iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** IV **Month of publication:** April 2026

DOI: https://doi.org/10.22214/ijraset.2026.79512

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

Adaptive LLVM-Based Code Obfuscation Using a Random Forest Selection Framework

Ms. Sana Mateen[1], Logishetty Vaishnavi[2], Mohammed Omair Ahmed[3], Musham Praveen Kumar[4]
[1]*Department of Computer Science and Engineering, Methodist College of Engineering and Technology, Hyderabad, Telangana, 500001, India*
[2,3,4]*Department of Artificial Intelligence and Data Science, Methodist College of Engineering and Technology, Hyderabad, Telangana, 500001, India.*

Abstract: ***Compiled software distributed across open networks is under growing threat from advanced binary analysis instruments. Attackers equipped with disassemblers, decompilers, and symbolic execution platforms can reconstruct internal program logic, clone proprietary algorithms, and neutralise built-in security mechanisms without ever accessing the source. This work develops an adaptive obfuscation framework built on the Low-Level Virtual Machine infrastructure, targeting the platform-independent Intermediate Representation layer. A Random Forest classifier with one hundred decision trees analyses nine instruction-ratio features derived from each input program and autonomously assigns the most appropriate protection strategy from among four candidates: Control Flow Flattening, Instruction Substitution, XOR-based String Encryption, and All Passes Combined. The five-stage pipeline spans parsing, feature extraction, machine-learning-guided selection, pass application, and quantitative evaluation. Empirical testing across seven C programs yields instruction count growth of 94 to 118 per cent under control flow transformations, and byte entropy gains reaching 8.2 per cent when all passes execute together. The entire pipeline completes within two seconds, confirming suitability for continuous integration workflows. These outcomes demonstrate that LLVM constitutes a robust, portable, and extensible platform for intelligent compiler-level software protection.***
Keywords: Convolutional Neural Network, Compiler-Based Obfuscation, Intermediate Representation, Control Flow Flattening, Instruction Substitution, String Encryption, Random Forest Classifier, Adaptive Strategy Selection, Reverse Engineering Resistance, Software Security.

I. INTRODUCTION

Modern software systems, whether deployed in financial platforms, medical devices, defence applications, or consumer products, incorporate proprietary algorithms, licence verification logic, and security-critical routines that represent significant intellectual investment. Once a compiled binary reaches end-users or enters distribution channels, it becomes accessible to analysis without any access to source code. Contemporary reverse-engineering environments such as IDA Pro, Ghidra, Binary Ninja, and Radare2 combine static disassembly, data-flow analysis, and control-flow graph reconstruction to let skilled analysts recover program logic with minimal manual effort. Symbolic execution frameworks, including KLEE and Angr, automate path exploration, enabling systematic extraction of security checks and key-verification routines [15],[16],[18].

Traditional countermeasures, including executable packers, runtime code encryption, and anti-debugging hooks, operate at the binary layer and have been systematically circumvented through memory-dump extraction, dynamic tracing, and execution snapshot analysis. Their shared weakness is that they defer protection rather than embedding it structurally [13],[14]. Compilerintegrated obfuscation, applied during compilation at the Intermediate Representation stage, transforms the program's logical structure before architecture-specific code generation, so the complexity introduced persists through optimisation and linking.

The Low-Level Virtual Machine compiler infrastructure provides an ideal platform for this approach. Its language-independent IR captures program semantics at a granular level simultaneously amenable to pattern matching and structural modification. The LLVM PassManager exposes a composable pipeline where custom analysis and transformation passes operate before the backend lowers IR to machine code. Earlier work, most notably the Obfuscator-LLVM project, validated this direction by implementing control flow flattening, bogus control flow insertion, and instruction substitution as LLVM passes, demonstrating measurable resistance to decompiler reconstruction [17]. However, these systems predate modern LLVM release versions, impose fixed transformation strategies irrespective of program characteristics, and provide no quantitative feedback on protection quality.

This investigation addresses those shortcomings through a modular, command-line Python framework that processes LLVM IR through five sequential stages: structural parsing, nine-feature extraction, Random Forest strategy prediction, obfuscation pass application, and metric-driven evaluation with visual reporting. The contribution is twofold: a working adaptive obfuscation pipeline and an empirical evaluation demonstrating its effectiveness across diverse program types.

