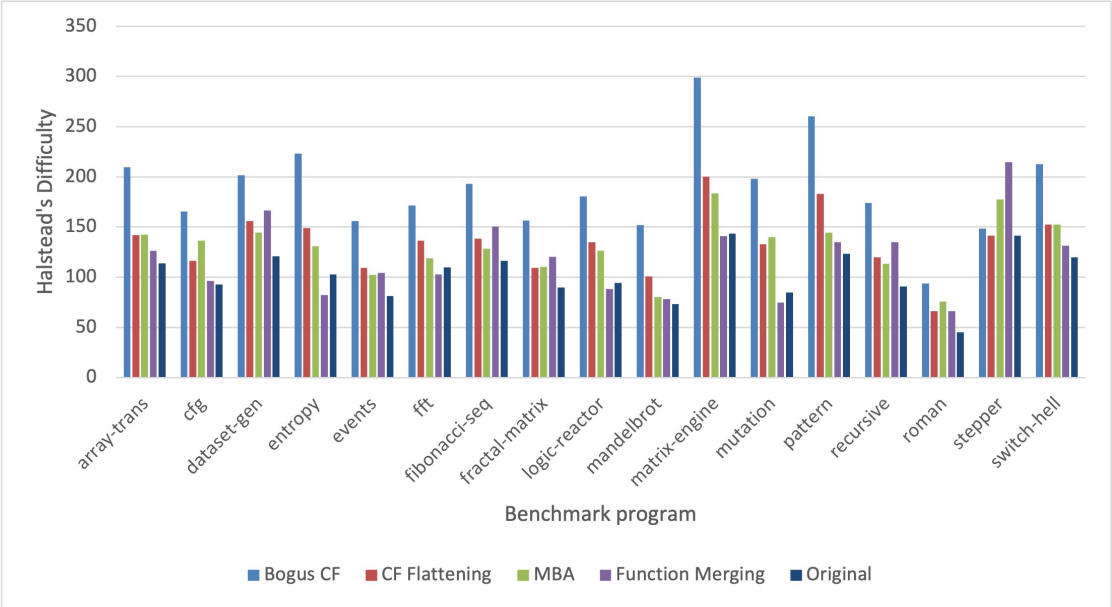


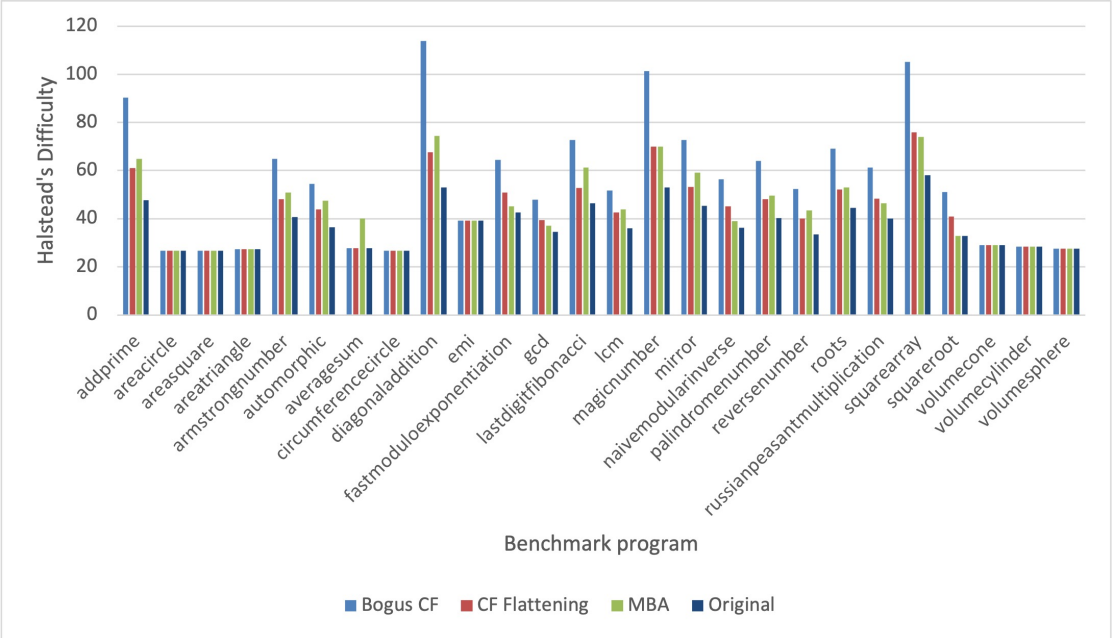
Figure 5.9: Lines of Code score of artificial benchmark group for combinations of obfuscation techniques

5.4.5 Halstead Difficulty

In terms of score distribution among obfuscations, Halstead Difficulty shown in Figure 5.10 is similar to ABC metric for both benchmark groups. Thus, Control Flow Flattening, MBA and Function Merging result in additional 10% to 30% of the original score. Bogus Control Flow produces the highest scores for all code samples except for *stepper* program, which obtains the highest score on Function Merging. It is explained by the internal structure of the program shown in Listing 5.1 - it is composed of 12 small functions, a couple lines of code each, so Function Merging has a broader effect compared to control flow modification employed inside the functions.



