



МЕТОД РОЗМІТКИ ФРАГМЕНТІВ ПРОМІЖНОГО КОДУ .NET ОРАКУЛОМ-ОБФУСКАТОРОМ ДЛЯ АДАПТИВНОГО ЗАХИСТУ ВИХІДНОГО КОДУ

Проаналізовано наявний спосіб формування навчальних даних для адаптивної обфускації платформи .NET, за якого множини придатних стратегій обфускації коду зіставляють фрагментів проміжного коду експертним судженням або евристичними правилами на зразок "фрагмент містить рядкові константи, отже шифруй його рядки". З'ясовано, що такий спосіб формування навчальних даних не масштабується і не гарантує узгодженості мітки з критерієм, за яким оцінюють остаточний результат, через що моделі навчають прогнозувати одне, а оцінюють результат за іншим критерієм. Розроблено метод розмітки фрагментів проміжного коду, у якому еталонну множину придатних стратегій обфускації фрагмента проміжного коду становлять δ -оптимальні за інтегральною метрикою якості обфускації перетворення, тобто той самий критерій, за яким оцінюють остаточний результат, а мітку подано бінарним вектором належності без згортання до однієї стратегії обфускації коду. Інстанційовано компоненту структурної складності на рівні окремого фрагмента проміжного коду через згладжену насиченість, що зберігає розрізняльну здатність метрики на коротких фрагментах коду, де пряме відносно нормування приросту складності вироджується. Реалізовано двоступеневий контур верифікації семантичної еквівалентності, який допускає до розмітки тільки коректні перетворення. Проведено перевірку методу оракул-розмітки на корпусі з дев'яти бібліотек .NET з дозвільними ліцензіями: виконано 29 544 вимірювання пар "фрагмент – стратегія" на 9848 фрагментах коду, причому контур відхилив 0,58 % пар. Встановлено, що присутність стратегій обфускації коду в еталонних множинах різко асиметрична: вставляння непрозорих предикатів входить до 96,12 % множин (довірчий інтервал 95,70–96,50 %) проти 3,80 % – для шифрування рядків і 3,41 % – для рефлексивної обфускації коду. Досліджено природу асиметрії присутності стратегій обфускації коду декомпозицією їх застосовності та заслуги і показано, що вона є структурною властивістю простору стратегій обфускації коду, а не перевагою методу: непрозорий предикат спрацьовує на 99,31 % фрагментів коду, тоді як шифрування рядків застосовне тільки до 7,47 %. Підтверджено стійкість домінування непрозорих предикатів за всіх нормалізацій і допусків, а медіана запасу якості 0,1640 у кілька разів перевищує допуск. Виявлено, що шифрування рядків потрапляє до еталонної множини у 50,67 % фрагментів коду з рядковими літералами, тож правило на основі наявності конструкції було б там не точнішим за випадкове вгадування. Виявлено також null-клас обсягом 8,2 % фрагментів коду, для якого жодна стратегія обфускації коду не дає виграшу і який має структурний механізм: серед фрагментів переміщення даних він сягає 22,3 %. Показано, що метод оракул-розмітки дає змогу одноразово заплатити обчислювально витратною процедурою вимірювання якості обфускації коду, щоб надалі отримувати її результат прогнозом навченої моделі за мілісекунди.

Ключові слова: обфускація програмного коду; цикломатична складність коду; семантична еквівалентність перетворення; непрозорі предикати; машинне навчання моделі; інтегральна метрика якості обфускації коду.

Вступ / Introduction

Обфускація програмного коду залишається основним практичним засобом захисту інтелектуальної власності у програмах для платформи .NET. Причина архітектурна: компілятор мови C# породжує не машинний код, а спільну проміжну мову (англ. *Common Intermediate Language*, далі – проміжний код), яка зберігає імена типів, методів і полів, сигнатури та структуру метаданих, тож декомпілятори відновлюють з .NET-збірки читабельний програмний код, близький до початкового.

Обфускація перетворює код так, щоб зберегти поведінку програми і, водночас, ускладнити розуміння результату декомпіляції.

Класична обфускація застосовує сталий набір перетворень до всієї збірки однаково, витрачаючи ресурс там, де захист непотрібний, і не добираючи найдієвішого перетворення там, де воно критичне. Оскільки різні ділянки коду мають різну структуру, звідси випливає ідея адаптивної обфускації коду – вибирати стратегію перетворення коду для кожної ділянки окремо за її оз-

© Copyright 2026 by the author(s)

Головко Ігор Геннадійович, аспірант, кафедра комп'ютерної інженерії та інформаційних систем.

Email: i85.golovko@gmail.com; <https://orcid.org/0009-0004-2173-5126>

Савенко Олег Станіславович, д-р техн. наук, професор, кафедра комп'ютерної інженерії та інформаційних систем.

Email: savenko_oleg_st@ukr.net; <https://orcid.org/0000-0002-4104-745X>

Цитування за ДСТУ: Головко І. Г., Савенко О. С. Метод розмітки фрагментів проміжного коду .NET оракулом-обфускатором для адаптивного захисту вихідного коду. Науковий вісник НЛТУ України. 2026, т. 36, № 4. С. 121–129.

Citation APA: Golovko, I. G., & Savenko, O. S. (2026). An oracle-obfuscator labelling method for .NET intermediate code fragments in adaptive source code protection. *Scientific Bulletin of UNFU*, 36(4), 121–129. <https://doi.org/10.36930/40360414>

наками; саме цей підхід автори розвивали у попередніх працях, запропонувавши модуль машинного навчання для вибору стратегії обфускації коду та інтегральну метрику якості обфускації коду.

Адаптивний вибір стратегії обфускації коду є завданням навчання моделі з учителем, тож потребує розмічених навчальних даних – для кожного фрагмента проміжного коду має бути відома множина придатних стратегій обфускації коду. Це і є невирішений лімітувальний чинник. Наявні роботи призначають мітки експертно або за евристичними правилами "фрагмент з рядковими константами – шифруй рядки". Такий спосіб призначення міток має дві вади: він не масштабується (ручна розмітка – сотні-тисячі фрагментів коду проти десятків тисяч, потрібних для глибоких моделей) і, головне, евристична мітка не має гарантованого зв'язку з критерієм, за яким оцінюють остаточний результат – модель навчають прогнозувати експертну думку, а оцінюють за метрикою якості обфускації коду, і цей розрив створює систематичне зміщення, яке жодним обсягом даних не усунути. Прогалина, отже, не у браку моделей чи метрик (і те, й інше в літературі є), а у відсутності відтворюваного способу отримати мітки, узгоджені з метрикою якості обфускації коду, у масштабі, придатному для навчання моделі.

Об'єкт дослідження – формування навчальних даних для адаптивної обфускації платформи .NET.

Предмет дослідження – методи і засоби автоматичної розмітки фрагментів проміжного коду за інтегральною метрикою якості обфускації коду, які дадуть змогу масштабовано формувати навчальні набори міток для адаптивної обфускації коду без участі експерта.

Мета роботи – розробити та експериментально перевірити метод розмітки фрагментів проміжного коду, який визначає еталонну множину придатних стратегій обфускації коду цим самим критерієм, за яким оцінюють остаточну якість обфускації коду, і який масштабується на реальні збірки .NET.