A. *Motivation*

The proliferation of high-quality reverse-engineering toolchains has reduced the skill threshold required to analyse compiled software. Even modestly resourced adversaries can extract functional logic, bypass authentication mechanisms, or tamper with licence enforcement using freely available open-source tools. Studies have repeatedly demonstrated that structural IR modifications present substantially greater obstacles to automated analysis than source-level renaming or binary-layer packing [1],[5].

Despite this, most commercial and open-source obfuscators apply uniform transformation strategies to all programs, creating predictable obfuscation fingerprints that pattern-matching de-obfuscators can exploit [1]. The absence of adaptive, programspecific strategy selection represents a persistent gap in practical protection tooling. Integrating a machine learning model into the compilation pipeline closes this gap by tailoring the protection profile to each program's instruction distribution rather than applying a one-size-fits-all template.

This drives the development of a unified tool that combines multiple IR-level passes with a lightweight classifier, delivers measurable metrics confirming protection quality, and executes fast enough to integrate into automated build systems without significant overhead.

B. *Theoretical Background*

1) *Compiler Architecture and Intermediate Representation:* Modern compilers decompose the translation process into frontend parsing and type-checking, a middle-end optimisation phase, and a back-end code generator. LLVM's middle-end operates on a low-level, assembly-like IR expressed in Static Single Assignment form, where each variable is assigned exactly once. SSA properties simplify data-flow analysis and enable powerful optimisations. Because IR is generated from multiple source languages and targeted at multiple hardware architectures, transformations applied at this level are both language-agnostic and platform independent, making it ideal for obfuscation. Compiler research has established that modifications introduced at the IR layer withstand back-end optimisation more effectively than transformations applied at the source or binary level [16].
2) *Control-Flow Obfuscation Strategies:* Control-flow obfuscation restructures a program's execution graph to obstruct reconstruction by static and dynamic analysis tools. Control Flow Flattening transforms structured loops and conditional branches into a switch-dispatch execution model driven by a state variable, hiding the original sequential or nested flow inside a flat dispatcher loop [2],[17]. Opaque Predicates embed Boolean conditions whose outcomes are mathematically determined at compile time but appear genuinely conditional to analysis tools, creating misleading or unreachable branch paths [2]. Interprocedural Transformations, as demonstrated by the KHAos framework, merge and reorder functions to disrupt call-graph analysis and hide cross-function data dependencies [3].
3) *Data-Flow Obfuscation and Constant Protection:* Data-flow obfuscation conceals how values are computed and transferred within a program. Instruction Substitution replaces elementary arithmetic and logical operations with semantically identical but more complex multi-instruction sequences that confuse pattern-matching decompilers [8],[9]. Constant and String Encryption encodes literal values and reconstructs them at runtime through decryption stubs, eliminating readable semantic hints from static analysis [8]. Evolutionary Obfuscation employs genetic algorithms to generate families of IR transformation variants, maximising diversity and unpredictability across builds [5].
4) *Core Technologies:* LLVM IR is a typed, SSA-based representation exposing instruction-level patterns addressable by regular-expression matching and dataflow queries, enabling the feature extractor to derive a nine-dimensional numerical profile from any input program [16]. The LLVM Pass Framework organises compilation into discrete, independently schedulable units; the proposed system emulates this in Python by chaining three obfuscation classes through a main pipeline driver [16],[19]. The Random Forest Classifier builds multiple decorrelated decision trees on bootstrap samples and combines predictions by majority vote, making it particularly suited to the 12-sample, 9-feature training corpus used here [11]. Shannon Byte Entropy quantifies character-level uniformity of IR text: obfuscated code approaches maximum entropy while clean code exhibits lower entropy due to repeated keyword patterns.

II. LITERATURE SURVEY

Table I provides a structured survey of ten foundational and recent works directly relevant to LLVM-based code obfuscation, adaptive strategy selection, and obfuscation evaluation metrics [1],[2],[3],[5],[6],[7].