(a) Artificial benchmark group



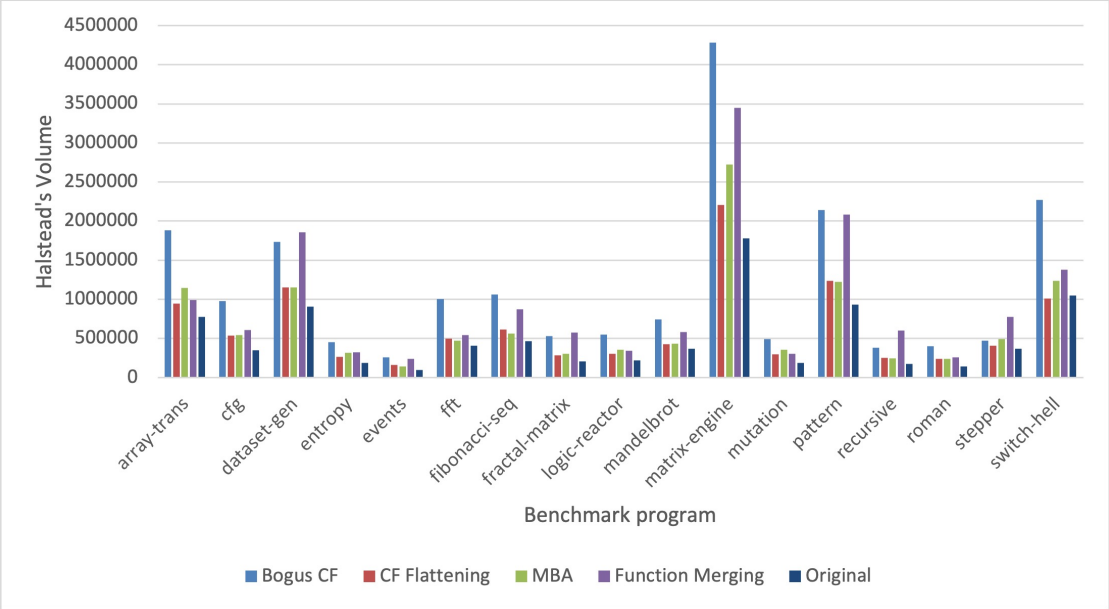
(b) OSAGE mathematical benchmark group

Figure 5.10: Halstead Difficulty score

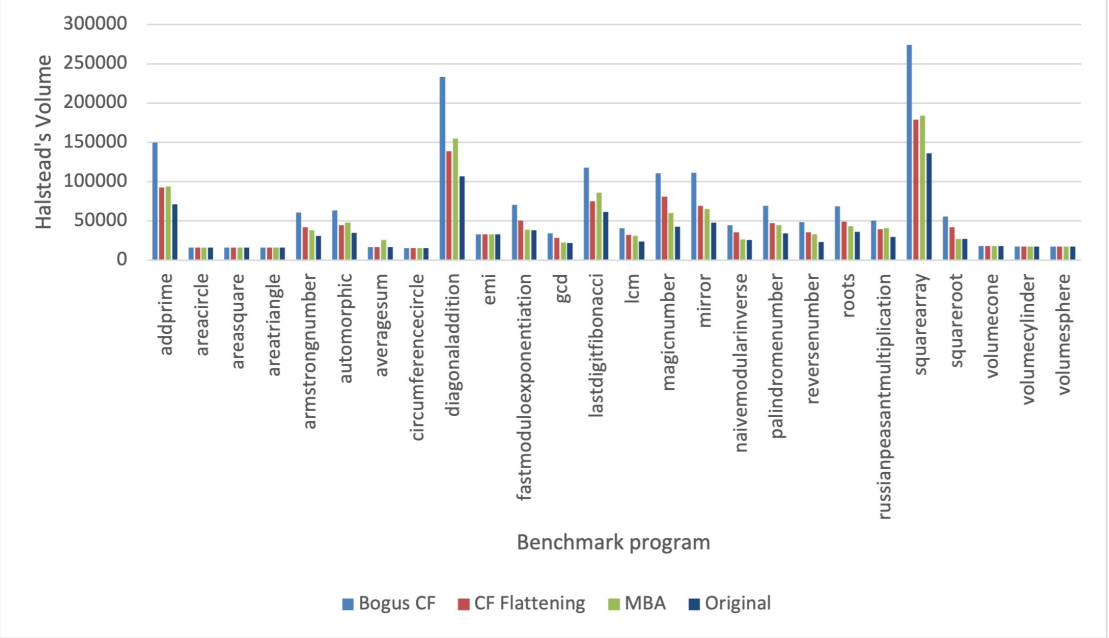
5.4.6 Halstead Volume

Halstead Volume reflects code size in terms of informational content based on the number of unique operators and operands, and their total number. According to the benchmark results depicted in Figure 5.11, Function Merging produces a larger increase in the original Volume score, compared to the corresponding values of Lines of Code metric (Figure 5.8). For example, $matrix - engine$ program has doubled the Halstead Volume score after Merging, while Lines of Code score has increased by only around 30%.

Function Merging and Bogus Control Flow exhibit the highest Halstead Volume scores in both artificial and OSAGE benchmark groups, such as for $events$, $stepper$, $dataset - gen$, $matrix - engine$, and $squarearray$. It correlates with an intuitive interpretation of Halstead Volume, describing how much effort is required to understand the program considering code syntax and semantics, rather than the total number of instructions as in Lines of Code. Formally, this trend is explained strictly based on the Halstead Volume formula: $V = N * log_2(n)$, where N is a total number of operators and operands, and n is a number of distinct operators and operands. Bogus Control Flow performs a straightforward code duplication inside function bodies, which increases the program's number of instructions and increases both N and n components. On the other hand, Function Merging has a slight influence on N by adding a dispatcher mechanism inside the unified function, but drastically increases n by merging function arguments and assigning them unique identifiers, and additionally renaming temporary variables allocated inside functions. For example, *stepper* functions shown in Listing 5.1 use the same parameter named *data*, while the unified function would generate a unique name for each merged argument.