Для досягнення зазначеної мети визначено такі основні завдання дослідження:

- формалізувати метод оракул-розмітки, у якому еталонну множину стратегій обфускації фрагмента проміжного коду становлять δ -оптимальні за інтегральною метрикою якості перетворення, а мітка є бінарним вектором належності без згортання до однієї стратегії обфускації коду, що дасть змогу усунути систематичне зміщення між критерієм навчання моделі та критерієм оцінювання якості обфускації коду й отримувати мітки відтворювано, без участі експерта;
- інстанціювати компоненту структурної складності метрики на рівні окремого фрагмента проміжного коду, усунувши насичення, властиве прямому перенесенню збіркового означення, що збереже розрізняльну здатність метрики на коротких фрагментах коду, де пряме відносне нормування приросту складності вироджується;
- реалізувати метод оракул-розмітки із двоступеневим контуром верифікації семантичної еквівалентності, який забезпечує розмічування тільки коректних перетворень, що гарантує потрапляння до навчального набору міток тільки семантично коректних перетворень;
- перевірити метод оракул-розмітки на корпусі реальних .NET-збірок зі статистичним опрацюванням первинних даних і встановити, як розподіляється придатність стратегій обфускації коду за фрагментами проміжного коду, що дасть змогу оцінити практичну придатність цього методу та встановити структурні закономірності розпо-

ділу придатності стратегій обфускації коду за фрагментами проміжного коду.

Аналіз останніх досліджень та публікацій. У дослідженні [4] автори запропонували таксономію обфускаційних перетворень, поділивши їх на лексичні, структурні та перетворення потоку керування, і ввели чотири виміри оцінювання обфускаційних перетворень – потужність, стійкість, непомітність та вартість. Ця схема залишається понятійним каркасом галузі, а інтегральна метрика, застосована в цій роботі, є її кількісною інстанціацією.

У роботі [7] автори означили цикломатичну складність програми як кількість лінійно незалежних шляхів у графі потоку керування. Для задач обфускації програмного коду ця метрика цінна прямим зв'язком з механізмом перетворення, а саме: вставляння непрозорого предиката додає розгалуження, тобто збільшує складність шляхом її вимірювання.

У роботі [1] автори показали, що стійкість обфускації коду визначається не самим перетворенням, а його співвідношенням з конкретною атакою, тож оцінювати перетворення поза моделлю загроз некоректно. Звідси випливає висновок, істотний для цієї роботи: вибір перетворення має залежати від захищуваного програмного коду, а не бути сталим.

У роботі [11] автори встановили для платформи Android, що вставляння непрозорих предикатів дає найбільший приріст складності відновлення серед структурних перетворень.

У дослідженні [8] автори з'ясували, що рефлексивному переписуванню піддаються не всі виклики, а тільки ті, що належать самій програмі, а не бібліотекам платформи. Це прямо узгоджується з отриманими тут даними про фактичну застосовність рефлексивного перетворення.

У роботі [10] автори розглянули автоматичний вибір параметрів обфускації коду як задачу оптимізації і застосували коеволюційні алгоритми. Такий підхід шукає рішення для кожної збірки заново і не породжує придатної до перенесення моделі.

У роботі [3] автори узагальнили застосування машинного навчання моделі до аналізу коду і вказали на те, що якість моделей коду визначається насамперед якістю та обсягом розмічених даних, а лімітувальною ланкою більшості задач є саме розмітка, а не архітектура моделі.

Проведений аналіз останніх досліджень та публікацій свідчить, що обидві потрібні складові наявні в літературі порізно: означено метрики якості обфускації коду, придатні до автоматичного обчислення [1, 4], і показано визначальну роль розмітки для моделей коду [3]. Проте жоден з відомих методів не визначає мітку фрагмента проміжного коду цим самим критерієм, за яким оцінюють остаточну якість обфускації коду, – метрики обчислюють постфактум, для оцінювання результату, а не для породження навчального сигналу. Власні попередні праці [9, 12] дали інтегральну метрику обфускації коду та модульну архітектуру обфускатора з модулем машинного навчання моделі, проте питання масштабованого джерела міток залишили відкритим. Саме цю прогалину заповнює запропонований метод, що й зумовлює потребу проведення цього дослідження.

Матеріали та методи дослідження. Дослідження виконано на корпусі з дев'яти бібліотек .NET з дозвіль-

ними ліцензіями, проміжний код яких сегментовано на фрагменти за правилами базових блоків. Застосовано метод оракул-розмітки (термін "оракул" ужито в машиннонавчальному значенні – це процедура, яка постає еталонну мітку на етапі формування навчального набору й працює до складання захищеної збірки, а не під час її виконання): для кожної пари "фрагмент – стратегія" виконано перетворення, перевірено семантичну еквівалентність двоступеневим контуром й обчислено інтегральну метрику якості обфускації коду. Опрацьовано 29 544 пари на 9848 фрагментах коду. Використано методи статистичного опрацювання первинних даних: інтервальне оцінювання часток за Вілсоном, декомпозицію застосовності та заслуги, аналіз чутливості за сіткою параметрів і знаковий тест для перевірки консистентності часток за членами корпусу.

Результати дослідження та їх обговорення / Research results and their discussion

Метод оракул-розмітки фрагментів проміжного коду. Об'єктом розмітки є фрагмент проміжного коду x – зв'язний підграф графа потоку керування методу збірки .NET, виділений сегментацією за правилами базових блоків. Кожному фрагменту проміжного коду метод оракул-розмітки ставить у відповідність не одну "правильну" стратегію обфускації коду, а множину придатних стратегій обфускації коду, оскільки і базові моделі, і оператор вибору належать до класу мультиміткової класифікації (multi-label).

Простір стратегій обфускації коду утворюють чотири елементи: $\Omega = \{\text{тотожне перетворення (no-op), вставлення непрозорих предикатів (opaque), шифрування рядкових літералів (strenc), рефлексивна обфускація викликів (meta)}\}$. Тотожне перетворення є базовою лінією: його якість дорівнює нулю за побудовою, і воно потрапляє до еталонної множини тільки тоді, коли жодна дієва стратегія обфускації коду не дає виграшу. Дисертаційний простір нараховує п'ять стратегій обфускації коду; дві з них (перейменування ідентифікаторів та структурне ускладнення) визначені на рівні цілої збірки, а не фрагмента проміжного коду, тому цей експеримент їх коректно унеможливило (обґрунтування – у розділі "Обговорення").

Якість застосування стратегії обфускації коду ω до фрагмента проміжного коду x методом оракул-розмітки оцінює інтегральною метрикою $Q(\cdot)$, успадкованою з попередньої роботи автора [9]:

$$Q(x, \omega) = \alpha R(x, \omega) + \beta S(x, \omega) + \gamma P(x, \omega) - \lambda O(x, \omega),$$

де: $R(\cdot)$ – стійкість до відновлення (частка інформації, яку декомпілятор не відновлює з обфускованого фрагмента проміжного коду); $S(\cdot)$ – структурна несхожість обфускованого фрагмента проміжного коду від вихідного; $P(\cdot)$ – приріст структурної складності; $O(\cdot)$ – накладні витрати. Ваги фіксовано: $\alpha = 0,35$, $\beta = 0,25$, $\gamma = 0,25$, $\lambda = 0,15$ (перші три доданки винагороджують захисний ефект, четвертий штрафувє його ціну). Саме так метод вирішує суперечність між рівнем захисту та його ціною: стратегія обфускації коду потрапляє до еталонної множини тільки тоді, коли її захисний ефект перевищує власну ціну щонайменше на допуск, тож ресурс витрачається адресно – на ті фрагменти проміжного коду, де захист окупається.