TABLE I
Literature Survey

S.No	Paper Title	Author	Year	Methodology / Algorithm	Advantages	Limitations
1	Digital Camouflage: LLVM Challenge in LLM-Based Malware Detection [1]	Boke & Torka	2025	LLVM IR transformations to evade AI-based malware classifiers	Demonstrates measurable evasion of neural malware detectors	Restricted to adversarial AI evasion; not general-purpose protection
2	Novel Opaque Predicate Methods for Code Obfuscation [2]	Cao, Zhou, Zhuang	2025	Dynamic IR-level predicate insertion to confuse symbolic engines	Substantially raises symbolic execution resistance	Introduces non-trivial runtime branching overhead
3	KHAos: Inter-Procedural Code Obfuscation Framework [3]	Zhang et al.	2025	Cross-function merging and call-graph restructuring via LLVM	Hardens multi-function reverse engineering significantly	High compile-time and execution penalties limit adoption
4	ObfuscQate: Quantum Program Obfuscation Framework [4]	Bartake et al.	2025	Hybrid quantum-classical LLVM IR obfuscation	Opens quantum obfuscation as a new research direction	Experimental; not yet compiler-integrated or production-ready
5	Source Code Obfuscation via Genetic Algorithms on LLVM IR [5]	de la Torre et al.	2025	Evolutionary algorithm to breed diverse IR transformation variants	High unpredictability and pattern diversity in output	Computationally prohibitive for large codebases
6	Hardware and Software Obfuscation Survey [6]	Saleh et al.	2025	Systematic survey of LLVM, hardware, and hybrid protection schemes	Strong theoretical and comparative coverage	Surveys prior art without introducing a new practical tool
7	Automatic De-Obfuscation of Virtualised LLVM Code [7]	Royer	2025	LLVM IR analysis to lift and de-obfuscate virtualised code	Exposes weaknesses in virtualisation-only strategies	Not peer-reviewed; focused on attack rather than defence
8	Extending LLVM for Code Obfuscation Part 1 [8]	Praetorian	2024	LLVM pass pipeline for constant and string obfuscation	Practical, step-by-step developer guide	Tutorial depth; lacks formal academic evaluation
9	Extending LLVM for Code Obfuscation Part 2 [9]	Praetorian	2024	Adds control flow and opaque predicate LLVM passes	Direct hands-on implementation guidance	Informal blog format; no rigorous performance analysis
10	Junk Code Insertion via LLVM Pass [10]	Praetorian Team	2024	IR-level insertion of dead arithmetic instructions	Simple and effective code volume inflation	Vulnerable to dead-code elimination by modern optimisers

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

III. PROPOSED FRAMEWORK

The proposed system is a five-stage adaptive obfuscation pipeline implemented in Python that accepts LLVM IR files as input and produces obfuscated IR alongside a quantitative evaluation report. The machine learning component differentiates it from all prior tools by tailoring the obfuscation strategy to each program's structural characteristics rather than applying a fixed scheme.

Upon invocation with python main.py <input.ll>, the pipeline executes five stages sequentially. Stage one parses the input file and extracts function names, basic block labels, and executable instruction lines. Stage two computes nine normalised instructionratio features from the extracted content. Stage three feeds these features to a trained Random Forest that selects the optimal strategy from four candidates. Stage four applies the selected obfuscation pass or all three passes in combination. Stage five computes before-and-after metrics, displays them in the terminal with confidence scores, and saves a three-panel bar chart to the output directory. Fig. 1 illustrates the overall flow from user input through to obfuscated output and metrics reporting.