(a) Artificial benchmark group



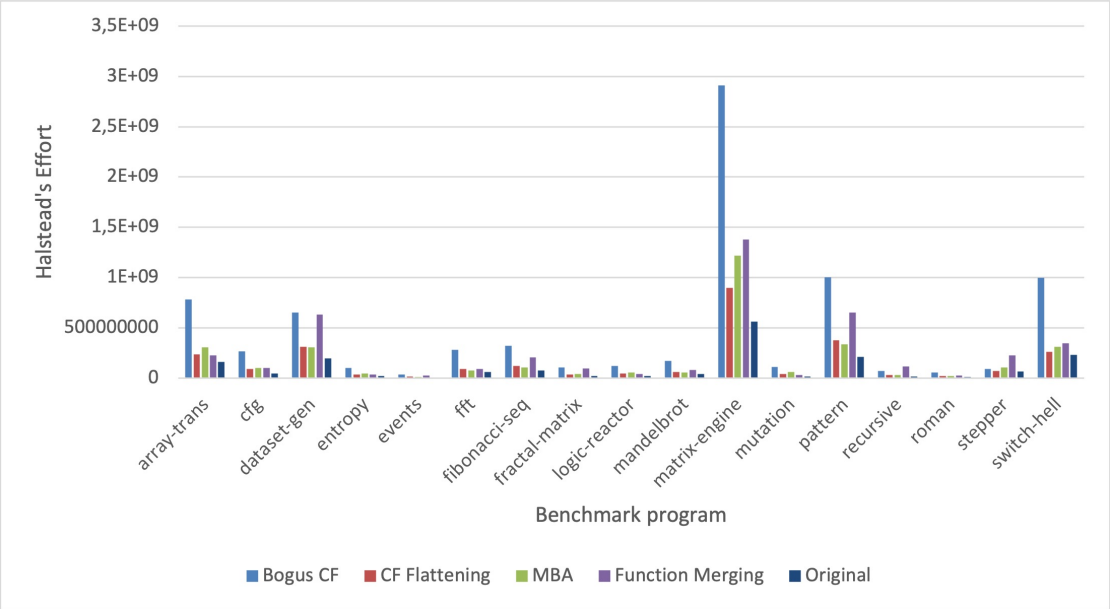
(b) OSAGE mathematical benchmark group

Figure 5.11: Halstead Volume score

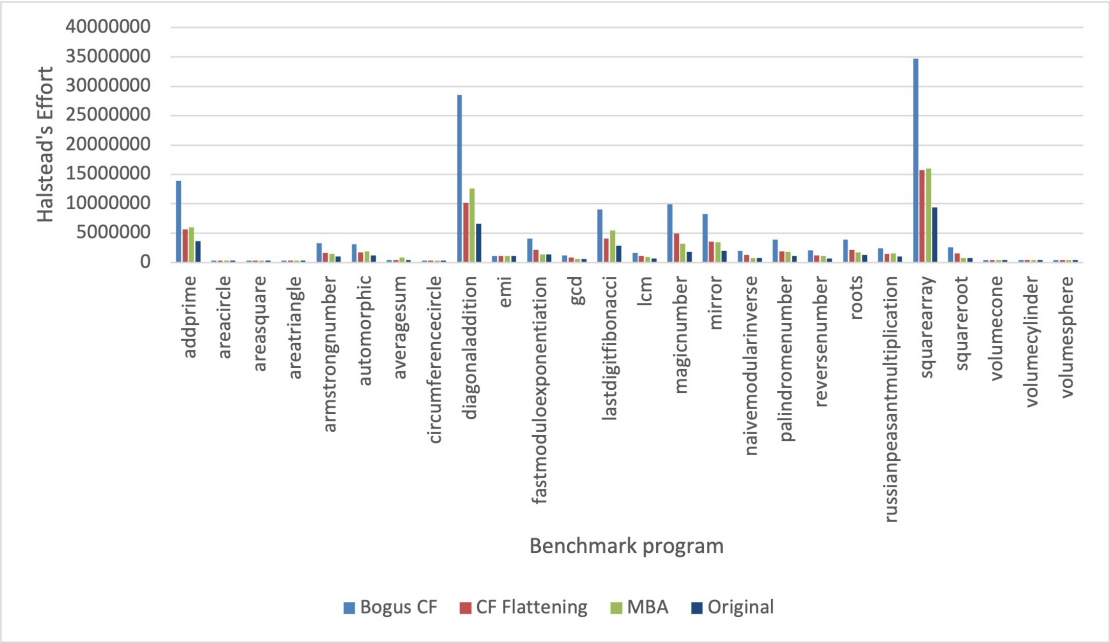
5.4.7 Halstead Effort

Halstead Effort accumulates both volume and difficulty and produces a comprehensive metric to evaluate the effort required to understand and maintain the code. As visualised in Figure 5.12, Bogus Control Flow leads to highly different, outlying scores than other obfuscating techniques and the original version, with values reaching 4 times that of the corresponding original scores, for example for $matrix - engine$ of the artificial set and *diagonaladdition* of the OSAGE group.

Another trend of Halstead Effort metric is its high variation among both sets of benchmark programs. For example, *squarearray* and *diaginaladdition* scores for Bogus Control Flow versions are over 25 million, while other entries of OSAGE benchmark group are between 1 and 4 million. By definition, Halstead Effort is calculated as a multiplication of difficulty and volume both resolving around the number of operands and operators in the code, and while all OSAGE programs achieve almost equal difficulty score, their volume score indicates the same outliers - *squarearray* and *diaginaladdition*. Thus, effort highlights this difference by aggregating both metrics.