Компоненту складності $P(\cdot)$ метод оракул-розмітки нормує згладженою насиченістю:

$$P(x, \omega) = \frac{\Delta V(x, \omega)}{\Delta V(x, \omega) + \kappa_P}, \Delta V(x, \omega) = \max(V(G_x') - V(G_x), 0),$$

де: $V(G_x)$ – цикломатична складність підграфа фрагмента проміжного коду; x' – обфускований варіант; κ_P – параметр напівнасичення. Це вирішує проблему масштабу: цикломатична складність базового блоку мала ($V(G_x) \in \{1, 2\}$), тому пряме відносне нормування цієї складності насичувало б приріст структурної складності $P(\cdot)$ до одиниці від єдиного доданого предиката, і компонента перестала б розрізняти фрагменти. Згладжена насиченість обмежена, монотонна, зберігає фрагментну точність і не насичується стрибком (одиничний предикат дає $P = 1/(1 + \kappa_P)$). Значення $\kappa_P = 3$ прийнято як середину діапазону нечутливості, підтвердженого сканом.

Еталонну множину метод оракул-розмітки визначає як множину δ_o -оптимальних стратегій обфускації коду – тобто тих, чия якість відстає від найкращої не більш ніж на допуск $\delta_o > 0$:

$$Q^*(x) = \max_{\omega \in \Omega} Q(x, \omega), \Omega^*(x) = \{ \omega \in \Omega : Q(x, \omega) \geq Q^*(x) - \delta_o \}.$$

Стратегія обфускації коду належить еталону, якщо вона нерозрізнявана від найкращої в межах похибки вимірювання. Навчальною ціллю слугує бінарний вектор належності $y(x) = 1_{\Omega^*(x)}$ (multi-hot); згортання множини до однієї мітки не виконується, чим усувається зміщення розмітки до менш інвазивних стратегій обфускації коду. У цьому експерименті $\delta_o = 0,05$ (провізорно; чутливість перевірено окремо).

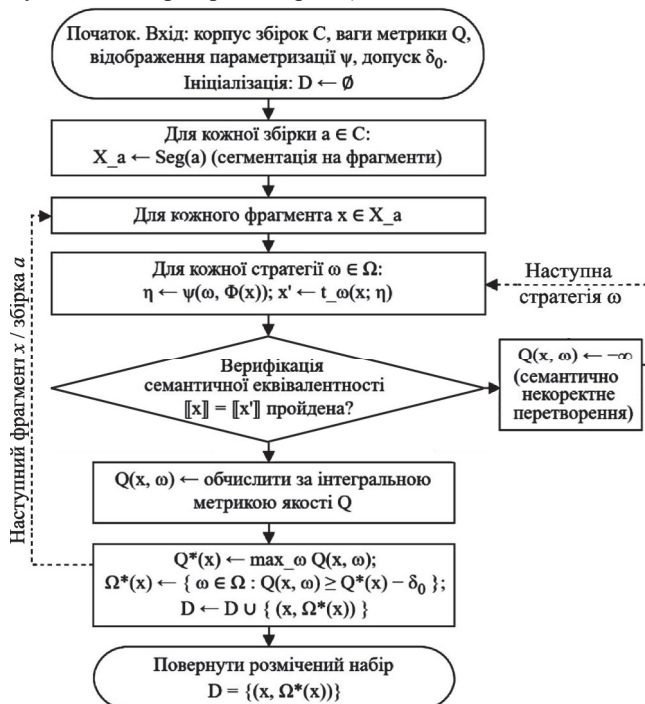


Рисунок. Блок-схема методу оракул-розмітки навчального набору міток / Flowchart of the oracle labelling method for a training set

Наукова новизна методу оракул-розмітки полягає в тому, що мітку визначає той самий критерій, за яким оцінюють остаточний результат; розрив між критерієм розмітки та критерієм оцінювання якості обфускації коду створював би систематичне зміщення. Означення не утворює замкненого кола: Q обчислюється з результату фактичного застосування стратегії обфускації коду (декомпіляція, порівняння структури, вимірювання складності та витрат), а не з міток чи прогнозів; мітки є нас-

лідком вимірювань, а моделі навчаються апроксимувати обчислювально витратну процедуру вимірювання малозатратним прогнозом.

Процедуру подано блок-схемою на рисунку: для кожного фрагмента проміжного коду кожної збірки оракул послідовно застосовує всі стратегії обфускації коду, перевіряє семантичну еквівалентність, обчислює Q для коректних перетворень і формує еталонну множину як множину δ_o -оптимальних стратегій обфускації коду.

На рисунку введено такі умовні позначення: C – корпус збірок; a – збірка; $Seg(a)$ – сегментація на фрагменти; X_a – множина фрагментів збірки a ; x – фрагмент; Ω – простір стратегій обфускації коду; ω – стратегія обфускації коду; $\Phi(x)$ – ознаки фрагмента проміжного коду; ψ – відображення параметризації; η – параметри перетворення; t_ω – перетворення стратегії обфускації коду ω ; x' – обфускований фрагмент коду; $[[\cdot]]$ – семантика

(результат виконання); Q – інтегральна метрика якості; $Q^*(x)$ – максимум Q за стратегіями обфускації фрагмента коду; δ_o – допуск; $\Omega^*(x)$ – еталонна множина; D – розмічений набір міток.

Ітерації за фрагментами проміжного коду незалежні, тому процедура оракул-розмітки навчального набору міток паралелізується (10 робочих потоків), а витрати одноразові: повторна розмітка потрібна тільки в разі зміни корпусу, ваг метрики або просторів параметрів.

Корпус збірок .NET. Бібліотеки відбирали за чотирма критеріями: дозвільна ліцензія; наявність автоматичних тестів (xUnit або NUnit) як основи верифікації семантичної еквівалентності; запуск тестів під наявним інструментарієм .NET; доменна різноманітність у скупності. Корпус нараховує дев'ять членів шести доменів; для кожного закріплено точну версію (хеш коміту), платформу та ліцензію, що робить його відтворюваним.

Табл. 1. Склад корпусу .NET-збірок: домен, ліцензія, цільова платформа та кількість опрацьованих фрагментів проміжного коду; для трьох найбільших членів наведено обсяг підвибірки та повний обсяг у дужках ($n = 9848$ фрагментів коду, 9 членів) / Composition of the .NET assembly corpus: domain, licence, target platform and the number of processed intermediate-code fragments; for the three largest members the subsample size is given with the full size in parentheses ($n = 9848$ fragments, 9 members)

Член корпусу	Домен	Ліцензія	Платформа	Фрагментів коду	Частка, %
Markdig	Розмітка Markdown	BSD-2-Clause	net10.0	2100 (з 8872)	21,3
YamlDotNet	Оброблення YAML	MIT	net10.0	2100 (з 5466)	21,3
Scriban	Шаблонізація	BSD-2-Clause	net10.0	2100 (з 11 100)	21,3
Fractions	Арифметика раціональних чисел	MIT	net9.0	1794	18,2
SimpleBase	Кодеки base16/32/58/64	Apache-2.0	net10.0	1325	13,5
Hashids .NET	кодування ідентифікаторів	MIT	net7.0	233	2,4
Slugify.Core	Оброблення тексту	MIT	net8	95	1,0
SampleLib	Внутрішня тестова бібліотека	внутрішня	net10.0	54	0,5
nanoid-net	Генерування ідентифікаторів	MIT	net8	47	0,5
Разом				9848	100