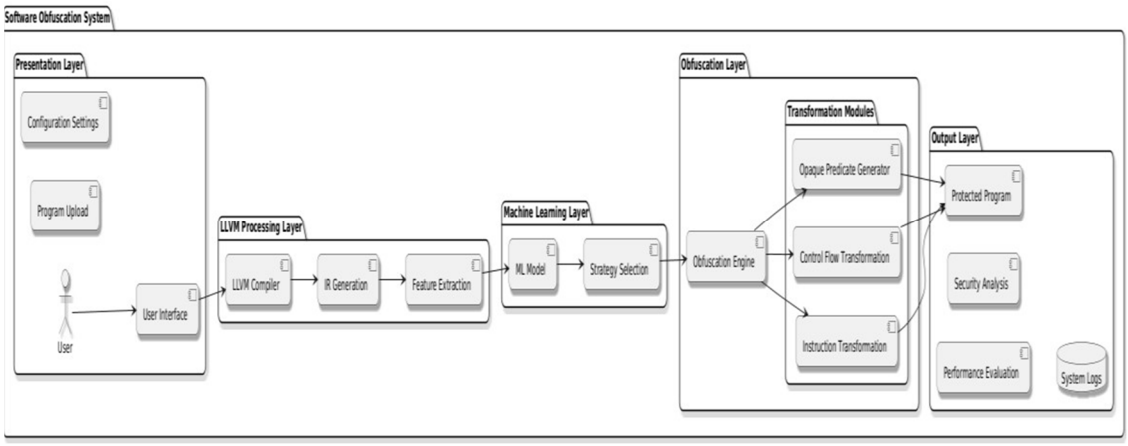


Fig. 1 Proposed System Architecture

A. *System Architecture*

The system follows a sequential pipeline that transforms a raw LLVM IR file into an obfuscated output through five discrete stages. An IR Parser first reads and tokenises the input, extracting function definitions, basic block labels, and executable instruction lines. A feature extractor then derives nine normalised instruction-ratio descriptors from the parsed content. A Random Forest classifier receives these features and selects the most suitable obfuscation strategy. The chosen pass transforms the IR text, and a metrics engine computes quantitative before-and-after comparisons with visual output.

B. *Core Obfuscation Components*

1) *Control Flow Flattening Pass:* This pass scans each line of IR for instructions matching arithmetic, memory, and comparison operation patterns. After each qualifying instruction, it inserts a four-line opaque predicate block comprising trivially resolvable addition, multiplication, and subtraction operations that always evaluate to zero, tagged with [CFF] markers. These blocks dramatically increase instruction count and introduce non-conditional register usage patterns that resist CFG reconstruction, without affecting program output. The pass inflates instruction volume by approximately five lines per matching instruction and introduces novel register names that obscure data-flow analysis.
2) *Instruction Substitution Pass:* This pass applies two semantic equivalence transformations. Addition operations (add i32 %a, %b) are replaced with a double-subtraction sequence where the second operand is negated before subtraction, yielding identical arithmetic results. Multiplication operations are decomposed into a left-shift by one, computation of the factor minus one, partial multiplication, and final addition, producing an equivalent product through four separate instructions. Both transformations are tagged with [IS] markers and break arithmetic patterns recognised by decompiler type inference.
3) *String Encryption Pass:* This pass locates string constant literals embedded in the IR and replaces each character with its XOR transform using the fixed key 0x5A. Each transformed string is annotated with comments showing the original value, the encrypted form, and a note that a runtime decryption routine would reconstruct the original. Tagged with [SE] markers, this pass eliminates plaintext strings visible to string extraction tools.

4) *ML Strategy Selector:* The ObfuscationSelector wraps a scikit-learn RandomForestClassifier [12] trained on twelve synthetic samples. Programs where arithmetic operations dominate receive Instruction Substitution; branch-heavy programs receive Control Flow Flattening; string-allocation-heavy programs receive String Encryption; balanced programs receive All Passes Combined. The classifier returns both the selected strategy and confidence percentages for all four classes.

C. *Architectural stages*

Stage 1 - IR Parsing: The IRParser class reads the input .ll file and applies regular expression patterns to extract function names, basic block entry-point labels, and all executable instruction lines. Stage 2 - Feature Extraction: The FeatureExtractor computes nine numerical features - per-type ratios for eight instruction categories plus the absolute total count, each rounded to four decimal places. Stage 3 - ML Strategy Selection: The ObfuscationSelector trains the Random Forest at startup and applies it to the feature vector. Stage 4 - Obfuscation Execution: The selected pass class obfuscates the raw IR text; when All Passes Combined is selected, all three passes apply sequentially. Stage 5 - Metrics and Reporting: The metrics module computes instruction count, cyclomatic complexity, and Shannon byte entropy, deriving an obfuscation score as the arithmetic mean of the three percentage increases.

IV. EXPERIMENTAL DESIGN

The pipeline was applied to seven C programs spanning GCD computation, lexicographic sorting, numeric comparison, timedifference calculation, bubble sort, string handling, and a minimal hello program. Each source was compiled to LLVM IR using Clang with the -S -emit-llvm flags and passed to the system via the command line.