(a) Artificial benchmark group



(b) OSAGE mathematical benchmark group

Figure 5.12: Halstead Effort score

5.4.8 Summary

Figure 5.13 visualises average relative software metrics scores of the obfuscated programs, compared to the original programs, aggregating metrics across both OSAGE and the mathematical benchmark suite:

$$AvgScore = AVG(score_{obfuscated}/score_{original})$$

- Control Flow Flattening results in an average overhead of under 25% for all metrics except for Halstead Effort, where it reaches 50%.
- Bogus Control Flow produces an overhead of slightly over 50% for ABC, Cyclomatic Complexity and Lines of Code, and of around 90% for Halstead Volume. Halstead Effort reaches up to 3.3 as much as of the original program's score.
- Function Merging overhead varies greatly for A and ABC metrics, in average of 50%, and almost does not affect B, C metrics, and Cyclomatic Complexity. In average, the increase is up to 20% for Halstead Difficulty, around 40% for Lines of Code, 90% for Halstead Volume, and about 175% for Halstead Effort.
- MBA expressions produces an average overhead in the range of 30% to 55% for A, ABC metrics, and Halstead Effort, and under 25% for other metrics, causing no increase in B metric and Cyclomatic Complexity.

The summary accumulates metrics across all programs and does not reflect a high dispersion of scores present for certain metrics, such as ABC metric with Function Merging obfuscation (Figure 5.1a).

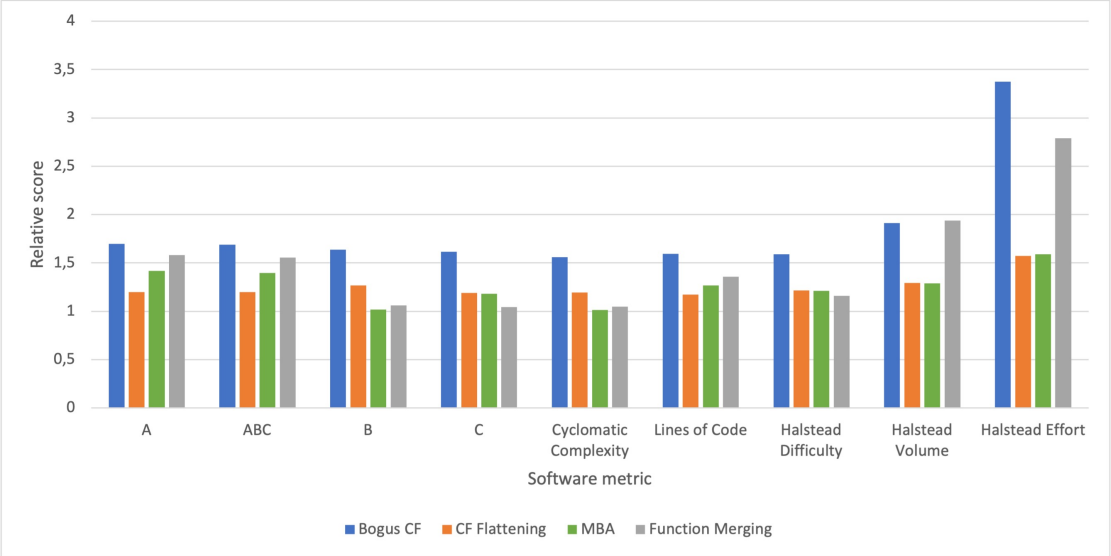


Figure 5.13: Average relative software metric scores of obfuscated benchmark programs, compared to the original programs. A value of 1 stands for the original program's metric score

5.5 Performance. Runtime Behaviour

5.5.1 Real time and CPU time

Benchmark program execution is tracked using two metrics - real time and CPU time. These two measurements describe fundamentally different aspects of the code runtime characteristics.

CPU time, also known as process time, refers to the amount of time for which a CPU is exclusively executing the instructions of a particular process. It is typically composed of the user time, spent executing code in user space, and system time, spent within kernel-level operations on behalf of the current process. The core difference between CPU time and real time is that CPU time does not account for periods the process spends in an idle state while waiting for external resources, such as input and output operations.

In contrast, real time refers to the total elapsed time from the start to the completion of a program, acting as a real-life stopwatch. It includes CPU execution time and also all intervals of inactivity, such as waiting for disk reads and writes completion, synchronisation with other processes, and time spent in the scheduler queue. Thus, real time provides a view of the program's behaviour from an end-user perspective, encapsulating the overall duration of code execution in the real setup.

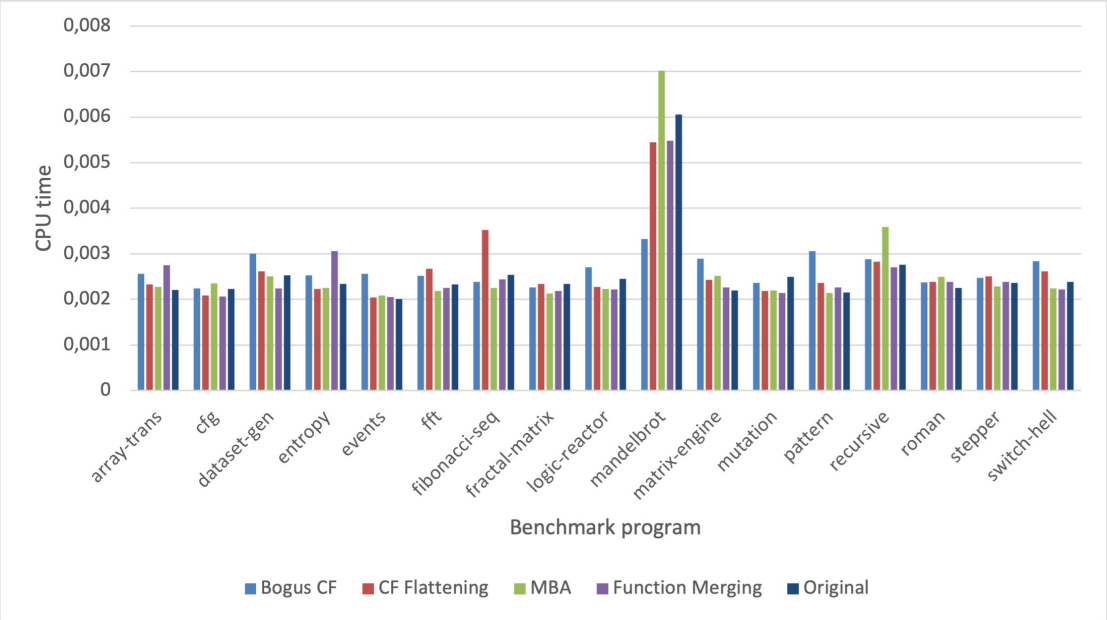
CPU time provides more informative insights into the relative program's complexity in the context of performance profiling, while versions of the same code sample differ only in control flow structure, ordering of instructions, in-memory operations, and design of internal abstractions such as functions. However, real time suggests a broader user-