Markdig, YamlDotNet та Scriban дають разом понад 25 тис. фрагментів коду, тож повний перебір коштував би тижнів машинного часу. Тому для кожного опрацьовано рівномірну підвибірку 2100 фрагментів коду із фіксованим зерном (1337): фрагменти впорядковують за адресою, відбирають із фіксованим зерном і знову впорядковують, тож повторний запуск відбирає ті самі адреси. Підвибірка рівномірна, отже – частки на ній є незміщеними оцінками часток повного члена (ціна – ширші довірчі інтервали, наведені разом з оцінками). Решту шести членів опрацьовано повністю. Такий склад забезпечує збалансованість: жоден член не перевищує 40 % корпусу (найбільший, Markdig, – 21,3 %), а чотири найбільші члени представляють чотири різні домени; це важливо, бо об'єднані частки зважені розміром члена. Сегментація дала 9848 фрагментів коду, з яких 1252 (12,7 %) належать до згенерованих компілятором методів (ітератори, асинхронні автомати, лямбда-замикання), які свідомо не вилучали, бо вони присутні у реальній збірці й підлягають захисту нарівні з рукописними.

Протокол вимірювання. Для кожної пари "фрагмент $\times$ стратегія" процедура застосовує перетворення до копії збірки, перевіряє семантичну еквівалентність, вимірює компоненти Q і кешує результат; загальний обсяг – 9848 фрагментів коду $\times$ 3 дієві стратегії обфускації коду = 29 544 пари. Верифікація семантичної еквівалентності – двоступенева. Перший ступінь (статичний): середовище CLR завантажує обфусковану збірку і перевіряє коректність метаданих і типів. Другий (динамічний): виконується повний набір автоматичних тестів бібліотеки проти обфускованої збірки. Стратегія обфускації коду проходить контур тільки за успіху обох сту-

пенів; інакше $Q(x, \omega) = -\infty$ із фіксацією причини. Так гарантовано розмічування тільки семантично коректних перетворень. Збої ізолюються: аварія однієї пари не перериває перебір. Контур відхилив 170 пар з 29 544 (0,58 %): 169 – на динамічному ступені, одна – аварія обфускатора; отже, перетворення стійкі до реального коду різних доменів.

Апроксимації пілотного рівня (свідомі, повідомлені прямо): $R(\cdot)$ обчислено як частку невідновлюваних рядкових літералів замість усереднення за трьома декомпіляторами (ILSpy, dotPeek, dnSpy); рефлексивну складову $R(\cdot)$ – як діагностичну, поза підсумком; $S(\cdot)$ – як косинусну відстань на сирих гістограмах кодів операцій без зважування за оберненою частотою; $O(\cdot)$ – тільки за приростом розміру коду. Спрощення діють однаково на всі стратегії обфускації коду, тож переваги жодній не створюють. Відтворюваність: процедура детермінована (в т.ч. з відбором підвибірки за фіксованим зерном); повторний прогін трьох найменших повністю опрацьованих членів (SampleLib, Hashids .NET, nanoid-net; 334 фрагменти) відтворив усі записи побітово, з розбіжністю тільки в телеметрії.

План статистичного опрацювання первинних даних. Первинними даними є 9848 записів "фрагмент $\rightarrow$ компоненти Q за стратегіями обфускації фрагмента коду $\rightarrow$ еталонна множина". Опрацювання: (1) null-клас – частка фрагментів коду з $Q^*(x) \leq \delta_o$ (жодна дієва стратегія обфускації коду не переважає тотожного перетворення) з довірчим інтервалом, подано окремо; решту названо дієвими; (2) частки присутності стратегій обфускації коду в $\Omega^*(x)$ на дієвих фрагментах коду з 95 % довірчим інтервалом Вілсона (обраним замість нор-

мального наближення, бо він коректний на частках, близьких до 0 і 1); (3) декомпозиція "застосовність × заслуга" – частка застосовності (двома оцінками: наявність конструкції за ознаками та фактичне спрацювання $Q(x, \omega) \neq 0$) і частка входження до Ω^* серед застосовних; розбіжність оцінок сама є результатом; (4) запас якості – медіана, міжквартильний розмах і частка $\leq \delta_0$ для різниці $Q(\text{opaque}) - \max\{Q(\text{strenc}), Q(\text{meta})\}$; (5) аналіз чутливості за сіткою $\delta_0 \in \{0,02; 0,05; 0,10; 0,15; 0,20\}$, $\kappa_P \in [2,5]$ та трьома нормалізаціями P (приналежність перераховується з Q на кожному кроці, а не переноситься); (6) консистентність за членами з довірчими інтервалами і знаковим тестом (описово, зважаючи на малу кількість членів).

Загальний обсяг та null-клас. Оракул опрацював 9848 фрагментів проміжного коду дев'яти бібліотек .NET, виконавши 29 544 вимірювання пар "фрагмент × стратегія". Контур верифікації семантичної еквівалентності відхилив 170 пар (0,58 %): 169 не пройшли динамічного ступеня (розбіжність результатів автоматичних тестів) і тільки одна спричинила аварію обфускатора.

Для 807 фрагментів коду (8,2 %; 95 % довірчий інтервал Вілсона 7,7–8,8 %) жодна дієва стратегія обфускації коду не перевершила тотожного перетворення ($Q^*(x) \leq \delta_0$). Цей null-клас подано окремо, бо він відповідає на самостійне питання: для якої частини коду обфускація не дає вимірюваного виграшу. Решта 9041 фрагмента проміжного коду (91,8 %) утворюють дієву підмножину, на якій побудовано всі подальші частки. Механізм утворення null-класу перевірено прямо: розмір його не пояснює (медіана інструкцій 5 проти 4 у дієвих), накладні витрати – теж (жоден фрагмент не має від'ємної якості); натомість 695 фрагментів коду (86,1 %) мають $Q(\text{opaque})$ точно нуль, а 112 (13,9 %) не пройшли верифікації. Нуль означає, що обфускатор не знайшов місця для непрозорого предиката: приріст цикломатичної складності нульовий, отже $P = 0$.

Справжнім розрізнявачем є шаблонний тип фрагмента проміжного коду: частка null-класу серед фрагментів коду типу "дані" сягає 22,3 % (221 з 989), тоді як для розгалужених типів вона становить 2,8 % (ітеративний) – 3,6 % (умовний), для сервісного – 10,1 %, для викликів API – 8,0 %, а для обчислювального – 0,0 %. Фрагменти "дані" є лінійними блоками переміщення даних без розгалужень, тож непрозорий предикат не має до чого прикріпитися. Отже, null-клас відображає не витратність обфускації коду, а відсутність структури, придатної для трансформації, і невіддатливість фрагмента проміжного коду прогнозована з його структури.

Частки присутності стратегій обфускації коду в еталонних множинах. Тотожне перетворення не потрапило до жодної еталонної множини дієвих фрагментів коду (табл. 2), що підтверджує коректність розділення класів. Вставлення непрозорих предикатів присутнє у 96,12 % множин проти 3,80 % – для шифрування рядків і 3,41 % – для рефлексивної обфускації коду; середня потужність множини – 1,033 стратегії обфускації коду на фрагмент. Ці частки виглядають як беззастережне домінування однієї стратегії обфускації коду, проте наступний підрозділ показує передчасність такої інтерпретації. Нагадаємо, у дослідженнях з кібербезпеки та зворотного інжинірингу частки присутності стратегій обфускації коду в еталонних множинах (наприклад, у наборах даних шкідливого ПЗ або бенчмарках на зразок

O-LLVM чи Tigress) суттєво відрізняються залежно від мети та вартості реалізації захисту.

Табл. 2. Частка дієвих фрагментів коду, для яких стратегія обфускації коду належить до еталонної множини $\Omega^*(x)$, з 95 % довірчим інтервалом Вілсона ($n = 9041$ дієвий фрагмент, допуск $\delta_0 = 0,05$) / Share of effective fragments for which a strategy belongs to the reference set $\Omega^*(x)$, with a 95 % Wilson confidence interval ($n = 9041$ effective fragments, tolerance $\delta_0 = 0,05$)

Стратегія обфускації коду	Фрагментів коду у Ω^*	Частка, %	95 % ДІ, %
Тотожне перетворення (no-op)	0	0,00	0,00-0,04
Непрозорі предикати (opaque)	8690	96,12	95,70-96,50
Шифрування рядків (strenc)	344	3,80	3,43-4,22
Рефлексивна обфускація (meta)	308	3,41	3,05-3,80

Декомпозиція "застосовність × заслуга". Частка присутності змішує дві величини: чи стратегія обфускації коду застосовна до фрагмента проміжного коду і чи вона переважає там, де застосовна. Шифрування рядків не визначене на фрагменті без рядкових літералів, рефлексивна обфускація – без викликів методів програми. Застосовність оцінено двома способами, бо вони розходяться: за наявністю конструкції (ознаки фрагмента проміжного коду) та за фактичним спрацюванням перетворення ($Q(x, \omega) \neq 0$; нульова якість означає, що перетворення виконалося й не змінило фрагмент). Другий спосіб є прямим виміром і дає чесніший знаменник.

Табл. 3. Декомпозиція присутності стратегії обфускації коду в еталонній множині на застосовність та заслугу за двома оцінками застосовності: наявність конструкції за ознаками та фактичне спрацювання перетворення; з 95 % довірчим інтервалом Вілсона ($n = 9041$ дієвий фрагмент, $\delta_0 = 0,05$) / Decomposition of strategy presence in the reference set into applicability and merit under two applicability estimates: presence of the construct according to fragment features, and actual firing of the transformation; with a 95 % Wilson confidence interval ($n = 9041$ effective fragments, $\delta_0 = 0,05$)

Стратегія	Передумова	Наявність конструкції, шт. (%)	Заслуга за наявністю, %	Фактично спрацювало, шт. (%)	Заслуга за спрацюванням, %	95 % ДІ, %
Непрозорі предикати	Немає (застосовна завжди)	9041 (100,00)	96,12	8979 (99,31)	96,78	96,40 – 97,13
Шифрування рядків	Рядкові літерали	675 (7,47)	50,67	671 (7,42)	50,97	47,19 – 54,73
Рефлексивна обфускація	Виклики методів	4863 (53,79)	6,17	1329 (14,70)	22,57	20,41 – 24,90

Декомпозиція (табл. 3) перевертає інтерпретацію одразу для двох стратегій обфускації коду. Шифрування рядків застосовне тільки до 7,47 % дієвих фрагментів коду, але там, де застосовне, потрапляє до еталонної множини у 50,67 % випадків – це не слабка, а вузько застосовна сильна стратегія обфускації коду (обидві оцінки для неї збігаються: 7,47 % проти 7,42 %). Для рефлексивної обфускації коду оцінки розходяться: ознака "наявність викликів" враховує усі виклики, зокрема – виклики бібліотек платформи, які перетворення не переписує, тож за ознакою стратегія обфускації коду

виглядає широко застосовною (53,79 %) і слабкою (6,17 %); прямий вимір дає 14,70 % фактичного спрацювання із заслугою 22,57 %. Отже, вона є вузько застосовною й помірно результативною, а занижена заслуга за ознакою – артефакт зовеликого знаменника; наведення обох оцінок робить цей ефект видимим.

Домінування непрозорих предикатів має, натомість, структурну природу і зберігається за обома оцінками: предикат спрацьовує майже на кожному фрагменті (99,31 %), і універсальність, помножена на істотний запас якості обфускації коду, дає 96,12 %. Це властивість простору стратегій обфускації коду, а не наслідок налаштування метрики, і його не усунути перезважуванням компонент Q . Практична цінність методу оракул-розмітки впливає з цієї асиметрії: існує ніша 7,47 % фрагментів коду, де шифрування рядків конкурує нарівні, ще 14,70 %, де рефлексивна обфускація дієва приблизно у кожному п'ятому випадку, і 8,2 % null-класу, де обфускація недоцільна. Локалізацію цих ніш і дає метод автоматично й масштабовано. Ніші не зводяться до правила "є рядки – шифруй рядки": серед фрагментів коду з рядковими літералами шифрування переважає тільки у 50,67 % випадків, тобто таке правило було б не точнішим за випадкове вгадування; розмітка оракулом цю градацію передає, бо вимірює якість, а не наявність конструкції.

Запас якості непрозорих предикатів. Проти висновку про домінування непрозорих предикатів є деяке заперечення: можливо, непрозорі предикати переважають з незначним відривом, а допуск δ_o зовеликий, щоб це помітити. Розподіл різниці $Q(\text{opaque}) - \max\{Q(\text{strenc}), Q(\text{meta})\}$ на 8690 фрагментах коду, де непрозорі предикати належать до еталонної множини, це знімає: медіана 0,1640, міжквартильний розмах 0,1241–0,1968, а частка випадків з різницею не більшою за $\delta_o = 0,05$ – тільки 3,43 %. Отже, у понад 96 % випадків перевага непрозорих предикатів перевищує допуск у кілька разів, і їхнє домінування не є артефактом межі їхньої нерозрізності.

Аналіз чутливості розмітки. Частка непрозорих предикатів залишається у діапазоні 90,1–96,4 % на всій сітці допусків (табл. 4), тобто головний висновок стійкий. Натомість null-клас реагує різко (7,8 % при $\delta_o = 0,02$ проти 76,8 % при $\delta_o = 0,20$), бо зростання допуску втягує до δ_o -множини дедалі слабші стратегії обфускації коду. Тому робочу точку доцільно тримати в діапазоні 0,02–0,10, де null-клас залишається змістовним; прийняте значення 0,05 належить до нього.

Табл. 4. Чутливість розмітки до допуску δ_o : частка null-класу, обсяг дієвої підмножини, частки присутності стратегій обфускації коду та середня потужність еталонної множини ($n = 9848$ фрагментів коду; приналежність перераховано з компонент Q на кожному кроці сітки) / Sensitivity of the labelling to the tolerance δ_o : the null-class share, the size of the effective subset, the strategy presence shares and the mean cardinality of the reference set ($n = 9848$ fragments; membership is recomputed from the Q components at every step of the grid)

δ_o	Null-клас %	Дієвих	Непрозорі предикати %	Шифрування рядків %	Рефлексивна %	Середня $ Q^* $
0,02	7,8	9080	95,3	4,1	2,0	1,014
0,05	8,2	9041	96,1	3,8	3,4	1,033
0,10	13,8	8486	96,4	5,5	8,9	1,108
0,15	32,4	6659	95,6	12,8	17,0	1,254
0,20	76,8	2283	90,1	30,5	38,0	1,586

Стійкість домінування непрозорих предикатів до нормалізації компоненти P підтверджено окремим сканом за трьома нормалізаціями (відносна фрагментна, відносна методу, що містить фрагмент, згладжена насиченість) у всіх дев'яти членах корпусу: частка непрозорих предикатів залишається у діапазоні 85,1–99,6 %, тоді як частка насичення P спадає від 0,51–0,64 – за відносною фрагментної нормалізації до 0,00 – за згладженою насиченості. Отже, насичення було артефактом нормалізації, а домінування непрозорих предикатів від неї не залежить; сканування параметра напівнасичення $\kappa_P \in [2,5]$ складу еталонних множин не змінює, що й обґрунтовує вибір $\kappa_P = 3$.

Консистентність часток за членами корпусу. Частка непрозорих предикатів перевищує 86 % у кожному з дев'яти членів корпусу (табл. 5), а довірчі інтервали взаємно перекриваються, попри те, що обсяги членів відрізняються на два порядки. Знаковий тест щодо гіпотези про частку 0,5 дає дев'ять перевищень із дев'яти ($p = 0,004$) і тлумачиться описово: дев'ять спостережень дають низьку потужність, тож тест підтверджує напрям ефекту, а не його силу. Змістовно важливо, що жоден член не поводить інакше, в т.ч. з бібліотеками різних доменів (розмітка Markdown, шаблонізація, оброблення YAML, кодеки, арифметика раціональних чисел). Присутність вузько застосовних стратегій обфускації коду, натомість, залежить від домену: рядкові літерали часті у бібліотеках оброблення тексту й майже відсутні в арифметичній, що підтверджує – передумова застосовності є властивістю коду, а не метрики.

Табл. 5. Частка дієвих фрагментів коду з непрозорими предикатами в еталонній множині за членами корпусу, з 95 % довірчим інтервалом Вілсона; знаменником є тільки дієві фрагменти члена ($\delta_o = 0,05$; n подано окремо для кожного члена) / Share of effective fragments with opaque predicates in the reference set by corpus member, with a 95 % Wilson confidence interval; the denominator is the effective fragments of that member only ($\delta_o = 0,05$; n is given separately for each member)

Член корпусу	Дієвих фрагментів коду	Частка, %	95 % ДІ, %
Markdig	1974	96,9	96,0–97,5
Scriban	1959	96,8	95,9–97,5
YamlDotNet	1783	94,3	93,1–95,3
Fractions	1758	98,4	97,7–98,9
SimpleBase	1196	94,0	92,5–95,2
Hashids.Net	204	95,1	91,2–97,3
Slugify.Core	74	93,2	85,1–97,1
SampleLib	50	94,0	83,8–97,9
nanoid-net	43	86,0	72,7–93,4

Природа домінування непрозорих предикатів та потреба в розмітці фрагментів проміжного коду. Присутність непрозорих предикатів у 96,12 % еталонних множин є результатом вимірювання простору стратегій обфускації коду, а не свідченням переваги запропонованого методу. На джерело домінування непрозорих предикатів вказує декомпозиція "застосовність×заслуга": непрозорий предикат спрацьовує майже на кожному фрагменті (99,31 %), тоді як шифрування рядків потребує рядкових літералів (7,47 % фрагментів коду), а рефлексивна обфускація – придатних до переписування викликів (14,70 %). Домінування непрозорих предикатів є добутком універсальної застосовності на істотний запас якості (медіана 0,1640) і зберігається за всіх трьох нормалізацій компоненти складності (85,1–99,6 %), на

всій сітці допусків (90,1-96,4 %) та за потроєння корпусу, яке зрушило частку тільки на 0,15 відсоткового пункту. Отже, воно є властивістю коду, а не наслідком добору ваг α , β , γ , λ , і його не усунути перезважуванням компонент Q .

Звідси випливає відповідь на заперечення "якщо відповідь майже завжди однакова, розмітка зайва". По-перше, стало правило "завжди непрозорий предикат" помиляється на 8,2 % фрагментів коду pull-класу (обфускація – чисті витрати) та ще на $\approx 3,9$ % дієвих фрагментів коду, де непрозорі предикати до еталонної множини не входять. По-друге, правило на основі наявності конструкції теж не працює: серед фрагментів коду з рядковими літералами шифрування є оптимальним тільки у 50,67 % випадків, тобто не точніше за випадкове вгадування на тій самій підмножині, заради якої правило й запроваджують. Придатність стратегії обфускації коду визначає не наявність синтаксису, а виміряна якість – саме цю градацію передає розмітка оракулом. По-третє, pull-клас має структурний механізм (22,3 % для фрагментів коду типу "дані" проти 2,8-3,6 % для розгалужених), тобто невіддатливості фрагмента проміжного коду прогнозована з його структури і є придатною ціллю навчання моделі.

Узгодження простору стратегій обфускації коду. Метод оракул-розмітки у загальному формулюванні оперує п'ятьма стратегіями обфускації фрагмента коду, а експеримент вимірює три дієві та тотожне перетворення. Це не спрощення: перейменування ідентифікаторів і структурне ускладнення визначені на рівні цілої збірки (змінюють таблицю метаданих чи граф типів), тож їхня якість на окремому фрагменті не визначена. Оскільки одиницею розмітки є фрагмент, простір звужено саме до стратегій обфускації коду, для яких вимірювана величина існує; це знімає заперечення про "замалий простір стратегій обфускації коду". Оцінювання збіркових стратегій обфускації коду потребує іншої одиниці аналізу і є предметом окремої роботи.

Апроксимації щільного рівня. Три компоненти метрики обчислено спрощено: стійкість R – як частку невідновлюваних рядкових літералів замість усереднення за трьома декомпіляторами (ILSpy, dotPeek, dnSpy); структурну несхожість S – на сирих гістограмах кодів операцій без зважування за оберненою частотою; накладні витрати O – тільки за приростом розміру коду. Спрощення діють однаково на всі стратегії обфускації коду, тому не пояснюють асиметрії *присутності стратегій обфускації коду*, але занижують абсолютні значення Q , роблячи наведені числа оцінкою низу. Повне обчислення цих компонент є предметом подальшої роботи.

Провізорність допуску та оцінок застосовності. Значення $\delta_o = 0,05$ прийнято провізорно; канонічне означення прив'язує допуск до шуму вимірювання, $\delta_o = c\delta_c$. Аналіз чутливості показав стійкість головного висновку на сітці 0,02–0,20, але склад pull-класу до допуску чутливий (7,8–76,8 %). Ознака-передумова завищує застосовність (для рефлексивної обфускації коду 53,79 % за ознакою проти 14,70 % фактичного спрацювання), тому оцінки за ознаками є верхніми межами, а відповідні заслуги – нижніми; у роботі наведено обидві оцінки.

Атрибуція на рівні фрагмента проміжного коду та генеровані методи. Приріст складності графа потоку керування приписується фрагментові відніманням решти методу, що містить фрагмент, тож перетворення

методу, що містить фрагмент, може дати ненульову зміну у фрагменті без відповідної конструкції (788 випадків – для шифрування рядків, 1003 – для рефлексивної обфускації коду); на висновки це не впливає, бо до еталонних множин потрапляють тільки два та вісім таких фрагментів коду. Розмітка проміжного коду охоплює й фрагменти генерованих компілятором методів – ітераторів, асинхронних автоматів, лямбда-замикань (1252; 12,7 %), структура яких може відрізнитися від рукописного коду.

Узагальнення. Дев'ять бібліотек – замало для узагальнення на всю сукупність .NET-коду; знаковий тест на дев'яти членах має низьку потужність і тлумачиться описово. Три найбільші члени опрацьовано рівномірною підвибіркою (2100 фрагментів коду, фіксоване зерно), що дає незміщені оцінки, але ширші довірчі інтервали. Корпус не покриває вебзастосунки, роботу з базами даних, багатопотокові бібліотеки; частки ніш для інших доменів очікувано будуть іншими, проте структура результату – ні, оскільки універсальність непрозорого предиката від домену не залежить. Процедура детермінована: повторний прогін трьох найменших повністю опрацьованих членів (SampleLib, Hashids .NET, nanoid-net) відтворив усі 334 записи побітово.

Обговорення результатів дослідження. У дослідженні [13] автори запропонували рамку кількісного оцінювання якості обфускації коду за трьома категоріями – потужністю, стійкістю та вартістю – і визначили для кожної власні показники вимірювання. Отримані тут результати узгоджуються з цим підходом за складом компонент, проте відрізняються призначенням: у зазначеній роботі метрику обчислюють для оцінювання якості обфускації коду вже застосованого захисту, тоді як тут та сама за природою величина породжує навчальний сигнал, тобто виконує роль оракула розмітки.

У роботі [5] автори застосували машинне навчання моделі до векторних подань проміжного коду LLVM і класифікували застосовані обфускувальні перетворення. Одиниця аналізу там збігається з ужитою тут, але задача обернена – розпізнати уже застосоване перетворення, а не дібрати придатне наперед; отримані тут дані показують, що для другої задачі мітку не можна вивести з наявності синтаксичної конструкції, оскільки шифрування рядків переважає тільки у 50,67 % фрагментів коду з рядковими літералами.

У роботі [6] автори запропонували непрозорі предикати, стійкі до динамічного символічного виконання, і показали, що приріст стійкості досягається за незначних накладних витрат. Це узгоджується з отриманим тут домінуванням непрозорих предикатів у 96,12 % еталонних множин, проте пояснення тут інше: декомпозиція застосовності та заслуги показує, що домінування непрозорих предикатів зумовлене насамперед універсальною застосовністю перетворення (99,31 %), а не винятковою його силою.

У дослідженні [14] автори експериментально виміряли стійкість бінарного порівняння програм до обфускації коду. Такий вимір відповідає компоненті стійкості вжитої тут метрики; водночас у цій роботі компоненту стійкості обчислено спрощено – як частку невідновлюваних рядкових літералів, – тому наведені абсолютні значення якості є оцінкою низу.

У роботі [8] автори дослідили навчуваність, робастність та адаптивність глибоких моделей на обфускова-

ному коді. Отримані тут дані доповнюють цю лінію з іншого боку: показано, що придатність стратегії обфускації коду до фрагмента проміжного коду є прогнозованою величиною, оскільки невіддатливість має структурний механізм – серед фрагментів переміщення даних null-клас сягає 22,3 % проти 2,8–3,6 % для розгалужених типів.

У дослідженні [3] автори виявляли факт застосування обфускації коду у застосунках Android глибокими моделями, до того ж мітки для навчання моделі були відомі за побудовою – з процедури, якою обфускацію застосовували. Методологічно – це найближча паралель до запропонованого нами підходу; відмінність тільки у тому, що тут міткою є не факт застосування перетворення, а його оптимальність за вимірюваною якістю, тому мітка узгоджена з критерієм остаточного оцінювання якості обфускації коду.

Зіставлення з наведеними роботами підтверджує доцільність проведеного саме нашого дослідження: у літературі окремо розвинено кількісне оцінювання якості обфускації коду [13, 14], машинне навчання моделі на проміжному коді [5, 8] та пошукові процедури добору перетворень [10], проте жодна з цих ліній не використовує метрики обфускації коду як джерело міток для навчання моделі. Саме таке поєднання, за якого критерій розмітки збігається з критерієм оцінювання якості обфускації коду, і відрізняє запропонований метод від наявних результатів.

Отже, внаслідок виконаної роботи можна сформулювати такі наукову новизну та практичну значущість результатів дослідження.

Наукова новизна отриманих результатів дослідження – отримав подальший розвиток метод розмітки фрагментів проміжного коду, у якому, на відміну від наявних методів, еталонну множину придатних стратегій обфускації коду визначає той самий критерій, за яким оцінюють остаточну якість обфускації коду, а не експертна евристика чи правило на основі наявності синтаксичної конструкції, що практично усуває методологічний розрив між критерієм навчання моделі та критерієм оцінювання якості обфускації коду і робить розмітку відтворюваною та масштабованою на реальні збірки .NET.

Практична значущість результатів дослідження – можна використати для автоматичного формування навчального набору міток фрагментів проміжного коду .NET, що дає змогу замінити обчислювально витратну процедуру вимірювання якості обфускації коду, яка потребує циклу "обфускація – верифікація – вимірювання" за кожною стратегією і триває секунди на кожен фрагмент, малозатратним прогнозом навченої моделі за мілісекунди, а отримані кількісні дані про ніші застосовності стратегій обфускації коду – 7,47 % фрагментів коду для шифрування рядків, 14,70 % для рефлексивної обфускації коду та 8,2 % null-класу – можуть бути застосовані для налаштування адаптивних обфускаторів платформи .NET.

Висновки / Conclusions

Розроблено та експериментально перевірено метод розмітки фрагментів проміжного коду, який визначає еталонну множину придатних стратегій обфускації коду цим самим критерієм, за яким оцінюють остаточну

якість обфускації коду, і який масштабується на реальні збірки .NET. За результатами проведеного дослідження можна зробити такі основні висновки.

1. Розроблено метод розмітки фрагментів проміжного коду .NET, у якому еталонну множину придатних стратегій обфускації коду становлять оптимальні за інтегральною метрикою якості перетворення в межах заданого допуску, а мітку сформовано як бінарний вектор належності без згортання до однієї стратегії обфускації коду.
2. Інстанційовано компоненту структурної складності на рівні окремого фрагмента проміжного коду через згладжену насиченість, чим усунуто насичення метрики на коротких фрагментах коду: частка насичених значень впала від 51–64 % до нуля в усіх дев'яти членах корпусу.
3. Проведено перевірку розробленого методу на корпусі з дев'яти бібліотек .NET з дозвільними ліцензіями, виконавши 29 544 вимірювання пар "фрагмент – стратегія" на 9848 фрагментах коду, до того ж кожне перетворення проходило двоступеневий контур верифікації семантичної еквівалентності, який відхилив 0,58 % пар.
4. Встановлено, що присутність стратегій обфускації коду в еталонних множинах різко асиметрична: вставлення непрозорих предикатів входить до 96,12 % множин (довірчий інтервал 95,70–96,50 %), шифрування рядків – до 3,80 %, а рефлексивна обфускація – до 3,41 %.
5. Досліджено природу асиметрії присутності стратегій обфускації коду декомпозицією застосовності та заслуги і показано, що вона є структурною властивістю простору стратегій обфускації коду, а не наслідком добору ваг: непрозорий предикат спрацьовує на 99,31 % фрагментів коду, шифрування рядків застосовне до 7,47 %, рефлексивна обфускація – до 14,70 %. Підтверджено стійкість домінування непрозорих предикатів за всіх трьох нормалізацій компоненти складності (85,1–99,6 %) та на всій сітці допусків (90,1–96,4 %), а медіана запасяк 0,1640 у кілька разів перевищує допуск.
6. Виявлено, що шифрування рядків потрапляє до еталонної множини у 50,67 % фрагментів коду з рядковими літералами, тож правило на основі наявності конструкції було б на цій підмножині не точнішим за випадкове вгадування, а також виявлено null-клас обсягом 8,2 % фрагментів коду зі структурним механізмом утворення: серед фрагментів коду типу "дані" він сягає 22,3 % проти 2,8–3,6 % для розгалужених типів.
7. Показано, що практичне значення розробленого методу полягає в заміні витратної процедури вимірювання малозатратним прогнозом, а нетривіальна цінність такого прогнозу зосереджена у нішах – 7,47 % фрагментів коду для шифрування рядків, 14,70 % для рефлексивної обфускації коду та 8,2 % null-класу. Подальші дослідження спрямовано на обчислення компоненти стійкості за трьома живими декомпіляторами, вимірювання реальних накладних витрат за часом і пам'яттю, калібрування допуску за шумом метрики та розширення корпусу на не покриті домени.

Декларація про використання засобів ШІ. Програмні засоби ChatGPT та Grammarly було використано авторами виправдано і винятково для лінгвістичного редагування, стилістичного коригування та перефразування окремих фрагментів тексту. Автори критично перевірили, систематизували й остаточно затвердили зміст роботи, беручи на себе повну відповідальність за поданий матеріал.

References

1. Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4), 81:1–37. <https://doi.org/10.1145/3212695>

2. Aragón-Jurado, J. M., Jareño, J., de la Torre, J. C., Dorronsoro, B., & Ruiz, P. (2024). Two-level software obfuscation with co-operative co-evolutionary algorithms. In *2024 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. (pp. 1–8). <https://doi.org/10.1109/CEC60901.2024.10612116>
3. Balachandran, V., Sufatrio, Tan, D. J. J., & Thing, V. L. L. (2016). Control flow obfuscation for Android applications. *Computers & Security*, 61, 72–93. <https://doi.org/10.1016/j.cose.2016.05.003>
4. Banescu, S., & Pretschner, A. (2018). A tutorial on software obfuscation. *Advances in Computers*, 108, 283–353. <https://doi.org/10.1016/bs.adcom.2017.09.004>
5. Cao, Y., Zhou, Z., & Zhuang, Y. (2025). Advancing code obfuscation: Novel opaque predicate techniques to counter dynamic symbolic execution. *Computers, Materials & Continua*, 84(1), 1545–1565. <https://doi.org/10.32604/cmc.2025.062743>
6. Cohen, R., David, R., Mori, R., Yger, F., & Rossi, F. (2025). Experimental study of binary diffing resilience on obfuscated programs. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2025)*, Lecture Notes in Computer Science (pp. 223–243). Springer. https://doi.org/10.1007/978-3-031-97620-9_13
7. Collberg, C., Thomborson, C., & Low, D. (1997). *A taxonomy of obfuscating transformations* (Technical Report No. 148). Department of Computer Science, University of Auckland. URL: <https://hdl.handle.NET/2292/3491>
8. Conti, M., Vinod, P., & Vitella, A. (2022). Obfuscation detection in Android applications using deep learning. *Journal of Information Security and Applications*, 70, article ID 103311. <https://doi.org/10.1016/j.jisa.2022.103311>
9. Golovko, I., Savenko, O., Vizhevskiy, P., & Sachenko, A. (2024). Obfuscation process with machine learning module. In *Proceedings of the 2024 IEEE 14th International Conference on Dependable Systems, Services and Technologies (DESSERT)*. (pp. 1–7). IEEE. <https://doi.org/10.1109/DESSERT65323.2024.11122185>
10. Jin, H., Lee, J., Yang, S., Kim, K., & Lee, D. H. (2024). A framework to quantify the quality of source code obfuscation. *Applied Sciences*, 14(12), article ID 5056. <https://doi.org/10.3390/app14125056>
11. Marinaro, T., Martinelli, F., Mercaldo, F., & Santone, A. (2021). Detecting call indirection obfuscation through equivalence checking in Android environment. *Procedia Computer Science*, 192, 1659–1669. <https://doi.org/10.1016/j.procs.2021.08.170>
12. McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), 308–320. <https://doi.org/10.1109/TSE.1976.233837>
13. Raubitsek, S., Felbauer, P., Mallinger, K., & Schrittwieser, S. (2025). Classification of obfuscation techniques in LLVM IR: Machine learning on vector representations. *Machine Learning and Knowledge Extraction*, 7(4), article number 125. <https://doi.org/10.3390/make7040125>
14. Uhm, J., Kwon, Y., & Koo, H. (2025). On the learnability, robustness, and adaptability of deep learning models for obfuscation-applied code. In *Proceedings of the 2025 Workshop on Software Understanding and Reverse Engineering ACM*. (pp. 68–75). <https://doi.org/10.1145/3733822.3764676>

I. G. Golovko, O. S. Savenko

Khmelnytskyi National University, Khmelnytskyi, Ukraine

AN ORACLE-OBFUSCATOR LABELLING METHOD FOR .NET INTERMEDIATE CODE FRAGMENTS IN ADAPTIVE SOURCE CODE PROTECTION

The existing practice of building training data for adaptive obfuscation of .NET code is analysed, in which the set of suitable strategies is assigned to a fragment by expert judgement or by heuristic rules of the form "the fragment holds string constants, so encrypt its strings". It is established that this practice does not scale and guarantees no link between the label and the criterion by which the final result is judged, so models are trained to predict one thing while being measured by another. A method of automatic labelling of intermediate-code fragments is developed, in which the reference set of suitable strategies for a fragment consists of the transformations that are δ -optimal under an integral obfuscation quality metric, that is, under the very criterion by which the final result is judged, while the label is given as a multi-hot membership vector with no collapse to a single strategy. The structural complexity component is instantiated at the scope of an individual fragment through soft saturation, which preserves the discriminating power of the metric on short fragments where direct relative normalisation degenerates. A two-stage semantic equivalence gate is implemented, admitting only correct transformations to the labelling. The method is evaluated on a corpus of nine permissively licensed .NET libraries: 29,544 fragment-strategy pairs are measured across 9,848 fragments, and the gate rejects 0.58 % of the pairs. It is established that strategy presence in the reference sets is sharply asymmetric: opaque predicate insertion enters 96.12 % of the sets (confidence interval 95.70–96.50 %) against 3.80 % for string encryption and 3.41 % for reflective obfuscation. The nature of the asymmetry is examined through an applicability-versus-merit decomposition, and it is shown to be a structural property of the strategy space rather than a merit of the method: an opaque predicate fires on 99.31 % of the fragments, whereas string encryption applies to only 7.47 %. The dominance is confirmed to persist across all normalisations and tolerances, and the median quality margin of 0.1640 exceeds the tolerance several times over. It is found that string encryption enters the reference set in 50.67 % of the fragments carrying string literals, so a presence-based rule would be no more accurate there than a coin toss. A null class covering 8.2 % of the fragments is also identified, for which no strategy pays off and which has a structural mechanism: among data-movement fragments it reaches 22.3 %. It is shown that the method allows a costly measurement procedure to be paid for once and thereafter replaced by millisecond inference of a trained model.

Keywords: software code obfuscation; cyclomatic complexity; semantic equivalence; opaque predicates; machine learning; integral obfuscation quality indicator.