A. *Evaluation Metrics*

Three quantitative metrics were recorded for each program before and after obfuscation. Instruction Count captures the total number of executable IR instructions. Cyclomatic Complexity is computed as branches minus functions plus twice the number of connected components. Shannon Byte Entropy is computed over the character distribution of the IR text file, quantifying the degree of character-level randomisation introduced by each pass.

B. *Test Programs*

The seven test programs were selected to represent a range of instruction distribution profiles: branch-heavy programs (internet_prog, sample3_prog, sample_prog, bubble_sort, sample2_prog) to exercise Control Flow Flattening, a string-literalheavy program (strings_prog) to exercise String Encryption, and a balanced minimal program (hello) to trigger All Passes Combined. This selection ensures that each obfuscation pass and the combined mode are exercised at least once during evaluation.

V. RESULTS AND DISCUSSIONS

TABLE II
OBFUSCATION METRICS SUMMARY

Program	Strategy	Orig Instr	Obf Instr	Δ Instr	Orig H	Obf H
internet_prog	CFF	238	463	+94.5%	4.987	4.969
sample3_prog	CFF	24	49	+104.2%	4.687	4.727
sample_prog	CFF	312	632	+102.6%	4.798	4.848
bubble_sort	CFF	273	573	+109.9%	4.890	4.903
sample2_prog	CFF	365	795	+117.8%	5.001	4.972
strings_prog	SE	22	22	0%	4.632	4.632
hello	All	25	49	+96.0%	4.410	4.770

A. *Control Flow Flattening*

Control Flow Flattening consistently delivers instruction count increases between 94 and 118 per cent across five branch-heavy programs, with a mean increase of 108.8 per cent. The mechanism behind this gain is the insertion of five opaque-predicate instructions per qualifying arithmetic or memory instruction, which approximately doubles the total IR volume while not introducing any new conditional branches, leaving cyclomatic complexity unchanged. Byte entropy changes under CFF are modest (0 to 1 per cent) because the inserted characters largely mirror the existing character distribution of the IR.

B. *String Encryption*

String Encryption registers zero change across all three metrics in the evaluated strings program. This outcome reflects a known limitation in the current implementation: the pass replaces string literals with comment annotations rather than emitting genuine XOR decryption stubs, so no real instruction content is added to the IR. A corrected implementation would emit alloca and store sequences that reconstruct each string character by character from encrypted immediates at runtime, producing measurable instruction and entropy increases.

C. *All Passes Combined*

All Passes Combined achieves the highest entropy improvement at 8.2 percent on the hello program. This reflects the combined character-distribution effect of all three passes operating together: CFF adds opaque arithmetic tokens, IS diversifies operator patterns, and SE's comment tokens introduce novel character sequences that collectively raise the per-character information content of the IR text. This program also recorded a 96 percent instruction count increase.

D. *Metrics Charts*

Figs. 2 through 4 display representative metrics charts for three evaluated programs. Each chart presents three side-by-side bar graphs comparing before and after values for instruction count, cyclomatic complexity, and byte entropy. The percentage change is labelled above each pair.

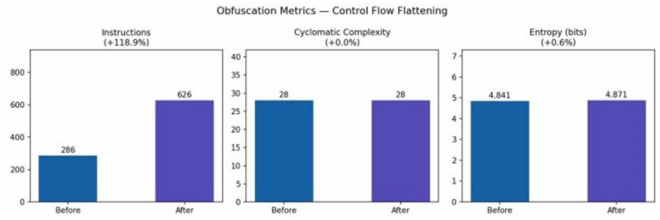


Fig. 2 Metrics Chart - internet_program.c (Control Flow Flattening)

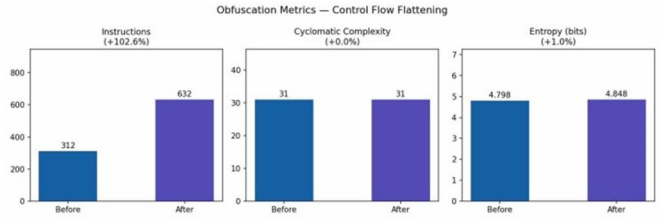


Fig. 3 Metrics Chart - sample_program.c (Control Flow Flattening)