oriented view on the program execution and needs to also be studied in conjunction with CPU time.

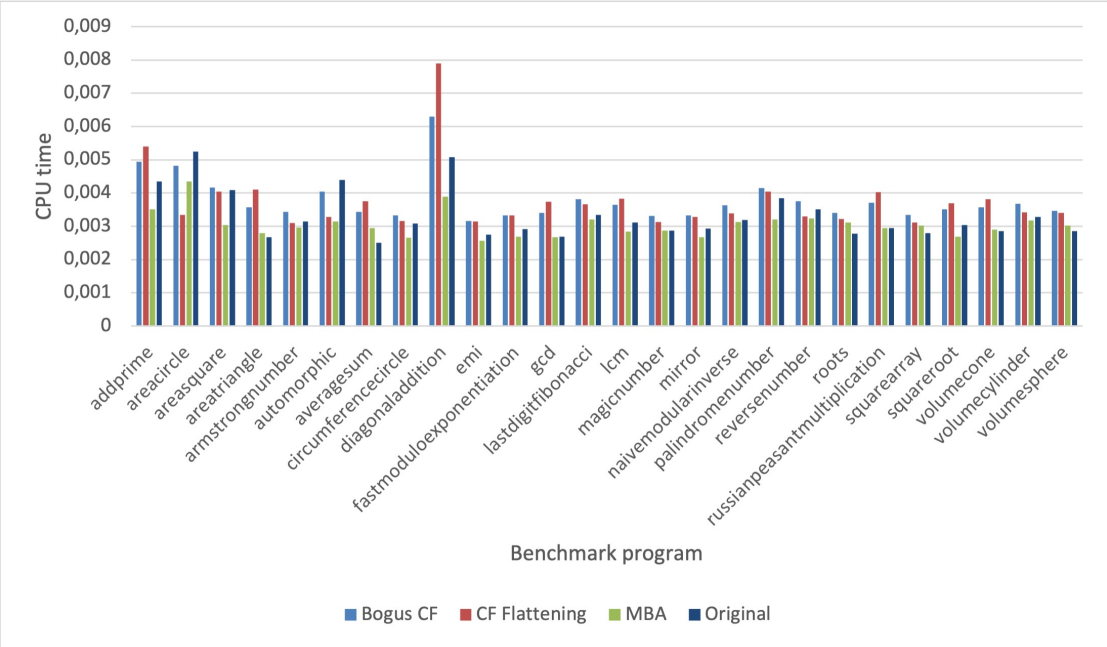
5.5.2 Benchmark Results

Figure 5.14 and Figure 5.15 visualise CPU time and total execution time of benchmark programs. The measurements show that there is no correlation between obfuscating transformations and execution time, meaning that obfuscated binaries demonstrate the same performance as the original executables. Combinations of obfuscating techniques displayed in Figure 5.16 and Figure 5.17 produce a similar outcome. Occasional outliers, such as for *mandelbrot* entry, are not consistent and do not follow any pattern throughout the entire benchmark set.

There are several factors explaining the execution profiling results. First, the absolute CPU time for programs in both target sets remains around 0.0025 seconds, which can be too small and error-prone to produce any recognisable patterns. Then, the absolute size and complexity of benchmark code are not sufficient to be significantly affected by obfuscation techniques and result in performance overhead. And finally, the designed transformations are actually lightweight enough to be seamlessly integrated into target programs without affecting their performance, considering implementation details of obfuscations. Namely, Bogus Control Flow consistently doubled complexity metric scores and according to software metrics it poses a possible performance overhead, but essentially it generates reachable duplicated code sections which have the exact same complexity as their original counterparts, and the only slight potential performance overhead lies in the additional options in a *switch* dispatcher generated by Control Flow Flattening, that may complicate branch predictions in a CPU. As for Control Flow Flattening, it did not produce outlying scores for complexity-oriented or size-oriented metrics, similarly to Function Merging and MBA expressions.

