Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
____________ Іван ПАРХОМЕНКО
«__» _________ 2025 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань *12 Інформаційні технології*
(шифр і назва галузі знань)

спеціальність *125 Кібербезпека та захист інформації*
(код і назва спеціальності)

освітній ступень *магістр*

освітньо-наукова програма *Кібербезпека*
(назва освітньої програми)

на тему: «Механізми захисту вихідного коду програмного застосунку на базі мови програмування Python»

Виконавець: студент II курсу, групи КБм-22

_______________ **Сергій КУЛІБАБА**
(підпис) (Ім'я, ПРІЗВИЩЕ)

	Ім'я, ПРІЗВИЩЕ	**Підпис**
Науковий керівник	Іван ПАРХОМЕНКО	
Нормоконтроль	Юрій БАБЕНКО	

Київ 2025

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
____________ Іван ПАРХОМЕНКО
«25» жовтня 2024 р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи

спеціальності *125 Кібербезпека та захист інформації*
(код і назва спеціальності)

освітній ступень *магістр*

Здобувача(ки) КБм-22 Кулібаби Сергія Олександровича
(група) (прізвище ім'я по-батькові)

Тема кваліфікаційної роботи Механізми захисту вихідного коду програмного застосунку на базі мови програмування Python

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Рішення засідання кафедри кібербезпеки та захисту інформації факультету інформаційних технологій протокол № 4 від 24.10.2024 р.

2. МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень Процес захисту вихідного коду програмного забезпечення.

Предмет досліджень Методи та механізми захисту програмного забезпечення на рівні вихідного коду.

Мета	Розробити механізми захисту вихідного коду програмних застосунків із використанням мови програмування Python, щоб підвищити конфіденційність інформації та захисту інтелектуальної власності розробників.
Вихідні дані для проведення роботи	Інтерпретована мова програмування Python, засоби атак на програмне забезпечення, засоби захисту програмного забезпечення.

3. ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна	Розроблені новітні механізми обробки та захисту вихідного коду, які дозволяють забезпечити захист інтелектуальної власності розробників та об'єктів операційних систем.
Практична цінність	Практичне значення роботи полягає в розробці механізмів захисту вихідного коду мови програмування Python, яке дозволить захистити інтелектуальну власність розробників.

4. ЕТАПИ ВИКОНАННЯ РОБОТИ

Найменування етапів робіт	Строки виконання робіт (початок-кінець)
Розробка плану для досягнення мети роботи	25.10.2024 – 29.11.2024
Аналіз особливостей мови програмування Python	30.11.2024 – 25.12.2024
Аналіз та тестування методів зламу й захисту програмних застосунків	21.01.2025 – 13.02.2025
Створення тестового вихідного коду та розробка механізмів захисту для мови програмування Python	15.02.2025 – 09.04.2025
Визначення ефективності розроблених механізмів	10.04.2025 – 01.05.2025
Оформлення пояснювальної записки згідно методичних рекомендацій	02.05.2025 – 15.05.2025
Подача пакету документів на розгляд ЕК	15.05.2025 – 19.05.2025

Завдання видав ____________ Іван ПАРХОМЕНКО
(підпис) (Ім'я, ПРІЗВИЩЕ)

Завдання прийняв
до виконання ____________ Сергій КУЛІБАБА
(підпис) (Ім'я, ПРІЗВИЩЕ)

Дата видачі завдання: 25.10.2024 р.
Термін подання кваліфікаційної роботи до ЕК 19.05.2025 р.

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Механізми захисту вихідного коду програмного застосунку на базі мови програмування Python»: 95 сторінок, 50 рисунків та 10 таблиць. 80 літературних джерел.

Об’єкт дослідження – процес захисту вихідного коду програмного забезпечення.

Мета роботи – розробити механізми захисту вихідного коду програмних застосунків із використанням мови програмування Python, щоб підвищити конфіденційність інформації та захисту інтелектуальної власності розробників.

У роботі досліджено особливості мови програмування Python. Проведено аналіз зламу та захисту програмних застосунків, які написані на мові програмування Python. Розроблено новітні механізми захисту вихідного коду для мови програмування Python. Визначено ефективність розроблених механізмів.

Наукова новизна: розроблені новітні механізми обробки та захисту вихідного коду, які дозволяють забезпечити захист інтелектуальної власності розробників та об’єктів операційних систем.

Актуальність теми: У сучасних умовах стрімкого розвитку IT особливої актуальності набуває захист вихідного коду програм, зокрема написаних мовою Python, яка зберігає код у відкритому або легко декомпільованому вигляді. Обфускація є одним із ключових механізмів протидії реверс-інжинірингу та викраденню інтелектуальної власності. Створення власного обфускатора дозволяє підвищити рівень безпеки програмного забезпечення та адаптувати захист до специфіки Python. Це робить тему дослідження актуальною як у науковому, так і в прикладному аспекті.

Ключові слова: обфускація, деобфускація, захист, злам, вихідний код, мова програмування, Python.

ЗМІСТ

ВСТУП 7
РОЗДІЛ 1. ТЕХНОЛОГІЇ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 9
1.1 Python як мова програмування для розробки програмного забезпечення 9
1.2 Особливості мови програмування Python 11
1.3 Допоміжні засоби для розробки програмного забезпечення 14
Висновки до першого розділу 20
РОЗДІЛ 2. ОСНОВНІ АСПЕКТИ ЗАХИСТУ ВИХІДНОГО КОДУ 21
2.1 Обфускація та деобфускація вихідного коду 21
2.2 Методи зламу та аналізу вихідного коду 29
2.3 Основні методи захисту вихідного коду 35
2.4 Огляд програми-аналога для обфускації вихідного коду 38
Висновки до другого розділу 45
РОЗДІЛ 3. РОЗРОБКА МЕХАНІЗМІВ ЗАХИСТУ ВИХІДНОГО КОДУ 47
3.1 Визначення основних компонентів вихідного коду 47
3.2 Структури проєктів 49
3.3 Ідентифікатори вихідного коду 54
3.4 Рядкові типи вихідного коду 59
3.5 Коментарі та документації вихідного коду 64
Висновки до третього розділу 67
РОЗДІЛ 4. ВИЗНАЧЕННЯ ЕФЕКТИВНОСТІ РОЗРОБЛЕНИХ МЕХАНІЗМІВ ЗАХИСТУ ВИХІДНОГО КОДУ 69
4.1 Формалізація оцінювання ефективності механізмів захисту 69
4.2 Оцінка ефективності зміни структури проєкту 74
4.3 Оцінка обфускації ідентифікаторів у вихідному коді 75
4.4 Оцінка обробка рядкових типів вихідному коді 76
4.5 Оцінка захисту коментарів та документацій вихідного коду 77

4.6 Аналіз отриманих результатів 79
Висновки до четвертого розділу 83
ВИСНОВКИ 85
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 87
ДОДАТОК А 96

ВСТУП

Актуальність даної теми обумовлена недостатнім рівнем поширення ефективних, відкритих інструментів захисту коду для Python, а також зростаючим попитом на безпечне розповсюдження програмного забезпечення – як у комерційних проектах, так і в дослідницьких розробках. Існуючі обфускатори або забезпечують поверхневий рівень захисту, що не витримує аналізу досвідченого зловмисника, або методи зламу таких механізмів вже висвітлені в мережі. Крім того, обфускація повинна не тільки заплутувати код, але й зберігати коректну роботу програми, що вимагає глибокого розуміння механізмів виконання Python-коду, зберігання метаданих та багато іншого.

Основною задачею цієї кваліфікаційної роботи є знайти, розглянути, проаналізувати мову програмування Python, методи зламу та захисту програмних застосунків, розробити власні механізми захисту вихідного коду, а також визначити їх ефективність.

Науковою новизною цієї кваліфікаційної роботи є розроблені новітні механізми обробки та захисту вихідного коду, які дозволяють забезпечити захист інтелектуальної власності розробників та об'єктів операційних систем.

Об'єктом дослідження є процес захисту вихідного коду програмного забезпечення, що є критично важливим при передачі вихідного коду третім особам або ж при публікації самого коду у відкриті джерела.

Предметом дослідження є методи та механізми захисту програмного забезпечення на рівні вихідного коду.

У сучасному цифровому світі програмне забезпечення відіграє ключову роль у всіх сферах людської діяльності – від медицини, банківської справи, освіти до оборонної промисловості. Зі зростанням обсягів і складності програмних продуктів, створених мовами високого рівня, зростає і загроза їх несанкціонованого копіювання, реверс-інжинірингу та неправомірного використання. Особливо це стосується інтерпретованих мов програмування, серед яких Python займає провідне місце

завдяки своїй простоті, читабельності, широкому застосуванню в науці, штучному інтелекті, веб-розробці та автоматизації.

Python є відкритою мовою, що поширюється безкоштовно, і її код зазвичай надається у відкритому вигляді. Це створює сприятливі умови як для колективної розробки, так і для потенційних зловживань: викрадення логіки програми, копіювання алгоритмів, компрометація безпеки тощо. Більшість рішень, написаних на Python, доставляються у вигляді вихідного або напівскомпільованого байт-коду, що значно полегшує їх зворотну інженерію. У зв’язку з цим особливої важливості набуває проблема захисту вихідного коду Python-застосунків від несанкціонованого доступу, модифікації або розповсюдження.

Одним із найпоширеніших та ефективних способів ускладнення аналізу та модифікації коду є обфускація – процес перетворення програмного коду у вигляд, що зберігає функціональність, але ускладнює його розуміння та аналіз. На практиці це означає зміну імен змінних, функцій, класів, структур, а також модифікацію структури програми без втрати її функціональності. Проте для мови Python, яка зберігає багато метаданих під час свого виконання, розробка якісних обфускаторів становить складну інженерну задачу.

В межах цієї роботи є потреба в розробці нових механізмів обфускації, який виконуватиме трансформації на рівні вихідного коду без зміни логіки програми, з урахуванням особливостей Python. Розроблений інструмент дозволить автоматизувати процес захисту програмного застосунку перед його публікацією або передачею третім особам, що значно підвищить рівень безпеки інтелектуальної власності.

РОЗДІЛ 1
ТЕХНОЛОГІЇ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Python як мова програмування для розробки програмного забезпечення

Мова програмування Python була створена наприкінці 1980-х років нідерландським програмістом Гвідо ван Россумом (Guido van Rossum). Офіційно перша версія Python побачила світ у лютому 1991 року. Вона була задумана як проста й інтуїтивно зрозуміла мова, яка поєднувала б переваги інших мов, таких як ABC, C, Modula-3 та Lisp [1].

Початково Python мав на меті стати мовою, що значно спрощує розробку програмного забезпечення, забезпечуючи читабельний синтаксис, який був би зрозумілий навіть програмістам-початківцям. Протягом 1990-х і 2000-х років мова поступово набувала популярності завдяки своїй простоті, відкритості, та можливостям швидкого створення прототипів.

Ключовою віхою у розвитку Python стало створення версії Python 2.0 у 2000 році, яка значно розширила можливості мови, зокрема введенням спискових виразів та розширенням стандартної бібліотеки. У 2008 році було випущено Python 3.0, який запропонував значні зміни в синтаксисі та базовій структурі мови, що потребувало певного перехідного періоду для адаптації програмістів.

Сьогодні Python є однією з найпопулярніших мов програмування у світі, займаючи стабільно високі позиції в рейтингах мов програмування [2]. Широке застосування Python знайшов у сфері веб-розробки, аналізі даних, машинному навчанні, штучному інтелекті, автоматизації процесів, DevOps та в наукових дослідженнях.

Завдяки відкритому коду та великій екосистемі сторонніх бібліотек Python активно розвивається спільнотою програмістів з усього світу, що дозволяє швидко адаптувати мову до нових технологічних викликів.

Сучасний світ важко уявити без програмного забезпечення, яке забезпечує роботу мобільних додатків, веб-сервісів, банківських систем і навіть автомобілів з автопілотом. Однією з ключових мов програмування, що лежить в основі створення різноманітних рішень, є Python. Завдяки своїй простоті, гнучкості та великій екосистемі бібліотек, ця мова знаходить застосування в веб-розробці, аналізі даних, штучному інтелекті та автоматизації завдань [3].

Python є основою багатьох популярних продуктів і сервісів: наприклад, платформи для веб-розробки, такі як Instagram, використовують його для ефективної обробки даних користувачів, а хмарні сервіси, як Dropbox, покладаються на Python для синхронізації файлів. У науковій сфері інструменти на базі Python, зокрема Jupyter Notebook, дозволяють дослідникам зручно аналізувати дані та тестувати алгоритми машинного навчання. Крім того, допоміжні засоби, для прикладу, TensorFlow і PyTorch стали стандартом для створення нейронних мереж, що використовуються, наприклад, для розпізнавання облич у сервісі Google Photos [4].

Порівнюючи Python з такими мовами, як C++ та Java, можна відзначити, що Python має більш лаконічний синтаксис і дозволяє швидко створювати прототипи завдяки великій кількості готових допоміжних засобів або бібліотек мови програмування. Водночас, для завдань, де критично важлива оптимізація продуктивності та детальний контроль над апаратними ресурсами, перевагу надають C++ чи Java [5]. У сфері кібербезпеки Python застосовується для розробки прототипів та автоматизації, тоді як C++ забезпечує високу ефективність у роботі з системними ресурсами.

Завдяки своїй універсальності Python знаходить застосування не лише у веб-розробці та аналізі даних, а й у створенні ігор, а також у фінансових технологіях та робототехніці. Таким чином, Python продовжує залишатися однією з найвпливовіших мов програмування сучасності, підтримуючи як інноваційні проєкти, так і великі комерційні рішення.

1.2 Особливості мови програмування Python

Python – це мова програмування, що вирізняється винятковою простотою, виразністю та зрозумілістю, завдяки яким розробники можуть зосередитись на вирішенні завдань, а не на синтаксичних нюансах. Її дизайн-філософія базується на чистоті коду та легкості читання, що забезпечує простоту підтримки програм навіть у великих проєктах. Оскільки Python є інтерпретованою мовою, зміни в коді можна тестувати оперативно без потреби проходження етапу компіляції, що значно пришвидшує цикл розробки [6].

Динамічна типізація дає змогу працювати з даними без попереднього оголошення типів, що сприяє гнучкості при створенні прототипів і швидкому виявленню помилок під час виконання. Завдяки підтримці різних парадигм програмування, Python є універсальним інструментом для вирішення широкого спектру завдань, від веб-розробки до аналізу даних та автоматизації. Мова активно застосовується у створенні різноманітних додатків, де важлива швидкість розробки та можливість легко адаптувати програму під змінні вимоги ринку [7].

Величезна стандартна бібліотека надає готові інструменти для роботи з мережевими протоколами, обробкою файлів, базами даних та іншими завданнями, що дозволяє розробникам інтегрувати існуючі рішення без потреби писати код з нуля. Крім того, можливість легко інтегрувати Python з іншими мовами, як-от C або C++, дозволяє оптимізувати критично важливі за продуктивністю частини коду, що є важливим аспектом при розробці масштабованих систем [8].

Крос-платформеність Python забезпечує виконання розроблених програм на різних операційних системах без значних змін, що підвищує гнучкість і масштабованість застосунків. Активна спільнота розробників постійно розширює екосистему мови новими бібліотеками та інструментами, що дозволяє вирішувати все більш складні завдання в галузях штучного інтелекту, аналізу даних, веб-розробки та автоматизації.

Особливу роль у структуризації коду відіграє підтримка об’єктно-орієнтованого програмування (ООП). ООП – це парадигма, яка організовує код у вигляді об’єктів, що об’єднують дані і методи для їх обробки [9]. Такий підхід дозволяє створювати окремі, повторно використовувані компоненти, що значно спрощує розширення, підтримку та тестування складних систем. Використання ООП дозволяє логічно розбивати програму на модулі, які можуть розроблятися та перевірятися окремо, що є необхідним для створення надійних програмних рішень. На рисунку 1.1 відображено приклад організації вихідного коду за парадигмою програмування ООП.

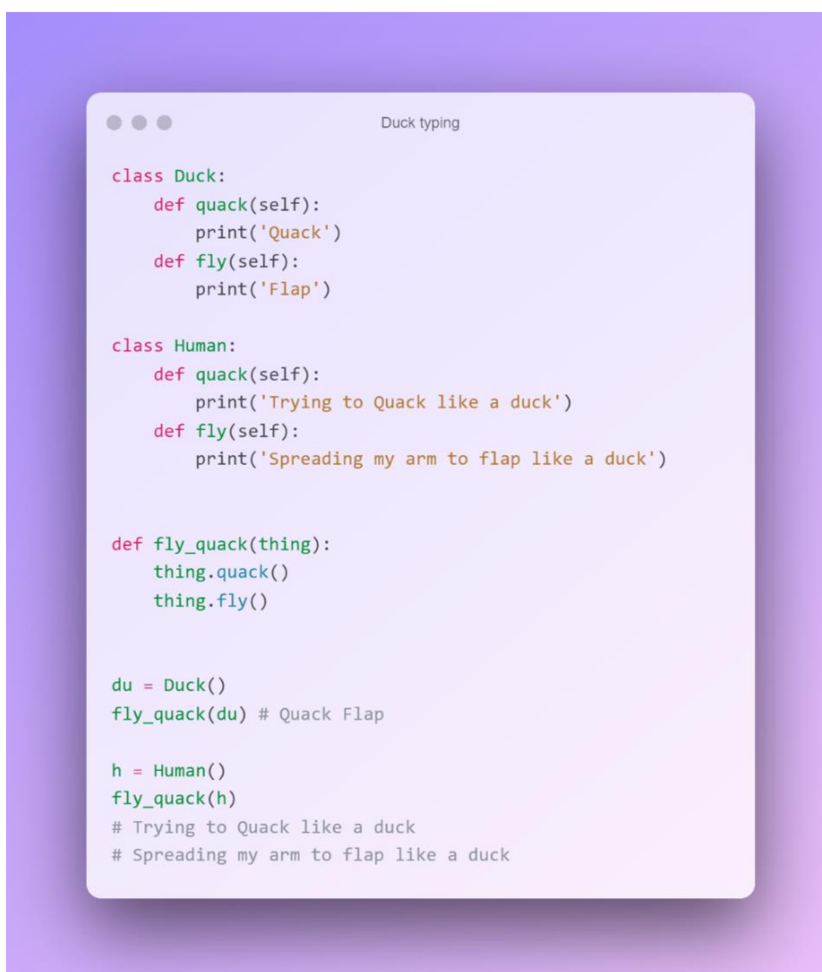

Рисунок 1.1 – Організація коду за парадигмою програмування ООП

Крім того, можливості метапрограмування Python, такі як використання декораторів, лямбда-функцій та генераторів, забезпечують додаткову гнучкість. Вони дозволяють змінювати поведінку програмних компонентів під час виконання та

створювати адаптивний код, здатний реагувати на змінні умови роботи. Такий підхід не лише збагачує функціональність програми, а й сприяє зменшенню дублювання коду, полегшуючи його подальшу підтримку [10]. На рисунку 1.2 відображено приклад метапрограмування за допомогою лямба-функцій.

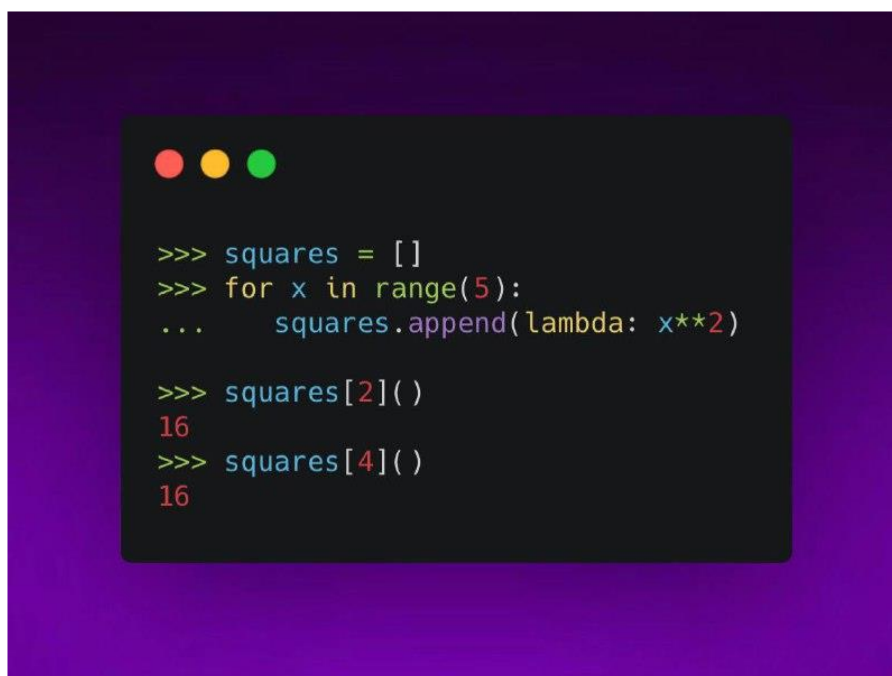

Рисунок 1.2 – Метапрограмування із використанням лямба-функцій

Немало важливою частиною мови програмування є стилі написання коду. Ця мова програмування має свій стиль написання коду, який має назву «PEP 8» (Python Enhancement Proposal 8) – це офіційний стильовий принцип для написання коду на Python. Він визначає загальні правила форматування, які сприяють єдиному стилю коду, підвищенню його читабельності та зручності підтримки. Документ створений для того, щоб усі розробники Python слідували однаковим стандартам, що особливо важливо у великих командах і відкритих проєктах [11].

Python має філософію «чистий, простий і зрозумілий код». PEP 8 є частиною цієї філософії, оскільки допомагає писати код уніфіковано та без зайвої складності. Дотримання цього стилю полегшує читання коду іншими розробниками та дозволяє зменшити кількість помилок, пов’язаних з неузгодженістю стилю.

Основна суть PEP 8 полягає в тому, що код повинен бути чистим, логічно структурованим і зрозумілим для всіх. Це досягається через правильні відступи,

обмеження довжини рядків, єдині правила іменування змінних та функцій, а також правильне форматування коментарів і документації. Документ містить детальні рекомендації щодо організації імпортів, використання пробілів у виразах, розташування дужок і багатьох інших аспектів написання коду.

Дотримання PEP 8 є стандартом у професійній розробці на Python. Більшість популярних середовищ розробки та інструментів автоматично перевіряють код на відповідність цьому стилю, а сам Python-інтерпретатор та лінтери (наприклад, flake8, pylint, black) можуть допомогти у підтриманні відповідності стандарту.

Отже, поєднання простоти синтаксису, динамічної типізації, підтримки ООП, можливостей метапрограмування та визначення стилів написання коду робить Python однією з найпотужніших та найгнучкіших мов програмування сучасності. Ці особливості забезпечують розробникам всі необхідні інструменти для створення як невеликих скриптів, так і великих, масштабованих систем, що відповідають сучасним вимогам ринку.

1.3 Допоміжні засоби для розробки програмного забезпечення

Допоміжні засоби – це інструменти, бібліотеки, пакети та середовища, які сприяють спрощенню, пришвидшенню та підвищенню якості розробки програмного забезпечення [12]. До них належать як окремі бібліотеки та пакети, так і комплексні інтегровані середовища розробки (IDE), системи контролю версій, інструменти для тестування, налагодження та збірки проектів.

Однією з найважливіших переваг мови програмування Python є її потужна та активна спільнота. Велика кількість розробників з усього світу регулярно створюють, підтримують і розвивають бібліотеки та інструменти, які суттєво спрощують процес розробки. Завдяки цьому мова має надзвичайно широку екосистему, що містить десятки тисяч готових бібліотек і модулів для різноманітних задач.

Бібліотека представляє собою набір готових функцій і модулів, що дозволяють розробникам використовувати попередньо реалізовані рішення для виконання

специфічних завдань, не витрачаючи час на написання коду з нуля. Бібліотеки активно застосовуються в різних галузях:

1) Обробка даних та аналітика: Наприклад, бібліотеки Pandas та NumPy дозволяють ефективно працювати з великими обсягами даних, що є критично важливим для фінансового аналізу та наукових досліджень [13].

2) Машинне навчання та штучний інтелект: Бібліотеки, такі як TensorFlow та PyTorch, спрощують розробку моделей штучного інтелекту, що знаходять застосування у розпізнаванні образів, обробці природних мов та інших напрямках [14].

3) Веб-розробка: Бібліотеки, зокрема Flask та Django, дозволяють швидко створювати веб-додатки, RESTful API та інші інтернет-сервіси [15].

4) Інші сфери: Бібліотеки для роботи з базами даних, графікою, мережею та безпекою забезпечують додатковий функціонал, необхідний для комплексних систем [16; 17].

Спільнота Python характеризується високою активністю та відкритістю: більшість популярних бібліотек є безкоштовними та мають відкритий вихідний код, доступний на платформах, таких як GitHub та GitLab [18; 19]. Такий підхід значно спрощує процес розробки програмного забезпечення, дозволяючи швидко знаходити готові рішення чи інтегрувати вже написаний код у нові проєкти.

Завдяки цим чинникам Python не тільки посів провідні позиції у рейтингах популярності мов програмування, а й став основою для розвитку таких ключових технологічних напрямків, як штучний інтелект, машинне навчання, аналіз даних, автоматизація, та веб-розробка. Ця популярність забезпечує Python стабільною підтримкою, регулярними оновленнями та постійним розширенням його можливостей, роблячи мову привабливою як для початківців, так і для досвідчених програмістів.

Крім бібліотек, до допоміжних засобів належать інтегровані середовища розробки (наприклад, PyCharm, Visual Studio Code), які надають розширені можливості [20]:

- Підсвічування синтаксису та автодоповнення коду.
- Інтегроване налагодження та тестування.
- Зручний інтерфейс для роботи з системами контролю версій (Git, SVN).
- Розширення для аналізу якості коду та інтеграції з системами CI/CD (Continuous Integration, Continuous Delivery, Continuous Deployment; безперервне розгортання застосунків) [21].

Використання допоміжних засобів дозволяє:

- Заощаджувати час: Готові рішення значно скорочують обсяг коду, який потрібно писати вручну.
- Підвищувати продуктивність: Інструменти автоматизації та налагодження сприяють швидшому виявленню помилок і оптимізації коду.
- Покращувати якість: Стандартизовані бібліотеки та платформи забезпечують високу якість реалізації, що важливо для підтримки та масштабування програмного забезпечення.

Можна виділити наступні практичні приклади:

1) TensorFlow використовується для створення та навчання нейронних мереж, що застосовуються в сервісах, таких як Google Photos для автоматичного розпізнавання обличчя та сортування зображень [22].

2) Flask дозволяє швидко розгортати веб-додатки та API, що використовуються як для невеликих внутрішніх корпоративних проектів, так і для масштабних мікросервісних архітектур у компаніях типу Netflix [23].

3) Pandas є ключовим інструментом для аналізу та обробки фінансових даних, що застосовується в аналітичних системах, таких як платформи Bloomberg [24].

Для роботи з бібліотеками в мові програмування Python допомагає «Pip». Pip (Pip Installs Packages) – це стандартний менеджер пакетів для мови програмування Python, який використовується для встановлення, оновлення та управління сторонніми бібліотеками та модулями Python із репозиторію пакетів Python Package Index (PyPI) [25].

Основні можливості Pip [26]:

1) Встановлення пакетів із офіційних або сторонніх репозиторіїв.

2) Оновлення пакетів до останніх версій.

3) Видалення пакетів і контроль встановлених версій.

4) Перегляд інформації про встановлені пакети та їхні залежності.

5) Робота з файлами залежностей (requirements.txt) для простого розгортання проєктів.

Особливості роботи [27]:

- Вбудований у Python, починаючи з версії Python 2.7.9 та Python 3.4 і вище.
- Забезпечує вирішення залежностей пакетів (встановлює необхідні бібліотеки, від яких залежить пакет).
- Підтримує створення ізольованих середовищ розробки (у поєднанні з "virtualenv або вбудованим модулем venv у Python).

Завдяки готовим рішенням, а саме бібліотекам із даними, можна робити збірку розробленого програмного забезпечення для різних операційних систем – Windows та Unix-подібних систем – це бібліотека «PyInstaller». PyInstaller – це інструмент для мови програмування Python, який дозволяє створювати автономні (standalone) виконувані файли для програм Python. Основне призначення PyInstaller – це зручне розповсюдження Python-додатків без необхідності встановлення інтерпретатора Python чи додаткових бібліотек на комп'ютер користувача [28].

Основні можливості PyInstaller [29]:

1) Створення автономних виконуваних файлів: Генерує файли .exe (для Windows), або виконувані файли для Linux і macOS, які включають у себе Python-інтерпретатор і всі необхідні залежності.

2) Підтримка різних платформ: PyInstaller працює з Windows, Linux і macOS, дозволяючи створювати виконувані файли для відповідної платформи.

3) Режими пакування: Дозволяє створювати як єдиний файл («onefile»), який містить усе необхідне для запуску програми, так й окремий проєкт з виконуючим файлом та додатковими ресурсами («onedir»).

4) Підтримка зовнішніх ресурсів: Дозволяє включати додаткові ресурси (іконки, зображення, звуки тощо) до виконуваного файлу.

5) Підтримка консолі та графічних додатків: Можливість створювати як консольні, так і графічні застосунки.

Основні переваги використання PyInstaller [30]:

- Спрощення розповсюдження програм, оскільки користувачам не потрібно встановлювати Python або будь-які залежності вручну.
- Приховування вихідного коду (часткова обфускація) через компіляцію в байт-код та інтеграцію у виконуваний файл.
- Підтримка більшості бібліотек Python, включаючи популярні графічні інтерфейси (Tkinter, PyQt, Kivy, PyGame).

Недоліки та обмеження:

- Через включення Python-інтерпретатора та всіх залежностей розмір вихідного файлу може бути значно більшим, ніж початковий Python-скрипт.
- В режимі «onefile» програма має розпакуватись у тимчасову директорію перед виконанням, що може збільшити час запуску.
- Хоча PyInstaller ускладнює доступ до вихідного коду, його можна витягнути за допомогою спеціалізованих інструментів.

При розробці програмного забезпечення, можуть бути задачі із обробкою текстових даних. Мова програмування Python має вбудовану бібліотеку із даними для обробки тексту, яка називається «re» – це стандартна бібліотека для роботи з регулярними виразами, яка надає широкі можливості для обробки текстових даних. Вона дозволяє ефективно знаходити, аналізувати, перевіряти та перетворювати текст за заданими шаблонами [31].

Основні можливості бібліотеки «re»:

- Пошук тексту за шаблоном: дозволяє знаходити окремі слова, фрази чи комбінації символів у великих текстах відповідно до визначеного шаблону.

- Перевірка відповідності тексту: можна перевірити, чи відповідає текст певним правилам (наприклад, формат номера телефону, email-адреси або ідентифікатора користувача).
- Замінювання частин тексту: дає змогу автоматично виконувати заміну знайдених фрагментів тексту за певним правилом чи шаблоном.
- Розбиття тексту: підтримує розбиття рядків на частини за допомогою складних розділювачів або визначених комбінацій символів.
- Витягування інформації з тексту: дозволяє легко та ефективно отримувати конкретні дані або частини тексту (наприклад, дати, адреси, числа) з великих за обсягом текстових даних.

Переваги бібліотеки «re»:

- Вбудована в стандартну бібліотеку Python, тому не потребує додаткового встановлення та доступна одразу для використання.
- Висока швидкість роботи навіть із великими обсягами текстової інформації завдяки оптимізованим алгоритмам пошуку та аналізу.
- Підтримка складних шаблонів, що забезпечує гнучкість роботи з текстом будь-якої складності.
- Можливість гнучкого використання, що дозволяє застосовувати бібліотеку як для простих, так і для складних завдань текстового аналізу.

Корисність використання бібліотеки «re»:

- Аналіз лог-файлів або журналів подій для пошуку помилок чи специфічних записів.
- Перевірка введених користувачем даних (наприклад, перевірка валідності електронних адрес чи номерів телефонів).
- Автоматизація очищення, перетворення та обробки великих масивів текстових даних.
- Розробка додатків для збору інформації з веб-сторінок або інших текстових джерел.

Висновки до першого розділу

В першому розділі було розглянуто мову програмування зі сторони практичної реалізації, її особливості, а також допоміжні засоби даної мови для розробки програмного забезпечення.

Проаналізувавши наукові джерела, можна визначити, що мова програмування Python наразі має попит серед розробників, адже дана мова дозволяє зручно, якісно та швидко розробляти програмні забезпечення як для локального, так і масштабного використання.

Серед особливостей можна виділити парадигми мови програмування, а саме об’єктно-орієнтованість та модульність. Саме ці парадигми є ключовими при розробці застосунків для різних сфер діяльності. Більш важливим при розробці є застосування принципу ООП, що дозволяє якісно організувати код для подальшого його використання розробниками.

Створення застосунку в сучасному інформаційному світі неможливе без застосування допоміжних засобів. Розглянуто різні засоби, які дозволяють створювати програмні забезпечення за різним призначенням. Наприклад, цілі застосування бібліотек мов програмування, середовища розробки програмного забезпечення, переваги застосування допоміжних засобів, а також засоби, які дозволяють безпосередньо створювати готовий програмний засіб для різних операційних систем. Завдяки великій підтримці зі сторони інших розробників, які є ключовими особами, дозволяють швидко та якісно розробляти або доповнювати програмні засобі, за рахунок розроблених допоміжних засобів.

РОЗДІЛ 2
ОСНОВНІ АСПЕКТИ ЗАХИСТУ ВИХІДНОГО КОДУ

2.1 Обфускація та деобфускація вихідного коду

Обфускація вихідного коду – це навмисне перетворення програмного коду у форму, що зберігає його функціональність, але робить його важким для читання і розуміння людиною [32]. Іншими словами, після обфускації програма працює так само, але її вихідний або виконуваний код виглядає “заплутаним” та неочевидним. Основні цілі такого перетворення – ускладнити аналіз і зворотне проектування програми (reverse engineering), захистити алгоритми та логіку від стороннього вивчення, а також запобігти несанкціонованій модифікації або крадіжці інтелектуальної власності [33].

Зворотний інжиніринг (Reverse Engineering) – це процес аналізу програмного забезпечення, апаратних засобів або інших систем з метою вивчення їхньої структури, логіки роботи та принципів функціонування. Він передбачає розбір коду, витягнення алгоритмів або схем, щоб відновити початкову логіку, виявити вразливості або створити сумісні продукти. Зворотний інжиніринг може застосовуватися для аналізу безпеки, модифікації програм, дослідження шкідливого ПЗ, відновлення вихідного коду та інших завдань.

Обфускація особливо важлива для захисту комерційного програмного забезпечення: вона підвищує стійкість до реверс-інжинірингу, ускладнює пошук вразливостей чи обходу ліцензійних перевірок, і загалом додає рівень безпеки до застосунку. Хоча сам по собі цей підхід не робить програму невзламуваною, він помітно підвищує поріг зусиль, необхідних для її аналізу та злому.

Існує багато методів обфускації коду, які можуть застосовуватися окремо або в комплексі. Нижче розглянуто основні з них [34]:

1) *Зміна імен змінних та функцій*. Один з найпростіших способів заплутати код – це перейменувати ідентифікатори (змінні, функції, класи) на непрозорі, беззмістовні назви. Замість зрозумілих імен, наприклад, «COUNT» чи «PHONE_NUMBER» після обфускації можуть використовуватися назви типу «a», «b», «c» тощо. У результаті логіка програми приховується за беззмістовними позначеннями, і аналіз коду ускладнюється. Подібне перейменування не впливає на виконання програми, але позбавляє вихідний код семантичних підказок, які допомагають зрозуміти його роботу. Багато обфускаторів автоматично замінюють всі імена на короткі комбінації літер і цифр, роблячи декомпільований код важким для розуміння.

На рисунках 2.1 – 2.2 відображено приклад обфускації вихідного коду мови програмування Python.

```python
import base64

from models.encryptions.tdm import TDM


class Shape:
    def calculate_area(self):
        """Base area"""
        pass

    def calculate_perimeter(self):
        """Base perimeter"""
        pass


class Rectangle(Shape):
    # Initialize the Rectangle object with given length and width
    def __init__(self, length, width):
        self.length = length
        self.width = width

    # Calculate and return the area of the rectangle using the formula: length * width
    def calculate_area(self):
        return self.length * self.width

    # Calculate and return the perimeter of the rectangle using the formula: 2 * (length + width)
    def calculate_perimeter(self):
        return 2 * (self.length + self.width)


# Example usage
l = 5
w = 7
rectangle = Rectangle(l, w)

print("Area:", rectangle.calculate_area())
print("Perimeter:", rectangle.calculate_perimeter())
```

Рисунок 2.1 – Оригінальний вихідний код

```
import base64

from models.encryptions.tdm import TDM


class lgntc:
    def mipmrpiet_iuti(self):
        """ 4__B0__a1&&5&&8__e3&&7__r6__s2"""
        pass

    def mipmrpiet_ltuactetu(self):
        """ 4__B0__a1__e3&&6&&10&&12__i8__m9__p5__r7&&13__s2__t11"""
        pass


class RcepnahSc(lgntc):
    #  10&&14&&24&&31&&36&&42&&49&&53__I0__R15__a5&&19&&50__b26__c17&&29__d52&&56__e9&&13&&16&&23&&28&&40&&44__g21&&37&&46__h12&&35&&
    def __init__(self, acrdiw, htgiw):
        self.acrdiw = acrdiw
        self.htgiw = htgiw

    #  9&&13&&20&&24&&29&&32&&36&&46&&52&&56&&65&&72&&74__*73__:64__C0__a1&&6&&10&&25&&28&&41&&63__c3&&39__d12&&77__e8&&15&&23&&27&&3
    def mipmrpiet_iuti(self):
        return self.acrdiw * self.htgiw

    #  9&&13&&20&&24&&34&&37&&41&&51&&57&&61&&70&&72&&74&&82&&84__(75__)90__*73__+83__271__:69__C0__a1&&6&&10&&46&&68__c3&&44__d12&&8
    def mipmrpiet_ltuactetu(self):
        return 2 * (self.acrdiw + self.htgiw)


#  7__E0__a2&&10__e6&&12__g11__l5__m3__p4__s9__u8__x1
a = 5
h = 7
nceilrdac = RcepnahSc(a, h)

exec(base64.b64decode(TDM().decrypt("xym1pFjLGccG1yF9plldXdpRCKpkHcbnJpkFyQoIWZIiE6mNlwgbWaZG").encode()).decode())
exec(base64.b64decode(TDM().decrypt("WYbWMuXJwlwbWaX2V0WFjx0dGddSV0Q==gpKHcbnJplBlQoImcZXltjoiRlcCLY2BuHJkVpb").encode()).decode())
```

Рисунок 2.2 – Приклад обфускованого вихідного коду

2) *Вставка зайвого або фальшивого коду*. Цей підхід передбачає додавання в програму фрагментів коду, які не впливають на її кінцевий результат, але заплутують структуру і логіку. Приклади – вставка «недіючих» інструкцій (noop-операцій) на рівні машинного коду або додавання умов, які завжди хибні/істинні, та зайвих циклів. Такий «мертвий» код збільшує об’єм програми і створює хибні шляхи виконання, щоб збити аналітика з пантелику. Наприклад, обфускатор може генерувати фрагменти коду, які виконують зайві обчислення, результатами яких ніхто не користується, або викликати функції-заглушки. Хоча ці вставки не змінюють функціональність, вони ускладнюють автоматичний аналіз: інструментам реверс-інжинірингу важче відрізнити справжню логіку від штучно доданого шуму.

На рисунку 2.3 відображено приклад вставки фальшивого коду, саме «fake_computation» для ускладнення зворотного інжинірингу.

```
import random


def noop():
    """Функція-заглушка, яка нічого не робить."""
    pass


def fake_computation():
    """Функція, що виконує зайві обчислення, але їх результат ніде не використовується."""
    temp = 0
    for _ in range(1000):  # Безглуздий цикл
        temp += random.randint(1, 100) * 0  # Множення на нуль робить операцію марною
    return temp  # Результат ніколи не використовується


def always_false_condition():
    """Фрагмент коду з завжди хибною умовою."""
    x = random.randint(1, 10)
    if x > 100:  # Це ніколи не буде істинним
        print("Цей код ніколи не виконається")
```

Рисунок 2.3 – Вставка фальшивого коду

3) *Використання шифрування рядків*. Багато обфускаторів приділяють особливу увагу текстовим рядкам у коді (наприклад, повідомленням, ключам API, URL-адресам). Шифрування рядків полягає у тому, що всі літеральні рядки у програмі зберігаються у зашифрованому вигляді і розшифровуються лише під час виконання програми. Це не дозволяє легко прочитати важливі дані при статичному аналізі коду або декомпіляції. Наприклад, зрозумівши призначення рядка "License verification failed" або "DELETE FROM Users", зловмисник міг би легше знайти відповідні ділянки коду. Якщо ж такі рядки зашифровані (набіром безглуздих символів) і розкодовується тільки в процесі роботи програми, аналізатору доведеться витратити додаткові зусилля, щоб отримати їх значення. Цей метод захищає

конфіденційну інформацію в коді і ускладнює розуміння намірів програми без її запуску.

На рисунку 2.4 відображено приклад використання шифрування рядків.

```
from cryptography.fernet import Fernet
import base64

# Генерація ключа (у реальному випадку ключ має бути фіксованим або отриманим з безпечного сховища)
KEY = base64.urlsafe_b64encode(b'my_secret_key_123456789012')  # 32 байти
CIPHER = Fernet(KEY)


# Функція для шифрування рядка
def encrypt_string(plain_text):
    return CIPHER.encrypt(plain_text.encode()).decode()


# Функція для розшифрування рядка
def decrypt_string(encrypted_text):
    return CIPHER.decrypt(encrypted_text.encode()).decode()


# Зашифровані рядки
ENCRYPTED_STRINGS = {
    "error": encrypt_string("License verification failed"),
    "query": encrypt_string("DELETE FROM Users"),
    "url": encrypt_string("https://example.com/api"),
}


# Функція для отримання рядка у розшифрованому вигляді
def get_string(key):
    if key in ENCRYPTED_STRINGS:
        return decrypt_string(ENCRYPTED_STRINGS[key])
    return None


# Використання у програмі
print("Decoded Error Message:", get_string("error"))
print("Decoded Query:", get_string("query"))
print("Decoded URL:", get_string("url"))
```

Рисунок 2.4 – Використання шифрування рядків

4) *Поліморфізм та динамічне створення коду*. Поліморфізм в контексті обфускації означає генерацію множинних варіантів коду, що виконують одну й ту саму функцію, але мають різну реалізацію. Наприклад, кожна компіляція програми

може продукувати трохи інший код (різні імена змінних, інший порядок еквівалентних інструкцій, випадкові вставки тощо) – таким чином жодні дві зібрані версії не будуть ідентичними. Це ускладнює аналіз, адже відсутній один статичний шаблон коду – розібравши одну версію, не обов'язково легко зрозуміти іншу. Поліморфні техніки часто застосовуються у шкідливому програмному забезпеченні, щоб уникнути виявлення сигнатурним аналізом антивірусів. Динамічне створення коду – ще більш радикальний підхід, при якому програма генерує або завантажує фрагменти коду «на льоту», під час виконання. Наприклад, застосунок може зберігати частину своєї логіки у зашифрованому вигляді і розшифровувати та виконувати її через «eval» або JIT-компіляцію вже під час роботи. У результаті статичний аналіз такого коду вкрай утруднений – допоки програму не запущено, повної картини її поведінки не отримати. Поєднання поліморфізму і динамічного виконання дозволяє створювати самомодифікуючийся код, який змінює свою структуру під час роботи і при кожному запуску виглядає інакше. Це серйозно ускладнює як ручний аналіз, так і спроби автоматичного відстеження шкідливих шаблонів.

На рисунку 2.5 відображено приклад створення поліморфізму.

```
def generate_polymorphic_function():
    """Генерує випадкову реалізацію еквівалентної функції."""
    var_name = f"var_{random.randint(1000, 9999)}"
    operation = random.choice(["+", "-", "*", "//"])
    code = f"""
def obfuscated_function(x):
    {var_name} = x {operation} {random.randint(1, 10)}
    return {var_name} {operation} {random.randint(1, 10)}
"""
    exec(code, globals())  # Додає згенеровану функцію у глобальний простір
    return obfuscated_function
```

Рисунок 2.5 – Створення поліморфізму

5) *Використання метапрограмування та динамічного виконання коду в Python*. У Python, на відміну від C/C++, відсутні препроцесори та макроси, однак схожих результатів можна досягти за допомогою метапрограмування, динамічного

створення коду та використання глобальних змінних. Один із методів обфускації – це імітація макросів, коли певні фрагменти коду замінюються функціями або словниками, що зберігають складні вирази. Це дозволяє приховати реальну логіку за непрямими викликами, що ускладнює її аналіз. Наприклад, замість явного виконання математичних операцій використовуються фабричні функції або динамічно підставлені вирази. Ще одним способом є динамічна генерація коду, коли програма формує частини своєї логіки "на льоту" через «exec()» або «eval()». Такий підхід дозволяє створювати різні варіанти коду при кожному запуску, змінюючи імена змінних, логіку обчислень або навіть додаючи випадкові вставки, що не впливають на результат. Це робить статичний аналіз практично неможливим, оскільки до запуску невідомо, яку саме версію програми буде виконано. Додатково використовується умовна логіка вибору фрагментів коду, яка імітує препроцесорні директиви. У Python код може містити різні варіанти реалізації функцій, які визначаються лише під час виконання. Це означає, що при різних запусках або залежно від зовнішніх параметрів будуть активовані різні версії алгоритму, що робить аналіз ще складнішим. Такі методи широко застосовуються для обфускації та захисту програмного коду, оскільки ускладнюють реверс-інжиніринг, приховують важливі ділянки логіки та роблять програму менш передбачуваною для аналізу.

На рисунку 2.6. відображено приклад метапрограмування та динамічного виконання коду.

```
MACROS = {
    "ADD_TWO": lambda x: x + 2,
    "MULTIPLY_RANDOM": lambda x: x * random.randint(1, 10)
}
result = MACROS

code = "def dynamic_function(x): return x * 2"
exec(code, globals())  # Додає функцію у глобальний простір
print(dynamic_function(10))  # 20
```

Рисунок 2.6 – Метапрограмування та створення динамічного виконання коду

Деобфускація вихідного коду – це процес зворотного перетворення обфускованого коду в більш зрозумілу та аналізовану форму. Мета деобфускації – спростити або повністю відновити початкову логіку програми, що була навмисно прихована чи ускладнена [35].

При деобфускації вихідного коду або програмного забезпечення можуть виникати складності та/або обмеження [36]:

1) Деобфускація не завжди дозволяє повністю відновити первинний код, особливо при використанні поліморфних або динамічних технік.

2) Якісно виконана обфускація значно збільшує час і ресурси, необхідні для аналізу програми. Навіть за допомогою автоматизованих інструментів процес може залишатися дуже трудомістким.

3) Високий рівень динамічної або самомодифікуючої обфускації може ускладнити деобфускацію настільки, що простіше переписати логіку з нуля, ніж відновлювати її з існуючого коду.

Попри те, що обфускація часто використовується для захисту програмного забезпечення, деобфускація також має практичну користь, серед якої можна виділити [37]:

1) Аналіз безпеки: Деобфускація дозволяє виявити потенційні вразливості або шкідливий функціонал, прихований у коді. Це особливо важливо у сфері кібербезпеки, де необхідно ретельно аналізувати програмне забезпечення на предмет можливих загроз.

2) Реверс-інжиніринг для сумісності: Допомагає зрозуміти, як працюють певні закриті або не задокументовані системи, з метою створення сумісного чи інтегрованого програмного забезпечення.

3) Відновлення втраченого вихідного коду: У разі втрати оригінальних вихідних кодів деобфускація може допомогти частково або повністю відновити логіку програм, що дозволяє продовжувати їх підтримку чи модернізацію.

4) Навчання та освітні цілі: Деобфускація сприяє кращому розумінню складних програмних систем, алгоритмів або методів захисту, допомагаючи студентам, дослідникам і спеціалістам покращити навички програмування та аналізу.

2.2 Методи зламу та аналізу вихідного коду

Для аналізу вихідного коду програмного забезпечення з метою зворотного інжинірингу, обходу захисту або виявлення вразливостей використовують низку підходів.

1) **Декомпіляція** – це процес зворотного перетворення виконуваного файлу (бінарного або байт-коду) у вихідний код високого рівня, близький до початкового. Зазвичай застосовується для мов програмування з проміжним виконанням (наприклад, Java, .NET, Python).

Особливості декомпіляції [38]:

- Дозволяє відновити алгоритми та логіку, що містяться у програмі.
- Результатом є код, який може бути зрозумілим людині без глибоких знань машинного коду.
- Якість декомпіляції залежить від мови програмування та використовуваних засобів обфускації.

Приклади інструментів:

- Java: JD-GUI, JADX.
- .NET: ILSpy, dotPeek, dnSpy.
- Python: uncompyle6, pycdc.

2) **Дизасемблювання** – це перетворення виконуваного машинного коду у код низького рівня (асемблерний код). Застосовується переважно для програм, написаних мовами, що компілюються в машинний код (наприклад, C, C++, Rust).

Особливості дизасемблювання [39]:

- Дозволяє аналізувати структуру і поведінку програми на рівні процесора.

- Вимагає хорошого знання архітектури комп'ютерів та інструкцій процесора.
- Отриманий код є складнішим для аналізу порівняно з декомпіляцією.

Приклади інструментів:

- IDA Pro.
- Ghidra.
- Radare2.
- OllyDbg, x64dbg.

3) **Статичний аналіз** – аналіз коду без запуску програми. Використовується для пошуку потенційних вразливостей, розуміння структури та виявлення прихованих алгоритмів чи функцій.

Особливості статичного аналізу [40]:

- Не вимагає виконання коду, що дозволяє аналізувати потенційно небезпечні програми без ризику для системи.
- Можливість автоматизації аналізу великих обсягів коду.
- Недоліком є те, що певні аспекти поведінки програми (наприклад, динамічно створений код) можуть бути не помічені.

Приклади інструментів:

- SonarQube, Semgrep (для вихідного коду).
- IDA Pro, Ghidra (для бінарного коду).
- Fortify, Checkmarx (для пошуку вразливостей).

4) **Динамічний аналіз** – це аналіз поведінки програми під час її виконання в реальному часі або в контрольованому середовищі (наприклад, емулятор чи дебагер).

Особливості динамічного аналізу [41]:

- Дозволяє виявити динамічну поведінку програми, яка неможлива до аналізу статичним способом.
- Дає змогу відстежувати зміну стану пам'яті, файлову активність, мережеві взаємодії.

- Потребує особливої обережності при запуску потенційно небезпечного коду.

Приклади інструментів:

- Debuggers: WinDbg, GDB, LLDB.
- Монітори активності: Process Monitor, Wireshark, Fiddler.
- Sandboxes (пісочниці): Cuckoo Sandbox, FireEye Sandbox.

Після ознайомлення із основними методами зламу програмного забезпечення, можна застосувати декілька вказаних підходів для отримання інформації про розроблене невеличке програмне забезпечення. Буде застосовуватись статичний аналіз та декомпіляція застосунку. Для цього знадобляться наступні програмні засоби [42; 43; 44]:

1) Windows FlareVM – комплексна віртуальна операційна система, яка містить набір програм для різноманітного тестування програмного забезпечення.

2) PEStudio – це інструмент для статичного аналізу виконуваних файлів Windows (PE-файлів: EXE, DLL тощо). Він використовується для виявлення потенційних загроз, аналізу стиснення, обфускації, цифрових підписів, імпортованих API-функцій, секцій файлу та інших характеристик, які можуть вказувати на шкідливе програмне забезпечення або упаковку файлу.

3) Detect it Easy – це аналізатор виконуваних файлів, який використовується для визначення компілятора, лінкера, пакувальника, шифрування та інших характеристик файлів PE (Windows), ELF (Linux), Mach-O (macOS).

4) PyInstXtractor – це Python-скрипт, який використовується для витягування (розпакування) файлів, упакованих PyInstaller.

5) PycDC – це інструмент для декомпіляції байт-коду Python (.pyc файлів) у вихідний код .py. Він дозволяє відновити вихідний код Python із скомпільованих файлів, що можуть бути витягнуті, наприклад, за допомогою PyInstXtractor.

Для початку потрібно переглянути EXE-файл, та переконатись про створення застосунку завдяки інструменту «PyInstaller». Для цього проводиться статичний аналіз виконуючої програми через «PEStudio». На рисунках 2.7 – 2.8 відображено

результат про аналіз виконуючого файлу. В розділі «indicators» вказано «file-ratio» 95,43% – що може свідчити про стиснення заголовків файлу, а зафіксовані рядки в розділі «strings» свідчать про використання Python-утиліт в програмі. Перевірити дійсність стиснення даних можна завдяки програмі «Detect it Easy», яка ще може додатково надати інформацію про програму. На рисунку 2.9 відображено результат аналізу застосунку за допомогою програми «Detect it Easy», яка підтвердила стиснення даних, де додатково вказала рівень стиснення, де «best» відповідає рівню 7-9, а також програму для стискання («ZLIB»).

Рисунок 2.7 – Результат «indicators» при аналізу виконуючого файлу

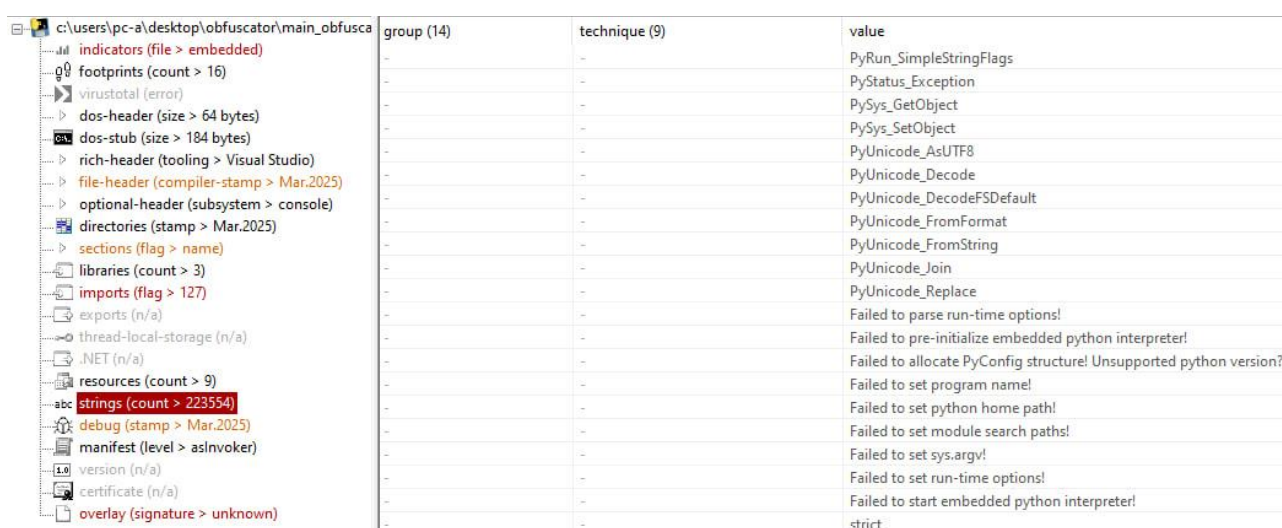

Рисунок 2.8 – Результат «strings» при аналізу виконуючого файлу

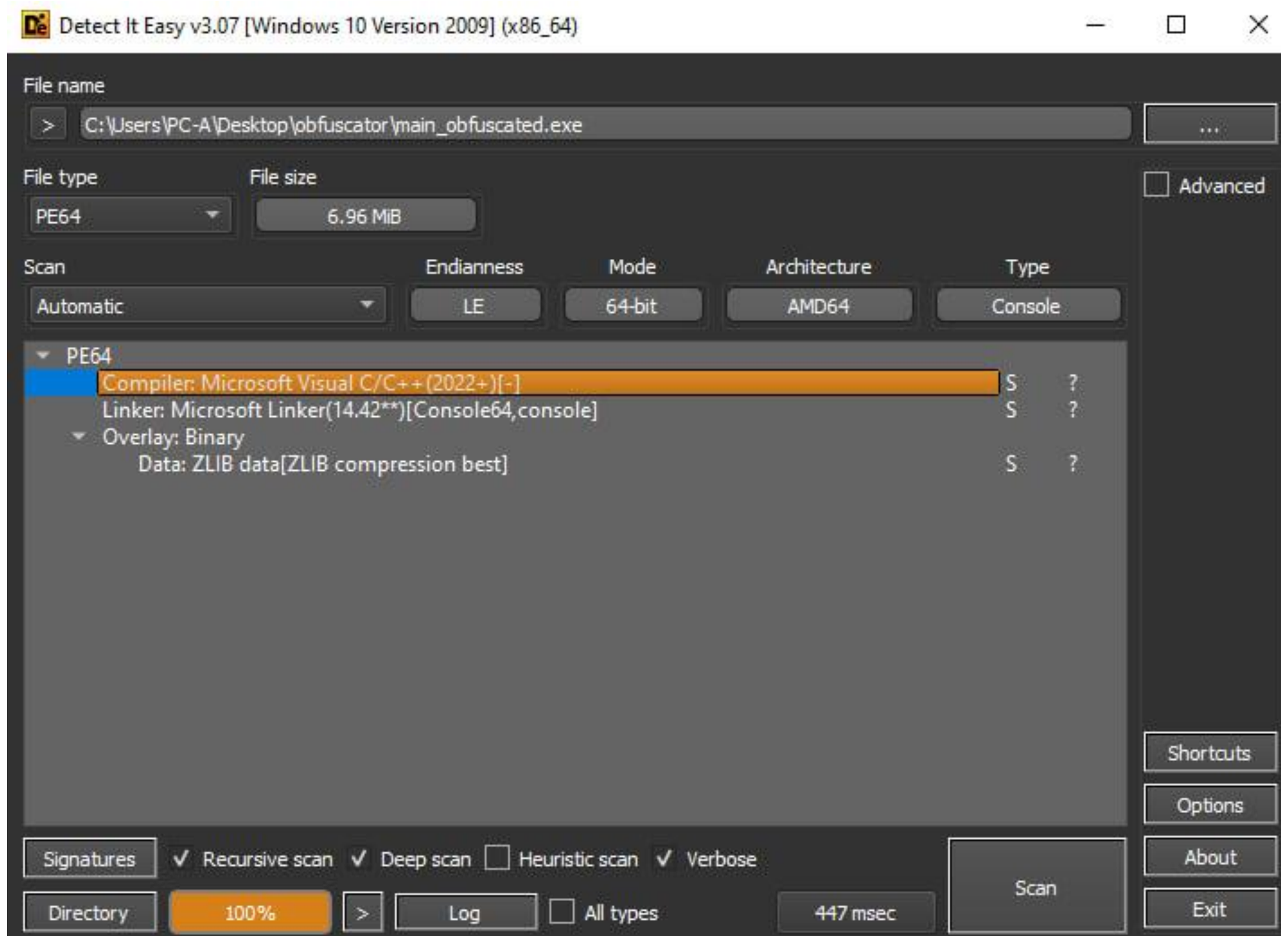

Рисунок 2.9 – Результат виявлення пакувальника за допомогою програми Detect it Easy

Розпакувати виконуючу Python-програму можна декількома шляхами:

1) Через debugger-програми виявити стиснуті секції, витягнути їх через редактор HEX (для прикладу, «HxD» для операційної системи Windows) та розпакувати дані (для прикладу, бібліотека «ZLIB» на мові програмування Python).

2) Застосувати готове рішення для автоматизованого розпакування Python EXE-програм та застосувати готовий декомпілятор для Python-програм.

Другий шлях зменшує час на аналіз Python-програм, тому буде саме він продемонстрований. На рисунках 2.10 – 2.12 відображено процес зламу EXE-програми.

```
PS F:\Python\Projects\Decompiler\original> python .\pyinstxtractor.py ..\targets\original.exe
[+] Processing ..\targets\original.exe
[+] Pyinstaller version: 2.1+
[+] Python version: 3.11
[+] Length of package: 6958426 bytes
[+] Found 60 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: pyi_rth_inspect.pyc
[+] Possible entry point: test_all.pyc
[+] Found 98 files in PYZ archive
[+] Successfully extracted pyinstaller archive: ..\targets\original.exe

You can now use a python decompiler on the pyc files within the extracted directory
```

Рисунок 2.10 – Розпакування EXE-програми інструментом «PyInstXtractor»

```
PS F:\Python\Projects\Decompiler\original> ls .\original.exe_extracted\*.pyc


    Каталог: F:\Python\Projects\Decompiler\original\original.exe_extracted


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        07.03.2025      0:56           1856 pyiboot01_bootstrap.pyc
-a----        07.03.2025      0:56           5323 pyimod01_archive.pyc
-a----        07.03.2025      0:56          34146 pyimod02_importers.pyc
-a----        07.03.2025      0:56           7053 pyimod03_ctypes.pyc
-a----        07.03.2025      0:56           1958 pyimod04_pywin32.pyc
-a----        07.03.2025      0:56           3127 pyi_rth_inspect.pyc
-a----        07.03.2025      0:56            360 struct.pyc
-a----        07.03.2025      0:56           1826 test_all.pyc
```

Рисунок 2.11 – Перевірка наявності .pyc-файлів після розпакування застосунку

```
PS F:\Python\Projects\Decompiler\original\pyarmor_obfuscated.exe_extracted\pycdc\Debug> .\pycdc.exe ..\..\..\original.exe_extracted\test_all.pyc
# Source Generated with Decompyle++
# File: test_all.pyc (Python 3.11)

import base64
from models.encryptions.tdm import TDM

class Shape:

    def calculate_area(self):
        '''Base area'''
        pass


    def calculate_perimeter(self):
        '''Base perimeter'''
        pass



class Rectangle(Shape):

    def __init__(self, length, width):
        self.length = length
        self.width = width


    def calculate_area(self):
        return self.length * self.width


    def calculate_perimeter(self):
        return 2 * (self.length + self.width)


l = 5
w = 7
rectangle = Rectangle(l, w)
print('Area:', rectangle.calculate_area())
print('Perimeter:', rectangle.calculate_perimeter())
PS F:\Python\Projects\Decompiler\original\pyarmor_obfuscated.exe_extracted\pycdc\Debug>
```

Рисунок 2.12 – Декомпіляція застосунку завдяки інструменту «PycDC»

Отриманий результат, саме вихідний код застосунку, отриманий повністю, окрім коментарів. Відсутність коментарів вихідного коду можуть лише мінімально ускладнити процес аналізу коду, але самі коментарі ніяк не впливають на виконання програми.

Таким чином, можна переконатись, що програми для різних операційних систем, які створюються завдяки мові програмування Python, не є надійно захищеними.

2.3 Основні методи захисту вихідного коду

Як було попередньо визначено, існує певний підхід до обробки вихідного коду застосунку для ускладнення його до зламу та/або аналізу – який має назву обфускація вихідного коду [45]. Окрім такого підходу, існують ще інші, які додатково можуть підійняти рівень захищеності коду або програмного забезпечення.

1) *Шифрування та упаковка виконуваних файлів* – цей метод захисту вихідного коду передбачає обробку вже скомпільованих програмних файлів з метою приховування їхньої внутрішньої структури та логіки роботи від несанкціонованого аналізу або зворотного інжинірингу [46].

Упаковка виконуваних файлів полягає в стисканні виконуваного коду та створенні саморозпаковуваного виконуваного файлу, який розпаковує (розшифровує) захищений код лише в пам’яті під час виконання програми. Це значно ускладнює статичний аналіз, оскільки реальний код програми доступний лише після запуску, що обмежує можливості аналізу за допомогою дизасемблерів та декомпіляторів.

Шифрування виконуваних файлів або їх частин передбачає використання криптографічних алгоритмів для приховування критичних частин коду. Під час запуску програма розшифровує ці частини в оперативній пам'яті, що додатково ускладнює виявлення та аналіз важливих фрагментів логіки.

Інструменти для шифрування та упаковки [47; 48; 49]:

- UPX (Ultimate Packer for eXecutables) – популярний інструмент для упаковки виконуваних файлів, дозволяє значно зменшити розмір програм і частково ускладнити статичний аналіз.
- VMProtect – більш складний інструмент, який не лише упаковує, але й створює віртуальну машину для виконання захищеного коду, що значно підвищує складність аналізу та злому.
- Themida – потужний засіб захисту виконуваних файлів, який включає в себе шифрування, упаковку, антидебагінг і методи протидії зворотному інжинірингу, що робить процес злому дуже ресурсомістким та важким.

2) *Антидебагінг та захист від емуляції* – це набір спеціалізованих технік і методів, що призначені для ускладнення або неможливості аналізу програмного забезпечення за допомогою інструментів налагодження (дебагерів) та емуляторів. Основна мета таких методів – приховати реальну логіку програми від аналітиків, які намагаються зрозуміти її роботу через динамічний аналіз [50].

Докладніше про методи антидебагінгу та захисту від емуляції:

- Перевірка точок переривання (breakpoints): Програма періодично перевіряє пам'ять або певні системні регістри на наявність активних точок переривання, які використовують дебагери. Якщо виявляється, що програма працює під контролем дебагера, вона може припинити роботу або змінити свою поведінку, що ускладнює аналіз.

- Використання таймерів (time checks): Метод полягає у вимірюванні часу виконання окремих операцій, оскільки процес налагодження часто уповільнює виконання програми. Якщо операція триває довше певного порогу, програма може вважати, що її аналізують, та відповідно зупинити чи змінити роботу.

- Перевірка середовища емуляції: Програма аналізує своє середовище на наявність ознак емуляції або віртуального середовища (наприклад, специфічні назви процесів, характерні для середовищ віртуалізації чи емуляції, такі як «VBoxService.exe» чи «vmtoolsd.exe»). У разі виявлення цих ознак, програма блокує виконання чи активує фіктивну логіку.

- Нестандартна поведінка системних викликів або інструкцій процесора: Використання специфічних системних викликів чи машинних інструкцій, поведінка яких у віртуальному середовищі або під час налагодження відрізняється від реального обладнання. Це дозволяє програмі виявити наявність дебагера або емулятора та перервати нормальний хід виконання.

3) *Перевірка цілісності* – це процес, що дозволяє переконатися у незмінності та автентичності програмного забезпечення після його розповсюдження або установки. Головна ідея полягає в тому, що будь-які несанкціоновані зміни у виконуваних файлах чи інших ресурсах програми можна легко виявити завдяки спеціальним методам.

Основні підходи до перевірки цілісності включають:

- Контрольні суми та хешування: Використання алгоритмів хешування (наприклад, SHA-256, MD5), які генерують унікальну контрольну суму для файлу. Навіть незначна зміна у файлі призведе до повної зміни контрольної суми, що дає змогу швидко виявити модифікацію.

- Цифрові підписи: Використання криптографічних ключів для підпису програмного коду, що гарантує його авторство та автентичність. При запуску програми або її оновленні система перевіряє цифровий підпис, і якщо він не відповідає оригіналу, програма може відмовитися запускатися.

4) *Переміщення критичної логіки на сервер* – це підхід, при якому найважливіші частини програмної логіки або алгоритми не включаються у клієнтську частину програми, а виконуються на віддаленому сервері. При такій архітектурі клієнт отримує лише результати виконання необхідних обчислень або інформацію, необхідну для роботи, без доступу до самої логіки роботи програми [51].

Основні переваги такого методу:

- Захист інтелектуальної власності: важливі алгоритми, бізнес-логіка чи формули, які є ключовими для бізнесу, залишаються недоступними для аналізу та крадіжки.
- Перешкода зворотному інжинірингу: клієнтська сторона програми не містить критично важливої інформації, яку можна було б легко проаналізувати або змінити.
- Покращення контролю: оновлення або зміни логіки проводяться на сервері, що дозволяє швидше реагувати на виявлені проблеми чи вразливості без необхідності поширення оновлень серед всіх користувачів.

2.4 Огляд програми-аналога для обфускації вихідного коду

Наразі існує мала кількість програмних засобів для захисту вихідного коду Python-застосункуів. Серед існуючого, немаловідомого рішення, є PyArmor – це інструмент для захисту Python-коду за допомогою обфускації, шифрування та ліцензування. Він використовується для ускладнення реверс-інжинірингу Python-програм [52].

Основні можливості PyArmor:

1) Обфускація коду – змінює імена змінних, функцій, класів.

2) Шифрування вихідного коду – .py файли перетворюються на зашифровані .pyc файли.

3) Захист від декомпіляції – ускладнює зворотну інженерію (uncompyle6, pycdc працюють гірше).

4) Ліцензування – можна обмежити запуск за ключем або на певному пристрої.

5) Підтримка PyInstaller – працює разом із PyInstaller.

Інструмент PyArmor працює наступним чином:

1) Відбувається шифрування Python-коду за допомогою AES (Advanced Encryption Standard).

2) Створюється «loader», який розшифровує код тільки під час виконання.

3) Маскує логіку програми, щоб ускладнити аналіз хакерам.

Якщо PyArmor працює в пам’яті, то процес зламу може бути наступний:

- Використання PyInstXtractor (якщо .exe).
- Перехоплення розшифрованого коду в пам’яті (за допомогою x64dbg).
- Аналіз вихідного коду через IDA Pro, Ghidra або Frida.

Використання PyArmor, а також інших подібних програмних засобів, може відбуватись для наступного:

- Захист комерційного Python-коду.
- Ускладнення реверс-інжинірингу.
- Ліцензування Python-програм.

Окрім PyArmor, також існують інші готові рішення, які дозволять обробити належним чином код для ускладнення його зламу та аналізу:

- Cython (компілює Python у C).
- Nuitka (перетворює виконуючий файл Python-коду у C++).
- pyobfuscate (інструмент для базової обфускації).

На рисунку 2.13 відображено приклад обфускації коду за допомогою інструменту PyArmor. Такий оброблений вихідний код працює із допоміжними

засобами інструмента PyArmor. Також ці додаткові ресурси дозволяють блокувати модифікацію вихідного коду, де результат такої зміни відображений на рисунку 2.14.

```
# Pyarmor 9.1.1 (trial), 000000, non-profits, 2025-03-06T23:31:49.094158
from pyarmor_runtime_000000 import __pyarmor__
__pyarmor__(__name__, __file__, b'PY000000\x00\x03\x0b\x00\xa7\r\r\n\x80\x00\x01\x00\x08\x00\x00\x00\x04\x00\x00\x00@\x00\
```

Рисунок 2.13 – Обфускація вихідного коду за допомогою PyArmor

```
1  # Pyarmor 9.1.1 (trial), 000000, non-profits, 2025-03-
2  from pyarmor_runtime_000000 import __pyarmor__
3  __pyarmor__(__name__, __file__, b'PY000000\x00\x03\x0b
4  print('hello, world')
```

```
pyarmour (1)
F:\Python\Projects\Obfuscator_Final\venv\Scripts\python.exe
Traceback (most recent call last):
  File "<frozen __main__>", line 3, in <module>
RuntimeError: unauthorized use of script (1:1107)

Process finished with exit code 1
```

Рисунок 2.14 – Результат запуску застосунку після модифікації коду

Для більш точного аналізу роботи даної бібліотеки, було прийнято рішення застосувати інструмент дебагінгу, який вбудований в застосунок PyCarhm, щоб виявити механізми обробки вихідного коду. Отримана інформація може бути корисною як для розробки особистих механізмів захисту вихідного коду та застосунків, так і для модифікації вже розроблених механізмів. Для цього буде проводитись статичний та динамічний аналіз.

Структура проєкту бібліотеки PyArmor відображена на рисунку 2.15. На ній видно ряд компонентів, які є важливими для повноцінної роботи даного інструменту. Так, як користувач, в основному, використовує CLI (Command-line Interface), то більш

цікавою директивою є «cli». На рисунку 2.16 відображено зміст каталогу «cli» із вкладними папками та файлами.

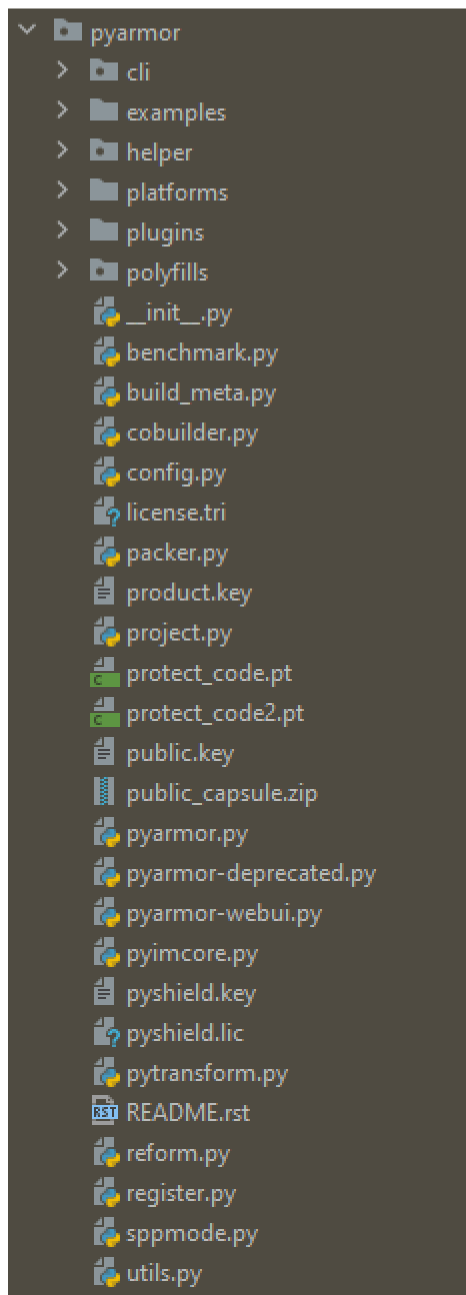

Рисунок 2.15 – Структура проєкту PyArmor

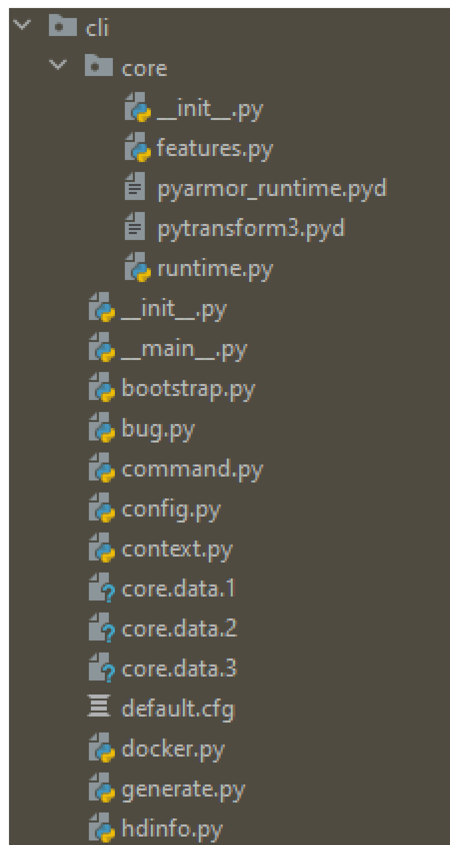

Рисунок 2.16 – Зміст каталогу «cli»

В даному змісті можна помітити модуль «__main__.py», а також файли «pyarmor_runtime» та «pytransform3.pyd». Модуль «__main__.py» відповідає за роботу в CLI-режимі (Command-line interface), а файли із розширенням .pyd служать як DLL-файлами – бібліотеками динамічної компоновки, створені на мові Python, які можуть запускатись із іншим кодом мови Python під час виконання.

Суть .pyd файлів полягає в тому, щоб розширити можливості Python, дозволяючи використовувати бібліотеки, написані на C або C++, як звичайні модулі Python. Вона містить скомпільований машинний код, а не байткод Python (.pyc), тому працює швидше і може мати доступ до низькорівневих можливостей ОС або апаратури.

Якщо створюються такі файли, виникає гіпотеза про приховування важливої логіки інструменту. Для виявлення цього було запущено процес тестування збірки тестового вихідного коду, за допомогою відладки у текстовому редакторі коду PyCharm. Під час дослідження працездатності інструменту, було виявлено

використання вказаних допоміжних файлів із розширенням pyd, які утворюють екземпляри обробленого вихідного коду. На рисунку 2.17 відображено структуру оброблених даних, а також ключову функцію, яка працює разом із кодом, який скомпільовано у pyd-файлі.

Рисунок 2.17 – Структура оброблених даних тестового вихідного коду

Шаблон даних, який починається з префікса «co», є характерним для структури об’єкта типу Code Object у мові програмування Python. Цей тип даних є ключовим елементом при формуванні байт-коду, оскільки він містить повну інформацію про структуру програми: імена змінних, інструкції, константи, таблиці відлагодження

тощо. Саме Code Object використовується при компіляції вихідного коду у формат pyc – скомпільований байт-код Python.

У процесі обфускації програма динамічно формує ці об'єкти, після чого виконує їх обробку. Алгоритм, який здійснює цю обробку, прихований у зовнішньому модулі, що реалізований у вигляді pyd-файлу. Такі файли є динамічними бібліотеками для Python, написаними мовами низького рівня, як-от C або C++, та скомпільовані у машинний код. Це унеможливлює їх безпосереднє прочитання або аналіз звичайними засобами Python.

Як результат, інструмент створює нову версію програми, яка виглядає як модифікований вихідний код, але насправді є результатом глибокої перебудови структури коду (див. рис. 2.13).

Якщо спробувати здійснити запуск обфускованого коду, а також визначити точки зупинок (breakpoints), то можна помітити, що допоміжні файли pyd опрацьовують обфускований код – ознака цьому є виведення «frame not available» у текстовому редакторів PyCharm під час відладки (рис. 2.18). При цьому, якщо ітераційно виконувати програму, то далі можна виявити частини вихідного коду, саме назва класу, його поля та значення, що є основним тестового розробленого та обфускованого засобу (рис. 2.19).

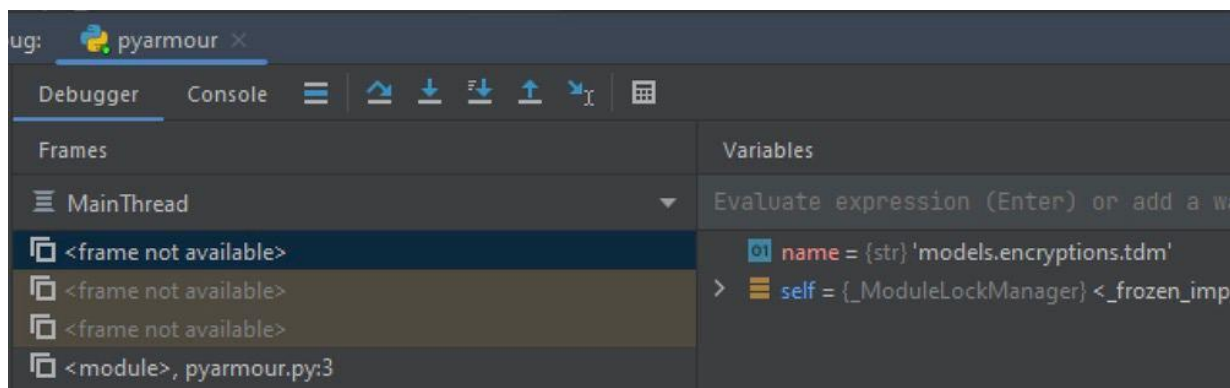

Рисунок 2.18 – Виявлення роботи pyd-файлу із розпакуванням коду

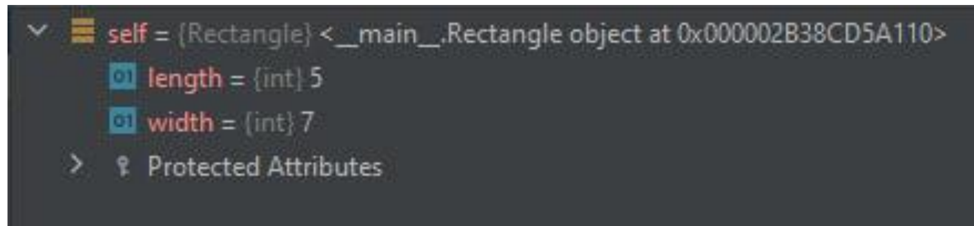

Рисунок 2.19 – Виявлення класу, полів та значень

На основі проведених досліджень із працездатності програми-обфускатора, можна сказати, що застосунок лише обгортає вихідний код, що може бути вразливим при отриманні навіть такого вихідного коду. Адже завдяки засобам відладки, можна переглянути змінні, значення, найменування в оригінальному вигляді. Інакше кажучи, даний інструмент здійснює лише ізоляцію вихідного коду, але не перетворює безпосередньо сам вихідний код.

Висновки до другого розділу

В другому розділі було розглянуто поняття та принципи обфускації й деобфускації, злам програмного забезпечення, захист від зламу, а також розглянуто програму для обфускації вихідного коду мови програмування Python.

Було визначено, що обфускація потрібна для захисту вихідного коду від зловмисників. Зловмисники можуть статично аналізувати вихідний код, щоб отримати необхідну інформацію. Також було визначено процес деобфускації, який може бути потрібен для різних цілей: надання програмному засобу комерційної важливості, ліцензування тощо.

Далі потрібно було ознайомитись із способами зламу програмного забезпечення. Було розглянуто низку програмних засобів, які дозволяють зламувати та аналізувати програми, які можуть бути написані на різних мовах програмування. Попередньо було класифіковано ці програми до підходи або способів зламу програм: декомпіляція, статичний аналіз, динамічний аналіз, дизасемблювання. Додатково було розроблено невеличкий програмний засіб для спроби її зламу – де результат виявився позитивним.

Після тестування способів зламу програмного забезпечення, було розглянуто способи захисту вихідного коду та програм. Реалізація захисту відбувається також за допомогою різних програм, які можуть ускладнювати процес отримання оригінальних даних для зловмисників.

Наприкінці, важливим був огляд та тестування інструмента, який здійснює обфускацію вихідного коду мови програмування Python – PyArmor. Даний інструмент має популярність серед розробників, а також підприємств, адже він дозволяє захистити інтелектуальну власність розробника, зробити ліцензію на використання програмного засобу та інше. Для подальшої розробки механізмів захисту вихідного коду мови програмування Python, також було визначено, як здійснюється захист вихідного коду даним інструментом. Додатково, було детально протестовано застосунок для виявлення алгоритмів його роботи. Результати показали, що вихідний код перетворюється у формат даних Code Object, де потім відбуваються додаткові обробки – що є лише обгорткою основного коду, де при запуску обфускованої програми, інструмент здійснює деобфускацію та починає виконання оригінального вихідного коду.

РОЗДІЛ 3
РОЗРОБКА МЕХАНІЗМІВ ЗАХИСТУ ВИХІДНОГО КОДУ

3.1 Визначення основних компонентів вихідного коду

Захист вихідного коду є важливим аспектом у забезпеченні безпеки програмного забезпечення, особливо коли йдеться про комерційні застосунки, інтелектуальну власність або критичну логіку, доступ до якої потрібно обмежити. Попри наявність готових інструментів, у деяких випадках виникає потреба у створенні власних, персоналізованих механізмів захисту. Це може бути зумовлено специфічними вимогами до безпеки, особливостями програмного продукту чи необхідністю унікальних підходів, які не охоплюються стандартними рішеннями.

Можна визначити наступні напрямки вихідного коду, які повинні підлягати до обробки механізмами. На рисунку 3.1 відображено зв'язки проєкта вихідного коду, які формують кінцевий програмний продукт.

Для створення ефективного обфускатора необхідно враховувати особливості кожного компонента програмного забезпечення, який потребує захисту. У процесі розробки планується застосовувати механізми обфускації до компонентів, які відображені на рисунку 3.1, які, у свою чергу, відображають зв'язки між усіма компонентами програмного забезпечення.

Рисунок 3.1 – Схема зв’язків компонентів програмного забезпечення

1) **Структура проєкта**: Обфускація структури проєкта полягає у приховуванні чи ускладненні логічної організації файлів і модулів, яка може бути досягнуто шляхом перейменування модулів, директорій і файлів на випадкові або неінформативні назви.

2) **Змінні та значення**: Обфускація змінних та їх значень забезпечує приховування логіки і робить вихідний код менш зрозумілим через перейменування змінних на короткі та/ або неінформативні назви, а також деякі значення, які можуть підлягати до такої обробки.

3) **Методи**: Захист методів полягає в ускладненні їх ідентифікації та аналізу логіки виконання, яке може включати перейменування методів на випадкові або беззмістовні назви.

4) **Класи**: Обфускація класів забезпечує захист інформації про призначення, функціональність і логіку класів через перейменування класів на неінформативні назви, які не відображають їх функціонального призначення.

5) **Рядкові типи вихідного коду**: Обфускація рядкових типів спрямована на приховування чутливих або інформативних текстових даних. Для досягнення захисту рядкових типів буде застосовуватись шифрування рядків, які розшифровуються лише в процесі виконання, а також маскування алгоритму дешифрування рядків.

6) **Коментарі та документації**: Захист коментарів та документацій немало важливий, оскільки цей компонент суттєво спрощує аналіз вихідного коду. Дії можуть включати зміни назв цих коментарів.

Таким чином, комплексна обфускація вказаних компонентів значно ускладнює зворотний інжиніринг та аналіз програмного забезпечення, що підвищує загальну безпеку кінцевого програмного продукту.

3.2 Структури проєктів

Структура проєкта при розробці застосунків — це організована схема розташування файлів, папок і модулів, яка визначає логіку, взаємозв’язки та архітектуру застосунку. Вона включає вихідний код, конфігураційні файли, бібліотеки, ресурси, документацію та інші необхідні компоненти.

Ключові елементи структури проєкта [53; 54]:

1) Головний каталог:

- Містить основні файли та директорії проєкта.
- Часто містить файл README.md, документацію та конфігураційні файли.

2) Каталог вихідного коду (src, app, або інша назва):

- Основна логіка застосунку, включаючи модулі, компоненти та бізнес-логіку.
- Може бути розбита на підкаталоги (наприклад, models/, services/, controllers/ у бекенді).

3) Конфігураційні файли:

- Файли для налаштування середовищ (наприклад, .env, config.json, settings.py).
- Можуть включати параметри підключення до бази даних, змінні середовища тощо.

4) Тестові файли:

- Тести для перевірки функціональності застосунку.
- Можуть бути організовані окремим каталогом або розташовані поруч із тестованими модулями.

5) Каталог статичних файлів (static, assets, public):

- Зображення, шрифти, CSS/JS-файли.
- Може містити файли, які не генеруються динамічно.

6) Каталог шаблонів (templates, views):

- HTML-шаблони або файли представлення (наприклад, у Flask/Django).

7) Логи та тимчасові файли (logs, temp):

- Використовуються для запису роботи застосунку або збереження тимчасових даних.

8) Залежності та пакунки (venv):

- Містять сторонні бібліотеки, встановлені за допомогою пакетних менеджерів (pip).

9) Файл керування залежностями (requirements.txt)

- Визначає список необхідних бібліотек і їхні версії.

10) Скрипти для автоматизації (scripts, bin):

- Файли, що автоматизують рутинні процеси (наприклад, запуск сервера, розгортання тощо).

Кінцевий програмний продукт формують файли та шляхи до файлів (каталоги), що є структурою проєкту. Для опрацювання таких даних було розроблено симетричний метод шифрування Pattern Reverse Multiplication (PRM). Даний метод може обробляти як рядкові типи даних, так і байти – що дозволяє захистити файли та програми проєкту [55].

При обробці байтів потрібно отримати усі внутрішні символи цих даних, які закодовано. На таблиці 3.1 відображено, для прикладу, частину кодувальних символів ASCII, а на наступній формулі принцип отримання байтів із закодованих символів (3.1):

$$\sum_{k=1}^{m}(n_{k-1} \times 2^{m-k})_{10} \tag{3.1}$$

Таблиця 3.1

Частина кодувальних символів ASCII

Символ	2-й код	Символ	2-й код
	00100000	a	01100001
!	00100001	b	01100010
"	00100010	c	01100011
#	00100011	d	01100100
$	00100100	e	01100101
%	00100101	f	01100110
&	00100110	g	01100111
‘	00100111	i	01101001
(	00101001	j	01101010
)	00101010	k	01101011
*	00101011	l	01101100
+	00101100	m	01101101
,	00101101	n	01101110
-	00101110	o	01101111
0	00110000	p	01110000
1	00110001	q	01110001
2	00110010	r	01110010
3	00110011	s	01110011
4	00110100	t	01110100
5	00110101	u	01110101
6	00110110	v	01110110

Обробка та підвищення рівня захисту повинна відбуватись завдяки відповідному шаблону – ключу. Ключ може бути створений індивідуально під власне розроблений алгоритм, адже підхід до опрацювання даних дає можливість реалізувати дану задачу [56].

Дані, які отримані, повинні взаємодіяти із ключем. Алгоритм опрацювання може бути будь-яким. В даному методі використовується наступний алгоритм (рис. 3.2):

1) Утворення ключа із довжиною 9 ≤ N.
2) Зворотній запис ключа.
3) Отримання конвертованих даних.
4) Відповідне число помножити на довжину ключа.
5) Результат добутку розділяється на окремі числа.
6) Кожне число береться як індекс із ключа.
7) Формування захищених даних.

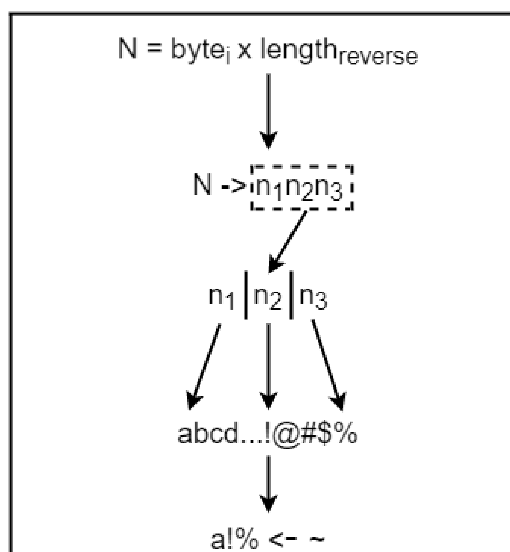

Рисунок 3.2 – Схема обробки даних

В запропонованому методі довжина повинна бути не менше 9. Це через те, що результат розділеного числа, який утворився від добутку, може бути від 0 до 9.

Надійність вихідних даних може залежати від довжини ключа – кількість ітерацій на обробку. При використанні мінімальної довжини можна отримати менш надійний захист й більш швидку обробку.

На рисунках 3.3 відображено результат застосування алгоритму PRM для зміни структури проєкта.

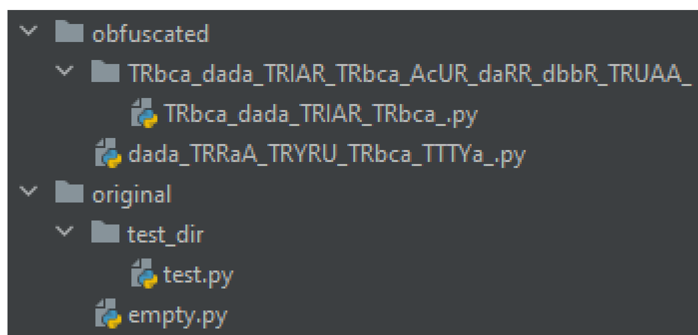

Рисунок 3.3 – Результат обробки структури проєкта за алгоритмом PRM

Окрім зміни структури проєктів, а саме найменувань, розроблений алгоритм може також змінювати структуру файлів на низькому рівні – саме байт-код файлів, які утворюють файлові системи різних операційних систем для якісної обробки цих даних, а також в інших програмних засобах, які зберігають текстовий формат даних. На рисунках 3.4 – 3.5 відображено приклад обробки файлу та текстових даних в базі даних.

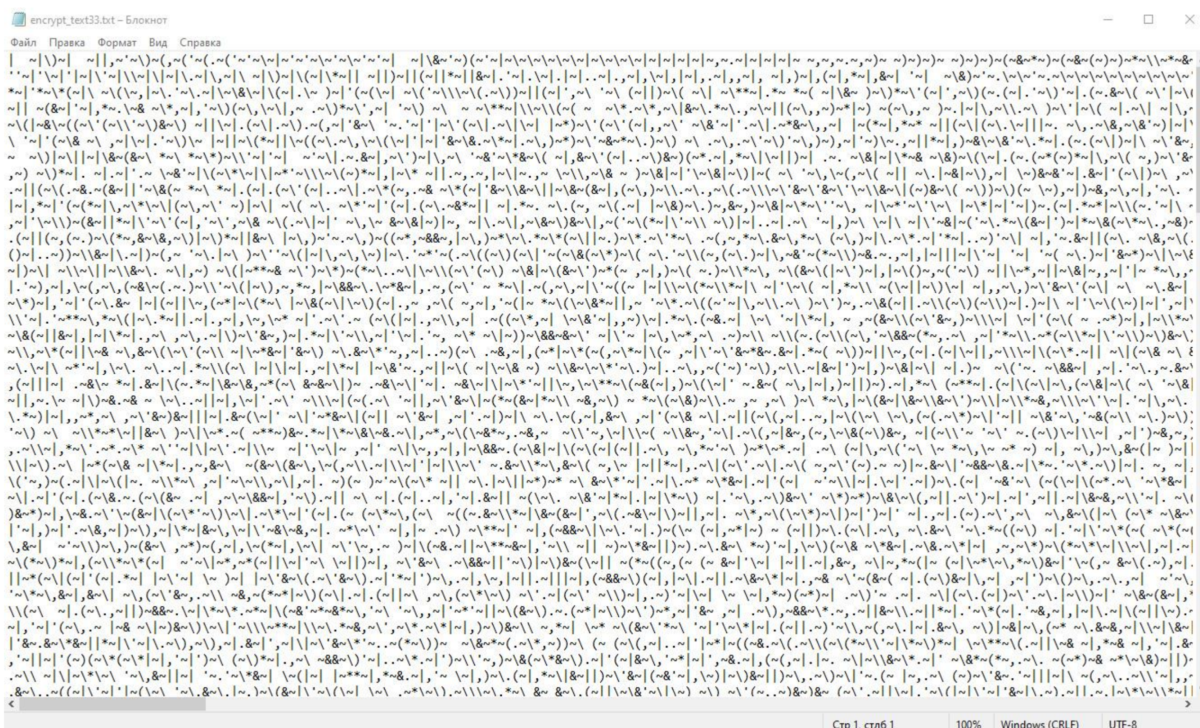

Рисунок 3.4 – Результат обробки файлу алгоритмом PRM

<table>
<tr><th></th><th>id</th><th>content</th><th>user</th></tr>
<tr><td>▶</td><td>1</td><td>)(|^~</td><td>.|*&,~%,%,~.|(^|~.|*&,~^&)|~%,%,~.|.,|...</td></tr>
<tr><td></td><td>2</td><td>^%,)~</td><td>.|*&,~%,%,~.|(^|~.|*&,~^&)|~%,%,~.|.,|...</td></tr>
<tr><td></td><td>3</td><td>)(|^~</td><td>.|*&,~%,%,~.|(^|~.|*&,~^&)|~%,%,~.|.,|...</td></tr>
<tr><td></td><td>4</td><td>^%,)~</td><td>.|*&,~%,%,~.|(^|~.|*&,~^&)|~%,%,~.|.,|...</td></tr>
<tr><td></td><td>5</td><td>)(|^~</td><td>.|*&,~%,%,~.|(^|~.|*&,~^&)|~%,%,~.|.,|...</td></tr>
<tr><td></td><td>6</td><td>^%,)~</td><td>.|*&,~%,%,~.|(^|~.|*&,~^&)|~%,%,~.|.,|...</td></tr>
<tr><td></td><td>7</td><td>)(|^~</td><td>.|*&,~%,%,~.|(^|~.|*&,~^&)|~%,%,~.|.,|...</td></tr>
<tr><td></td><td>8</td><td>)*%,~</td><td>.|*&,~%,%,~.|(^|~.|*&,~^&)|~%,%,~.|.,|...</td></tr>
</table>

Рисунок 3.5 – Результат обробки текстових даних в базі даних алгоритмом PRM

3.3 Ідентифікатори вихідного коду

До ідентифікаторів вихідного коду належать: класи, методи та змінні – це основні складові вихідного коду, які визначають логіку та функціональність застосунку. Вони формують структуру програмного забезпечення, забезпечуючи модульність, повторне використання коду та зручність підтримки.

Ключові елементи класів, методів та змінних вихідного коду [57; 58]:

1) **Класи:**

- Опис: Об'єкти, що інкапсулюють дані та методи для роботи з ними.
- Призначення: Використовуються для моделювання сутностей та організації коду.

2) **Методи:**

- Опис: Функції, які визначені всередині класів та виконують певну логіку.
- Призначення: Використовуються для обробки даних та взаємодії з іншими об'єктами.
- Типи методів:
  - Звичайні методи (self) – працюють із конкретним екземпляром класу.
  - Статичні методи (@staticmethod) – не залежать від стану класу або його екземплярів.
  - Класові методи (@classmethod) – працюють із самим класом, а не його екземпляром.

3) **Змінні:**

- Опис: Іменовані області пам'яті, що зберігають дані.
- Призначення: Використовуються для зберігання стану об'єктів або проміжних значень.
- Типи змінних:
  - Локальні – існують лише в межах певного методу.
  - Екземплярні (self.variable) – прив'язані до конкретного об'єкта.
  - Класові (ClassName.variable) – спільні для всіх екземплярів класу.

На рисунках 3.6 – 3.8 відображено приклади організації вихідного коду за класовим, методом та змінним представленням, що відповідає парадигмі програмування ООП (об'єктно-орієнтованому програмуванню).

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def drive(self):
        print(f"{self.brand} {self.model} is driving.")
```

Рисунок 3.6 – Класове представлення вихідного коду за принципом ООП

```
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b
```

Рисунок 3.7 – Представлення у вигляді методів вихідного коду за принципом ООП

```
class Example:
    class_variable = "I belong to the class"

    def __init__(self, value):
        self.instance_variable = value
```

Рисунок 3.8 – Змінне представлення вихідного коду за принципом ООП

Обробка вказаних компонентів (класів, методів, змін) відбувається за наступним алгоритмом:

1) Пошук усіх найменувань класів, методів та змінних.

2) Створення ключа для обробки назв, через об’єднання усіх найменувань класів, методів та змінних із прибиранням повторних символів.

3) Створюється зворотній запис ключа окремо для класів, методів та змінних.

4) Ітераційний пошук індексів кожних символів із найменувань в ключах.

5) Підстановка символів за індексами реверсивного ключа.

Зумовленість використання спільного алгоритму обробки цих компонентів пов’язано із спільним принципом утворення коду – саме назв. Таким чином, можна визначити, що складність формуючих назв залежить від кількості та унікальності тексту. На рисунках 3.9 – 3.14 відображено результат обробки класів, методів та змінних за вказаним алгоритмом на тестовому вихідному коді.

```
class Test:
    pass


class Rectangle:
    pass


class Circle:
    pass
```

Рисунок 3.9 – Тестовий код для обробки класів

```
class riCl:
    pass


class ginlacRti:
    pass


class seTnti:
    pass
```

Рисунок 3.10 – Результат обробки класів

```
def rectangle():
    pass


def circle(x1, y1):
    pass


class Test:
    def __init__(self):
        pass

    def test_case(self, input_data):
        pass


rectangle()

print()
```

Рисунок 3.11 – Тестовий код для обробки методів

```
def silgnatci():
    pass


def leslci(a1, d1):
    pass


class tseT:
    def __init__(self):
        pass

    def girg_lnri(self, tupni_yxix):
        pass


silgnatci()

print()
```

Рисунок 3.12 – Результат обробки методів, а також додатково класів

```
number = 1

array = [1, 2, 3]


def test(data):
    print(data)


class Test:
    def test_case(self, test_data):
        return test_data
```

Рисунок 3.13 – Тестовий код для обробки змінних

```
stdyar = 1

erreb = [1, 2, 3]


def acsa(meue):
    print(meue)


class tseT:
    def acsa_etsc(self, uanu_meue):
        return uanu_meue
```

Рисунок 3.14 – Результат обробки змінних, а також додатково класів та методів

3.4 Рядкові типи вихідного коду

Рядкові типи у вихідному коді – це текстові дані, що використовуються для зберігання, обробки та виводу інформації в програмі. Вони можуть містити символи, слова, речення або навіть спеціальні формати даних. Даний тип може містити критичну інформацію: API-дані (Application Programming Interface; спосіб інтеграції), логіни та паролі, інформація про ресурси та інше.

Ключові елементи рядкових типів вихідного коду [59]:

1) **Рядки**:

- Опис: Набір символів, укладених у лапки («'...'» або «"..."»).
- Призначення: Використовуються для збереження текстових даних.

2) **Форматування рядків**:

- Опис: Методи вставки змінних або динамічних значень у рядки.

3) **Мультистрокові рядки**:

- Опис: Рядки, що містять кілька рядків тексту.
- Використання: Часто застосовуються для збереження великих текстів, документування коду.

На рисунках 3.15 – 3.17 відображено приклади створення такого компоненту.

```python
message = "Hello, world!"
print(message)
```

Рисунок 3.15 – Створення звичайних рядкових типів

```python
name = "Alice"
greeting = "Hello, " + name + "!"

greeting2 = "Hello, {}!".format(name)

greeting3 = f"Hello, {name}!"
```

Рисунок 3.16 – Створення рядкових типів із їх форматуванням

```python
multiline_text = """This is
a multiline
string."""
```

Рисунок 3.17 – Створення мультистрокових рядків

Алгоритм опрацювання даного компоненту відбуватиметься за розробленим алгоритмом Transposition Data Method (TDM) [60]. Алгоритм методу наступний:

1. Отримання вхідних даних та, при необхідності, конвертування їх у байти.
2. Розподілення даних на 6 частин із їх опрацюванням з кроком 12:

a) Частина 1 та 2 із початковою позицією обробки 0 здійснюють заміну значень N та N + 1.

b) Частина 3 та 4 із початковою позицією обробки 2 здійснюють заміну значень від N до N + 2 та від N + 2 до N + 4.

c) Частина 5 та 6 із початковою позицією обробки 6 здійснюють заміну значень від N до N + 3 та від N + 3 до N + 6.

3. Заміна місцями першої половини вихідних значень із іншою.

4. Збереження результату.

На рисунку 3.18 відображено приклад застосування метода TDM.

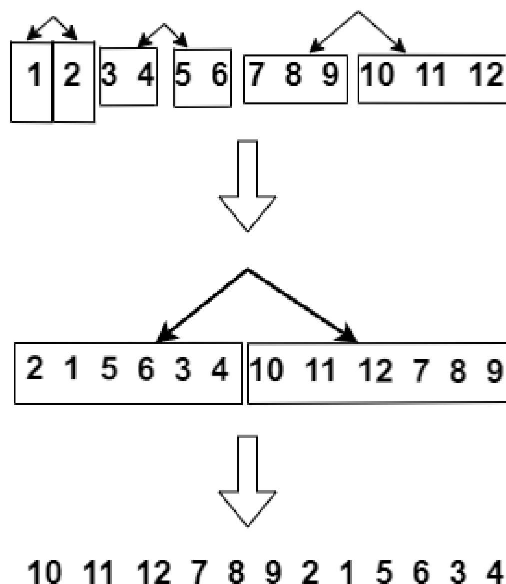

Рисунок 3.18 – Транспонування даних за алгоритмом

Для розшифрування слід враховувати, що значення довжини вхідних зашифрованих даних може бути не парне, тому необхідно на початку розшифрування визначити відсоток від ділення довжини на 2 для подальшої коректності результату – перестановки половин. Далі розшифрування в зворотному порядку.

При використанні даного підходу можуть виникнути складнощі із обробкою великих даних. Як приклад, 1 ГБ може оброблятись відносно довго. У такому випадку слід розділити вхідні дані на блоки, які опрацьовуватимуть частини по такому же алгоритму [61]. Задля отримання більшої унікальності розміщених даних, необхідно наприкінці обробки здійснити перестановку декількох частин, щонайбільше трьох, для великих об’ємів (рис. 3.19).

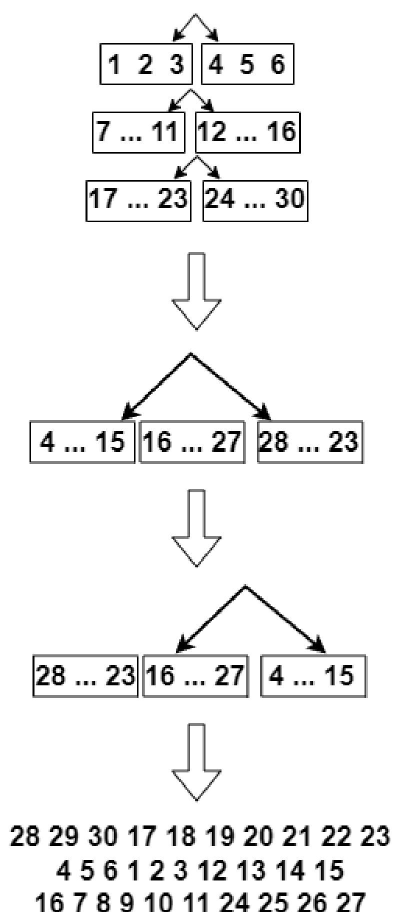

Рисунок 3.19 – Оптимізація для великих об'ємів даних

Даний алгоритм впроваджується як заміна рядкового значення. Інакше кажучи, отримання рядкових даних відбувається через виклик метода, який, у свою чергу, перетворює зашифровані текстові дані у дійсний їх вигляд. Зловмисник може проаналізувати функції дешифрування, тому слід додатково обробити виклик метода. Було прийнято рішення застосувати наступні вбудовані функції:

1) «exec»: Дана функція отримує на вхід рядок, який має відповідати звичайному синтаксичному запису мови програмування Python. Приклад використання даної функції: «exec('var = 1')». Таким чином буде створена звичайна змінна із вказаним значенням. Відповідно, так можна замаскувати метод дешифрування даних текстового типу.

2) «base64»: Цей модуль надає функції кодування двійкових даних у друковані символи ASCII і декодування таких кодувань назад у двійкові дані. В даному модулі застосовуються функції кодування та декодування даних. Це потрібно

для ускладнення процесу відладки (debugging) програмного забезпечення та статичного аналізу.

На рисунках 3.20 – 3.21 відображено результат обробки рядкового типу даних на тестовому вихідному коді.

```
string = 'E-mail'

f_string = f'Your e-mail: {string}'
```

Рисунок 3.20 – Тестовий код для обробки рядкових типів

```
exec(base64.b64decode(TDM().decrypt("0JbWUtCc=FpbmZaXduD0gJ0I").encode()).decode())

exec(base64.b64decode(TDM().decrypt("SZYW1tiB7lsOmZaXduSc=J0f1cZ29mnQg5pcSPJ1BmXIglvd").encode()).decode())
```

Рисунок 3.21 – Результат обробки рядкових типів

Окрім обробки рядкових типів даних, розроблений алгоритм може працювати із іншими типами даних, наприклад, для захисту даних, які передаються у мережі [62]. На рисунках 3.22 – 3.23 відображено приклад обробки даних, які передаються по мережі, саме від зовнішнього джерела (178.133.136.43; публічна IP-адреса) до внутрішнього (192.168.1.112; локальна IP-адреса).

```
Protocol: TCP 178.133.136.43 -> 192.168.1.112
Data: b'hello, world'
Protocol: TCP 192.168.1.112 -> 178.133.136.43
Data: b'hello, world'
```

Рисунок 3.22 – Перегляд останніх даних пакетів

```
PS D:\Projects\Linked System\Package encryption> python .\sniff_decoder.py
Protocol: TCP 178.133.136.43 -> 192.168.1.112
Data: b'rld woeho,ll'
Protocol: TCP 192.168.1.112 -> 178.133.136.43
Data: b'rld woeho,ll'
```

Рисунок 3.23 – Перегляд останніх даних пакетів, у яких застосовано метод TDM

3.5 Коментарі та документації вихідного коду

Коментарі та документація – це текстові пояснення в коді, які не виконуються програмою, але допомагають розробникам розуміти логіку та структуру застосунку. Вони покращують читабельність коду, полегшують його підтримку та сприяють ефективній командній роботі.

Ключові елементи коментарів та документацій вихідного коду [63; 64]:

1) **Коментарі:**

- Опис: Текстові пояснення в коді, які ігноруються інтерпретатором або компілятором.
- Призначення: Використовуються для пояснення складних ділянок коду, тимчасового вимкнення рядків або позначення завдань.
- Типи коментарів:
  - Однорядкові («#»).
  - Багаторядкові («""" """» або «''' '''»).

2) **Документування коду:**

- Опис: Спеціальний тип багаторядкових коментарів, що використовується для документування функцій, класів та модулів.
- Призначення: Допомагає іншим розробникам швидко зрозуміти призначення та використання коду.

На рисунках 3.24 – 3.25 відображено приклад коментарів та документацій вихідного коду.

```
# Це однорядковий коментар
print("Hello, world!")  # Вивід тексту на екран

"""
Це багаторядковий коментар.
Він використовується для пояснення великих фрагментів коду.
"""
```

Рисунок 3.24 – Створення коментарів вихідного коду

```python
def add(a, b):
    """
    Функція додає два числа.

    Параметри:
    a (int, float): Перше число.
    b (int, float): Друге число.

    Повертає:
    int, float: Сума чисел.
    """
    return a + b


class Car2:
    """
    Клас для представлення автомобіля.

    Атрибути:
    brand (str): Марка автомобіля.
    model (str): Модель автомобіля.
    """
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
```

Рисунок 3.25 – Створення документації вихідного коду

Так, як коментарі та документації схожі за своїм принципом створення та призначенням, тому було вирішено розробити наступний спільний алгоритм обробки для цих компонентів:

1) Пошук усіх коментарів та документацій.

2) Розділення усіх символів знайденого рядку, який містить коментар або документацію частини коду, із прибиранням повторних символів.

3) Пошук кількості входжень кожного символу в рядку.

4) Підстановка знайдених відсортованих індексів (за вбудованим алгоритмом сортування мови програмування Python) з рядка для кожного символу:

- Спочатку підставляється кількість пробілів у рядку, а саме індекси пробілів.

- Потім підставляються інші знайдені символи, а саме знайдений символ та його позиція.
- Якщо в рядку є декілька повторних входжень одного й того самого символу, підставляється відповідний знак («&&»; символ, який свідчить про повторення входжень).
- Після кожної обробки символів, підставляється роздільний знак («__»; два нижніх підкреслення).

Підстановка додаткових символів, таких як символів для визначення повторів та розділення, потрібні у майбутньому для якісної деобфускації вихідного коду. На рисунках 3.26 – 3.29 відображено результат обробки коментарів та документацій на тестовому вихідному коді.

```
# Some text to test

base_num = 1  # Base variable
```

Рисунок 3.26 – Тестовий код для обробки коментарів

```
#  4&&9&&12__S0__e3&&6&&14__m2__o1&&11__s15__t5&&8&&10&&13&&16__x7

mune_sab = 1  #  4__B0__a1&&6&&9__b10__e3&&12__i8__l11__r7__s2__v5
```

Рисунок 3.27 – Результат обробки коментарів

```
""" Test documentation """


class Test:
    """ Class description """
    pass
```

Рисунок 3.28 – Тестовий код для обробки документації

```
""" 0&&5&&19__T1__a14__c8__d6__e2&&11__i16__m10__n12&&18__o7&&17__s3__t4&&13&&15__u9"""


class tseT:
    """ 0&&6&&18__C1__a3__c10__d7__e8__i12&&15__l2__n17__o16__p13__r11__s4&&5&&9__t14"""
    pass
```

Рисунок 3.29 – Результат обробки документацій

Висновки до третього розділу

В третьому розділі було описано розроблені механізми обфускації вихідного коду. Для розробки таких механізмів слід було виділити компоненти, які підлягають до обробки: структура проєкта, змінні, методи, класи, рядкові типи, коментарі та документації.

Структура проєкта складається із файлів та папок, які формують кінцевий програмний продукт. Обробка цих даних відбувається за алгоритмом Pattern Reverse Multiplication, який працює за допомогою ключа для зміни даних, що відповідає симетричним ознакам методів шифрування даних. Додатково, розроблений алгоритм може опрацьовувати й інші дані, наприклад, файли операційної системи, текстові дані тощо.

Змінні, методи та класи обробляються за спільним алгоритмом: відбір найменувань, їх об’єднання без повторних символів (ключ), зворотній запис ключа, підстановка нових символів за допомогою поточних індексів та реверсивного ключа.

Рядкові типи також можна класифікувати як важливий компонент програмного забезпечення, адже такі дані можуть містити різну інформацію. Для цього компоненту застосовуються вбудовані компоненти обробки вихідного коду мови програмування Python, які мають свої особливості у розробці додатків, що призводять до ускладнення процесу зворотного інжинірингу програмних додатків. Також розроблений алгоритм TDM дозволяє працювати з іншими даними, наприклад, текстовий формат даних, мережеві дані, файли операційної системи тощо.

Коментарі та документації вихідного коду дозволяють швидко зрозуміти принцип роботи програмного забезпечення без необхідності його тестування. Для цього компоненту застосовується алгоритм збору кількості входжень та підстановки випадковим чином значень із вказівкою повторних символів у відповідних місцях.

Таким чином, розроблені механізми обфускації забезпечують багаторівневий підхід до захисту вихідного коду, охоплюючи як структурні, так і змістовні компоненти програмного продукту. Запропоновані алгоритми не лише ускладнюють процес зворотного інжинірингу, але й можуть адаптуватися до потреб захисту інших типів даних, що свідчить про їхню гнучкість і потенціал до подальшого вдосконалення та практичного застосування в різних сферах розробки програмного забезпечення.

РОЗДІЛ 4
ВИЗНАЧЕННЯ ЕФЕКТИВНОСТІ РОЗРОБЛЕНИХ МЕХАНІЗМІВ ЗАХИСТУ ВИХІДНОГО КОДУ

4.1 Формалізація оцінювання ефективності механізмів захисту

Визначення критеріїв є ключовим етапом при розробці новітніх засобів, механізмів тощо. Ці критерії допомагають оцінити, наскільки успішно нові засоби можуть впроваджуватись у відповідні процеси.

Критерій – це певний набір параметрів або характеристик, за якими оцінюється конкретний аспект [65]. Використання критеріїв дозволяє структуровано та систематично підходити до оцінки та аналізу даних.

Для отримання точних оцінок, потрібно визначити кількісний та якісний метод оцінювання. Одним із популярних методів оцінювання є шкала Лікерта, яка, зазвичай, містить кілька рівнів – наприклад, від 1 до 5 [66]. Ця шкала дозволяє респондентам оцінювати своє ставлення чи згоду з певними твердженнями, що допомагає отримати структуровану інформацію. Використання шкали Лікерта також спрощує процес аналізу, адже кожен рівень шкали може відповідати певній інтерпретації, що дозволяє узагальнювати і систематизувати результати дослідження.

Завдяки введенню такої шкали оцінювання можна отримати чіткі уявлення про загальні тенденції в поглядах респондентів, що значно полегшує інтерпретацію та аналіз даних. Основні причини введення шкали оцінювання [67]:

- Кількісне вимірювання суб'єктивних даних: Дозволяє перетворити якісні думки та ставлення респондентів у кількісні дані, які можна аналізувати статистично.
- Стандартизація відповідей: Забезпечує єдиний формат відповідей для всіх респондентів, що полегшує порівняння та аналіз.
- Глибокий аналіз ставлень: Дозволяє вимірювати інтенсивність або ступінь згоди/незгоди з певними твердженнями.

- Спрощення обробки даних: Кількісні дані легше обробляти за допомогою статистичних методів.

Введення рівнів (низький, середній, високий) полягає в групуванні відповідей для спрощення аналізу та інтерпретації даних:

- Спрощення даних: Зменшує кількість категорій, що полегшує аналіз, особливо при великих обсягах даних.
- Покращення статистичної надійності: Об'єднання категорій може збільшити очікувані частоти у таблицях спряженості, що є важливим для коректного застосування певних методів математичної статистики.
- Полегшення інтерпретації результатів: Легше зрозуміти та пояснити результати, коли вони представлені в узагальненій формі (наприклад, високий рівень згоди).
- Виявлення загальних тенденцій: Допомагає виявити основні патерни у відповідях респондентів.

Визначити рівні на основі шкали оцінювання можна наступним чином:

- Низький рівень: Оцінки 1 і 2.
- Середній рівень: Оцінка 3.
- Достатній рівень: Оцінки 4 і 5.

Запропоновано наступні критерії, які дозволять визначити ефективність розроблених алгоритмів для компонентів вихідного коду: структура проєктів, ідентифікатори, рядкові типи, коментарі та документації:

1) **Ентропія** – це кількісна міра випадковості або непередбачуваності символів у назві. Вона показує, наскільки "хаотичною" є назва з точки зору розподілу символів [68]. Вища ентропія означає, що назва менш читається людиною і менш ймовірно, що вона буде вгадана або реверсивно отримана. Це ключовий показник стійкості обфускації до статичного аналізу. Для даного критерію діапазони визначення оцінок наступні:

- $N \leq 2.0$ – оцінка 1 (низький рівень).
- $N \leq 2.5$ – оцінка 2 (низький рівень).

- $N \leq 3.0$ – оцінка 3 (середній рівень).
- $N \leq 3.5$ – оцінка 4 (високий рівень).
- $N > 3.5$ – оцінка 5 (високий рівень).

Даний критерій буде обраховуватись за формулою Шеннона (4.2) [69]:

$$H = -\sum_{i=1}^{n} p_i * log_2(p_i), \tag{4.2}$$

де p_i – ймовірність i-символу в рядку.

2) **Довжина назв** – середня кількість символів у назвах файлів та папок. Більша довжина імен зазвичай ускладнює читання, аналіз і зворотне інтуїтивне відновлення. Надто короткі обфусковані імена можуть бути вразливими до брутфорсу або евристики [70; 71]. Також це показник імітації складності. Для даного критерію діапазони визначення оцінок наступні:

- $N \leq 10$ – оцінка 1 (низький рівень).
- $N \leq 15$ – оцінка 2 (низький рівень).
- $N \leq 20$ – оцінка 3 (середній рівень).
- $N \leq 25$ – оцінка 4 (високий рівень).
- $N > 25$ – оцінка 5 (високий рівень).

Даний критерій буде обраховуватись за наступною формулою (4.3) [72]:

$$L = \frac{1}{n}\sum_{i=1}^{n} |b_i|, \tag{4.3}$$

де b_i – довжина i-назви,

n – кількість вхідних даних.

3) **Шаблонність** – наявність у назвах повторюваних структур, префіксів, суфіксів або відомих слів, які знижують унікальність імені. Висока шаблонність свідчить про вразливість до зворотної інженерії, оскільки дозволяє зловмиснику виявити патерн і масово відновити логіку назв. Ідеальні обфусковані імена мають бути псевдовипадковими [73]. Для даного критерію діапазони визначення оцінок наступні:

- $N \geq 0.8$ – оцінка 1 (низький рівень).

- N ≥ 0.6 – оцінка 2 (низький рівень).
- N ≥ 0.4 – оцінка 3 (середній рівень).
- N ≥ 0.2 – оцінка 4 (високий рівень).
- N < 0.2 – оцінка 5 (високий рівень).

Даний критерій буде обраховуватись за наступними формулами (4.4 – 4.9) [74]:

$$S = \{s_1, s_2, \dots, s_i\}, \tag{4.4}$$

де *S* – множина обфускованих рядків,

s_i – обфускований рядок.

$$Sub(s_i), \tag{4.5}$$

де *Sub* – множина усіх підрядків довжини ≥ *k* у рядку s_i.

$$F = \bigcup_{i=1}^{n} Sub(s_i), \tag{4.6}$$

де *F* – множина усіх унікальних підрядків.

$$C(f), \tag{4.7}$$

де *C* – кількість рядків s_i, у яких зустрічається фрагмент *f*.

$$G(s_i) = \begin{cases} 1, \text{якщо } \exists f \ \in C(f) \geq 2 \\ 0, \text{інакше} \end{cases}, \tag{4.8}$$

де *G(*s_i*)* – обрахунок s_i-рядку.

$$T = \frac{1}{n}\sum_{i=1}^{n} G(s_i), \tag{4.9}$$

де *T* – обрахунок для множини рядків,

n – кількість входжень.

4) **Відмінність від оригіналу** – ступінь схожості (або відмінності) між обфускованою назвою та її оригіналом [75]. Найчастіше вимірюється як 1. Чим менша схожість, тим важче зрозуміти, до якого файлу відноситься назва. Це основний показник ефективності іменної трансформації. Для даного критерію діапазони визначення оцінок наступні:

- N ≤ 0.3 – оцінка 1 (низький рівень).
- N ≤ 0.5 – оцінка 2 (низький рівень).

- N ≤ 0.7 – оцінка 3 (середній рівень).
- N ≤ 0.85 – оцінка 4 (високий рівень).
- N > 0.85 – оцінка 5 (високий рівень).

Даний критерій буде обраховуватись за формулою, яка альтернативна формулі Левенштейна, а саме найдовшої спільної послідовності (4.10) [76; 77]:

$$R = \frac{2 * M}{len(a) + len(b)}, \tag{4.10}$$

де M – кількість символів у спільних сегментах.

5) **Консистентність** – всі імена у структурі (файлів, папок) були змінені в процесі обфускації, і жодне не залишилося у відкритому вигляді [78]. Навіть одне не обфусковане ім'я може розкрити логіку всієї структури або бути вхідною точкою для реверс-інжинірингу. Це показник повноти і послідовності обфускації. Для даного критерію діапазони визначення оцінок наступні:

- N ≤ 0.6 – оцінка 1 (низький рівень).
- N ≤ 0.75 – оцінка 2 (низький рівень).
- N ≤ 0.85 – оцінка 3 (середній рівень).
- N ≤ 0.95 – оцінка 4 (високий рівень).
- N > 0.95 – оцінка 5 (високий рівень).

Даний критерій буде обраховуватись за наступними формулами (4.11 – 4.13) [79]:

$$S = \{(o_1, b_1), (o_2, b_2), \dots, (o_i, b_i)\}, \tag{4.11}$$

де o_i та b_i – оригінальні та обфусковані значення.

$$\delta_i = \begin{cases} 1, \text{якщо } o_n \neq b_n \\ 0, \text{якщо } o_n = b_n \end{cases}, \tag{4.12}$$

де δ_i – результат обрахунків для i-входження.

$$C = \frac{1}{n}\sum_{i=1}^{n} \delta_i, \tag{4.13}$$

де C – множина обфускованих значень,

n – загальна кількість входжень.

Визначення ефективності розроблених механізмів відбуватиметься на тестовому розробленому вихідному коді, який також був оброблений кожним механізмом (див. рис. 2.1 – 2.2) [80].

4.2 Оцінка ефективності зміни структури проєкту

Для оцінки механізмів обробки структури проєктів, відібрано 3 вхідних значення, які є назвами каталогів та файлів, а також обфускований варіант (табл. 4.1).

Таблиця 4.1

Визначення вхідних та вихідних даних структури проєкту

№	Вхідні дані	Вихідні дані
1	test_dir	TRbca_dada_TRIAR_TRbca_AcUR_daRR_dbbR_TRUAA_
2	test.py	TRbca_dada_TRIAR_TRbca_.py
3	empty.py	dada_TRRaA_TRYRU_TRbca_TTTYa_.py

Далі потрібно обрахувати значення за визначеними критеріями. На таблиці 4.2 відображено результат обрахунків за усіма критеріями, а також визначення оцінки за відповідним діапазоном.

Таблиця 4.2

Результат обрахунків даних за критеріями та визначення оцінок

№	Критерій	Значення	Оцінка
1	Ентропія	3.26	4
2	Довжина	32.0	5
3	Шаблонність	1.0	1
4	Відмінність	0.86	5
5	Консистентність	1.0	5

4.3 Оцінка обфускації ідентифікаторів у вихідному коді

Для оцінки механізмів обробки ідентифікаторів вихідного коду, відібрано декілька класів, методів та змінних, де усього 9 найменувань (табл. 4.3 – 4.4).

Таблиця 4.3

Визначення вхідних та вихідних даних ідентифікаторів вихідного коду

№	**Вхідні дані**	**Вихідні дані**
1	Shape	lgntc
2	calculate_area	mipmrpiet_iuti
3	calculate_perimeter	mipmrpiet_ltuachetu
4	Rectangle	RcepnahSc
5	length	Acrdiw
6	width	htgiw
7	l	a
8	w	h
9	rectangle	nceilrdac

Таблиця 4.4

Результат обрахунків даних за критеріями та визначення оцінок

№	**Критерій**	**Значення**	**Оцінка**
1	Ентропія	2.14	2
2	Довжина	7.67	1
3	Шаблонність	0.0	5
4	Відмінність	0.88	5
5	Консистентність	1.0	5

4.4 Оцінка обробка рядкових типів вихідному коді

Для оцінки механізмів зміни рядкових типів, відібрано 3 вхідних значення, який містить більшість шляхів роботи із цим типом у мові програмування Python (табл. 4.5 – 4.6).

Таблиця 4.5

Визначення вхідних та вихідних даних рядкових типів вихідного коду

№	Вхідні дані	Вихідні дані
1	string1 = 'E-mail'	exec(base64.b64decode(TDM().decrypt("0JbWUtCc=FpbmZaXduD0gJ0I").encode()).decode())
2	string2 = f'Your e-mail: {string}'	exec(base64.b64decode(TDM().decrypt("SZYW1tiB7lsOmZaXduSc=J0f1cZ29mnQg5pcSPJ1BmXIglvd").encode()).decode())
3	print("Area:", rectangle.calculate_area())	exec(base64.b64decode(TDM().decrypt("xym1pFjLGccG1yF9plldXdpRCKpkHcbnJpkFyQoIWZIiE6mNlwgbWaZG").encode()).decode())

Таблиця 4.6

Результат обрахунків даних за критеріями та визначення оцінок

№	Критерій	Значення	Оцінка
1	Ентропія	4.97	4
2	Довжина	101.67	5
3	Шаблонність	1.0	1
4	Відмінність	0.93	5
5	Консистентність	1.0	5

4.5 Оцінка захисту коментарів та документацій вихідного коду

Для оцінки механізмів обробки коментарів та документацій, відібрано 6 вхідних значень, серед яких є як коментарі, так і документації, які складаються із різною довжиною, а також символами (табл. 4.7 – 4.8).

Таблиця 4.7

Визначення вхідних та вихідних даних рядкових типів вихідного коду

№	Вхідні дані	Вихідні дані
1	Base area	4__B0__a1&&5&&8__e3&&7__r6__s2
2	Base perimeter	4__B0__a1__e3&&6&&10&&12__i8__m9__p5__r7&&13__s2__t11
3	Initialize the Rectangle object with given length and width	10&&14&&24&&31&&36&&42&&49&&53__I0__R15__a5&&19&&50__b26__c17&&29__d52&&56__e9&&13&&16&&23&&28&&40&&44__g21&&37&&46__h12&&35&&48&&58__i2&&4&&7&&33&&38&&55__j27__l6&&22&&43__n1&&20&&41&&45&&51__o25__t3&&11&&18&&30&&34&&47&&57__v39__w32&&54__z8
4	Calculate and return the area of the rectangle using the formula: length * width	9&&13&&20&&24&&29&&32&&36&&46&&52&&56&&65&&72&&74__*73__:64__C0__a1&&6&&10&&25&&28&&41&&63__c3&&39__d12&&77__e8&&15&&23&&27&&35&&38&&45&&55&&67__f31&&57__g43&&51&&69__h22&&34&&54&&71&&79__i49&&76__l2&&5&&44&&62&&66__m60__n11&&19&&42&&50&&68__o30&&58__r14&&18&&26&&37&&59__s48__t7&&16&&21&&33&&40&&53&&70&&78__u4&&17&&47&&61__w75

продовження таблиці 4.7

5	Calculate and return the perimeter of the rectangle using the formula: 2 * (length + width)	9&&13&&20&&24&&34&&37&&41&&51&&57&&61&&70&&72&&74&&82&&84__(75__)90__*73__+83__271__:69__C0__a1&&6&&10&&46&&68__c3&&44__d12&&87__e8&&15&&23&&26&&30&&32&&40&&43&&50&&60&&77__f36&&62__g48&&56&&79__h22&&39&&59&&81&&89__i28&&54&&86__l2&&5&&49&&67&&76__m29&&65__n11&&19&&47&&55&&78__o35&&63__p25__r14&&18&&27&&33&&42&&64__s53__t7&&16&&21&&31&&38&&45&&58&&80&&88__u4&&17&&52&&66__w85
6	Example usage	7__E0__a2&&10__e6&&12__g11__l5__m3__p4__s9__u8__x1

Таблиця 4.8

Результат обрахунків даних за критеріями та визначення оцінок

№	**Критерій**	**Значення**	**Оцінка**
1	Ентропія	3.54	5
2	Довжина	184.33	5
3	Шаблонність	1.0	1
4	Відмінність	0.86	5
5	Консистентність	1.0	1

Таким чином, було отримано усі дані, які потрібні для подальшого його аналізу, саме визначення оцінок розроблених механізмів.

4.6 Аналіз отриманих результатів

Для аналізу усіх обрахованих результатів, було відведено кожен результат у підсумкову таблицю (табл. 4.9).

Таблиця 4.9

Підсумкова таблиця оцінок кожного критерію кожного компоненту

Назва	Критерій	Значення	Оцінка	Рівень
Структура проєктів	Ентропія	3.26	4	Високий
	Довжина	34.0	5	Високий
	Шаблонність	1.0	1	Низький
	Відмінність	0.86	5	Високий
	Консистентність	1.0	5	Високий
Ідентифікатори	Ентропія	2.14	2	Низький
	Довжина	7.67	1	Низький
	Шаблонність	0.0	5	Високий
	Відмінність	0.88	5	Високий
	Консистентність	1.0	5	Високий
Рядкові типи	Ентропія	4.97	4	Високий
	Довжина	101.67	5	Високий
	Шаблонність	1.0	1	Низький
	Відмінність	0.93	5	Високий
	Консистентність	1.0	5	Високий
Коментарі та документації	Ентропія	3.54	5	Високий
	Довжина	184.33	5	Високий
	Шаблонність	1.0	1	Низький
	Відмінність	0.86	5	Високий
	Консистентність	1.0	1	Низький

Можна помітити, що отримано рівні для значень низький та високий. Потрібно проаналізувати кожен отриманий результат для кожного компоненту.

1) Структура проєктів.

Середнє значення отриманих результатів відображено на наступній формулі (4.14):

$$\frac{4+5+1+5+5}{5}=4 \quad (4.14)$$

Середнє значення ефективності для компонента структури проєкту становить 4.0, що відповідає високому рівню захисту згідно зі шкалою Лікерта. Це свідчить про успішну реалізацію механізму перейменування файлів і папок, який порушує оригінальну зрозумілу структуру директорій. Усі ключові критерії, окрім одного, отримали високі оцінки, що свідчить про складність відновлення структури без знання внутрішніх правил перетворення.

Водночас найгірший результат зафіксовано за критерієм шаблонності, який отримав 1 бал (низький рівень). Це означає, що в обфускованих іменах присутні повторювані фрагменти, характерні суфікси або структурні закономірності, що можуть бути виявлені шляхом порівняння кількох імен. Подібні шаблони полегшують зловмиснику побудову гіпотез щодо принципу генерації назв, що потенційно спрощує реверсивний аналіз структури проєкту.

Для покращення механізму обфускації структури доцільно реалізувати алгоритм підстановки елементів на основі псевдовипадкових значень, отриманих із ключа або згенерованих на етапі компіляції. Замість фіксованих фрагментів і шаблонних суфіксів рекомендується використовувати динамічну комбінацію символів із різних алфавітів, включаючи спеціальні знаки, цифри та регістрові зміни. Також доцільно вставляти службові або випадкові символи, які не мають відношення до оригінального імені, що дозволить одночасно підвищити ентропію та зменшити схожість між іменами [77].

2) Ідентифікатори.

Середнє значення отриманих результатів відображено на наступній формулі (4.15):

$$\frac{2+1+5+5+5}{5}=3.6 \quad (4.15)$$

Середнє значення ефективності для обфускації ідентифікаторів становить 3.6, що відповідає середньому рівню за шкалою Лікерта. Найнижчі бали були зафіксовані за критеріями ентропії (2 бали) та довжини (1 бал), що свідчить про низьку хаотичність та відносну лаконічність обфускованих імен.

Низька ентропія потенційно створює передумови для формування гіпотези про можливість відновлення оригінальних імен, оскільки низький рівень випадковості символів часто асоціюється з наявністю структурованих шаблонів. Втім, слід зазначити, що навіть за умов низької ентропії, висока візуальна та лексична нечитабельність імен виступає додатковим фактором захисту. Обфусковані ідентифікатори в аналізованому наборі даних не мають очевидного лексичного змісту, не демонструють відповідності з оригінальними назвами, і не дозволяють ідентифікувати їх функціональне призначення без контексту. Таким чином, відсутність семантики компенсує брак хаотичності, утворюючи ефективний бар'єр проти зворотного аналізу.

Окрім ентропії, на захисну якість обфускації суттєво впливає довжина ідентифікаторів. У наведеному прикладі середня довжина виявилася низькою, що знижує загальну складність обробки таких імен як людиною, так і засобами автоматичного аналізу.

Для покращення роботи механізму на рівні ідентифікаторів доцільно передбачити включення до складу обфускованих імен додаткових символів, не пов'язаних зі змістом оригінального імені (наприклад, випадкових префіксів, суфіксів, або символів із різних алфавітів). Це дозволить одночасно підвищити як ентропію, так і довжину імен, що сприятиме зміцненню загального рівня захисту [78].

3) Рядкові типи.

Середнє значення отриманих результатів відображено на наступній формулі (4.16):

$$\frac{4+5+1+5+5}{5} = 4 \tag{4.16}$$

Середнє значення ефективності для компонента рядкових типів становить 4.0, що відповідає високому рівню захисту. Це вказує на те, що більшість обфускованих

рядків складно відновити або зрозуміти безпосередньо, особливо з огляду на використання шифрування та додаткового кодування.

Проте найнижчий результат було отримано за критерієм шаблонності, який отримав 1 бал (низький рівень). Аналогічно до структури проєктів, це свідчить про використання однакової конструкції обробки рядків, зокрема через функцію «exec(base64.b64decode(TDM().decrypt(...)))». Повторюваність такої структури дозволяє зловмиснику встановити шаблон обробки і сформувати гіпотезу щодо зворотного алгоритму, навіть якщо самі дані залишаються зашифрованими.

Для посилення захисту рядкових типів доцільно впровадити різноманітність у структурі декодування, наприклад, за рахунок генерації додаткового проміжного коду або функцій, які будуть розташовані у скомпільованих модулях (.pyd), написаних на C або C++. Такий підхід значно ускладнить зворотню інженерію та аналіз обробки рядків на рівні Python [79].

Крім того, з метою підвищення ентропії та зниження ймовірності відновлення даних, можна модифікувати алгоритм TDM, доповнюючи вхідні дані випадковими або структурно нейтральними символами перед шифруванням. Це створить додатковий рівень складності для автоматизованих засобів дешифрування або евристичного аналізу.

4) Коментарі та документації.

Середнє значення отриманих результатів відображено на наступній формулі (4.17):

$$\frac{5+5+1+5+5}{5} = 4.2 \tag{4.17}$$

Середнє значення ефективності для компонента коментарів та документацій становить 4.2, що відповідає високому рівню захисту. Такий результат вказує на те, що всі коментарі були оброблені, втратили початковий зміст і стали повністю нечитабельними. Це унеможливлює їхнє використання для розуміння логіки програми, що позитивно впливає на загальну захищеність вихідного коду.

Разом з тим, найнижчу оцінку отримав критерій шаблонності – 1 бал, що вказує на повторювану структуру обфускованих коментарів. Основною причиною цього є

використання однакових спеціальних роздільників, зокрема подвійних підкреслень «__» та зв’язних символів «&&», які зустрічаються в кожному з коментарів. Така одноманітність може бути виявлена під час статичного або евристичного аналізу, що створює потенційні шляхи для побудови шаблонів деобфускації зловмисником.

Для підвищення ефективності обфускації коментарів доцільно змінити принцип формування вставок. Зокрема, замість фіксованих символів доцільно використовувати множину спеціальних або службових роздільників, які обираються випадковим або псевдовипадковим чином при генерації обфускованих коментарів. Це дозволить знизити шаблонність структури, ускладнити аналіз і підвищити оцінку за цим критерієм без зменшення загальної консистентності та нечитабельності [80].

Висновки до четвертого розділу

Четвертому розділу було присвячено визначення ефективності розроблених механізмів захисту вихідного коду.

Для вимірювання ефективності, було визначено наступні критерії: ентропія, довжина рядка, шаблонність, відповідність з оригіналом та консистентність. Для усіх критеріїв визначено шкалу оцінювання Лікерта, а також визначено рівні для кожних оцінок (діапазонів). Кожен критерій обрахувався за окремою формулою.

Найкращий результат показав компонент коментарів та документацій, середнє значення якого становило 4.2 – це свідчить про високу нечитабельність і консистентність перетворень. Єдиним недоліком виявилась шаблонність, зумовлена повторюваними службовими символами, що може бути покращено шляхом їх варіативного використання.

Рядкові типи також продемонстрували високий рівень захисту (4.0). Основна вразливість – знову шаблонність, що проявляється в єдиному підході до декодування за допомогою «exec» і «base64». Незважаючи на це, рівень ентропії та консистентність були дуже високими, що забезпечує складність для зловмисника при аналізі.

Структура проєктів отримала середню оцінку 4.0. Механізм перейменування добре порушує оригінальну ієрархію, однак повторювані патерни в іменах знижують унікальність та сприяють шаблонному аналізу. Рекомендовано посилити варіативність у побудові нових назв директорій.

Найнижче середнє значення виявлено у компонента ідентифікаторів, яке склало 3.6 бали, що відповідає середньому рівню ефективності. Хоча загальний рівень консистентності та нечитабельності залишився високим, низькі значення ентропії та довжини знижують загальну стійкість до деобфускації. Це створює потенційні передумови до відновлення вихідного змісту, якщо використовуються евристичні або словникові підходи.

Враховуючи те, що розроблені механізми можуть бути деобфусковані, що і є унікальністю цих механізмів, результати свідчать про їх ефективність, однак демонструють також і наявність окремих вразливостей, які можуть бути усунуті подальшою оптимізацією методами генерації – зокрема, шляхом підвищення шаблонної варіативності та ентропійної складності.

ВИСНОВКИ

На основі проведених досліджень, ознайомлення із мовою програмування Python та аналізу особливостей даної мови програмування, аналізом методів зламу та захисту вихідного коду, можна зробити наступні висновки. Мова програмування Python має великий попит серед розробників, при цьому стає більше зацікавлених сторін до зламу програмних додатків, які створені на цій мові програмування.

1. Робота із мовою програмування Python дозволила дізнатись історію утворення й іншу загальну інформацію цієї мови. Важливою частиною було більше поглибитись в механізми роботи Python, тому розглянуто особливості даної мови програмування, а також допоміжні засоби для розробки програмних додатків. Існує безліч програмних додатків, але особливу уваги приділено тим, які дозволяють як розробити якісний програмний продукт, так і утворити виконуючі файли на основі вихідного коду.

2. Досліджено методи зламу програмних застосунків, які створені завдяки мові програмування Python, зокрема зазначені методи можуть застосовуватись для додатків, які створені іншими мовами програмування. Було практично перевірено декілька методів зламу, які дозволили отримати повну інформацію про програмний продукт, саме вихідний код. Для захисту програмних додатків, які створюються на основі мови програмування Python, наведено методи захисту додатків. Для подальшої роботи, слід було ознайомитись із програмою-обфускатором, яка дозволила на основі свого принципу роботи, утворити власні механізми захисту вихідного коду для мови програмування Python.

3. Розроблено механізми захисту вихідного коду мови програмування Python, які дозволяють обробляти усі компоненти кінцевого програмного продукту: структуру проєкта, класи, методи та змінні, рядкові типи даних, коментарів та документації. Серед механізмів обробки вихідного коду є як симетричні методи шифрування, так і звичайна заміна текстових даних. Розроблені механізми можуть

додатково використовуватись для обробки файлів, що дозволить надати більший рівень безпеки даних на рівні операційної системи. Прикладом може бути симетричний методи шифрування PRM та безключовий метод шифрування TDM, адже вони можуть працювати на рівні байт-коду – що відповідає більшості структурам файлів операційних систем.

4. Було проведено формалізовану оцінку ефективності розроблених механізмів захисту вихідного коду. Аналіз охопив основні компоненти коду, включаючи структуру проєкту, ідентифікатори, рядкові типи, коментарі та документацію. Для кожного з них були визначені відповідні критерії оцінювання, що дозволило здійснити як кількісний, так і якісний аналіз рівнів захисту. Результати свідчать про середній рівень ефективності, саме 3.95 бали розроблених механізмів. Незважаючи на наявність окремих вразливостей – таких як повторюваність структур або недостатня ентропія в окремих компонентах – загальний рівень нечитабельності, консистентності змін і стійкості до зворотного аналізу є задовільним. Таким чином, розроблені механізми можуть успішно застосовуватись для ускладнення аналізу та декодування вихідного коду, забезпечуючи відповідний рівень захисту інформації в програмному забезпеченні. У майбутньому їх можна вдосконалити шляхом підвищення варіативності структур, довжини та збільшення ентропії.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sharma V., Kumar, V., Sharma, S., Pathak, S. Python Programming: A Practical Approach. Chapman and Hall/CRC, 2021. – P. 309.
2. Медведєва М., Жмурко О., Криворучко І., Ковтанюк М. Використання ігрових онлайн-сервісів у процесі вивчення мов програмування. Актуальні питання гуманітарних наук, 2021. – Т. 2. – № 36. – С. 248-255.
3. Turkmen G., Sengul G., Sezen A. Comparative Analysis of Programming Languages Utilized in Artificial Intelligence Applications: Features, Performance, and Suitability. International Journal of Computational and Experimental Science and Engineering, 2024. – V. 10. – №3. – P. 461-469.
4. R. Singhla, P. Singh, R. Madaan, S. Panda. Image Classification Using Tensor Flow. 2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS), 2021. – P. 398-401.
5. Ogala, J.O. and Ojie, D.V. Comparative analysis of c, c++, c# and java programming languages. Global Scientific Journal, 2020. – V. 8. – №5. – P.1899-1913.
6. Кухар М. А. Аналіз можливостей мови програмування Python для роботи з просторовими даними. ScienceRise, 2019. – Т. 4. – №56. – С. 40-46.
7. Штаба В. Г., Макарова Л. М. Огляд фреймворку Kivy з точки зору розробки мобільних застосунків. Free and Open Source Software, 2025. – С. 187.
8. Грузин Н. А. Порівняння C++ і Python. Modern Science, 2020. – Т. 2. – № 1. – С. 343-348.
9. Калмиков Д. В. Реалізація основних принципів об’єктно-орієнтованого програмування у Python та застосування його у веб-розробці. Економіко-правові та управлінсько-технологічні виміри сьогодення: молодіжний погляд, 2021. – Т. 2. – С. 524.
10. Slabinoha M., et al. Comparative analysis of embedded databases performance on single board computer systems using Python. 2022 IEEE 17th International Conference on Computer Sciences and Information Technologies (CSIT), 2022. – P. 222-225.

11. de Oliveira P., Gheyi R., da Costa J, Ribeiro M. Assessing Python Style Guides: An Eye-Tracking Study with Novice Developers. In Simpósio Brasileiro de Engenharia de Software (SBES), 2024. – P. 136-146.

12. Парфьонов Ю., Колгатін О. Вибір служби вебхостингу для застосунків на базі фреймворку Django. Вісник Харківського національного автомобільно-дорожнього університету, 2022. – № 96. – С. 66.

13. Vagizov M., et al. Prepare and analyze taxation data using the Python Pandas library. IOP Conference Series: Earth and Environmental Science, 2021. – V. 876. – № 1. – P. 1-8.

14. Novac O.-C., et al. Analysis of the application efficiency of TensorFlow and PyTorch in convolutional neural network. Sensors, 2022. – Т. 22. – № 22. – С. 8872.

15. Sharma M., Khan M. S., Singh J. Python & Django the Fastest Growing Web Development Technology. 2024 IEEE 1st Karachi Section Humanitarian Technology Conference (KHI-HTC), 2024. – P. 1-9.

16. Shah S., Patel Y., Panchal K., Gandhi P., Patel P., Desai A. Python and MySQL based smart digital retail management system. 2021 6th International Conference for Convergence in Technology (I2CT), 2021. – С. 1-6.

17. Dautzenberg E., van Hurne S., Smulders M. M., de Smet L. C. GraphIAST: A graphical user interface software for Ideal Adsorption Solution Theory (IAST) calculations. Computer Physics Communications, 2022. – V. 280. – P. 1-6.

18. Ali Q., Riganelli O., Mariani L. Testing in the evolving world of DL systems: Insights from Python GitHub projects. 2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS), 2024. – P. 25-35.

19. Бондаренко В. В., Голуб Д. А., Єгоян В. Б. GIT – система керування версіями: GitHub, GitLab. Free and Open Source Software, 2025. – С. 188.

20. Kornilovska N. V., Vyshemyrska S. V., Kolmykov M. Development of Python electronic message information protection system using the PyCharm working area. Вісник Херсонського національного технічного університету, 2021. – V. 1. – №76. – P. 106-112.

21. Zampetti F., Geremia S., Bavota G., Di Penta M. CI/CD pipelines evolution and restructuring: A qualitative and quantitative study. 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2021. – P. 471-482.

22. Golchubian A., Marques O., Nojoumian M. Photo quality classification using deep learning. Multimedia Tools and Applications, 2021. – V. 80. – № 14. P. 22193-22208.

23. Modi A., Shah K., Shah S., Patel S., Shah M. Sentiment analysis of Twitter feeds using Flask environment: A superior application of data analysis. Annals of Data Science, 2024. – V. 11. – № 1. – P. 159-180.

24. Quinonez C., Meij E. A new era of AI-assisted journalism at Bloomberg. AI Magazine, 2024. – V. 45. – № 2. – P. 187-199.

25. Wang C., Wu R., Song H., Shu J., Li G. smartpip: A smart approach to resolving Python dependency conflict issues. Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022. – P. 1-12.

26. Liang G., Zhou X., Wang Q., Du Y., Huang C. Malicious packages lurking in user-friendly Python package index. 2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2021. – P. 606-613.

27. Shaffer T., Chard K., Thain D. An empirical study of package dependencies and lifetimes in binder Python containers. 2021 IEEE 17th International Conference on eScience (eScience), 2021. – P. 215-224.

28. Li J., Wang P., Jia L., Mao R., Li Q., He Y., et al. Design and implementation of an automated PDF drawing statistics tool based on Python. Sixth International Conference on Computer Information Science and Application Technology (CISAT 2023), 2023. – V. 12800. – P. 1686-1690.

29. Nasucha M., Rahmat R., Satrio M., Khornelius J. B., Hermawan H., Handoko P. GUI Development for an Image Enhancement Application to Support Computer Vision. 2024 3rd International Conference on Artificial Intelligence For Internet of Things (AIIoT), 2024. – P. 1-4.

30. More K. S., Wolkersdorfer C. Intelligent Mine Water Management Tools—e Metsi and Machine Learning GUI. Mine Water and the Environment, 2023. – V. 42. – № 1. – P. 111-120.

31. Morihiro K., Masahiro I. Translation Rules of Regular Expression Code for Hardware Accelerator. Proceedings of Asia Pacific Conference on Robot IoT System Development and Platform, 2021. – V. 2020. – P. 51-58.

32. Zhang X., Breitinger F., Luechinger E., O'Shaughnessy S. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. Forensic Science International: Digital Investigation, 2021. – V. 39. – P. 301285.

33. Buscemi A., Turcanu I., Castignani G., Panchenko A., Engel T., Shin K. G. A survey on controller area network reverse engineering. IEEE Communications Surveys & Tutorials, 2023. – V. 25. – № 3. – P. 1445-1481.

34. Vishwas G., Nithin N. S., Varshith P., Belwal M. Unveiling the world of code obfuscation: A comprehensive survey. 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), 2023. – P. 1-8.

35. Choi B., Jin H., Lee D. H., Choi W. ChatDEOB: An Effective Deobfuscation Method Based on Large Language Model. International Conference on Information Security Applications, 2024. – P. 151-163.

36. Kochberger P., Schrittwieser S., Schweighofer S., Kieseberg P., Weippl E. SOK: Automatic deobfuscation of virtualization-protected applications. Proceedings of the 16th International Conference on Availability, Reliability and Security, 2021. – P. 1-15.

37. You G., Kim G., Cho S., Han H. A Comparative Study on Optimization, Obfuscation, and Deobfuscation tools in Android. Journal of Internet Services and Information Security, 2021. – V. 11. – № 1. – P. 2-15.

38. Durfina L., Kroustek J., Zemek P., Kolar D., Hruska T., Masarik K., Meduna A. Advanced Static Analysis for Decompilation Using Scattered Context Grammars. Researches in Mathematical Methods in Electrical Engineering and Computer Science, 2011. – P. 164-170.

39. Макарова Л., Камінський С., Бризгалов М. Розробка програмного забезпечення для знаходження внесених змін в код виконуваних файлів. Вчені записки, 2023. – Т. 34 (73). – №2. – С. 206-212.

40. Горюк Н., Лавровський І. Статичний аналіз вихідного коду програмного забезпечення на базі рішення Fortify Static Code Analyzer. Сучасний захист інформації, 2021. – Т. 2. – С. 32-38.

41. Білоус І., Нагорний П. Статичний аналіз коду C# з використанням Roslyn API. Інформаційні моделюючі технології, системи та комплекси, 2021. – С. 54-57.

42. Umar R., Riadi I., Kusuma R. S. Analysis of conti ransomware attack on computer network with live forensic method. International Journal on Informatics for Development (IJID), 2021. – V. 10. – № 1. – P. 53-61.

43. Cvitić I., Periša M., Vladava J. Data Collection with Honeypot Server for Reverse Engineering of Malware. EAI International Conference on Management of Manufacturing Systems, 2023. – P. 61-77.

44. Wiedemeier J., Tarbet E., Zheng M., Ko S., Ouyang J., Cha S. K., Jee K. PYLINGUAL: Toward Perfect Decompilation of Evolving High-Level Languages. 2025 IEEE Symposium on Security and Privacy (SP), 2025. – P. 52.

45. Кулібаба С., Курченко О. Обробка вихідного коду для захисту інтелектуальної власності та забезпечення безпеки даних. Стратегічні комунікації у сфері забезпечення національної безпеки та оборони: проблеми, досвід та перспективи, 2023. – С. 182-185.

46. Levchuk A. Методи захисту від змін та дизасемблювання виконавчих файлів ОС Windows. Ukrainian Journal of Educational Studies and Information Technology, 2017. – V. 5. – № 1. – P. 166-169.

47. Kamble M. T. Feature Extraction and Analysis of Portable Executable Malicious File. 2022 Second International Conference on Computer Science, Engineering and Applications (ICCSEA), 2022. – P. 1-6.

48. Lee Y. B., Suk J. H., Lee D. H. Bypassing anti-analysis of commercial protector methods using DBI tools, 2021. – V. 9. – P. 7655-7673.

49. Lee J. H. A Study on Virtual Instruction Extraction Approaches for Themida VM TIGER Series. Journal of the Korea Institute of Information Security & Cryptology, 2024. – V. 34. – № 6. – P. 1297-1306.

50. Darche P. Debugging and Testing. Microprocessor 5: Software and Hardware Aspects of Development, Debugging and Testing, 2020. – P. 33-67.

51. Макаренко Д. Розробка застосунку для підвищення захисту додатків на операційній системі Android від реверс-інжиніринга: дипломна робота. Київ: Національний університет «Києво-Могилянська академія», 2023. – 75 с.

52. Catalano C., Specchia G., Totaro N. G. Enhancing Code Obfuscation Techniques: Exploring the Impact of Artificial Intelligence on Malware Detection. International Conference on Product-Focused Software Process Improvement, 2023. – P. 80-88.

53. Stillerman J., et al. Data catalog project–A browsable, searchable, metadata system. Fusion Engineering and Design 112, 2016. – P. 995-998.

54. Guta, Gabor. The Module: Organization of Program Parts into a Unit. Pragmatic Python Programming: Learning Python the Smart Way, 2022. – P. 147-169.

55. Кулібаба С. О., Курченко О. А. Криптографічний метод шифрування даних Pattern Reverse Multiplication. Кібербезпека: освіта, наука, техніка, 2022. – Т. 3. – №15. – С. 216-223.

56. Alshahrani A., Walker S. A novel encryption solution for real time applications. The Third International Conference on e-Technologies and Networks for Development (ICeND2014), 2014. – P. 75-78.

57. Huang C. Design and Implementation of the Cloud Platform Based on Python. 2024 IEEE International Conference on Information Technology, Electronics and Intelligent Communication Systems (ICITEICS), 2024. – P. 1-6.

58. Sauter N., Hattne J., Grosse-Kunstleve R., Echols N. New Python-based methods for data processing. Biological Crystallography, 2013. – V. 69. – №7. – P. 1274-1282.

59. Barone A., Sennrich R. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. Proceedings of the Eighth International Joint Conference on Natural Language Processing, 2017. – P. 314-319.

60. Кулібаба С. О., Курченко О. А. Розробка метода криптографічного шифрування без використання ключів для обробки даних. Proceedings of the XVII International Scientific and Practical Conference: The research process in science and the implementation of results, 2023. – С. 39-41.

61. Alam M., Badawy W., Jullien G. A novel pipelined threads architecture for AES encryption algorithm. Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors, 2002. – P. 296-302.

62. Kolluru D., Bhaskara P. IP to IP Calling Through Socket Programming. 2021 Asian Conference on Innovation in Technology (ASIANCON), 2021. – P. 1-7.

63. Mukherjee, R., Tripp, O., Liblit, B., & Wilson, M. Static analysis for AWS best practices in Python code. Leibniz International Proceedings in Informatics, 2022. – P. 1-28.

64. Wiggins G., Cage G., Smith R., Hitefield S., McDonnell M., Drane L., Malviya Thakur A. Best practices for documenting a scientific Python project. Oak Ridge National Laboratory (ORNL), 2023. – P.1-3.

65. Здирко Н. Г., Остапчук С. М. Критерій ефективності в аналізі та державному аудиті публічних закупівель. Облік і фінанси, 2020. – Т. 1. – №87. – С. 146-157.

66. Кононенко Т. Методики дослідження психологічного благополуччя осіб зрілого віку. Інноваційні проєкти для післявоєнного відновлення та розвитку України, 2023. – С. 64-65.

67. Лега Є., Ляшенко С. Методи та алгоритми оцінювання цифрової інфраструктури закладів вищої освіти. Сучасний стан наукових досліджень та технологій в промисловості, 2023. – Т. 2. – №24. – С. 90-103.

68. Hamid T., Al-Jumeily D., Mustafina J. Evaluation of the dynamic cybersecurity risk using the entropy weight method. Technology for smart futures, 2018. – P. 271-287.

69. Ma S. Calculation of entropy from data of motion. Journal of Statistical Physics, 1981. – V. 26. – P. 221-240.

70. Stiawan D., Idris M, Malik R. Nurmaini S., Alsharif N., Budiarto R. Investigating brute force attack patterns in IoT network. Journal of Electrical and Computer Engineering, 2019. – V. 1. – P. 1-13.

71. Nielsen J., Molich R. Heuristic evaluation of user interfaces. In Proceedings of the SIGCHI conference on Human factors in computing systems, 1990. – P. 249-256.

72. Aspin A. An examination and further development of a formula arising in the problem of comparing two mean values. Biometrika 35.1/2, 1948. – P. 88-96.

73. Raiyn J. A survey of cyber attack detection strategies. International Journal of Security and Its Applications, 2014. – V. 8. – №1. – P. 247-256.

74. Kolaczek G., Krzysztof J.. Attack pattern analysis framework for multiagent intrusion detection system. International Journal of Computational Intelligence Systems, 2008. – V. 1. – №3. – P. 215-224.

75. Behera, C., Bhaskari, D. Different obfuscation techniques for code protection. Procedia Computer Science, 2015. – V. 70. P. 757-763.

76. Zhang S., Hu Y., Bian G. Research on string similarity algorithm based on Levenshtein Distance. In 2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), 2017. – P. 2247-2251.

77. Di Crescenzo G. Cryptographic formula obfuscation. In International Symposium on Foundations and Practice of Security, 2018. – P. 208-224.

78. Ilic F., Zhao H., Pock T., Wildes R. Selective Interpretable and Motion Consistent Privacy Attribute Obfuscation for Action Recognition. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2024. – P. 18730-18739.

79. Ananth P., Gupta D., Ishai Y., Sahai A. Optimizing obfuscation: avoiding Barrington's theorem. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, 2014. – P. 646-658.

80. Ceccato M., Di Penta M., Falcarin P., Ricca F., Torchiano M., Tonella P. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. Empirical Software Engineering, 2014. – V. 19. – P. 1040-1074.

ДОДАТОК А

СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ ЗА ТЕМОЮ РОБОТИ

1. Кулібаба С. О., Курченко О. А. Криптографічний метод шифрування даних Pattern Reverse Multiplication. Кібербезпека: освіта, наука, техніка, 2022. Т. 3. – №15. – С. 216-223.

2. Кулібаба С. О., Курченко О. А. Розробка метода криптографічного шифрування без використання ключів для обробки даних. Proceedings of the XVII International Scientific and Practical Conference: The research process in science and the implementation of results, 2023. – С. 39-41.

3. Кулібаба С. О., Щебланін Ю. М., Курченко О. А. Обробка вихідного коду для захисту інтелектуальної власності та забезпечення безпеки даних. IV Міжнародна науково-практична конференція «Стратегічні комунікації у сфері забезпечення національної безпеки та оборони: проблеми, досвід та перспективи», 2023. – С. 182-185.

4. Кулібаба С., Пархоменко І. Механізми захисту вихідного коду програмного застосунку на базі мови програмування Python. VIII Міжнародна науково-практична конференція «Проблеми кібербезпеки інформаційно-комунікаційних систем» (PCSICS), 2025. – С. 103-106.