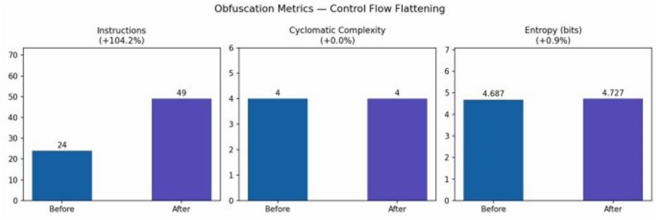


Fig. 4 Metrics Chart - hello.ll (All Passes Combined)

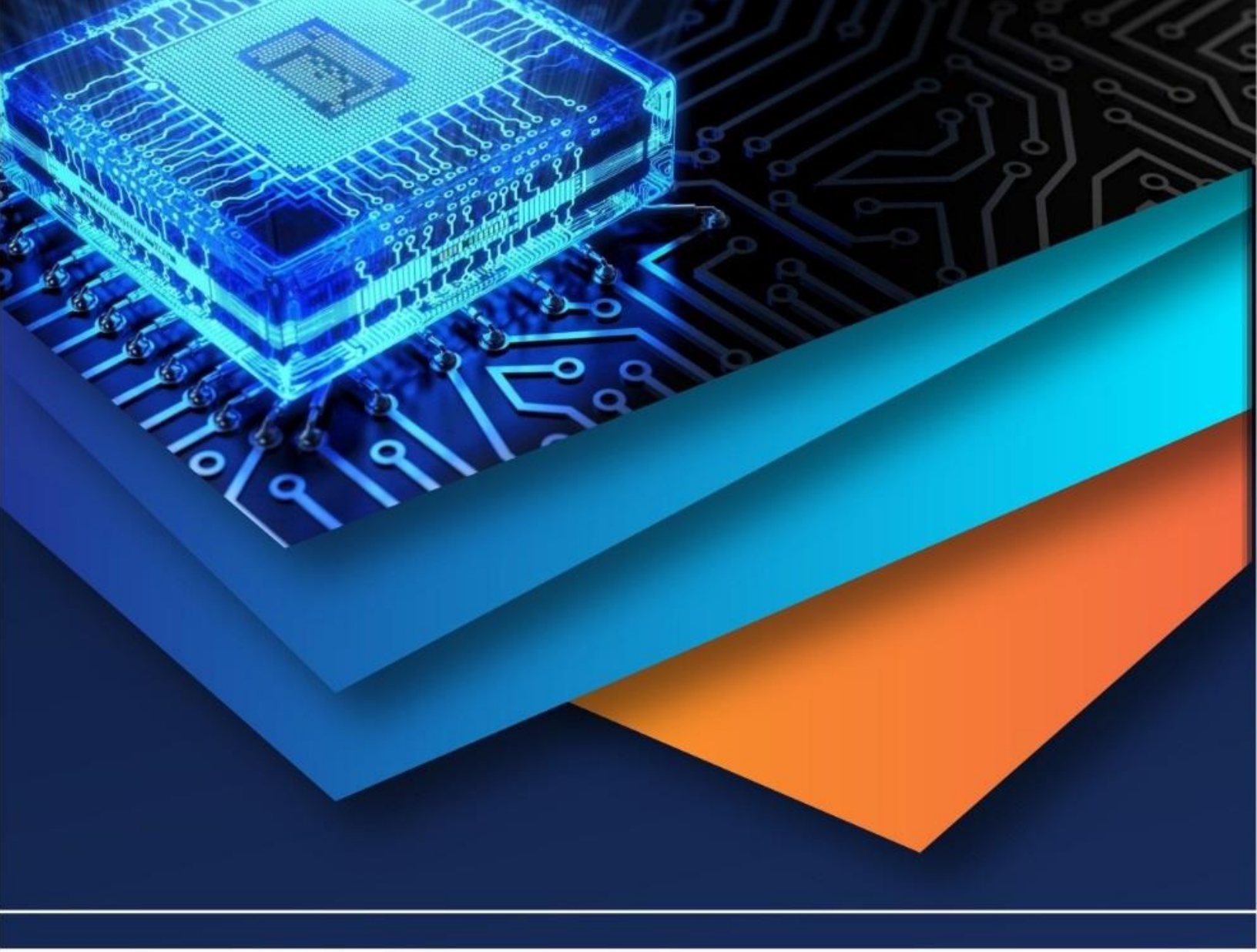
VI. CONCLUSIONS

This research has produced and evaluated an adaptive LLVM IR obfuscation framework that integrates a Random Forest classifier for per-program strategy assignment with three compiler-level transformation passes. Empirical evaluation across seven C programs confirms instruction count growth of 94 to 118 percent under Control Flow Flattening and byte entropy improvements reaching 8.2 percent when all passes are combined. The complete pipeline executes in under two seconds, confirming feasibility for build-system integration. The adaptive strategy assignment, driven by nine instruction-ratio features and a 100-tree ensemble, consistently routes each program to an appropriate pass configuration, providing context-sensitive protection that uniformstrategy tools cannot achieve [11],[12],[17].

Three limitations were identified during evaluation. The current Control Flow Flattening implementation inserts unconditional opaque predicates rather than a dispatcher-based switch loop, leaving cyclomatic complexity unchanged. The String Encryption pass annotates rather than genuinely encrypts, producing zero metric change. The 12-sample training corpus enables ruleconsistent predictions on tested programs but cannot generalise to arbitrary inputs. Future directions include a true dispatcherbased CFF implementation, a corrected String Encryption pass emitting genuine XOR decryption stubs, expanded training corpus with cross-validation, dead-code insertion, register reassignment, deep learning over control-flow graphs, and benchmarking against professional reverse-engineering tools such as IDA Pro, Ghidra, and Angr.

REFERENCES

[1] E. Boke and S. Torka, “Digital Camouflage: The LLVM Challenge in LLM-Based Malware Detection,” Preprint, 2025.