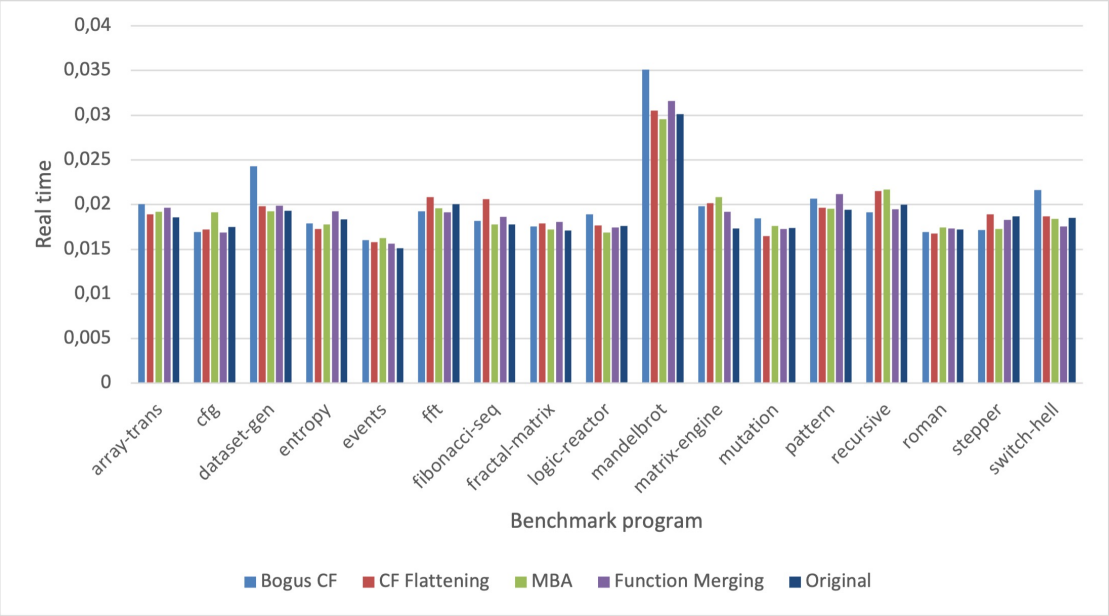


(a) Artificial benchmark group

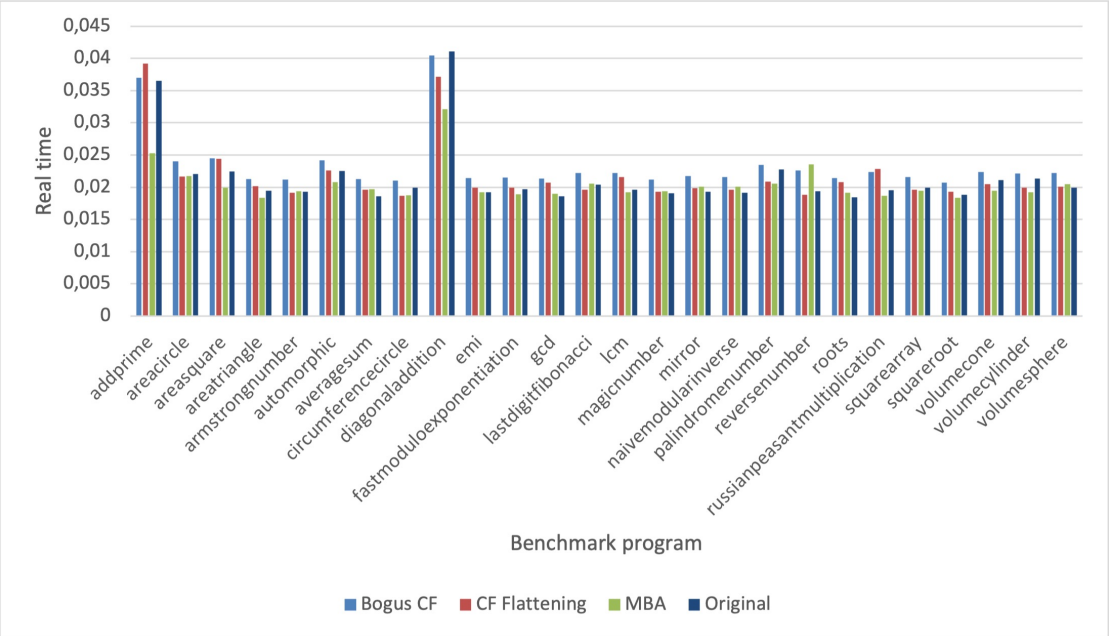


(b) OSAGE mathematical benchmark group

Figure 5.14: CPU execution time (in seconds)



(a) Artificial benchmark group



(b) OSAGE mathematical benchmark group

Figure 5.15: Real execution time (in seconds)

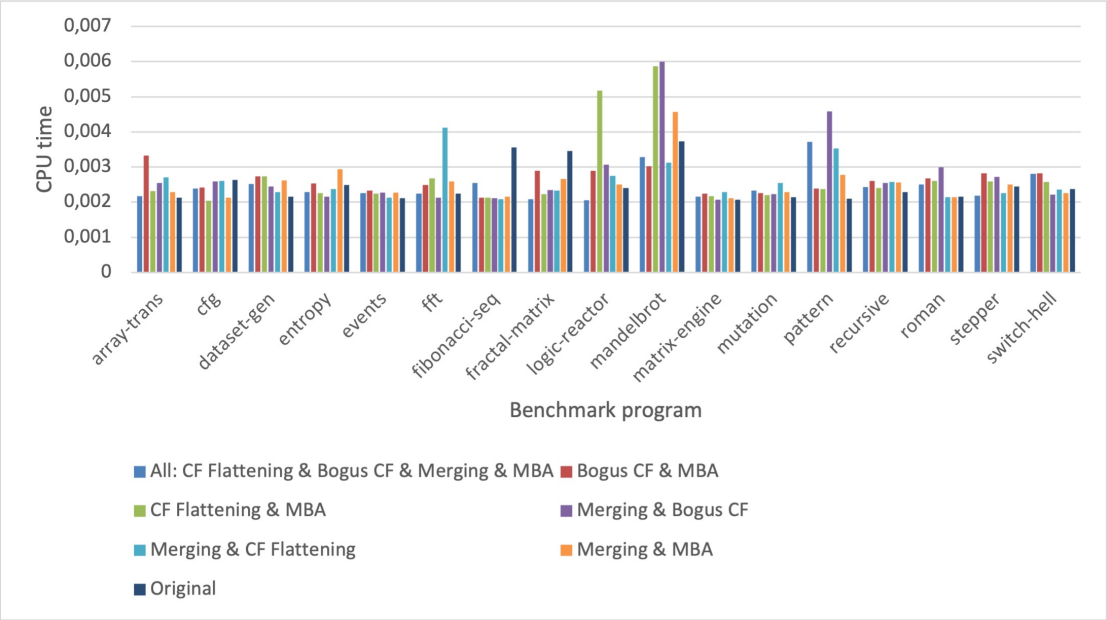


Figure 5.16: CPU execution time (in seconds) of artificial benchmark group for combinations of obfuscation techniques

Figure 5.17: Real execution time (in seconds) of artificial benchmark group for combinations of obfuscation techniques

6 Future Work

In the scope of this thesis, 4 static obfuscation techniques operating on LLVM IR level are implemented and profiled using a set of benchmark programs. However, there is still work that can be done to improve this implementation, namely, extensive performance evaluation and validation.

6.1 Extended Validation

In the current implementation, an obfuscation is considered valid if it preserves program's behaviour, meaning that for every program from a benchmark group, the obfuscated program and the original program produce identical outputs for the same inputs. This approach can be complemented by employing external tools to compare original and obfuscated binaries or IR files which compare their semantics. For example, Lopes et al. [29] suggest a publicly available web tool which checks if two programs have the same semantics and is claimed to produce only false negative errors. However, in practice, the instrument produces many false alarms when a code contains loops because the validation is based on loop unrolling, although it is possible to set a higher loop unrolling factor and run the validator on a host machine with sufficient computational resources.

6.2 Performance Profiling

Real time and CPU time profiling on a set of benchmark programs reveals no correlation between the execution time and applied obfuscation techniques, and it may happen because of the relatively small absolute execution time of around 0.02 seconds for real time and 0.0025 seconds for CPU time. Targeting time-consuming programs with more complex control flow graphs and intricate internal structure, such as the ones with dozens of interconnected functions, would yield more consistent runtime measurements and provide a comprehensive analysis of the relationship between obfuscation and performance.

Potential benchmark programs would include cryptographic algorithms, physic equations, algebra problem solvers, and open-source GNU core utilities[1]. For open-source implementations in C, the compilation setup poses a main challenge, requiring specific versions of core libraries linked to the host architecture and particular compiler. Additionally, it is essential that benchmark programs are logic oriented rather than data oriented, meaning that both real time and CPU time cannot be properly evaluated on programs which perform data-intensive operations, such as calculating large in-memory matrices in

[1] `https://github.com/coreutils/coreutils` (Accessed: 2025/04/10)

a simple cycle, as it may take a large amount of CPU time and interfere with performance measurements, while the program is hardly affected by obfuscations.

7 Conclusion

Software products deployed in untrusted environments and distributed in binary form become particularly vulnerable targets for reverse engineering, tampering, and intellectual property extraction, compromising the confidentiality and integrity of the system. Attackers apply dynamic and static analysis tools to reconstruct the program logic, gain a deeper understanding its internal structure and locate potential attack surface, such as cryptography routines, licence checks, proprietary algorithms, or security-critical sections. Obfuscation techniques obscure the code while preserving its semantics in order to complicate analysis and increase the amount of time and effort required to perform reverse engineering attacks and gain knowledge about the program's structure. In practice, obfuscation is actively employed by malware authors striving to prevent detection by anti-virus mechanisms, while game and mobile app developers use it to reduce the risk of cheating, tampering, or exposing confidential data. At the moment of the research, there are no well-known publicly available static obfuscator tools operating on the IR level of the code, although there are a number of related projects that rely on outdated LLVM versions or deploy source-to-source obfuscation.

In the scope of this study, multiple static obfuscation techniques are implemented in a form of modular LLVM transform passes, operating on the IR level of the program and applied at compile-time during optimisation. The transformations include Control Flow Flattening and its Bogus Control Flow extension, Function Merging, and instruction substitution based on MBA expressions, utilizing custom expression templates specifically designed as part of this project. Additionally, to provide developers a fine-grained control over the obfuscation setup, a mechanism is implemented to allow the annotation of target functions directly in the source code. The annotations are preserved through the compilation process and interpreted by the obfuscation passes, which then selectively apply transformations only to the functions marked with a corresponding obfuscation tag. This approach allows developers to determine which code sections require protections and which techniques are appropriate, considering that obfuscations differ from each other in terms of targeted abstractions, such as instructions or control flow, and performance overhead.

The implemented obfuscation techniques are validated for preserving the original program behaviour and evaluated in terms of execution time and software metrics. Profiling is focused on aspects such as control flow and data flow complexity, code volume, and maintainability, and employs well-established software metrics, including Cyclomatic Complexity and Halstead metrics, to compare the obfuscated binaries against the baseline scores of the original code. Both validation and profiling are deployed using a benchmark suite of 47 programs with high output entropy and strongly dependent on the inputs, which includes real-world mathematical algorithms and synthetic programs targeting complex

data transformations, designed exclusively for this project and intended to represent complex control flow to ensure comprehensive profiling results. Software metric scores reveal that Bogus Control Flow almost doubles complexity scores for all metrics, Function Merging and MBA expressions can double only ABC metric, and other combinations produce an overhead of up to around 30%. However, runtime profiling revealed that real time and CPU time are not affected by any of the protection techniques, and obfuscated binaries demonstrate the same performance as the original programs.

Looking forward, the suggested implementation can be further extended by extensive semantics validation and performance profiling on larger benchmark suites, as well as designing additional obfuscation techniques following the used approach.

Bibliography

[1] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. *Journal of Network and Computer Applications*, 48:44–57, 2015.

[2] David E Bakken, R Rarameswaran, Douglas M Blough, Andy A Franz, and Ty J Palmer. Data obfuscation: Anonymity and desensitization of usable data sets. *IEEE Security & Privacy*, 2(6):34–41, 2004.

[3] Vivek Balachandran and Sabu Emmanuel. Potent and stealthy control flow obfuscation by stack based self-modifying code. *IEEE Transactions on Information Forensics and Security*, 8(4):669–681, 2013.

[4] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200, 2016.

[5] Jan Cappaert and Bart Preneel. A general model for hiding control flow. In *Proceedings of the tenth annual ACM workshop on Digital rights management*, pages 35–42, 2010.

[6] Mariano Ceccato, Massimiliano Di Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19:1040–1074, 2014.

[7] Jien-Tsai Chan and Wuu Yang. Advanced obfuscation techniques for java bytecode. *Journal of systems and software*, 71(1-2):1–10, 2004.

[8] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations, 1997.

[9] Juan Carlos de la Torre, Javier Jareño, José Miguel Aragón-Jurado, Sébastien Varrette, and Bernabé Dorronsoro. Source code obfuscation with genetic algorithms using llvm code optimizations. *Logic Journal of the IGPL*, page jzae069, 2024.

[10] Philippe Biondi Fabrice Desclaux. Silver needle in the skype, 2006.

[11] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In *Security and privacy in communication networks: 14th international conference, secureComm 2018, Singapore, Singapore, August 8-10, 2018, proceedings, part i*, pages 172–192. Springer, 2018.

[12] Paolo Falcarin, Christian Collberg, Mikhail Atallah, and Mariusz Jakubowski. Guest editors' introduction: Software protection. *IEEE Software*, 28(2):24–27, 2011.

[13] Jerry Fitzpatrick. Applying the abc metric to c, c++, and java. *C++ report*, 1997.

[14] Peter Garba and Matteo Favaro. Saturn-software deobfuscation framework based on llvm. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 27–38, 2019.

[15] Ghidra. `https://ghidra-sre.org/`. (Accessed: 2025/04/12).

[16] Gimple. `https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html`. (Accessed: 2025/02/10).

[17] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[18] Hikari obfuscator. `https://github.com/HikariObfuscator/Hikari`, 2019. (Accessed: 2025/03/15).

[19] George Hotz. Hello hypervisor, I'm geohot. `https://web.archive.org/web/20100129034435/http://geohotps3.blogspot.com/2010/01/hello-hypervisor-im-geohot.html`, 2010. (Accessed: 2025/04/20).

[20] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm–software protection for the masses. In *2015 ieee/acm 1st international workshop on software protection*, pages 3–9. IEEE, 2015.

[21] Seoyeon Kang, Sujeong Lee, Yumin Kim, Seong-Kyun Mok, and Eun-Sun 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*, pages 309–311, 2021.

[22] Yuichiro Kanzaki, Akito Monden, and Christian Collberg. Code artificiality: a metric for the code stealth based on an n-gram model. In *2015 IEEE/ACM 1st international workshop on software protection*, pages 31–37. IEEE, 2015.

[23] Dhiru Kholia and Przemysław Węgrzyn. Looking inside the (drop) box. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, 2013.

[24] Michael Lackner, Reinhard Berlach, Reinhold Weiss, and Christian Steger. Countering type confusion and buffer overflow attacks on java smart cards by data type sensitive obfuscation. In *Proceedings of the First Workshop on Cryptography and Security in Computing Systems*, pages 19–24, 2014.

[25] Tımea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30(1):3–19, 2009.

[26] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, 2003.

[27] Binbin Liu, Weijie Feng, Qilong Zheng, Jing Li, and Dongpeng Xu. Software obfuscation with non-linear mixed boolean-arithmetic expressions. In *Information and Communications Security: 23rd International Conference, ICICS 2021, Chongqing, China, November 19-21, 2021, Proceedings, Part I 23*, pages 276–292. Springer, 2021.

[28] Llvm. `https://llvm.org/`. (Accessed: 2025/02/10).

[29] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 65–79, 2021.

[30] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, SE-2(4):308–320, 1976.

[31] Gary McGraw. Software security. *IEEE Security & Privacy*, 2(2):80–83, 2004.

[32] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015(S 91):1–91, 2015.

[33] Jasvir Nagra and Christian Collberg. *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Pearson Education, 2009.

[34] Justus Polzin. Mixed boolean-arithmetic generator. `https://plzin.github.io/posts/mba`, 2023. (Accessed: 2025/04/30).

[35] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.

[36] Cagri Sahin, Philip Tornquist, Ryan McKenna, Zachary Pearson, and James Clause. How does code obfuscation impact energy usage? In *2014 IEEE international conference on software maintenance and evolution*, pages 131–140. IEEE, 2014.

[37] Sebastian Schrittwieser and Stefan Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *Information Hiding: 13th International Conference, IH 2011, Prague, Czech Republic, May 18-20, 2011, Revised Selected Papers 13*, pages 270–284. Springer, 2011.

[38] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *Acm computing surveys (csur)*, 49(1):1–37, 2016.

[39] Monirul Islam Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[40] Tigress. `https://tigress.wtf/`. (Accessed: 2025/03/16).

[41] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[42] Sharath K Udupa, Saumya K Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE'05)*, pages 10–pp. IEEE, 2005.

[43] KU Leuven University. Security flaws leave keyless tesla cars vulnerable to theft. `https://nieuws.kuleuven.be/en/content/2018/security-flaws-leave-keyless-tesla-cars-vulnerable-to-theft`, 2018. (Accessed: 2025/04/23).

[44] Wei Xu, Fangfang Zhang, and Sencun Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 9–16. IEEE, 2012.

[45] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy*, pages 674–691. IEEE, 2015.

[46] Dimitrios Zissis and Dimitrios Lekkas. Addressing cloud computing security issues. *Future Generation computer systems*, 28(3):583–592, 2012.