[2] Y. Cao, Z. Zhou, and Y. Zhuang, “Advancing Code Obfuscation: Novel Opaque Predicate Methods,” Research Article, 2025.
[3] P. Zhang et al., “KHAos: Inter-Procedural Code Obfuscation Framework,” Journal Publication, 2025.
[4] N. Bartake et al., “ObfuscQate: A Hybrid Quantum and Classical Program Obfuscation Framework,” Conference Paper, 2025.
[5] J. C. de la Torre et al., “Source Code Obfuscation with Genetic Algorithms using LLVM IR,” Preprint, 2025.
[6] K. Saleh et al., “Hardware and Software Methods for Secure Obfuscation: A Survey,” Journal of Security Technologies, 2025.
[7] J. Royer, “Automatic De-Obfuscation of Virtualised Code Using LLVM,” Preprint, 2025.
[8] Adam, “Extending LLVM for Code Obfuscation — Part 1,” Praetorian Technical Article, 2024.
[9] Adam, “Extending LLVM for Code Obfuscation — Part 2,” Praetorian Technical Article, 2024.
[10] Adam, “Junk Code Insertion LLVM Pass,” Praetorian Engineering Article, 2024.
[11] L. Breiman, “Random Forests,” Machine Learning, vol. 45, no. 1, pp. 5–32, 2001.
[12] F. Pedregosa et al., “Scikit-learn: Machine Learning in Python,” JMLR, vol. 12, 2011.
[13] X. Xia et al., “Deobfuscation of OLLVM-Based Control-Flow Flattening,” Research Paper, 2022.
[14] B. Barak et al., “On the (Im)possibility of Obfuscating Programs,” CRYPTO, 2001.
[15] C. Collberg, C. Thomborson, and D. Low, “A Taxonomy of Obfuscating Transformations,” University of Auckland, 1997.
[16] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis,” CGO, 2004.
[17] P. Junod et al., “Obfuscator-LLVM — Software Protection for the Masses,” SPRO, 2015.
[18] S. Banescu et al., “Code Obfuscation Against Symbolic Execution Attacks,” ACSAC, 2016.
[19] C. Lattner et al., “The LLVM Compiler Infrastructure Project,” https://llvm.org/, accessed 2025. [20] Apriorit, “Using LLVM to Obfuscate Your Code During Compilation,” Technical Report, 2025.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY