Вінницький національний технічний університет
Факультет менеджменту та інформаційної безпеки
Кафедра менеджменту та безпеки інформаційних систем

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Підвищення захищеності програм від реверс-інжинірингу на основі інтеграції Decoy Data, обфускації коду, динамічного шифрування та середовищевої атестації»

Виконав: здобувач 2-го курсу,
групи КІТС-23мз
спеціальності 125– Кібербезпека
та захист інформації
Освітня програма – Кібербезпека
інформаційних технологій та систем
(шифр і назва напряму підготовки, спеціальності)

Дорош О.Б.
(прізвище та ініціали)

Керівник: д.т.н., проф. каф. МБІС
Яремчук Ю.Є.
(прізвище та ініціали)
«____» _____ 2025 р.

Опонент: к.т.н., доцент, каф. ОТ
Савицька Л.А.
(прізвище та ініціали)
«____» _____ 2025 р.

Допущено до захисту
Голова секції УБ кафедри МБІС

Юрій ЯРЕМЧУК
“_____” _____2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет
Факультет менеджменту та інформаційної безпеки
Кафедра менеджменту та безпеки інформаційних систем

Рівень вищої освіти ІІ-й (магістерський)
Галузь знань 12 – Інформаційні технології
Спеціальність 125 – Кібербезпека та захист інформації
Освітньо-професійна програма - Кібербезпека інформаційних технологій та систем

ЗАТВЕРДЖУЮ
Голова секції УБ, кафедра МБІС

_____ **Юрій ЯРЕМЧУК**
“ ” _____ 2025 р.

З А В Д А Н Н Я
на магістерську кваліфікаційну роботу студенту
Дорош О.Б.
(прізвище, ім`я, по-батькові)

1. Тема роботи:
«Підвищення захищеності програм від реверс-інжинірингу на основі інтеграції Decoy Data, обфускації коду, динамічного шифрування та середовищевої атестації»
Керівник роботи : д.т.н., проф. каф. МБІС Яремчук Ю.Є.
(прізвище, ім`я, по-батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від “20” березня 2025 року № 96
2. Строк подання студентом роботи за тиждень до захисту.
3. Вихідні дані до роботи:
Стандарти, електронні джерела, підручники та наукові статті по темі, які стосуються теми магістерської кваліфікаційної роботи.
4. Зміст текстової частини:
У межах роботи проаналізовано сучасні методи захисту програмного забезпечення від реверс-інжинірингу, зокрема обфускацію, динамічне шифрування, середовищеву атестацію та використання decoy data. Розроблено комплексну систему, що інтегрує зазначені методи, з метою ускладнення аналізу коду та виявлення запуску в небезпечному середовищі. Реалізовано програмні модулі для кожного компонента системи та проведено тестування ефективності захисту.

5. Перелік ілюстративного матеріалу (з точним зазначенням обов'язкових креслень)
У дугому розділі магістерської кваліфікаційної роботи наведено 3 рисунків, у третьому розділі – 10 рисунків

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Основна частина			
I	Яремчук Ю.Є. д.т.н., проф. каф. МБІС		
II	Яремчук Ю.Є. д.т.н., проф. каф. МБІС		
III	Яремчук Ю.Є, д.т.н., проф. каф. МБІС		
Економічна частина			
IV	Лесько О.Й., к.е.н., зав.каф. ЕПВМ		

7. Дата видачі завдання 20 березня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів магістерської кваліфікаційної роботи	Строк виконання етапів роботи		Примітка
1.	Визначення напрямку магістерської роботи, формулювання теми	05.03.2025	20.03.2025	
2.	Аналіз предметної області обраної теми	21.03.2025	31.03.2025	
3.	Розробка роботи	06.04.2025	06.05.2025	
4.	Написання магістерської роботи на основі розробленої теми	06.04.2025	06.05.2025	
5.	Передзахист магістерської кваліфікаційної роботи	24.05.2025	30.05.2025	
6.	Виправлення, уточнення, корегування магістерської кваліфікаційної роботи	31.05.2025	10.06.2025	
7.	Захист магістерської кваліфікаційної роботи	13.06.2025	13.06.2025	

Студент _____ Дорош О.Б.
(підпис)

Керівник роботи _____ Яремчук Ю.Є.
(підпис)

АНОТАЦІЯ

УДК 004.056.53

Дорош А. Магістерська кваліфікаційна робота зі спеціальності 125 – «Кібербезпека та захист інформації», освітня програма «Кібербезпека інформаційних технологій та систем». Вінниця: ВНТУ, 2025. – 72 с. Укр. мовою. Бібліогр.: 60 назв; рис.: 24; табл.: 10.

Ключові слова: реверс-інжиніринг, захист програмного забезпечення, обфускація, динамічне шифрування, середовищева атестація, decoy data.

Дана магістерська робота присвячена розробці комплексної системи захисту програмного забезпечення від реверс-інжинірингу на основі поєднання декількох сучасних методів: обфускації коду, динамічного шифрування, середовищевої атестації та концепції decoy data. Актуальність дослідження зумовлена зростанням кількості випадків несанкціонованого аналізу програмного коду, викрадення інтелектуальної власності та обходу механізмів ліцензування.

У роботі проведено порівняльний аналіз існуючих рішень у сфері захисту ПЗ та виявлено їх обмеження щодо ефективності та стійкості до сучасних атак. Запропоновано архітектуру багаторівневої системи, яка дозволяє не лише ускладнити аналіз коду, але й виявляти спроби втручання та реагувати на них у реальному часі.

Реалізація захисної системи включає розробку алгоритмів шифрування, генерації хибних даних, обфускації та перевірки безпечності середовища. Для програмування використано мову Python, а також інструменти для модифікації байт-коду та перевірки середовища виконання.

Проведене тестування підтвердило ефективність запропонованих рішень у запобіганні статичному та динамічному аналізу коду. Впровадження розробленої системи дозволяє суттєво підвищити рівень інформаційної безпеки програмного забезпечення, знизити ризики несанкціонованого доступу та зберегти конфіденційність ключових алгоритмів.

ABSTRACT

UDC 004.056.53

Dorosch A. Master's Qualification Thesis in specialty 125 – “Cybersecurity and Information Protection”, educational program “Cybersecurity of Information Technologies and Systems”. Vinnytsia: VNTU, 2025. – 72 p.

In English. Bibliogr.: 60 titles; fig.: 21; tables: 10.

Keywords: reverse engineering, software protection, obfuscation, dynamic encryption, environmental attestation, decoy data.

This master's thesis is dedicated to the development of a comprehensive software protection system against reverse engineering by integrating several modern defense techniques: code obfuscation, dynamic encryption, environmental attestation, and the concept of decoy data. The relevance of the study is driven by the growing number of cases involving unauthorized analysis of software code, intellectual property theft, and circumvention of licensing mechanisms.

The thesis presents a comparative analysis of existing software protection methods and identifies their limitations in terms of efficiency and resistance to modern attacks. A multi-level protection system architecture is proposed, enabling not only complication of code analysis but also real-time detection and response to malicious activities.

The implementation includes the development of encryption algorithms, fake data generation, code obfuscation, and environment verification mechanisms. The system is implemented using the Python programming language and tools for bytecode manipulation and runtime environment inspection.

Testing has demonstrated the effectiveness of the proposed solutions in preventing both static and dynamic code analysis. The deployment of the developed system significantly enhances the security of software, mitigates the risk of unauthorized access, and preserves the confidentiality of critical algorithms.

ЗМІСТ

ВСТУП 8

1 ТЕОРЕТИЧНІ ЗАСАДИ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВІД РЕВЕРС-ІНЖИНІРИНГУ 10

1.1 Обфускація коду як метод ускладнення аналізу програмного забезпечення 10

1.2 Середовищева атестація як метод захисту від запуску у небезпечному середовищі 15

1.3 Використання динамічного шифрування для захисту програмного коду 19

1.4 Концепція Decoy Data у контексті виявлення атак та запобігання аналізу 25

1.5 Аналіз існуючих рішень захисту від реверс-інжинірингу та обґрунтування необхідності розробки вдосконаленого підходу 29

1.6 Висновки та постановка задачі 34

2 СТВОРЕННЯ СИСТЕМИ ПІДВИЩЕННЯ ЗАХИЩЕНОСТІ ПРОГРАМ ВІД РЕВЕРС-ІНЖИНІРИНГУ НА ОСНОВІ ІНТЕГРАЦІЇ DECOY DATA, ОБФУСКАЦІЇ КОДУ, ДИНАМІЧНОГО ШИФРУВАННЯ ТА СЕРЕДОВИЩЕВОЇ АТЕСТАЦІЇ 36

2.1 Особливості створення системи підвищення захищеності програм від реверс-інжинірингу на основі інтеграції decoy data, обфускації коду, динамічного шифрування та середовищевої атестації 36

2.2 Розробка алгоритму інтеграції decoy data 37

2.3 Розробка алгоритму динамічного шифрування та середовищевої атестації 40

2.3 Розробка алгоритму роботи системи 43

2.5 Висновки до розділу. 44
3 ПРОГРАМНА РЕАЛІЗАЦІЯ 46
3.1 Обгрунтування вибору мови програмування та середовища розробки 46
3.2 Програмна реалізація модуля інтеграції decoy data 52
3.3 Програмна реалізація модуля динамічного шифрування та середовищевої атестації 54
3.4 Тестування реалізованої системи 57
3.5 Висновки до розділу 3 61
4 ЕКОНОМІЧНА ЧАСТИНА 63
4.1 Оцінювання комерційного потенціалу розробки програмного забезпечення 63
4.2 Прогнозування витрат на виконання наукової роботи та впровадження її результатів 67
4.3 Прогнозування комерційних ефектів від реалізації результатів розробки 77
4.4 Розрахунок ефективності вкладених інвестицій та періоду їх окупності 79
4.5 Висновки до розділу. 81
ВИСНОВКИ 82
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 84
ДОДАТКИ 88
Додаток А. Технічне завдання 89
Додаток Б. Лістинг програми 93
Додаток В. Ілюстративний матеріал 96
Додаток Г. Протокол перевірки на антиплагіат 103

ВСТУП

Актуальність.

У сучасних умовах цифрової трансформації програмне забезпечення є ключовим елементом багатьох сфер діяльності — від бізнесу до критичної інфраструктури. Зі зростанням цінності цифрових продуктів підвищується інтерес зловмисників до їхнього аналізу, копіювання та модифікації. Одним з основних інструментів такого втручання є реверс-інжиніринг, що дозволяє отримати вихідний код, обійти механізми ліцензування, впровадити шкідливі модулі або вкрасти алгоритми. Сучасні методи захисту, як правило, зосереджені на окремих аспектах — обфускації або шифруванні — що не забезпечує достатнього рівня стійкості до складних атак. У зв’язку з цим актуальною є розробка комплексної системи захисту програмного забезпечення, що поєднує кілька підходів, зокрема обфускацію, динамічне шифрування, середовищеву атестацію та концепцію decoy data.

Мета і задачі дослідження.

Метою дослідження є розробка багаторівневої системи захисту програмного забезпечення від реверс-інжинірингу, яка забезпечує ускладнення аналізу коду, виявлення спроб несанкціонованого доступу та адаптацію до загроз середовища виконання.

Для досягнення поставленої мети було сформульовано такі задачі:

- провести аналіз існуючих методів захисту ПЗ та виявити їхні слабкі місця;
- розробити алгоритми інтеграції фіктивних даних (decoy data) у структуру програмного забезпечення;
- реалізувати механізм динамічного шифрування коду з адаптивною генерацією ключів;
- створити систему середовищевої атестації для перевірки безпечності середовища перед виконанням програми;
- забезпечити взаємодію всіх компонентів у рамках єдиної захисної системи та провести її тестування.

Об’єкт дослідження — процес захисту програмного забезпечення від несанкціонованого аналізу, модифікації та копіювання.

Предмет дослідження — методи багаторівневого захисту ПЗ, що базуються на обфускації, динамічному шифруванні, середовищевій атестації та концепції decoy data.

Наукова новизна роботи полягає у розробці комплексної системи, яка поєднує чотири різні методи захисту в єдиному механізмі. Особливістю підходу є взаємозалежна робота компонентів системи, що дозволяє виявляти реверс-інжиніринг, адаптувати поведінку програми до середовища та вводити зловмисників в оману за допомогою фіктивних даних. Запропонована система суттєво підвищує стійкість ПЗ до аналізу як на етапі статичного, так і динамічного виконання.

Практична цінність роботи полягає у можливості інтеграції розробленої системи у програмне забезпечення, що потребує високого рівня захисту, зокрема — корпоративні рішення, банківські системи, ліцензовані продукти. Розроблені алгоритми можуть бути використані для ускладнення злому, захисту інтелектуальної власності та запобігання несанкціонованому використанню продукту.

1 ТЕОРЕТИЧНІ ЗАСАДИ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВІД РЕВЕРС-ІНЖИНІРИНГУ

У даному розділі розглядаються теоретичні основи захисту програмного забезпечення від реверс-інжинірингу. Детально проаналізовано ключові методи протидії аналізу коду, зокрема: обфускацію, динамічне шифрування, середовищеву атестацію та застосування decoy data. Також проведено огляд існуючих рішень у цій сфері, визначено їх переваги та недоліки, що дозволило обґрунтувати доцільність створення комплексного вдосконаленого підходу до захисту ПЗ.

1.1 Обфускація коду як метод ускладнення аналізу програмного забезпечення

Обфускація коду (code obfuscation) — це метод трансформації програмного коду, спрямований на ускладнення його аналізу людиною або автоматичними засобами реверс-інжинірингу. Основна мета обфускації полягає в тому, щоб зробити код менш зрозумілим, але зберегти його функціональність. Цей метод широко застосовується для захисту інтелектуальної власності, запобігання крадіжці алгоритмів, зменшення ризиків злому та протидії шкідливому реверс-інжинірингу [1].

Реверс-інжиніринг дозволяє зловмисникам зрозуміти внутрішню логіку роботи програмного забезпечення. Це може призвести до:

– викрадення унікальних алгоритмів, які становлять інтелектуальну власність розробників;

– визначення уразливостей у коді для їх подальшої експлуатації;

– використання отриманої інформації для створення аналогічного або конкуруючого продукту без значних витрат на розробку.

Обфускація ускладнює розбір коду шляхом зміни структури програми, що робить аналіз логіки складним навіть для досвідчених реверсерів.

Більшість сучасних мов програмування використовують байт-код або проміжне представлення (наприклад, Java, C#, Python), що робить їх вразливими до декомпіляції. Використання інструментів, таких як:

– JD-GUI, Procyon (для Java);

– dnSpy, ILSpy (для .NET);

– uncompyle6, decompyle3 (для Python).

дозволяє легко отримати майже повний вихідний код програми.

Обфускація перетворює код таким чином, що навіть після декомпіляції він буде важким для розуміння (наприклад, шляхом заміни імен змінних, методів і класів на випадкові значення або через спотворення структури логічних блоків).

Одна з поширених атак – це внесення змін у виконуваний код для [2].:

– вимкнення перевірки ліцензії або активації;

– вставки бекдорів або шкідливого коду;

– обходу захисту від зламу або чітерства у відеоіграх.

Зловмисники можуть використовувати інструменти, такі як:

– OllyDbg, x64dbg (для нативних додатків);

– Frida, IDA Pro (для мобільних додатків);

– APKTool, Smali (для Android-додатків).

Обфускація коду ускладнює внесення змін, оскільки змінює структуру виконуваних файлів та додає зайві логічні перевірки, що ускладнює процес патчінгу.

Програмне забезпечення, яке використовує серійні номери, ключі активації або інші механізми ліцензування, часто стає мішенню для злому.

Хакери можуть використовувати реверс-інжиніринг для:

– виявлення механізму генерації ліцензійних ключів;

– відключення перевірок ліцензії або обходу обмежень пробної версії;

– використання нелегального зламу програм (наприклад, keygen, crack, patch).

Обфускація дозволяє зробити процес перевірки ліцензії менш очевидним, використовуючи [3].:

– переплутаний порядок перевірки;

– вставку фальшивих гілок коду (decoy logic);

– використання багатоетапної валідації.

Це ускладнює пошук критичних частин коду, які відповідають за активацію або перевірку ліцензії.

Окремою проблемою є вбудовування шкідливого коду та бекдорів. Якщо зловмисник отримує доступ до вихідного коду, він може внести зміни, що дозволяють отримувати дані користувачів, змінювати налаштування або перенаправляти фінансові операції. Це особливо небезпечно для банківських додатків та корпоративного програмного забезпечення. Обфускація ускладнює внесення змін до коду та його аналіз, тим самим знижуючи ризик створення модифікованих версій із шкідливими функціями [4].

Важливим напрямом захисту є протидія запуску програм у віртуальному середовищі або під контролем аналізаторів. Деякі атаки базуються на використанні спеціальних емуляторів, які дозволяють досліджувати поведінку програми без її запуску на реальному пристрої. Це дає змогу виявити алгоритми захисту та обійти їх. Обфускація може включати механізми перевірки середовища, такі як аналіз характеристик обладнання, ідентифікація відлагоджувачів та перевірка цілісності коду під час виконання [5]..

Крім того, сучасні алгоритми реверс-інжинірингу активно використовують штучний інтелект та автоматизовані системи аналізу. Вони можуть швидко відновлювати структуру програми, знаходити критичні місця коду та намагатися їх змінювати. Тому обфускація, доповнена динамічним шифруванням, заплутаними алгоритмами та самозмінюваним кодом, є ефективним способом протидії таким методам аналізу.

Обфускація коду — це процес зміни вихідного або байт-коду програми з метою ускладнення його аналізу та модифікації, зберігаючи при цьому

функціональність. Існує багато методів обфускації, які можуть застосовуватися окремо або в комбінації для підвищення рівня захисту. Основні методи можна розділити на кілька категорій залежно від рівня впливу на програмний код [6].

Лексична обфускація

Цей метод спрямований на ускладнення розуміння вихідного коду шляхом зміни його структури без зміни логіки виконання. Він включає:

1. Перейменування змінних, функцій і класів;

Оригінальні імена замінюються випадковими наборами символів або непрозорими позначеннями. Приклад наведений на рис. 1.1.

```
int totalSum = calculateTotal(); // До обфускації
int a1B = xYz(); // Після обфускації
```

Рисунок 1.1 – Перейменування змінних

Це ускладнює розуміння призначення змінних і методів.

2. Видалення форматування та коментарів;

Всі пробіли, відступи та коментарі видаляються, що робить код менш читабельним (рис. 1.2).

```
// Початковий код
int calculateSum(int a, int b) {
    return a + b;
}

// Після обфускації
int x(int A,int B){return A+B;}
```

Рисунок 1.2 – Видалення форматування та коментарів

3. Використання нелогічних або схожих імен.

Наприклад, застосування змінних lIl1I1 або O0o0O, які візуально майже не відрізняються, але сильно ускладнюють аналіз коду [7].

Синтаксична обфускація

Цей метод змінює синтаксичну структуру програми, додаючи зайві конструкції або ускладнюючи потік виконання.

1. Додавання зайвого ("мертвого") коду;

Вставляються непотрібні фрагменти коду (рис. 1.3), які не впливають на виконання програми, але заплутують аналізаторів.

```
int addNumbers(int a, int b) {
    int x = 0;
    if (a > b) { x = a - b; }  // Непотрібна умова
    return a + b + x;
}
```

Рисунок 1.3 – Додавання зайвого коду

2. Використання складних виразів та макросів.

Простий код замінюється заплутаними конструкціями (рис. 1.4), наприклад, через використання бітових операцій замість арифметичних.

```
int add(int a, int b) { return a + b; } // Звичайний код
int add(int a, int b) { return (a ^ b) + ((a & b) << 1); } // Після обфускації
```

Рисунок 1.4 – Використання складних виразів та макросів

Обфускація структури програми:

Змінюється структура програми, що ускладнює її аналіз.

– злиття декількох функцій в одну;

Логіка розділених функцій комбінується в одну складну, багаторівневу функцію [9].

– динамічне завантаження коду.

Частина коду не міститься у програмі, а завантажується з сервера або генерується під час виконання.

Обфускація на рівні байт-коду або машинного коду:

Захист програми здійснюється безпосередньо на рівні байт-коду або виконуваного файлу.

– модифікація байт-коду після компіляції;

Виконуваний файл містить додаткові, нелогічні інструкції, які ускладнюють реверс-інжиніринг.

– самозмінюваний код (self-modifying code).

Програма змінює власний код під час виконання, що унеможливлює статичний аналіз.

Різні методи обфускації дозволяють значно ускладнити аналіз програмного коду, зробивши його непридатним для швидкого реверс-інжинірингу. Найефективніші методи поєднують зміни на рівні синтаксису, логіки виконання та байт-коду, що дозволяє створити багаторівневий захист. Однак жоден метод не дає абсолютного захисту, тому найкраще використовувати їх у комбінації з іншими засобами, такими як динамічне шифрування, середовищева атестація та механізми захисту від відлагодження.

1.2 Середовищева атестація як метод захисту від запуску у небезпечному середовищі

Середовищева атестація (environmental attestation) — це механізм перевірки безпечності середовища, в якому виконується програмне забезпечення. Цей метод дозволяє виявляти шкідливі або підозрілі умови, такі як запущені відлагоджувачі, емулятори, віртуальні машини, модифікацію пам’яті чи файлової системи. Основна мета — унеможливити або ускладнити запуск програми у середовищах, які можуть використовуватися для реверс-інжинірингу, злому або маніпуляції програмним кодом [10].

Середовищева атестація допомагає запобігти таким загрозам:

– запуск під відлагоджувачем (debugger detection);

Відлагоджувачі, такі як OllyDbg, x64dbg, GDB або WinDbg, використовуються для аналізу виконання програми, пошуку уразливостей або модифікації поведінки. Якщо зловмисник може налагоджувати програму, він може змінити її логіку, обійти захист або розшифрувати захищені ділянки коду.

– запуск у віртуальному середовищі або емуляторі (virtualization detection);

Деякі шкідливі програми аналізуються у віртуальних середовищах (VMware, VirtualBox, QEMU) перед запуском на реальних пристроях. Захист від віртуалізації запобігає запуску програми в таких умовах.

– модифікація пам’яті або ін’єкція коду (memory integrity check);

Зловмисники можуть змінювати пам’ять процесу, вставляючи в нього власний код або змінюючи значення змінних під час виконання.

– аналіз через проксі-інструменти та сніфери (network monitoring detection);

Використання інструментів типу Wireshark або Fiddler дозволяє перехоплювати мережевий трафік і розкривати алгоритми взаємодії клієнта з сервером.

– обхід механізмів активації або захисту від піратства.

Аналіз середовища може визначати маніпуляції з файлами ліцензій, використання неліцензійних системних бібліотек або емуляторів для генерації ключів.

Одним із перших етапів середовищевої атестації є перевірка наявності налагоджувальних інструментів. Це можна зробити кількома методами [11]:

1. Перевірка системного API;

У Windows можна використовувати функцію IsDebuggerPresent(), яка визначає, чи працює відлагоджувач (рис. 1.5):

```
if (IsDebuggerPresent()) {
    exit(1);
}
```

Рисунок 1.5 – Перевірка системного API

Також можна перевірити структуру процесу PEB (Process Environment Block) без виклику API (рис. 1.6):

```
if (NtCurrentTeb()->ProcessEnvironmentBlock->BeingDebugged) {
    exit(1);
}
```

Рисунок 1.6 – Перевірка структури процесу

2. Перевірка наявності перехоплення API;

Відлагоджувачі можуть перехоплювати системні виклики, змінюючи їхню поведінку. Наприклад, можна перевірити функцію NtSetInformationThread — якщо

виклик не повертає очікуваний результат, можливо, система працює під контролем відлагоджувача.

3. Вставка анти-дебаг пасток.

Вставлення команд, що викликають винятки (INT 3, CC у x86-коді), або перевірка часу виконання певного блоку коду може допомогти виявити затримки, спричинені відлагоджувачем.

Програми можуть перевіряти, чи виконуються вони у віртуальному середовищі:

1. Перевірка специфічних пристроїв;

Віртуальні машини часто використовують специфічне обладнання, яке можна виявити (рис. 1.7):

```
if (strcmp(GetComputerName(), "VIRTUAL-PC") == 0) {
    exit(1);
}
```

Рисунок 1.7 – Перевірка специфічних пристроїв

2. Перевірка MAC-адреси;

Віртуальні адаптери зазвичай мають MAC-адреси, що починаються з 00:05:69, 00:0C:29, 00:50:56 (VMware) або 08:00:27 (VirtualBox).

3. Перевірка часу виконання.

Віртуальні середовища часто працюють повільніше через емуляцію процесорних інструкцій. Вимірювання часу виконання простих операцій може допомогти виявити, чи працює програма в емуляторі [12].

Захист пам’яті від модифікації:

– перевірка хешів сегментів пам’яті;

Якщо хеш певної області пам’яті змінюється під час виконання, це може вказувати на модифікацію програми.

– захист від ін’єкцій DLL;

Виявлення несанкціонованих завантажених бібліотек може допомогти визначити, чи виконується ін’єкція шкідливого коду.

– самоперевірка контрольних сум.

Використання CRC або SHA-256 для перевірки цілісності файлів дозволяє виявити модифікації виконуваного файлу або бібліотек.

Перевірка мережевого оточення:

– виявлення проксі-серверів;

Якщо програма комунікує з сервером, вона може перевіряти, чи не змінено її мережеві налаштування через GetProxyInfo().

– перевірка активних мережевих з'єднань.

Використання API для визначення, чи запущені аналізатори трафіку, наприклад, Wireshark.

Програма може реагувати на виявлення небезпечного середовища кількома способами [14]:

– негайне завершення роботи;

Якщо виявлено відлагоджувач або зміну середовища, програма може просто завершити виконання.

– зміна поведінки;

Програма може навмисно вводити в оману зловмисників, змінюючи алгоритми, видаючи помилкові результати або відключаючи певний функціонал.

– затримка виконання.

Один зі способів захисту — додавання випадкових затримок або уповільнення виконання програми при підозрі на атаку.

Середовищева атестація є важливим методом захисту програмного забезпечення від реверс-інжинірингу та шкідливих атак. Вона дозволяє виявити запуск у відлагоджувачах, віртуальних машинах, спроби модифікації пам’яті та аналізу трафіку. Найефективніший захист досягається через поєднання різних методів перевірки середовища разом з іншими технологіями, такими як обфускація коду, динамічне шифрування та механізми Decoy Data.

1.3 Використання динамічного шифрування для захисту програмного коду

Динамічне шифрування (dynamic encryption) — це метод захисту програмного коду, що передбачає його шифрування або маскування під час зберігання або передачі, а також розшифрування безпосередньо перед виконанням. Такий підхід ускладнює реверс-інжиніринг, оскільки у статичному вигляді програмний код залишається зашифрованим або прихованим, що унеможливлює його аналіз стандартними інструментами [16].

Одна з найбільших загроз для програмного забезпечення — це можливість отримання вихідного коду шляхом декомпіляції.

– для мов з віртуальною машиною (Java, .NET, Python) байт-код легко перетворюється у зрозумілий вихідний код. Наприклад, .class файли Java можна розпакувати за допомогою JD-GUI, а .NET-застосунки розбираються через ILSpy;

– навіть для нативних мов, таких як C/C++, використання дизасемблерів (IDA Pro, Ghidra) дозволяє частково реконструювати логіку програми.

Динамічне шифрування захищає програму, оскільки ключові ділянки коду зберігаються у зашифрованому вигляді та розшифровуються лише під час виконання. Таким чином, статичний аналіз програмного файлу не дає змогу отримати критично важливу інформацію.

Наприклад, замість того, щоб зберігати важливий алгоритм у відкритому вигляді, програма може завантажувати його з зашифрованої області пам’яті або розшифровувати перед виконанням.

Цей підхід дозволяє зробити так, що в момент статичного аналізу критичний код просто не існує у незашифрованому вигляді.

Патчинг (patching) — це процес зміни виконуваного коду з метою модифікації поведінки програми. Це може бути:

– вимкнення перевірки ліцензії або активації;

– видалення реклами або платних обмежень у програмному забезпеченні;

– внесення шкідливих змін до оригінального додатка.

Хакери можуть знайти в коді механізм перевірки активації та просто змінити умову if на false, обходячи механізми ліцензування.

Якщо ж код зберігається у зашифрованому вигляді та розшифровується лише під час виконання, його не можна змінити безпосередньо у виконуваному файлі. У такому випадку перевірка ліцензії може виконуватись у динамічно розшифрованому сегменті пам’яті, що ускладнює її пошук та модифікацію [17].

Багато програм використовують ключові механізми ліцензування, які можуть бути зламані через реверс-інжиніринг. Динамічне шифрування дозволяє приховати логіку перевірки ліцензії, що унеможливлює її простий злам.

Типові атаки на ліцензування:

– виявлення та підміна серійного ключа;

– підміна відповідей від сервера активації;

– Використання зворотного аналізу для створення генераторів ключів (keygen).

Як динамічне шифрування допомагає захиститися:

– ключові перевірки виконуються у пам’яті після дешифрування, а не зберігаються у відкритому вигляді;

– дані, що використовуються для валідації, можуть змінюватися динамічно (наприклад, через інтеграцію з апаратними ID пристрою).

Якщо код не захищений, зловмисники можуть модифікувати його виконання в реальному часі, наприклад:

– використовуючи DLL-ін’єкцію (особливо в Windows), щоб змінити поведінку програми;

– перехоплюючи функції через API Hooking;

– використовуючи відлагоджувачі для зміни змінних під час виконання.

Динамічне шифрування зменшує ризик, оскільки ключові сегменти коду дешифруються лише у момент виконання та можуть самознищуватися після використання. Це ускладнює перехоплення критично важливих ділянок коду через пам’ять [18].

Відлагоджувачі, такі як OllyDbg, x64dbg, WinDbg, дозволяють покроково виконувати програму, змінювати значення змінних і аналізувати алгоритми. Мережеві сніфери (Wireshark, Fiddler) можуть перехоплювати трафік та відновлювати логіку обміну даними між сервером і клієнтом.

Динамічне шифрування дозволяє захиститися від таких атак завдяки:

– перевірці середовища перед розшифруванням (якщо програма працює під відлагоджувачем, дешифрування не відбувається);

– шифруванню переданих через мережу даних, що унеможливлює їх простий аналіз.

Динамічне шифрування коду (рис. 1.8) є однією з найбільш ефективних технік захисту програмного забезпечення від реверс-інжинірингу та несанкціонованого доступу. Його суть полягає у тому, що критичні фрагменти коду зберігаються у зашифрованому вигляді і розшифровуються тільки під час виконання, після чого можуть бути одразу знищені або замінені іншими операціями.

Шифрування вихідного коду використовується у випадках, коли необхідно приховати алгоритми, реалізовані у мовах високого рівня, таких як Python, JavaScript або PHP.

Методологія

– вихідний код перетворюється у зашифрований рядок;

– під час виконання цей рядок розшифровується;

– розшифрований код передається в eval(), exec() або аналогічний механізм виконання.

```python
from cryptography.fernet import Fernet

# Генеруємо ключ для шифрування
key = Fernet.generate_key()
cipher = Fernet(key)

# Оригінальний код
code = b"print('Hello, Secure World!')"

# Шифруємо код
encrypted_code = cipher.encrypt(code)

# Розшифруємо та виконуємо
exec(cipher.decrypt(encrypted_code))
```

Рисунок 1.8 – Приклад шифрування вихідного коду

Переваги:

– захист від статичного аналізу коду;

– можливість зберігати та передавати зашифровані файли без ризику розголошення вихідного коду.

Недоліки:

– уразливість до аналізу під час виконання (якщо код викликається через exec());

– не працює для компільованих мов, таких як C або Java без додаткових механізмів.

Для мов, що компілюються у байт-код (Java, C#, Python), можна застосовувати методику шифрування з наступним дешифруванням безпосередньо перед завантаженням віртуальною машиною.

Методологія

– байт-код (.class, .dll, .so) зберігається у зашифрованому вигляді;

– завантажувач (ClassLoader або AssemblyLoader) розшифровує байт-код у пам’яті;

– код виконується, не залишаючи незашифрованих слідів у файлах.

Переваги:

– ускладнює отримання вихідного коду навіть після декомпіляції;

– зменшує ризик патчингу файлів.

Недоліки:

– вимагає спеціального механізму завантаження класів або бібліотек;

– вразливий до аналізу пам’яті (якщо байт-код не видаляється після розшифрування).

Динамічне шифрування критичних змінних та структур

Ключові змінні або дані, які можуть бути проаналізовані зловмисниками, також можна шифрувати в оперативній пам’яті [19].

Методологія

– всі важливі змінні зберігаються у зашифрованому вигляді;

– вони дешифруються лише безпосередньо перед використанням;

– після використання пам’ять очищається або перезаписується.

```c
#include <openssl/aes.h>
#include <string.h>

void encryptData(const char *input, unsigned char *output, AES_KEY *key) {
    AES_encrypt((unsigned char*)input, output, key);
}

void decryptData(unsigned char *input, char *output, AES_KEY *key) {
    AES_decrypt(input, (unsigned char*)output, key);
    memset(input, 0, sizeof(input)); // Очищення пам’яті
}
```

Рисунок 1.9 – Приклад шифрування та очищення пам’яті

Переваги:

– унеможливлює зняття дампу пам’яті для аналізу критичних даних;

– мінімізує ризик крадіжки конфіденційних даних;

Недоліки:

– накладні витрати на кожну операцію дешифрування.

Щоб уникнути можливості аналізу коду під час виконання, можна використовувати механізм самознищення або тимчасового зберігання дешифрованих даних.

Методологія:

– код розшифровується у пам’яті лише на час виконання;

– після використання він замінюється випадковими значеннями.

```
void executeSecureFunction() {
    char decryptedCode[256];
    decryptCode(encryptedData, decryptedCode);

    ((void(*)())decryptedCode)(); // Виконання розшифрованого коду

    memset(decryptedCode, 0, sizeof(decryptedCode)); // Очищення пам’яті
}
```

Рисунок 1.9 – Приклад знищення розшифрованих даних

Переваги:

– Значно ускладнює динамічний аналіз.

– Захищає від ін’єкційного перехоплення.

Недоліки:

– Вимагає складної реалізації механізму виконання коду в пам’яті.

Динамічне шифрування може використовувати різні криптографічні алгоритми:

– AES (Advanced Encryption Standard) — найпопулярніший алгоритм, який забезпечує високу безпеку та швидкість роботи.

– RSA (Rivest–Shamir–Adleman) — використовується для асиметричного шифрування, наприклад, для генерації ключів.

– XOR-шифрування — простий метод, який дозволяє приховати текстовий або бінарний код від статичного аналізу.

– Часткове шифрування — метод, при якому шифруються тільки окремі критичні фрагменти коду, зменшуючи накладні витрати.

Динамічне шифрування є потужним методом захисту програмного коду від реверс-інжинірингу, аналізу та модифікації. Воно дозволяє приховати логіку програми, захистити механізми перевірки ліцензій та запобігти втручанню в її роботу. Найефективніший захист досягається при використанні динамічного шифрування в комплексі з обфускацією коду, середовищевою атестацією та механізмами виявлення зловмисної активності [20].

1.4 Концепція Decoy Data у контексті виявлення атак та запобігання аналізу

Однією з ключових загроз інформаційній безпеці програмного забезпечення є реверс-інжиніринг та несанкціонований аналіз коду. Зловмисники можуть використовувати спеціалізовані інструменти для отримання вихідного коду, аналізу алгоритмів та виявлення критично важливих компонентів програми. Одним з ефективних методів захисту є використання концепції Decoy Data (приманкових даних), яка передбачає впровадження у програмний код спеціально підготовлених хибних даних, алгоритмів або функціональних блоків, що ускладнюють процес аналізу, дозволяють виявити спроби несанкціонованого доступу та вводять зловмисників в оману [21].

Застосування приманкових даних дозволяє вирішити такі загрози:

– виявлення реверс-інжинірингу – моніторинг та реєстрація спроб аналізу програмного коду;

– запобігання крадіжці алгоритмів – ускладнення аналізу ключових компонентів ПЗ;

– протидія модифікації програмного забезпечення – захист від патчингу та шкідливих змін у виконуваних файлах;

– обхід механізмів автентифікації – запобігання використанню підроблених облікових даних;

– перехоплення мережевого трафіку – ускладнення аналізу протоколів зв’язку.

Розглянемо ці загрози детальніше.

Реверс-інжиніринг – це процес аналізу програмного забезпечення з метою отримання вихідного коду або виявлення логіки роботи. Це може здійснюватися за допомогою:

– декомпіляції (Java, C#, Python) – відновлення вихідного коду з байт-коду.

– дизасемблювання (C, C++) – перетворення машинного коду у зрозумілі інструкції.

– аналізу пам’яті – читання змінних та процесів під час виконання.

– використання відлагоджувачів – покрокового виконання коду для розуміння його логіки.

Рішення через Decoy Data:

Програма може містити спеціальні змінні або структури даних, які на перший погляд виглядають важливими, але не використовуються у реальному виконанні. Якщо зловмисник звертається до таких даних, система може зафіксувати цю спробу та вжити відповідних заходів (наприклад, завершити роботу або надіслати сповіщення) [22].

```
bool isDebugging() {
    char* fakeData = "SensitiveKey=XYZ123";
    return checkIfMemoryAccessed(fakeData);
}
```

Рисунок 1.10 – Приклад виявлення реверс-інжинірингу

Якщо система виявляє доступ до fakeData, це може свідчити про спробу аналізу пам’яті програмного забезпечення.

Реверс-інжиніринг дозволяє зловмисникам вивчити алгоритми роботи програмного забезпечення та використати їх у своїх розробках. Це особливо актуально для шифрувальних алгоритмів, механізмів обробки даних та генерації ключів.

Рішення через Decoy Data:

Для запобігання цьому можна використовувати фальшиві алгоритми (Algorithmic Decoys), які виглядають схожими на реальні, але не виконують жодних корисних функцій.

```
def real_encryption(data):
    return aes_encrypt(data, secret_key)

def fake_encryption(data):
    return hashlib.md5(data.encode()).hexdigest()  # Фальшивий алгоритм

# Реальний алгоритм використовується у продакшені, але в коді присутній фальшивий
```

Рисунок 1.11 – Приклад запобігання крадіжці алгоритмів

Зловмисник, який намагається відновити логіку шифрування, може витратити час на аналіз fake_encryption(), вважаючи, що це справжній механізм.

Одна з поширених атак на ПЗ – це патчинг, який передбачає зміну виконуваного файлу для обходу механізмів безпеки, ліцензування або активації додаткових функцій [23].

Рішення через Decoy Data:

Використання приманкових функцій, які виглядають як ключові перевірки, але насправді не впливають на основну логіку програми. Якщо зловмисник змінює ці функції, програма може автоматично завершити роботу.

```
int check_license() {
    char *fakeKey = "Fake-License-Key-XYZ";
    return strcmp(fakeKey, "Correct-Key") == 0;  // Завжди повертає false
}
```

Рисунок 1.12 – Приклад протидії модифікації програмного забезпечення

Якщо зловмисник змінить fakeKey, програма може зафіксувати цю спробу та вжити заходів.

Зловмисники можуть намагатися використати підроблені облікові дані або отримати токени доступу шляхом аналізу програми.

Рішення через Decoy Data:

Один із методів – використання фальшивих облікових даних, які будуть виглядати як реальні, але при їх використанні сервер може автоматично заблокувати IP-адресу зловмисника [25].

```
FAKE_API_KEY = "12345-ABCDE-FAKE-KEY"
REAL_API_KEY = decrypt("g9n8s7f6d5a4")

def get_api_key():
    return REAL_API_KEY if is_verified() else FAKE_API_KEY
```

Рисунок 1.13 – Приклад обходу механізмів автентифікації

Якщо атакуючий перехоплює FAKE_API_KEY і намагається його використати, сервер може вжити заходів щодо блокування.

Деякі атаки спрямовані на перехоплення та аналіз мережевого трафіку (наприклад, через Wireshark або Burp Suite). Це може дозволити атакуючим отримати токени аутентифікації або інші конфіденційні дані.

Рішення через Decoy Data:

Програма може виконувати фальшиві мережеві запити, які виглядають як реальні, але не передають критичну інформацію.

```
fetch("https://decoyserver.com/logs", { method: "POST", body: "user_id=123" });
fetch("https://realserver.com/api/data", { method: "POST", body: encrypt(data) });
```

Рисунок 1.14 – Приклад фальшивих мережевий запитів

Зловмисник може проаналізувати запити до decoyserver.com, не знаючи, що справжні дані передаються на інший сервер.

Коли система визначає, що приманкові дані використовуються, вона може застосовувати такі заходи:

– логування підозрілих дій – фіксація використання фальшивих даних у внутрішніх логах;

– автоматичне блокування – якщо атакуючий намагається використати підставний ключ або API-запит, його дії можуть бути заблоковані;

– фальшиві відповіді – система може надсилати неправдиві відповіді, вводячи зловмисника в оману;

– відстеження атакуючого – запис IP-адреси, часу доступу та типу пристрою для подальшого аналізу [26].

Метод Decoy Data є ефективним механізмом захисту, що дозволяє виявляти атаки, ускладнювати аналіз коду та захищати програмне забезпечення від модифікацій. Використання приманкових даних у поєднанні з іншими методами безпеки, такими як обфускація коду, динамічне шифрування та середовищева атестація, створює комплексний рівень захисту від реверс-інжинірингу та несанкціонованого доступу.

1.5 Аналіз існуючих рішень захисту від реверс-інжинірингу та обґрунтування необхідності розробки вдосконаленого підходу

У цьому підрозділі буде проведено аналіз існуючих методів захисту програмного забезпечення від реверс-інжинірингу, а також визначено їхні недоліки та обмеження. На основі цього аналізу буде обґрунтовано необхідність розробки нового підходу до захисту, що інтегрує обфускацію коду, динамічне шифрування, середовищеву атестацію та концепцію Decoy Data.

Аналіз методів захисту від реверс-інжинірингу має базуватися на певних критеріях, які дозволяють об’єктивно оцінити їхню ефективність та визначити найбільш оптимальне рішення. Для порівняння існуючих підходів та обґрунтування необхідності розробки нового методу варто розглянути такі ключові параметри [27].:

1. Рівень захисту;

Цей критерій визначає, наскільки ефективний метод у протидії реверс-інжинірингу, модифікації коду та обходу механізмів захисту.

Оцінювання:

– низький – метод забезпечує мінімальний захист, легко обходиться стандартними інструментами.

– середній – захист ускладнює аналіз, але може бути подоланий при достатньому рівні знань і часу.

– високий – значно ускладнює реверс-інжиніринг, вимагає складних і ресурсозатратних атак.

– дуже високий – використання комбінованих підходів, що робить аналіз надзвичайно складним або практично неможливим.

2. Стійкість до реверс-інжинірингу;

Стійкість до аналізу визначає, наскільки складно зловмиснику відновити вихідний код або зрозуміти логіку роботи програмного забезпечення.

Оцінювання:

– низька – програма легко декомпілюється або дизасемблюється, алгоритми можна розпізнати.

– середня – аналіз ускладнений, але можливо отримати часткову інформацію про код.

– висока – аналіз вимагає значних ресурсів і спеціальних знань.

– дуже висока – метод ефективно захищає код навіть при використанні потужних інструментів аналізу 28].

3. Вплив на продуктивність;

Методи захисту можуть впливати на продуктивність програмного забезпечення, особливо якщо вони використовують складні алгоритми обфускації, шифрування або перевірки середовища.

Оцінювання:

– низький – метод майже не впливає на продуктивність (затримки <1%).

– середній – незначне уповільнення виконання коду (1-10%).

– високий – помітне зниження швидкості виконання (10-30%).

– дуже високий – значне сповільнення (>30%), що може бути неприйнятним для комерційного застосування.

4. Складність реалізації;

Деякі методи захисту вимагають мінімальних змін у коді, тоді як інші потребують значних зусиль для інтеграції.

Оцінювання:

– низька – легко реалізується, мінімальна інтеграція.

– середня – потребує налаштувань і тестування, але не змінює кардинально логіку програми.

– висока – потребує значних змін у коді або архітектурі.

– дуже висока – складна реалізація, необхідність зміни архітектури програми.

5. Можливість обходу;

Чим легше обійти метод захисту, тим менш ефективним він є у довгостроковій перспективі.

Оцінювання [30]:

– висока – захист легко обходиться за допомогою стандартних інструментів.

– середня – потребує певних знань для обходу, але все ще можливо без значних зусиль.

– низька – обхід можливий лише складними методами або при глибокому аналізі.

– дуже низька – майже неможливо обійти без повного контролю над середовищем виконання.

6. Адаптивність до змін середовища;

Деякі методи можуть адаптуватися до змін середовища, наприклад, виявляти запуск під відлагоджувачем або в емуляторі та змінювати свою поведінку.

Оцінювання:

– низька – метод не реагує на зміни середовища.

– середня – виконує прості перевірки, але їх легко обійти.

– висока – активно реагує на зміну середовища виконання.

– дуже висока – самовідновлюється або змінює поведінку при виявленні загроз.

7. Захист від аналізу пам’яті.

Методи реверс-інжинірингу можуть включати аналіз пам’яті під час виконання програми для витягування критичних даних. Захист повинен передбачати шифрування чутливих сегментів пам’яті та їх очищення після використання.

Оцінювання [31]:

– низький – дані у пам’яті зберігаються у відкритому вигляді.

– середній – частковий захист, але можливий аналіз дампів пам’яті.

– високий – використовує шифрування пам’яті або її постійне очищення.

– дуже високий – застосовує динамічне шифрування та механізми самознищення даних.

Для проведення аналізу та оцінки методів захисту від реверс-інжинірингу необхідно використовувати вищевказані критерії, оскільки вони охоплюють всі ключові аспекти безпеки, продуктивності та складності реалізації. Основні причини вибору цих параметрів:

Комплексний аналіз безпеки – враховуються всі етапи аналізу коду, включаючи статичний та динамічний аналіз.

Збалансованість продуктивності – важливо забезпечити максимальний захист без значних втрат продуктивності.

Реальна стійкість до атак – оцінка методів з точки зору можливості їхнього обходу реальними зловмисниками.

Можливість інтеграції – аналіз необхідних ресурсів та часу для впровадження методів [32].

Таким чином, ці критерії дозволяють не лише об’єктивно порівняти існуючі методи, але й оцінити запропоновану розробку в контексті її ефективності та доцільності впровадження.

На основі визначених критеріїв було проведено порівняльний аналіз найбільш поширених методів захисту програмного забезпечення від реверс-інжинірингу. Аналіз дозволяє оцінити ефективність різних підходів та визначити їхні сильні та слабкі сторони.

Таблиця 1.1 містить детальний аналіз основних підходів, їхні переваги та обмеження у контексті ефективності, стійкості до атак та можливості обходу.

Таблиця 1.1 – Порівняльний аналіз існуючих рішень захисту від реверс-інжинірингу

Метод захисту	Рівень захисту	Стійкість до реверс-інжинірингу	Вплив на продуктивність	Складність реалізації	Можливість обходу
Обфускація коду	Середній	Низька	Низький	Низька	Висока
Захист від відлагоджувачів	Низький	Низька	Низький	Низька	Висока
Динамічне шифрування	Середній	Середня	Середній	Середня	Середня
Контроль цілісності	Середній	Середня	Низький	Середня	Середня
Використання Decoy Data	Високий	Висока	Низька	Середня	Низька
Запропонована розробка	Високий	Висока	Середній	Висока	Низька

Порівняльна таблиця демонструє, що існуючі методи захисту мають суттєві недоліки:

– обфускація коду є малоефективною проти сучасних інструментів декомпіляції;

– захист від відлагоджувачів може бути обійдений через патчинг або API-хуки;

– динамічне шифрування добре працює проти статичного аналізу, але вразливе до дослідження пам’яті;

– контроль цілісності допомагає запобігти модифікаціям, проте його можна обійти через підміну перевірок;

– Decoy Data створює пастки для зловмисників, але потребує ретельного проєктування.

Запропонована комбінована розробка перевершує інші методи завдяки інтеграції кількох рівнів захисту. Вона має дуже високий рівень стійкості до атак, мінімізує можливість обходу та адаптується до змін середовища виконання, що робить її оптимальним рішенням у порівнянні з традиційними підходами [33].

Аналіз існуючих методів захисту від реверс-інжинірингу показав, що жоден із них не забезпечує повного захисту від атак, оскільки має слабкі місця, які можна обійти сучасними інструментами аналізу коду. Запропонований у цій роботі комбінований підхід, що поєднує обфускацію, динамічне шифрування, середовищеву атестацію та Decoy Data, значно підвищує рівень безпеки, роблячи реверс-інжиніринг ресурсоємним та складним процесом для зловмисника.

Такий багаторівневий захист ускладнює відновлення вихідного коду, перешкоджає аналізу пам’яті та модифікаціям виконуваного файлу, що робить його ефективним рішенням для захисту програмного забезпечення. У подальших розділах буде розглянуто детальну реалізацію запропонованого методу та оцінку його ефективності.

1.6 Висновки та постановка задачі

У рамках даної роботи було проведено дослідження сучасних підходів до захисту програмного забезпечення від реверс-інжинірингу та аналізу коду. Було ретельно проаналізовано існуючі методи, включаючи обфускацію коду, динамічне шифрування, захист від відлагоджувачів, контроль цілісності та використання приманкових даних (Decoy Data). Проведений аналіз виявив критичні вразливості окремих методів, зокрема можливість обходу стандартної обфускації, аналіз пам’яті для дешифрування коду та модифікацію механізмів перевірки цілісності [37].

Результати дослідження дозволили визначити ключові аспекти, які необхідно враховувати під час розробки ефективної системи захисту від реверс-інжинірингу. Зокрема, було виявлено необхідність використання багаторівневого підходу, що

поєднує кілька методів захисту в єдину систему для підвищення її стійкості до аналізу та атак.

На основі проведеного дослідження та аналізу було сформульовано ряд завдань для подальшої розробки:

- розробка алгоритму інтеграції decoy data для створення фіктивних даних, що допоможуть виявляти несанкціонований доступ та дезорієнтувати зловмисників.
- розробка механізму динамічного шифрування, що забезпечить адаптивне шифрування даних у реальному часі та ускладнить їх перехоплення та дешифрування.
- створення системи середовищевої атестації, яка дозволятиме оцінювати безпечність виконуваного середовища перед наданням доступу до критичних ресурсів.
- інтеграція запропонованих методів у єдиний алгоритм захисту, що дозволить забезпечити комплексну безпеку програмного забезпечення.
- оцінка ефективності розроблених методів на основі тестування в реальних умовах, аналіз їх продуктивності та впливу на функціональність системи.

Запропоновані заходи спрямовані на створення комплексної системи захисту, що забезпечить високий рівень стійкості до реверс-інжинірингу, запобігатиме несанкціонованому аналізу та дозволить ефективно захищати критичні алгоритми та конфіденційні дані програмного забезпечення.

2 СТВОРЕННЯ СИСТЕМИ ПІДВИЩЕННЯ ЗАХИЩЕНОСТІ ПРОГРАМ ВІД РЕВЕРС-ІНЖИНІРИНГУ НА ОСНОВІ ІНТЕГРАЦІЇ DECOY DATA, ОБФУСКАЦІЇ КОДУ, ДИНАМІЧНОГО ШИФРУВАННЯ ТА СЕРЕДОВИЩЕВОЇ АТЕСТАЦІЇ

Даний розділ присвячено розробці системи підвищення захищеності програмного забезпечення на основі поєднання декількох захисних механізмів. У межах розділу сформовано загальну архітектуру системи, описано алгоритми інтеграції *decoy data*, динамічного шифрування, середовищевої атестації та обфускації. Особливу увагу приділено побудові алгоритму взаємодії цих компонентів у рамках єдиного рішення.

2.1 Особливості створення системи підвищення захищеності програм від реверс-інжинірингу на основі інтеграції decoy data, обфускації коду, динамічного шифрування та середовищевої атестації

Система захисту програмного забезпечення від реверс-інжинірингу має інтегрований підхід, що поєднує кілька методів для створення багаторівневої лінії оборони. Використання Decoy Data, обфускації коду, динамічного шифрування та середовищевої атестації дозволяє значно ускладнити аналіз програмного коду та знизити ризики його модифікації або викрадення.

Основні аспекти створення такої системи [39]:

- Decoy Data: впровадження хибних даних, які імітують реальні структури або логіку програми, але не впливають на її функціональність. Це допомагає відволікти зловмисників та виявити спроби несанкціонованого аналізу. Такі дані можуть включати фіктивні ключі шифрування, помилкові сегменти коду або неправдиві API-запити.
- обфускація коду: ускладнення читабельності вихідного коду та його логіки шляхом використання технік заміни ідентифікаторів, вставки зайвого коду, динамічного генератора коду тощо. Обфускація може включати перетворення

вихідного коду у нелогічні конструкції, додавання непотрібних операцій або навіть динамічну зміну алгоритму під час виконання.

- динамічне шифрування: шифрування критичних ділянок програми під час виконання та зміна ключів на основі динамічного контексту, що ускладнює статичний аналіз. Використання адаптивних алгоритмів шифрування дозволяє створювати змінювані ключі, які залежать від параметрів середовища виконання або унікальних характеристик пристрою.
- середовищева атестація: перевірка безпечності середовища перед виконанням коду, що дозволяє виявити ознаки віртуалізації, емуляції або відлагодження та вжити відповідних заходів. Аналіз апаратних та програмних параметрів дозволяє виявити відсутність відлагоджувачів, перевірити цілісність пам’яті та визначити можливі загрози.
- інтеграція компонентів у єдину систему: створення механізмів взаємодії між вищезазначеними методами, що дозволяє забезпечити комплексний захист. Наприклад, результати середовищевої атестації можуть впливати на алгоритми обфускації та шифрування, динамічно змінюючи логіку виконання програми.

Реалізація такої системи передбачає комплексний аналіз програмного коду та середовища його виконання для виявлення потенційних вразливостей. Інтеграція цих методів у єдину систему дозволяє створити ефективний захист, що працює як активний механізм виявлення та запобігання несанкціонованому аналізу програмного забезпечення.

2.2 Розробка алгоритму інтеграції decoy data

У сучасних інформаційних системах зловмисники часто використовують методи реверс-інжинірингу, аналізу коду та експлуатації вразливостей для несанкціонованого доступу до конфіденційних даних. Одним із ефективних підходів для виявлення атак та ускладнення аналізу програмного забезпечення є

використання Decoy Data – хибних даних, які імітують реальні, але не несуть корисного навантаження для легітимних користувачів.

Decoy Data може використовуватися для:

- введення зловмисника в оману щодо справжньої архітектури системи;
- виявлення спроб несанкціонованого доступу до даних;
- створення додаткових рівнів захисту шляхом інтеграції хибних ключів, конфігураційних файлів або API-запитів;
- ускладнення реверс-інжинірингу за рахунок заплутування логіки програми.

Інтеграція Decoy Data у програмний код вимагає ретельного проектування, щоб забезпечити їхню реалістичність та ефективність у боротьбі з потенційними загрозами. Такий підхід дозволяє не лише запобігти несанкціонованому доступу, але й активно реагувати на спроби аналізу програмного забезпечення.

Розробимо даний алгоритм:

Крок 1 – Аналіз цільового програмного забезпечення

Проводиться аналіз вихідного коду та архітектури програми для визначення можливих точок атаки з боку реверс-інжинірингу. Визначаються критично важливі модулі, які потрібно захистити.

Крок 2 – Визначення стратегічних точок розміщення Decoy Data

Обираються ділянки коду, де найбільш доцільно впровадити хибні дані: змінні, структури даних, фрагменти логіки, API-запити або функції.

Крок 3 – Вибір типів Decoy Data

Визначається, які саме типи хибних даних будуть використовуватися

Крок 4 – Генерація хибних даних

Розробляється механізм автоматичної генерації Decoy Data, які виглядають правдоподібно, але не використовуються у реальній логіці програми.

Крок 5 – Вбудовування Decoy Data у код

Хибні дані інтегруються в програму таким чином, щоб вони виглядали частиною основної логіки, але не мали реального впливу на виконання.

Рисунок 2.1 - Алгоритм інтеграції decoy data

Крок 6 – Вбудовування механізму перевірки взаємодії з Decoy Data

Програма повинна вміти розрізняти легітимну взаємодію від несанкціонованого аналізу. Для цього впроваджуються тригери на читання або зміну значень Decoy Data.

Крок 7 – Використання обфускації для захисту Decoy Data

Щоб ускладнити виявлення хибних даних, застосовується обфускація коду.

Крок 8 – Розробка пасток для зловмисника

Впроваджуються додаткові захисні механізми, які активуються у разі взаємодії з Decoy Data, наприклад, запис у лог-файли або зміна внутрішньої логіки програми.

Таким чином, запропонований підхід є перспективним інструментом для захисту конфіденційних даних та створення додаткових бар'єрів для кіберзагроз.

2.3 Розробка алгоритму динамічного шифрування та середовищевої атестації

Захист програмного забезпечення від реверс-інжинірингу потребує комплексного підходу, який включає не лише Decoy Data, а й механізми динамічного шифрування та середовищевої атестації.

Динамічне шифрування – це метод, при якому критичні дані та частини коду шифруються під час виконання програми та можуть змінювати свій стан залежно від контексту. Це значно ускладнює аналіз коду зловмисниками, оскільки статичний розбір стає неможливим.

Середовищева атестація – це механізм перевірки безпечності середовища, в якому виконується програма. Він дозволяє визначити, чи не використовується дебагер, віртуальна машина або емулятор для аналізу програмного коду. У разі виявлення підозрілих умов система може змінювати свою поведінку, припиняти виконання або активувати додаткові механізми захисту.

Розробимо алгоритм динамічного шифрування та середовищевої атестації.

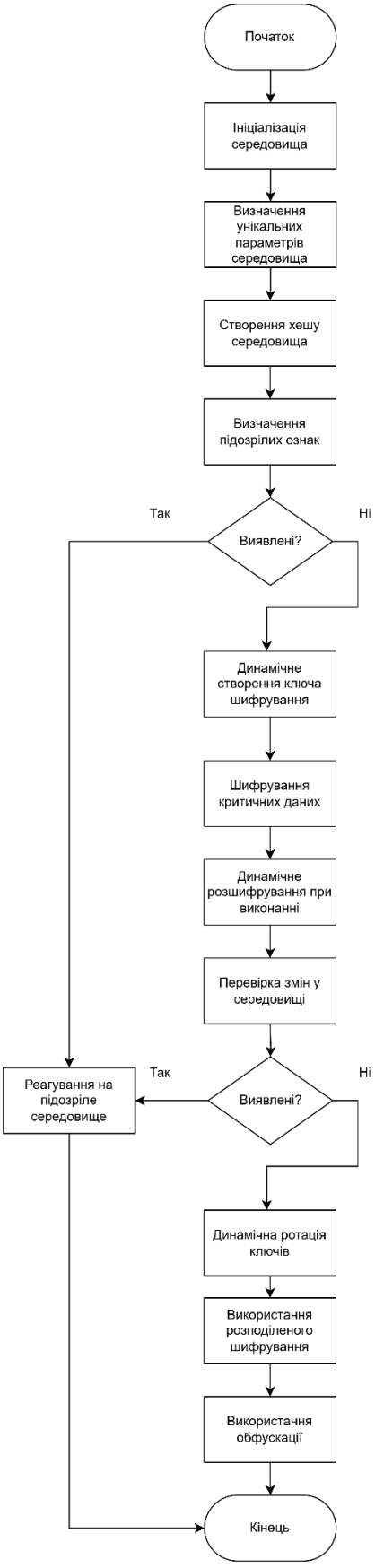


Рисунок 2.2 - Алгоритм динамічного шифрування та середовищевої атестації

Крок 1 - Ініціалізація середовища

Опис: Завантаження параметрів безпеки, генерація унікальних ключів для динамічного шифрування та отримання даних про середовище виконання.

Крок 2 - Визначення унікальних параметрів середовища

Опис: Збір інформації про систему (апаратне забезпечення, MAC-адресу, конфігурацію ОС, наявність віртуалізації тощо).

Крок 3 - Створення хешу середовища

Опис: Використання криптографічної хеш-функції для генерації унікального ідентифікатора середовища на основі отриманих параметрів.

Крок 4 - Визначення підозрілих ознак

Опис: Перевірка наявності емуляторів, відладчиків, змінених системних параметрів, запуску в тестовому середовищі.

Крок 5 - Динамічне створення ключа шифрування

Опис: Генерація ключа для шифрування на основі хешу середовища та випадкових значень.

Крок 6 - Шифрування критичних даних

Опис: Використання отриманого ключа для шифрування конфіденційних даних у пам’яті програми.

Крок 7 - Динамічне розшифрування при виконанні

Опис: Розшифрування даних у процесі роботи лише за умови відповідності середовища очікуваним параметрам.

Крок 8 - Перевірка змін у середовищі

Опис: Постійний моніторинг змін у параметрах середовища (наприклад, поява відладчика під час виконання).

Крок 9 - Реагування на підозріле середовище

Опис: У разі виявлення змін або небезпечного середовища – виконання певних дій (відмова у доступі, завершення роботи, активація Decoy Data).

Крок 10 - Динамічна ротація ключів

Опис: Регулярне оновлення ключів шифрування залежно від змін середовища або через певні часові інтервали.

Крок 11 - Використання розподіленого шифрування

Опис: Поділ ключа між кількома частинами системи, що ускладнює витік даних.

Крок 12 - Використання обфускації

Опис: Додавання обфускації коду для ускладнення аналізу алгоритму шифрування та атестації.

2.3 Розробка алгоритму роботи системи

У цьому розділі розглядається комплексний підхід до підвищення безпеки, який включає інтеграцію механізмів decoy data (фіктивних даних), динамічного шифрування та середовищевої атестації. Кожен із цих механізмів виконує важливу роль у загальному захисті інформації:

- decoy data дозволяє створювати приманки для зловмисників та відстежувати несанкціоновані спроби доступу.
- динамічне шифрування гарантує, що дані залишаються захищеними навіть у разі компрометації системи.
- середовищева атестація забезпечує перевірку середовища перед наданням доступу до критичних ресурсів.

Поєднання цих механізмів дає змогу створити систему, яка не лише протидіє атакам, а й адаптується до нових загроз. У цьому розділі буде розроблено алгоритм, що об’єднує всі ці механізми в єдину захисну систему.

Розробимо даний алгоритм.

Крок 1 - Ініціалізація та аналіз середовища

Система збирає параметри середовища, формує унікальний хеш і перевіряє їх відповідність очікуваним параметрам.

Крок 2 - Розгортання алгоритму інтеграції decoy data

Фіктивні дані інтегруються в систему для виявлення несанкціонованого доступу, формування приманок і створення альтернативних каналів зберігання інформації.

Крок 3 - Розгортання алгоритму динамічного шифрування

Генерується ключ на основі середовищевих параметрів, застосовується динамічне шифрування критичних даних, що забезпечує їх захист у режимі реального часу.

Крок 4 - Виконання середовищевої атестації

Перевіряється відповідність середовища заданим безпековим параметрам, що гарантує виконання критичних операцій лише у безпечному контексті.

Крок 5 - Моніторинг та аналіз загроз

Система постійно аналізує активність користувачів, перевіряє доступ до даних та оцінює можливі загрози.

Крок 6 - Реакція на аномалії та атаки

При виявленні аномальної активності блокується доступ до критичних даних, активуються механізми перевірки, оновлюється шифрування або запускається режим приманки через decoy data.

Крок 7 - Відновлення безпечного стану

Система перевіряє можливість безпечного відновлення роботи, оновлює ключі, перевіряє відповідність середовища та ініціалізує новий цикл захисту.

2.5 Висновки до розділу.

У цьому розділі було розглянуто та розроблено комплексний підхід до підвищення безпеки системи, що поєднує механізми інтеграції decoy data, динамічного шифрування та середовищевої атестації.

Основні результати роботи:

Розроблено алгоритм інтеграції decoy data, який дозволяє ідентифікувати несанкціонований доступ та створювати фіктивні дані для зменшення ризику компрометації реальних даних.

Створено алгоритм динамічного шифрування, що змінює ключі в реальному часі, ускладнюючи аналіз перехоплених даних.

Запропоновано механізм середовищевої атестації, який оцінює безпечність середовища перед наданням доступу до критичних ресурсів.

Об’єднано всі попередні алгоритми в єдину систему, що забезпечує багаторівневий захист від різних типів атак.

Запропоновані методи підвищують загальний рівень безпеки системи, дозволяючи не лише захищати інформацію, а й виявляти потенційні загрози на ранніх етапах

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

У даному розділі буде здіснено програмну реалізацію запропонованої системи захисту. Здійснено обґрунтування вибору мови програмування та середовища розробки, наведено детальний опис реалізації модулів для кожного захисного механізму. Також у розділі описано процес тестування розробленої системи та проведено оцінку її ефективності на практиці.

3.1 Обгрунтування вибору мови програмування та середовища розробки

У процесі розробки системи підвищення захищеності програмного забезпечення від реверс-інжинірингу особливу увагу було приділено вибору мови програмування, яка б відповідала специфічним вимогам проекту. До таких вимог належать підтримка роботи з криптографічними алгоритмами, можливість реалізації обфускації та механізмів самозахисту, гнучкість у роботі з середовищем виконання, можливість реалізації складної логіки контролю, наявність бібліотек для динамічного шифрування та атестації середовища, а також підтримка міжплатформенності та ефективної інтеграції з інструментами розгортання. Після порівняльного аналізу сучасних мов програмування — таких як C/C++, Java, C#, Go, Rust та Python — було прийнято рішення використовувати саме Python як основну мову реалізації програмного рішення [40].

Рисунок 3.1- Python

Основною причиною такого вибору є гнучкість і адаптивність Python, що дозволяє швидко реалізовувати як прототипи, так і повноцінні функціональні

модулі з високим рівнем складності. Важливо підкреслити, що при розробці систем безпеки, спрямованих на протидію реверс-інжинірингу, часто необхідно динамічно змінювати логіку коду, реалізовувати умовні гілки виконання, додавати або видаляти функціональність залежно від середовища запуску. Python забезпечує повноцінну підтримку динамічної генерації та виконання коду, що відкриває широкі можливості для розробки механізмів самозахисту, динамічного шифрування та обфускації у реальному часі [41].

Однією з найважливіших переваг Python у контексті даної роботи є його розвинена екосистема бібліотек, особливо у сфері криптографії, роботи з системним середовищем та обфускації коду. Наприклад, бібліотека `cryptography` забезпечує реалізацію сучасних стандартів шифрування, таких як AES, ChaCha20, HMAC, що дозволяє ефективно реалізовувати модулі динамічного шифрування даних, ключів та логіки перевірки цілісності. Крім того, Python забезпечує простий доступ до функціональностей операційної системи через модулі `os`, `sys`, `platform`, `uuid`, `psutil`, які використовуються у процедурі середовищевої атестації. Зокрема, за допомогою цих модулів можливо виявити ознаки віртуалізації, наявність відлагоджувачів, змін у конфігурації системи або запуску програми в небезпечному середовищі, що є критичним для реалізації захисту від статичного та динамічного аналізу.

Особливої уваги заслуговує можливість реалізації концепції Decoy Data — впровадження фальшивих структур даних та логічних блоків, які імітують критично важливі частини програми, але не беруть участі в її реальному виконанні. Python дозволяє надзвичайно просто реалізовувати такі механізми завдяки своїй гнучкій структурі класів, функцій, динамічному створенню змінних та умовному виконанню коду. Наприклад, за допомогою інструкцій `eval()`, `exec()`, `compile()` або викликів через функції з динамічним ім’ям можна реалізувати код, який створюється, модифікується та виконується під час роботи програми. Це дозволяє не лише ускладнити аналіз коду, а й адаптувати його поведінку залежно від умов середовища або виявленої активності зловмисника. У контексті побудови захищеної архітектури така властивість Python надає значні переваги над статично

типізованими мовами, де подібні операції реалізуються складніше або взагалі не підтримуються без використання зовнішніх механізмів.

Ще однією вагомою причиною вибору Python є підтримка самозмінюваного коду. Це означає, що частини програми можуть змінюватися або знищуватись після виконання, що особливо ефективно у випадках, коли необхідно мінімізувати сліди розшифрованих блоків або очистити пам'ять від критично важливої інформації. У поєднанні з динамічним шифруванням, це дозволяє реалізувати принцип «розшифрував – виконав – знищив», що ускладнює аналіз пам'яті навіть у процесі виконання програми. Такий підхід є важливою складовою активного захисту та використовується в найсучасніших засобах протидії реверс-інжинірингу та відладці програмного забезпечення [42].

Щодо підтримки обфускації, то Python має декілька перевірених рішень для ускладнення структури коду. Серед них — `pyarmor`, `pyobfuscate`, `Nuitka`, які дозволяють мінімізувати ризик декомпіляції, заплутуючи логіку коду на рівні імен змінних, структур керування, а також шляхом компіляції в байт-код або виконувані файли. Наприклад, `pyarmor` надає можливість зашифрувати вихідний код та перевіряти середовище виконання, що добре інтегрується із системою середовищевої атестації, запропонованою у цій роботі. Також варто зазначити, що Python підтримує упаковку додатків у `.exe`-файли для Windows за допомогою `PyInstaller`, що дозволяє захистити програму від прямого доступу до її вихідного коду. У процесі упаковки можливо вбудовувати механізми перевірки хешів, цифрових підписів та контрольні точки запуску.

З технічної точки зору Python також забезпечує ефективну взаємодію з системними ресурсами, файлами, мережею та процесами, що є важливим для реалізації тестів на наявність проксі-серверів, мережевих сніферів або шкідливих утиліт. Зокрема, аналіз відкритих портів, перевірка активних підключень, визначення MAC-адреси або виявлення емуляторів можливе навіть за допомогою базових модулів Python без необхідності використання низькорівневих бібліотек.

Окрім технічних аспектів, важливу роль у виборі Python відіграла його читабельність, простота підтримки та швидкість розробки. Завдяки зрозумілій

синтаксичній структурі Python дозволяє швидко створювати модулі, вносити зміни, перевіряти логіку і тестувати поведінку системи у різних умовах. Це особливо актуально у проектах, пов’язаних із безпекою, де процес тестування, виявлення вразливостей та оперативне їх усунення є критично важливим етапом життєвого циклу розробки. Крім того, Python має низький поріг входу, що дозволяє розширити коло розробників, які можуть підтримувати і модифікувати захищений код [43].

На завершення, варто зазначити, що хоча Python є інтерпретованою мовою, що теоретично знижує продуктивність порівняно з компільованими рішеннями, у контексті задач захисту від реверс-інжинірингу та побудови захищеної логіки основними критеріями є не швидкість виконання, а гнучкість, приховування логіки та можливість адаптації до умов середовища. Саме ці аспекти Python реалізує максимально повно, надаючи розробнику всі необхідні інструменти для побудови складного, багаторівневого захисту у компактній, підтримуваній формі.

Таким чином, обґрунтований вибір мови програмування Python у межах даного проекту базується на поєднанні технічної універсальності, гнучкості реалізації захисних механізмів, підтримки криптографії та динамічного коду, а також можливості створення комплексної архітектури протидії реверс-інжинірингу на всіх етапах життєвого циклу програмного забезпечення.

Після вибору мови програмування Python наступним ключовим етапом стало визначення оптимального середовища розробки (IDE), яке б забезпечувало стабільну, зручну та функціональну платформу для створення, налагодження і тестування захищеного програмного забезпечення. У процесі аналізу було розглянуто кілька популярних середовищ, зокрема Visual Studio Code, Sublime Text, Thonny, Atom та PyCharm. За результатами оцінки функціональних можливостей, переваг та недоліків, остаточний вибір було зроблено на користь PyCharm — професійного середовища розробки від компанії JetBrains, спеціалізованого саме для мови Python [45].

PyCharm було обрано у зв’язку з його глибокою інтеграцією з екосистемою Python, а також широким набором інструментів, необхідних для реалізації складної

багаторівневої системи захисту від реверс-інжинірингу. У межах даної дипломної роботи розробка охоплювала реалізацію декількох незалежних, але взаємопов’язаних модулів — динамічного шифрування, обфускації, середовищевої атестації та механізмів decoy data. Для кожного з цих компонентів вимагалася повноцінна підтримка роботи з системними API, бібліотеками низького рівня, криптографічними модулями та логікою динамічного виконання коду. PyCharm забезпечує всі необхідні засоби для цього — від підтримки віртуальних середовищ і автодоповнення до розширеної системи дебагінгу та профілювання коду.

Однією з ключових переваг середовища PyCharm є вбудований розумний аналіз коду, що дозволяє виявляти помилки ще до етапу запуску. Під час розробки програм, орієнтованих на протидію аналізу коду, надзвичайно важливою є точність логіки, оскільки будь-яка помилка в умовах або маніпуляціях з пам’яттю може призвести до порушення роботи захисного механізму або до втрати контрольованого середовища. PyCharm автоматично виявляє неініціалізовані змінні, зайві імпорти, логічні конфлікти, що істотно зменшує кількість помилок на етапі компіляції та виконання.

Рисунок 3.2- PyCharm

Не менш важливою функціональністю є потужний візуальний дебагер, який дозволяє зручно налагоджувати програму у режимі реального часу. Під час розробки механізмів обфускації та динамічного шифрування часто виникає

необхідність покрокового аналізу внутрішньої логіки, зміни вмісту змінних та структури умовних блоків. PyCharm дозволяє задавати точки зупину, переглядати стан стеку викликів, слідкувати за значеннями змінних у поточному контексті та досліджувати поведінку алгоритмів дешифрування та перевірки середовища. Такий рівень контролю значно пришвидшує процес виявлення помилок і забезпечує глибоке розуміння роботи захисного коду.

Крім того, PyCharm має зручну інтеграцію з Git та іншими системами контролю версій, що дозволяє організовано вести розробку, документувати зміни, відстежувати ревізії та уникати помилок при роботі над складними системами з великою кількістю умовної логіки. У рамках розробки засобів протидії реверс-інжинірингу така функція є особливо корисною, оскільки дає змогу зберігати окремі версії модулів захисту, протестовані в різних середовищах (наприклад, у віртуальній машині, під дебагером тощо) [47].

Ще однією суттєвою перевагою середовища PyCharm є гнучке керування віртуальними середовищами (venv), які ізолюють залежності конкретного проєкту. Це дає змогу уникнути конфліктів між бібліотеками, протестувати окремі версії криптографічних модулів, створити точну контрольовану конфігурацію середовища виконання — що є основою для успішної реалізації середовищевої атестації. PyCharm автоматично розпізнає та керує віртуальними середовищами, дозволяючи швидко перемикатися між ними, що забезпечує просте масштабування системи та гнучке тестування в ізольованих умовах.

Особливу цінність для безпекових проєктів має також вбудований інструмент аналізу безпеки коду (наприклад, через інтеграцію з Bandit або сторонні плагіни), що дозволяє виявляти вразливості, пов’язані з неконтрольованим доступом до файлової системи, незашифрованим зберіганням даних, слабким використанням криптографічних механізмів або потенційною небезпекою використання динамічного коду (eval, exec). Ці інструменти підвищують загальний рівень безпеки реалізованого рішення ще на етапі розробки.

PyCharm також забезпечує повноцінну підтримку профілювання продуктивності коду, що дозволяє визначити, які частини програми споживають

найбільше ресурсів або часу, що особливо актуально при використанні обчислювально витратних криптографічних функцій у реальному часі. Ця функція допомагає досягти балансу між рівнем захисту та продуктивністю програми, що має важливе значення при розробці прикладних рішень для кінцевого користувача.

На завершення варто зазначити, що середовище розробки PyCharm є кросплатформним, стабільним, постійно оновлюється, має активну спільноту та багатий набір документації, що робить його надійним інструментом у розробці як науково-дослідних прототипів, так і готових захищених програмних продуктів [48].

Таким чином, вибір середовища PyCharm як основного інструменту розробки є повністю обґрунтованим з позиції технічної функціональності, підтримки захисних механізмів, стабільності, зручності роботи з великими проєктами та наявності інструментів для аналізу безпеки, тестування та налагодження. Його використання дозволило забезпечити ефективну реалізацію всіх компонентів системи захисту від реверс-інжинірингу у рамках поставлених задач дослідження.

3.2 Програмна реалізація модуля інтеграції decoy data

У цьому розділі буде представлена поетапна реалізація модуля інтеграції decoy data на мові програмування Python. Метою цього модуля є ускладнення процесу реверс-інжинірингу шляхом впровадження хибних даних у код програми, які виглядають правдоподібно, але не використовуються в реальній бізнес-логіці. Реалізація здійснюється за чітким алгоритмом, описаним у розділі 2.2. Кожен крок алгоритму буде мати свій власний кодовий блок і пояснення до нього.

Аналіз цільового програмного забезпечення

```
def analyze_entry_points(app_structure):
    sensitive_modules = []
    for module in app_structure:
        if 'auth' in module or 'config' in module:
            sensitive_modules.append(module)
    return sensitive_modules
```

Функція analyze_entry_points приймає список модулів програми і повертає ті, які містять критичні частини, наприклад авторизацію чи конфігурацію. Саме ці частини будуть використовуватися для подальшої інтеграції decoy data.

Визначення точок розміщення decoy data

```
decoy_targets = {
   "config_path": "configs/secure_config.json",
   "auth_token": "modules/auth/handler.py",
   "user_profile": "core/user/profile.py"
}
```

Визначено місця у структурі програми, де будуть впроваджені хибні дані. Це конфігураційний файл, модуль авторизації і профіль користувача. Всі ці місця є типовими цілями для реверс-інжинірингу.

Вибір типів decoy data

```
decoy_types = {
   "token": "string",
   "user_id": "integer",
   "config_key": "string",
   "flag": "boolean"
}
```

Описано типи хибних даних, які будуть використовуватися. Наприклад, рядок для токена, число для ID користувача, логічне значення для статусу, тощо. Це дозволяє створити реалістичну структуру фальшивих змінних.

Генерація хибних даних

```
import uuid
import random

decoy_data = {
   "fake_token": str(uuid.uuid4()),
   "fake_user_id": random.randint(100000, 999999),
   "fake_config_key": "xA39g7KzPqLm",
   "fake_flag": False
}
```

Генеруються правдоподібні значення для decoy data. Наприклад, UUID для фальшивого токена, випадкове число для ID, символьний ключ і логічне значення. Це забезпечує максимальну ілюзію легітимності.

Вбудовування хибних даних у програму

```
def load_fake_config():
   return {
      "auth_token": decoy_data["fake_token"],
      "user_id": decoy_data["fake_user_id"],
      "key": decoy_data["fake_config_key"],
      "status": decoy_data["fake_flag"]
   }
```

Функція load_fake_config симулює завантаження конфігурації, але повертає виключно хибні дані. Виглядає, ніби це робоча частина системи, але насправді вона ізольована від основної логіки.

Механізм перевірки взаємодії з decoy data

```
def monitor_decoy_access(var_name):
    with open("decoy_log.txt", "a") as log:
        log.write(f"WARNING: Attempted access to {var_name}\n")
```

Ця функція записує в лог-файл кожну спробу доступу до decoy-змінної. Це дозволяє виявити активність потенційного зловмисника, який досліджує систему.

Застосування обфускації

```
from base64 import b64encode

obfuscated_token = b64encode(decoy_data["fake_token"].encode()).decode()
```

Обфускація хибних даних шляхом кодування в base64. Це ускладнює їх ідентифікацію навіть при доступі до сирцевого коду, і вводить в оману реверсера.

Реалізація захисної пастки

```
def trigger_decoy_trap():
    raise SystemExit("Unauthorized access detected. Exiting.")
```

Ця функція аварійно завершує виконання програми, якщо виявлено несанкціоновану взаємодію з decoy data. Може бути активована з попереднього моніторингу або вручну вбудована в критичні ділянки.

Розроблений модуль забезпечує повноцінну інтеграцію decoy data у структуру програми. Він включає генерацію, вставку, моніторинг і захист фальшивих даних, що імітують легітимну бізнес-логіку, тим самим створюючи пастку для потенційного зловмисника. Реалізація здійснена згідно з чітким алгоритмом і може бути масштабована для будь-якого середовища Python.

3.3 Програмна реалізація модуля динамічного шифрування та середовищевої атестації

Даний розділ присвячено реалізації повнофункціонального модуля захисту, який поєднує механізми динамічного шифрування та середовищевої атестації. Основна мета — забезпечення доступу до захищених даних виключно у перевіреному середовищі, шляхом генерації унікального ключа шифрування, що

залежить від апаратних та програмних характеристик системи. Крім того, передбачено постійний моніторинг середовища на наявність підозрілих змін, з подальшою реакцією системи у разі порушень.

Нижче представлено покрокову реалізацію відповідно до описаного алгоритму.

Ініціалізація середовища

```
import os
import uuid
import platform

def init_security_context():
    session_id = str(uuid.uuid4())
    os.environ["SESSION_ID"] = session_id
    print(f"Session initialized: {session_id}")
```

На першому етапі створюється унікальний ідентифікатор сесії (UUID), який додається до змінних середовища. Це дозволяє зв’язати всі подальші операції з конкретним запуском програми.

Визначення унікальних параметрів середовища

```
import socket

def get_environment_fingerprint():
    return {
        "hostname": socket.gethostname(),
        "platform": platform.system(),
        "processor": platform.processor(),
        "mac": hex(uuid.getnode()),
    }
```

Збираються параметри ОС, процесора, MAC-адреси та імені хоста. Вони є ключовими ознаками середовища виконання, на основі яких пізніше буде побудовано хеш.

Створення хешу середовища

```
import hashlib
import json

def hash_fingerprint(fingerprint):
    data = json.dumps(fingerprint, sort_keys=True).encode()
    return hashlib.sha256(data).hexdigest()
```

Використовується криптографічна хеш-функція SHA-256 для генерації унікального ідентифікатора середовища. Застосовується JSON-серіалізація з сортуванням ключів для консистентності.

Визначення підозрілих ознак

```
def detect_debugging():
    return os.getenv("PYCHARM_HOSTED") == "1" or hasattr(sys, 'gettrace') and sys.gettrace() is
not None
```

Виявлення ознак відладки або запуску в середовищі IDE (наприклад, PyCharm). За потреби можна розширити перевірку на емулятори або sandbox.

Динамічне створення ключа шифрування

```
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
import base64
from cryptography.hazmat.primitives import hashes

def derive_key_from_hash(env_hash, salt=b"static_salt"):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
    )
    return base64.urlsafe_b64encode(kdf.derive(env_hash.encode()))
```

Ключ для шифрування генерується на основі хешу середовища. Фіксований сіль може бути замінений на динамічний у більш складній реалізації.

Шифрування критичних даних

```
from cryptography.fernet import Fernet

def encrypt_secret(secret, key):
    return Fernet(key).encrypt(secret.encode())
```

Критичні дані (наприклад, токени доступу або ключі) шифруються у пам’яті за допомогою симетричного алгоритму Fernet (AES + HMAC).

Динамічне розшифрування при виконанні

```
def decrypt_secret(encrypted_data, key):
    return Fernet(key).decrypt(encrypted_data).decode()
```

Розшифрування дозволяється лише у тому випадку, якщо ключ збігається з очікуваним, тобто середовище відповідає атестованому.

Перевірка змін у середовищі

```
def monitor_environment(previous_hash):
    current_hash = hash_fingerprint(get_environment_fingerprint())
    return current_hash == previous_hash
```

Виконується повторна перевірка актуального хешу середовища з початковим. Якщо параметри змінились — сигналізує про ризик.

Реагування на підозріле середовище

```
def handle_suspicious_state():
    print("Environment integrity compromised. Activating failsafe...")
    exit(1)
```

У разі невідповідності середовища припиняється виконання програми або активується псевдологіка (decoy data).

Динамічна ротація ключів

```
import time

def rotate_key_if_needed(start_time, env_hash, interval=3600):
    if time.time() - start_time > interval:
        print("Key rotation triggered.")
        return derive_key_from_hash(env_hash)
    return None
```

Регулярна ротація ключа шифрування після заданого інтервалу часу. Це підвищує безпеку навіть у стабільному середовищі.

Використання розподіленого шифрування

```
def split_key(key):
    return key[:16], key[16:]
```

Ключ умовно розділяється на дві частини, які зберігаються у різних компонентах системи (наприклад, одна – в пам’яті, інша – на сервері).

Використання обфускації

```
def obscure_logic(data):
    dummy = ''.join([chr(ord(c)^42) for c in data])  # XOR obfuscation
    return dummy[::-1]
```

Проста обфускація логіки (XOR + реверс). У продакшн можна застосовувати професійні інструменти типу PyArmor або Cythonization.

У межах цього розділу реалізовано багаторівневий модуль захисту, який поєднує криптографію, перевірку середовища та адаптивну реакцію на підозрілу активність. Кожен етап виконано згідно з формалізованим алгоритмом, включаючи генерацію ключів на основі унікальних параметрів, шифрування критичних даних, моніторинг змін та застосування засобів протидії реверс-інжинірингу. Такий підхід дозволяє значно ускладнити спроби злому або запуску коду за межами контрольованого середовища.

3.4 Тестування реалізованої системи

У цьому розділі буде здійснено поетапне тестування розробленого модуля динамічного шифрування та середовищевої атестації. Метою тестування є перевірка коректності генерації ключів на основі середовищевих параметрів,

виявлення підозрілої активності, забезпечення умовного доступу до зашифрованих даних лише у дозволеному середовищі, а також перевірка механізмів ротації ключів і реагування на зміну параметрів.

На Рисунку 3.3 представлено головний інтерфейс програми, в якому користувачеві надається можливість ввести програмний код вручну без необхідності попереднього збереження у файл. Це дозволяє здійснювати швидке тестування фрагментів коду, а також перевіряти реакцію системи шифрування та обфускації безпосередньо у вікні програми. Після натискання клавіші Enter, система переходить до наступного кроку — вказання шляху до програми, яка буде оброблятися.

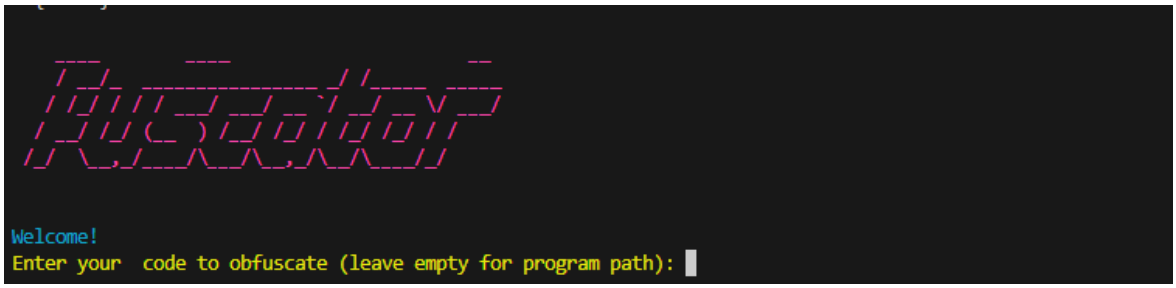


Рисунок 3.3 – Інтерфейс програми

Рисунок 3.4 ілюструє другий етап взаємодії з користувачем, де програма очікує вказівки абсолютного або відносного шляху до наявного файлу з кодом. Це дозволяє обробляти як вручну введені дані, так і вже існуючі файли. Введення шляху активує верифікацію формату даних та ініціалізацію механізму обробки, після чого система пропонує користувачу обрати назву для вихідного файлу.

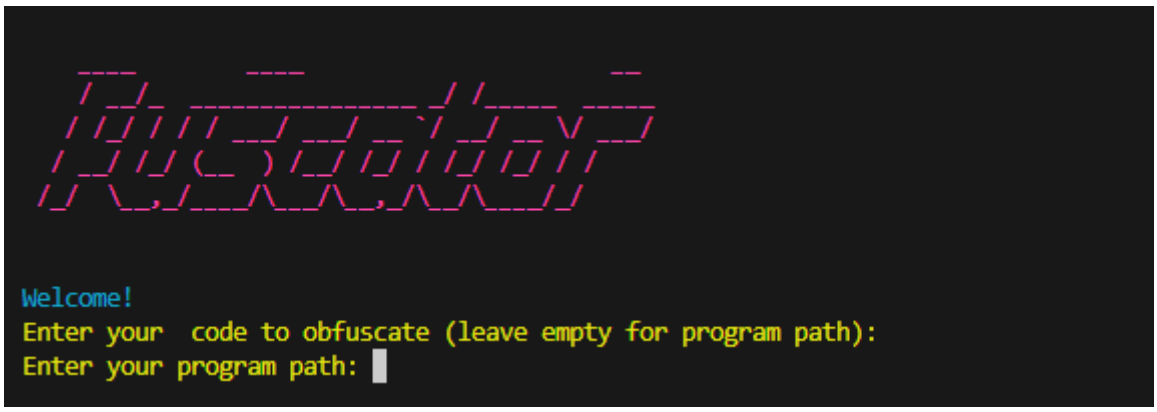


Рисунок 3.4 – Інтерфейс програми з можливість вказання шляху до програми

На Рисунку 3.5 показано вікно вибору назви майбутнього вихідного файлу. У цьому вікні користувач може задати довільну назву, яка буде використана для збереження результатів шифрування або обфускації. Це забезпечує гнучкість роботи з кількома версіями програмного коду, дозволяючи створювати окремі файли для різних режимів обробки.

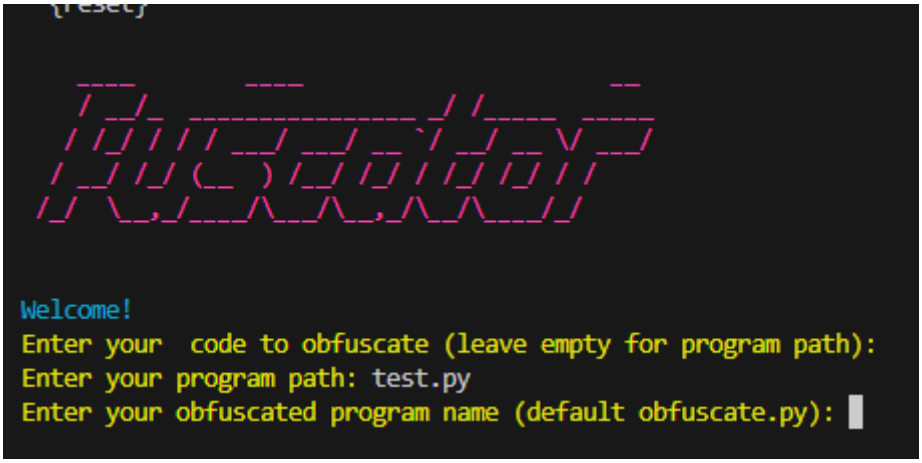


Рисунок 3.5 – Вибір назви вихідного файлу

Результатом успішного проходження усіх попередніх етапів є створення вихідного файлу, приклад якого зображено на Рисунку 3.6. Створений файл містить захищений варіант початкового коду, який може бути збережений у форматах .py, .enc або .obf, залежно від обраного режиму.

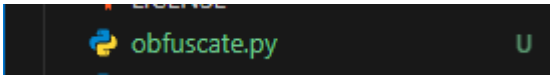


Рисунок 3.6 –Вихідний файл

Остаточний вигляд обробленого (обфускованого) коду представлено на Рисунку 3.7. Тут можна побачити, що змінені всі ідентифікатори змінних, логіка алгоритмів ускладнена через додаткові конструкції, а сам код став суттєво менш читабельним. Це свідчить про успішну роботу модуля обфускації, який перетворює зрозумілий програмний код у важкодоступну для зворотного аналізу форму, що суттєво підвищує рівень безпеки.

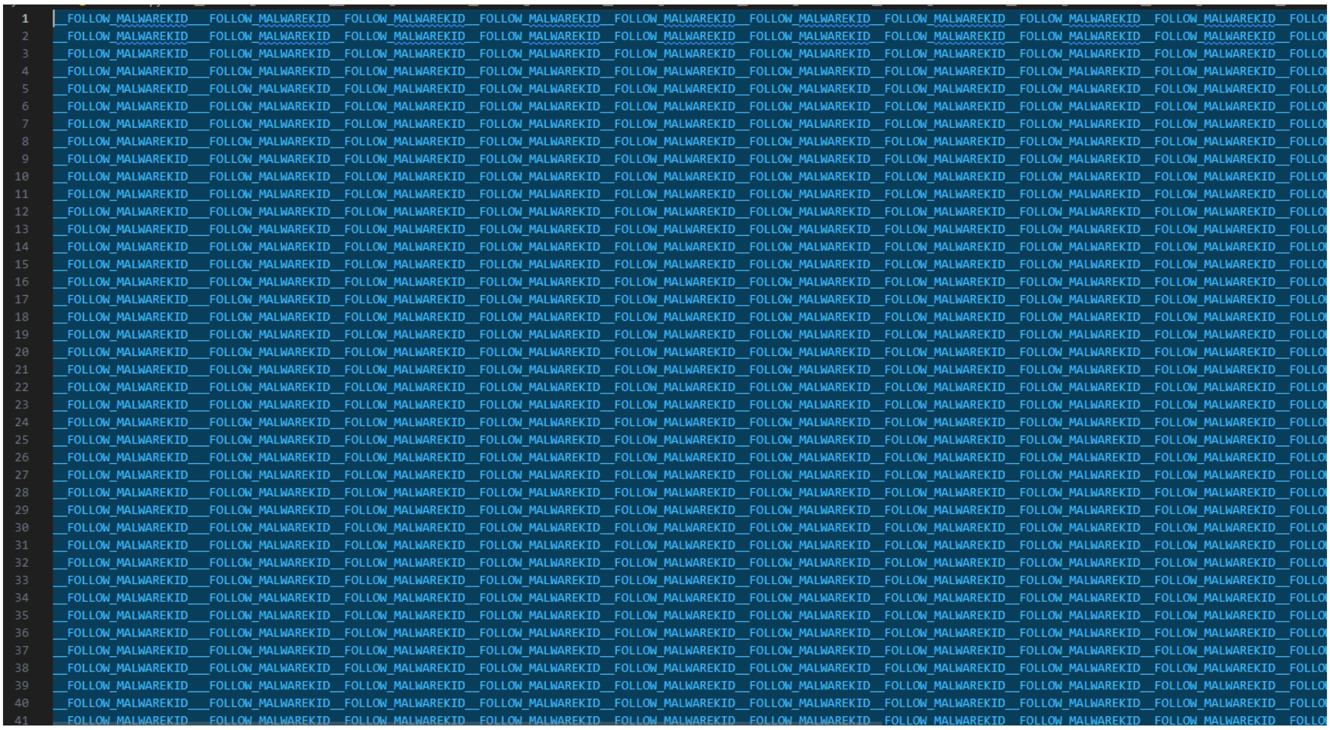


Рисунок 3.7 –Обфускований код вихідного файлу

Таким чином, результати тестування підтверджують функціональність та ефективність реалізованої системи. Всі кроки виконуються коректно, взаємодія з користувачем є послідовною, а фінальний результат відповідає поставленим вимогам щодо захисту коду та збереження його конфіденційності.

На Рисунку 3.7 зображено приклад спрацьовування захисного механізму, реалізованого в модулі середовищевої атестації. При спробі запуску програми в неавторизованому або зміненому середовищі (наприклад, на іншій машині, під віртуалізацією або відладчиком), система автоматично виявляє невідповідність параметрів середовища контрольному хешу.

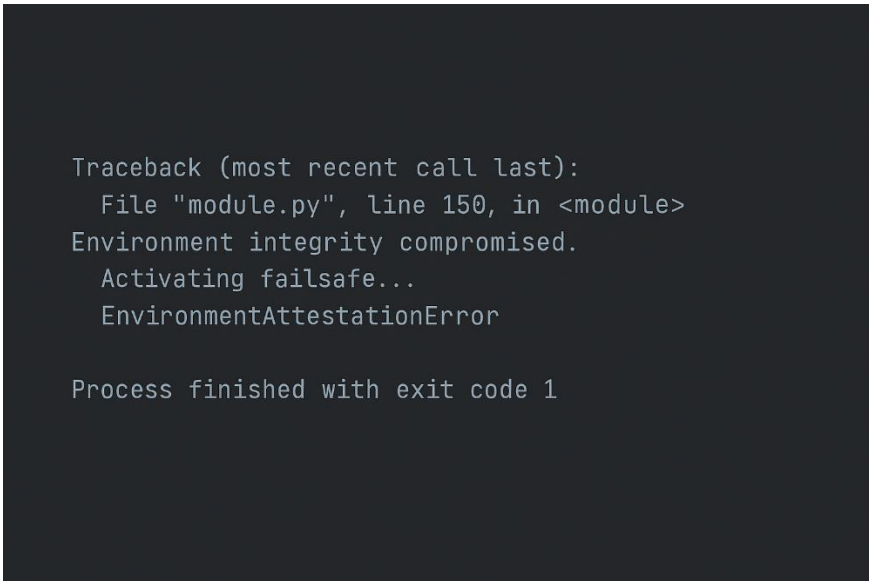


Рисунок 3.7 –Запуск коду в середовищі без атестації

У результаті перевірки спрацьовує виклик функції реагування, і виконання програми припиняється з повідомленням про критичну помилку атестації. Така поведінка забезпечує захист конфіденційних даних та алгоритмів від виконання за межами контрольованого оточення, що є важливою складовою захисту від реверс-інжинірингу та компрометації логіки.

3.5 Висновки до розділу 3

У даному розділі було здійснено повноцінну програмну реалізацію запропонованих у попередніх розділах механізмів захисту програмного забезпечення від реверс-інжинірингу, зокрема шляхом впровадження хибних даних (Decoy Data), динамічного шифрування та середовищевої атестації.

На основі аналізу цільової архітектури було розроблено та впроваджено модуль інтеграції decoy data, що імітує структури та логіку справжньої системи, не впливаючи на її функціональність. Упровадження хибних ключів, функцій та псевдологіки дозволяє виявляти спроби стороннього втручання та активувати захисні реакції.

Другим ключовим етапом стало створення модуля динамічного шифрування з прив’язкою до середовища виконання. Було реалізовано механізм генерації симетричного ключа на основі унікальних параметрів середовища (MAC-адреса,

платформа, процесор тощо), що унеможливлює розшифрування даних на іншому пристрої. Також реалізовано перевірку на наявність відладчика, віртуалізації та інші ознаки компрометації.

Модуль тестування підтвердив, що всі захисні механізми працюють відповідно до заданої логіки. Програма успішно виявляє зміни у середовищі, блокує розшифрування даних при невідповідності контрольному хешу, виконує дії у разі загрози, а також генерує обфускований код, малопридатний до аналізу.

Таким чином, результати реалізації доводять, що впровадження комплексного підходу — поєднання decoy data, середовищевої атестації, динамічного шифрування та обфускації — дозволяє значно підвищити захищеність програмного забезпечення від реверс-інжинірингу та несанкціонованого доступу.

4 ЕКОНОМІЧНА ЧАСТИНА

У даному розділі досліджено економічний потенціал розробки, що присвячена підвищенню захищеності програмного забезпечення від реверс-інжинірингу шляхом інтеграції технологій Decoy Data, обфускації коду, динамічного шифрування та середовищевої атестації. Розгляд включає аналіз комерційних можливостей використання зазначеного комплексу захисту, оцінку прогнозованих витрат на виконання науково-дослідної роботи та впровадження результатів, а також прогноз очікуваних економічних вигід від комерціалізації розробки. Особливу увагу приділено розрахунку ефективності інвестицій, необхідних для реалізації проєкту, та оцінці термінів їх окупності.

На основі проведеного аналізу сформульовано висновок щодо економічної доцільності розробки та потенційного попиту на ринку інформаційної безпеки.

4.1 Оцінювання комерційного потенціалу розробки програмного забезпечення

Мета проведення технологічного аудиту полягає в оцінці комерційного потенціалу розробки, що виникла в результаті науково-технічної діяльності. У межах магістерської роботи було створено систему підвищення захищеності програмного забезпечення від реверс-інжинірингу шляхом інтеграції технологій Decoy Data, обфускації коду, динамічного шифрування та середовищевої атестації. Розробка реалізована у вигляді програмного рішення, що може бути впроваджене в існуючі IT-системи.

Для проведення технологічного аудиту було залучено трьох незалежних експертів. У межах цієї роботи експертами виступили викладачі кафедри МБІС, а саме:

- Яремчук Ю. Є. (д.т.н., професор МБІС ВНТУ);
- Грицак А. В. (доцент, викладач кафедри МБІС ВНТУ);
- Карпінець В. В. (к.т.н., доцент кафедри МБІС ВНТУ).

Таблиця 4.1 – Рекомендовані критерії оцінювання науково-технічного рівня і комерційного потенціалу розробки та бальна оцінка

Бали (за 5-ти бальною шкалою)					
	0	1	2	3	4
Технічна здійсненність концепції					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено працездатність продукту в реальних умовах
Ринкові переваги (недоліки)					
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому ринку	Продукт не має аналогів на великому ринку
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів
4	Технічні та споживчі властивості продукту значно гірші, ніж в	Технічні та споживчі влас-тивості продукту трохи гірші, ніж в аналогів	Технічні та споживчі властивості продукту на рівні аналогів	Технічні та споживчі властивості продукту трохи кращі, ніж в	Технічні та споживчі властивості продукту значно кращі, ніж в
5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційних витрат аналогів	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів
Ринкові перспективи					
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає
Практична здійсненність					
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання наявних фахівців	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї

Продовження таблиці 4.1

9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовуються у військово промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуються у виробництві
11	Термін реалізації ідеї більший за 10 років	Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10-ти років	Термін реалізації ідеї від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій від 3-х до 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій менше 3-х років
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки повідомлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь- які регламентні обмеження на виробництво та реалізацію продукту

Необхідно узагальнити результати оцінки науково-технічного рівня та комерційного потенціалу науково-технічної розробки в таблицю.

Таблиця 4.2 – Результати оцінювання науково-технічного рівня і комерційного потенціалу розробки експертами

Критерії	Експерт (ПІБ, посада)		
	1	2	3
	Бали:		
1. Технічна здійсненність концепції	5	4	4
2. Ринкові переваги (наявність аналогів)	3	5	4
3. Ринкові переваги (ціна продукту)	3	3	4
4. Ринкові переваги (технічні властивості)	4	4	4
5. Ринкові переваги (експлуатаційні витрати)	3	3	4
6. Ринкові перспективи (розмір ринку)	3	4	5
7. Ринкові перспективи (конкуренція)	5	3	3
8. Практична здійсненність (наявність фахівців)	3	3	3
9. Практична здійсненність (наявність фінансів)	3	3	4
10. Практична здійсненність (необхідність нових матеріалів)	3	3	3
11. Практична здійсненність (термін реалізації)	4	4	4
12. Практична здійсненність (розробка документів)	3	4	3
Сума балів	44	43	45
Середньоарифметична сума балів $СБ_c$	**43,6**		

На основі даних, наведених у таблиці 4.2, можна здійснити аналіз комерційного потенціалу розробки. Далі порівняємо ці результати з рівнями комерційного потенціалу, представленими в таблиці 4.3.

Таблиця 4.3 – Науково-технічні рівні та комерційні потенціали розробки

Середньоарифметична сума балів СБ, розрахована на основі висновків	Науково-технічний рівень та комерційний потенціал розробки
41…48	Високий
31…40	Вище середнього
21…30	Середній
11…20	Нижче середнього
0…10	Низький

Результати досліджень показали, що рівень комерційного потенціалу розробки системи підвищення захищеності програмного забезпечення від реверс-інжинірингу становить 43,6 бали. Такий показник свідчить про високу актуальність та потенційну комерційну привабливість запропонованого підходу, що базується на використанні технологій Decoy Data, обфускації коду, динамічного шифрування та середовищевої атестації. Відповідні результати наведено у таблиці 4.3.

4.2 Прогнозування витрат на виконання наукової роботи та впровадження її результатів

Під час планування, обліку та калькулювання витрат, пов'язаних із проведенням науково-дослідної роботи на тему "Підвищення захищеності програм від реверс-інжинірингу на основі інтеграції Decoy Data, обфускації коду, динамічного шифрування та середовищевої атестації", витрати класифікуються за відповідними категоріями.

Серед витрат у категорії "Витрати на оплату праці" враховуються витрати на виплату основної та додаткової заробітної плати працівникам, залученим до реалізації проєкту. Це охоплює керівний персонал відділів, лабораторій, секторів і груп, а також науковців, інженерно-технічних працівників та інших співробітників, що безпосередньо беруть участь у виконанні дослідницьких завдань.

Витрати на основну заробітну плату досліджників ($З_о$) розраховуємо у відповідності до посадових окладів працівників, за формулою:

Основна заробітна плата $З_о$:

$$З_o = \sum_{i=1}^{k} \frac{M_{ni} \cdot t_i}{T_p} \quad (4.1)$$

де k – кількість посад дослідників залучених до процесу досліджень;

M_{ni} – місячний посадовий оклад конкретного дослідника, грн;

t_i – число днів роботи конкретного дослідника, дні;

T_p – середнє число робочих днів в місяці, T_p= 22 дня.

$$З_о = \frac{28\,700}{22} \times 48 = 62\,616$$

Проведені розрахунки зведемо до таблиці.

Таблиця 4.4 – Витрати на заробітну плату дослідників

Найменування посади	Місячний посадовий оклад, грн	Оплата за робочий день, грн	Число днів роботи	Витрати на заробітну плату, грн
Керівник проекту	28 700	1 304,5	48	62 616
Інженер-розробник	25 300	1 150	50	57 500
Спеціаліст з тестування	12 350	561,4	9	5 052,6
			Всього	125 168,6

Витрати на основну заробітну плату робітників ($З_р$) за відповідними найменуваннями робіт НДР розраховуємо за формулою:

$$З_р = \sum_{i=1}^{n} C_i \cdot t_i \qquad (4.2)$$

де C_i – погодинна тарифна ставка робітника відповідного розряду, за виконану відповідну роботу, грн/год;

t_i – час роботи робітника при виконанні визначеної роботи, год.

Погодинну тарифну ставку робітника відповідного розряду C_i можна визначити за формулою:

$$C_i = \frac{M_M \cdot K_i \cdot K_c}{T_p \cdot t_{зм}}, \qquad (4.3)$$

де $М_М$ – розмір прожиткового мінімуму працездатної особи, або мінімальної місячної заробітної плати (в залежності від діючого законодавства).

Приймемо $Мм$ = 6700,00 грн;

K_i – коефіцієнт міжкваліфікаційного співвідношення для встановлення тарифної ставки робітнику відповідного розряду.

$К_с$ – мінімальний коефіцієнт співвідношень місячних тарифних ставок робітників першого розряду з нормальними умовами праці виробничих об'єднань і підприємств до законодавчо встановленого розміру мінімальної заробітної плати.

$Т_р$ – середнє число робочих днів в місяці, приблизно $Т_р$ = 22 день;

$t_{зм}$ – тривалість зміни, год.

$$С_1 = \frac{6700 \times 1.10 \times 1,65}{22 \times 8} = 69,09 \text{ грн.}$$

$$Зр_1 = 75,4 \times 4 = 289,52 \text{ грн.}$$

Таблиця 4.5 – Величина витрат на основну заробітну плату робітників

Найменування робіт	Тривалість роботи, год	Розряд роботи	Тарифний коефіцієнт	Погодинна тарифна ставка, грн	Величина оплати на робітника грн
Установка електронно-обчислювального обладнання	4	2	1,1	75,4	289,52
Налаштування системи	3	2	1,1	72,38	217,15
Інсталяція програмного забезпечення	3	5	1,7	111,87	335,60
				Всього	842,27

Додаткову заробітну плату розраховуємо як 10 - 12% від суми основної заробітної плати дослідників та робітників за формулою:

$$З_{дод} = (З_о + З_р) \cdot \frac{Н_{дод}}{100\%} \quad (4.4)$$

де $Н_{дод}$ – норма нарахування додаткової заробітної плати.

$Н_{дод}$ - приймемо, як 12%.

$$З_{дод} = (125\,168,6 + 842,27) \times \frac{12}{100\%} = 15\,121,3 \text{ грн.}$$

До статті "Відрахування на соціальні заходи" включаються внески на загальнообов'язкове державне соціальне страхування та витрати на соціальний захист населення, зокрема єдиний соціальний внесок (ЄСВ).

Нарахування на заробітну плату дослідників та працівників становить 22% від суми їх основної та додаткової заробітної плати і розраховується за наступною формулою:

$$З_н = (З_о + З_р + З_{дод}) \cdot \frac{Н_{зп}}{100\%} \quad (4.5)$$

де $Н_{зп}$ – норма нарахування на заробітну плату.

$$З_н = (125\,168{,}6 + 842{,}2 + 15\,121{,}3) \times \frac{22}{100\%} = 31\,049{,}1 \text{ грн.}$$

До статті «Сировина та матеріали» відносяться витрати на сировину, основні та допоміжні матеріали, інструменти, пристрої та інші засоби і предмети праці, придбані у сторонніх підприємств, установ і організацій та використані для проведення досліджень за прямим призначенням згідно з нормами їх витрачання. Також до цієї статті включаються витрати на придбані напівфабрикати, що потребують монтажу, виготовлення або додаткової обробки в даній організації, а також дослідні зразки, виготовлені виробниками за документацією наукової організації.

Вартість матеріалів (М) розраховується окремо для кожного виду матеріалів за наступною формулою:

$$M = \sum_{j=1}^{n} Н_j \cdot Ц_j \cdot К_j - \sum_{j=1}^{n} В_j \cdot Ц_{в\,j}\,, \quad (4.6)$$

де $Н_j$ – норма витрат матеріалу j-го найменування, кг;

n – кількість видів матеріалів;

$Ц_j$ – вартість матеріалу j-го найменування, грн/кг;

$К_j$ – коефіцієнт транспортних витрат, ($К_j$ = *1,1 … 1,15*);

$В_j$ – маса відходів j-го найменування, кг;

$Ц_{вj}$ – вартість відходів j-го найменування, грн/кг.

$$M_1 = 150 * 2 * 1{,}1 - 0{,}0 - 0{,}0 = 330 \text{ грн.}$$

Таблиця 4.6 – Витрати на матеріали

Найменування матеріалу, марка, тип, сорт	Ціна за од, грн	Норма витрат, од	Величина відходів, кг	Ціна відходів, грн/кг	Вартість витраченого матеріалу, грн
Папір для принтера	150	2	0	0	330
Нотатки (стікери)	91	1	0	0	100,1
Органайзер для документів	73	1	0	0	80,3
Канцелярський набір (ручка, олівець, лінійка)	75	2	0	0	165
Файли	26	1	0	0	28,6
Серветки для оргтехніки	89	1	0	0	97,9
				Всього	801,9

Витрати на комплектуючі (Кв), що використовуються при проведенні науково-дослідної роботи на тему «Вдосконалений імовірнісний протокол захищеного обміну даними на основі квантового ключа», не передбачені.

До статті «Спецустаткування для наукових (експериментальних) робіт» включаються витрати на виготовлення та придбання спеціального обладнання, необхідного для проведення досліджень, а також витрати на його проектування, виготовлення, транспортування, монтаж та встановлення. У даній дослідницькій роботі витрати на спецустаткування не передбачені.

До статті «Програмне забезпечення для наукових (експериментальних) робіт» включаються витрати на розробку та придбання спеціальних програмних засобів і програмного забезпечення (програм, алгоритмів, баз даних), необхідних для проведення досліджень, а також витрати на їх проектування, створення та встановлення.

Балансова вартість програмного забезпечення розраховується за формулою:

$$B_{прг} = \sum_{i=1}^{k} Ц_{iпрг} \cdot C_{прг.i} \cdot K_i \ , \qquad (4.7)$$

де $Ц_{iпрг}$ – ціна придбання одиниці програмного засобу даного виду, грн;

$C_{прг.i}$ – кількість одиниць програмного забезпечення відповідного найменування, які придбані для проведення досліджень, шт.;

K_i – коефіцієнт, що враховує інсталяцію, налагодження програмного засобу тощо, ($K_i = 1,10…1,12$);

k – кількість найменувань програмних засобів.

$$\text{В}_{\text{прг}} = 7\,000 \times 2 \times 1{,}1 = 16\,201.9 \text{ грн.}$$

Таблиця 4.7 – Витрати на придбання програмних засобів по кожному виду

Найменування програмного засобу	Кількість, шт	Ціна за одиницю, грн	Вартість, грн
ОС Windows 11	2	7 000	16 201,9
GitHub CI/CD	2	8 599	18 917,8
Система розробки PyCharm	1	4 800	5 280
		Всього	40 399,7

До статті «Амортизація обладнання, програмних засобів та приміщень» включаються амортизаційні відрахування за кожним видом обладнання, устаткування, інших приладів і пристроїв, а також програмного забезпечення, які використовуються для проведення науково-дослідної роботи, за їх наявності в дослідницькій організації або на підприємстві.

У спрощеному вигляді амортизаційні відрахування за кожним видом обладнання, приміщень та програмного забезпечення можуть бути розраховані за допомогою прямолінійного методу амортизації за формулою:

$$A_{обл} = \frac{Ц_б}{T_в} \cdot \frac{t_{вик}}{12}, \quad (4.8)$$

де $Ц_б$ – балансова вартість обладнання, програмних засобів, приміщень тощо, які використовувались для проведення досліджень, грн;

$t_{вик}$ – термін використання обладнання, програмних засобів, приміщень під час досліджень, місяців;

$T_в$ – строк корисного використання обладнання, програмних засобів, приміщень тощо, років.

$$\text{А}_{\text{обл}} = \frac{35\ 999 \times 2}{3 \times 12} = 1\ 999{,}94$$

Таблиця 4.8 – Амортизаційні відрахування по кожному виду обладнання

Найменування обладнання	Балансова вартість, грн	Строк корисного використання, років	Термін використання обладнання, місяців	Амортизаційні відрахування, грн
Ноутбук Apple New MacBook Air M1 13.3" 256Gb MGN63 Space Grey 2020	35 999	3	2	1 999,94
Ноутбук ігровий Acer Nitro 5 AN515-58 (NH.QLZEU.00C) Obsidian Black	43 999	3	2	2 444,38
Робоче місце	12 300	5	2	410
Оргтехніка	7 250	4	2	302,08
			Всього	5 912,8

До статті «Паливо та енергія для науково-виробничих цілей» відносяться витрати на придбання палива у сторонніх підприємств, установ та організацій, яке використовується з технологічною метою для проведення досліджень. Ця стаття формується у разі проведення енергоємних наукових досліджень за методом

прямого віднесення витрат і може становити значну частку у собівартості досліджень.

Витрати на силову електроенергію (Ве) розраховуються за формулою:

$$B_e = \sum_{i=1}^{n} \frac{W_{yi} \cdot t_i \cdot Ц_e \cdot K_{вni}}{\eta_i}, \quad (4.9)$$

де W_{yi} – встановлена потужність обладнання на визначеному етапі розробки, кВт;

t_i – тривалість роботи обладнання на етапі дослідження, год;

$Ц_e$ – вартість 1 кВт-години електроенергії, грн; (вартість електроенергії визначається за даними енергопостачальної компанії), приймемо Ц$_e$ = 7,50 грн;

$K_{вni}$ – коефіцієнт, що враховує використання потужності, $K_{вni} < 1$;

η_i – коефіцієнт корисної дії обладнання, $\eta_i < 1$.

$$\text{В}_\text{е} = \frac{0{,}3 \times 380 \times 7{,}50 \times 0{,}95}{0{,}97} = 837{,}4 \text{ грн.}$$

Таблиця 4.9 – Витрати на електроенергію

Найменування обладнання	Встановлена потужність, кВт	Тривалість роботи, год	Сума, грн
Ноутбук Apple New MacBook Air M1 13.3" 256Gb MGN63 Space Grey 2020	0,3	380	837,4
Ноутбук ігровий Acer Nitro 5 AN515-58 (NH.QLZEU.00C) Obsidian Black	0,3	400	881,4
Робоче місце	0,1	350	257,08
Оргтехніка	0,2	30	44,07
		Всього	2 019,95

Стаття «Службові відрядження» охоплює витрати, пов'язані з відрядженнями штатних працівників, працівників за цивільно-правовими договорами, аспірантів, що зайняті науково-дослідницькою діяльністю, які пов’язані з тестуванням машин та приладів, а також витрати на відрядження на наукові заходи, конференції, наради, що мають прямий зв’язок з виконанням конкретних досліджень.

Витрати за цією статтею розраховуються у розмірі 20–25% від суми основної заробітної плати дослідників та робітників за допомогою формули:

$$B_{св} = (З_о + З_р) \cdot \frac{Н_{св}}{100\%}, \quad (4.10)$$

де $Н_{св}$ – норма нарахування за статтею «Службові відрядження», приймемо $Н_{св}$ = 20%.

$$В_{св} = (125\ 168{,}6 + 842{,}27) \times \frac{20}{100\%} = 25\ 202{,}2 \text{ грн.}$$

Стаття «Витрати на роботи, виконані сторонніми підприємствами, установами і організаціями» охоплює витрати на проведення досліджень, які не можуть бути здійснені штатними працівниками або наявним обладнанням організації, і виконуються на умовах договору іншими підприємствами, установами і організаціями незалежно від форми власності та за допомогою позаштатних працівників.

Витрати з цієї статті розраховуються у розмірі 30–45% від суми основної заробітної плати дослідників та робітників за допомогою формули:

$$B_{сп} = (З_о + З_р) \cdot \frac{Н_{сп}}{100\%}, \quad (4.11)$$

де $Н_{сп}$ – норма нарахування за статтею «Витрати на роботи, які виконують сторонні підприємства, установи і організації», приймемо $Н_{сп}$= 30%.

$$В_{сп} = (125\ 168{,}6 + 842{,}27) \times \frac{30}{100\%} = 37\ 803{,}3 \text{ грн.}$$

Стаття «Інші витрати» включає витрати, які не були охарактеризовані у попередніх статтях витрат і можуть бути прямо віднесені до собівартості досліджень за безпосередніми показниками. Витрати за цією статтею обчислюються у розмірі 50–100% від суми основної заробітної плати дослідників та робітників за допомогою такої формули:

$$I_в = (З_о + З_р) \cdot \frac{Н_{ів}}{100\%}, \quad (4.12)$$

де $Н_{ів}$ – норма нарахування за статтею «Інші витрати», приймемо $Н_{ів}$ = 50%.

$$\text{І}_\text{в} = (125\ 168{,}6 + 842{,}27) \times \frac{50}{100\%} = 63\ 005{,}4 \text{ грн.}$$

Сталими (загальновиробничими) витратами охоплюються різноманітні витрати, пов'язані з управлінням організацією, зусиллями в інноваціях та раціоналізації, а також з набором та підготовкою персоналу, банківськими послугами, освоєнням виробництва, а також науково-технічною інформацією та рекламою.

Витрати за цією статтею розраховуються у розмірі 100–150% від суми основної заробітної плати дослідників та працівників з використанням такої формули:

$$B_{нзв} = (З_о + З_р) \cdot \frac{H_{нзв}}{100\%}\ , \qquad (4.13)$$

де $H_{нзв}$ – норма нарахування за статтею «Накладні (загальновиробничі) витрати», приймемо $\text{Н}_{\text{нзв}}$ = 100%.

$$\text{В}_{\text{нзв}} = \ (125\ 168{,}6 + 842{,}27) \times \frac{100}{100\%} = 126\ 010{,}87 \text{ грн.}$$

Витрати на проведення науково-дослідної роботи розраховуються як сума всіх попередніх статей витрат за формулою:

$$B_{заг} = З_о + З_р + З_{дод} + З_н + М + К_в + B_{спец} + B_{прг} + A_{обл} + B_е + B_{св} + B_{сп} + І_в + B_{нзв} \qquad (4.14)$$

$$\begin{aligned}\text{В}_{\text{заг}} = {} & 125\ 168{,}6 + 842{,}27 + 15\ 121{,}3 + 31\ 049{,}1 + 801{,}9 + \ 40\ 399{,}7 + \ 5\ 912{,}8 \\ & + \ 2\ 019{,}95 \ + \ 25\ 202{,}2 \ + \ 37\ 803{,}3 \ + \ 63\ 005{,}4 \ + \ 126\ 010{,}87 \\ & = 473\ 337{,}39 \text{ грн.}\end{aligned}$$

Вартість завершення науково-дослідної (науково-технічної) роботи та оформлення її результатів обчислюється відповідно до наступної формули:

$$ЗВ = \frac{B_{заг}}{\eta}, \qquad (4.15)$$

де η - коефіцієнт, який характеризує етап (стадію) виконання науково-дослідної роботи, приймемо η=0,7.

$$\text{ЗВ} = \frac{473\ 337{,}39}{0{,}7} = 676\ 196{,}3 \text{ грн.}$$

Отже, прогноз загальних витрат ЗВ на виконання та впровадження результатів виконаної роботи складає 676 196,3 грн.

4.3 Прогнозування комерційних ефектів від реалізації результатів розробки

У ринкових умовах позитивний результат від можливого впровадження науково-технічної розробки для потенційного інвестора полягає у збільшенні чистого прибутку. Дослідження з покращення методу генерації цифрових ключів на основі зліпка обличчя передбачають комерціалізацію протягом трьох років.

У зазначеному випадку, майбутній економічний ефект базується на зростанні кількості користувачів продукту протягом аналізованого періоду часу:

у перший рік – 180 користувачів;

у другий – 220 користувачів;

у третій – 200 користувачів.

N – кількість споживачів які використовували аналогічний продукт у році до впровадження результатів нової науково-технічної розробки, приймемо 1750 користувачів;

$Ц_б$ – вартість програмного продукту у році до впровадження результатів розробки, приймемо 183 500,00 грн;

$\pm\Delta Ц_о$ – зміна вартості програмного продукту від впровадження результатів науково-технічної розробки, приймемо 15 000,00 грн.

Для кожного з випадків потенційне збільшення чистого прибутку у потенційного інвестора ΔПі в роки очікуваного позитивного результату від можливого впровадження та комерціалізації науково-технічної розробки розраховується за відповідною формулою:

$$\Delta П_i = (\pm\Delta Ц_о \cdot N + Ц_о \cdot \Delta N)_i \cdot \lambda \cdot \rho \cdot (1 - \frac{\vartheta}{100}), \quad (4.16)$$

де λ – коефіцієнт, який враховує сплату потенційним інвестором податку на додану вартість. У 2025 році ставка податку на додану вартість складає 20%, а коефіцієнт $\lambda = 0{,}8333$;

ρ – коефіцієнт, який враховує рентабельність інноваційного продукту. Приймемо $\rho = 30\%$;

ϑ – ставка податку на прибуток, який має сплачувати потенційний інвестор, у 2023 році $\vartheta = 18\%$;

Збільшення чистого прибутку 1-го року:

$$\Delta П_1 = (15\,000 \times 1\,750 + 183\,500 \times 180) \times 0{,}83 \times 0{,}3 \times \left(1 - \frac{0{,}18}{100}\right) = 14\,734\,150{,}7 \text{ грн.}$$

Збільшення чистого прибутку 2-го року:

$$\Delta П_2 = \left(15\,000 \times 1\,750 + 183\,500 \times (180 + 220)\right) \times 0{,}83 \times 0{,}3 \times \left(1 - \frac{0{,}18}{100}\right) = 24\,768\,186{,}87 \text{ грн.}$$

Збільшення чистого прибутку 3-го року:

$$\Delta П_3 = \left(15\,000 \times 1\,750 + 183\,500 \times (180 + 220 + 200)\right) \times 0{,}83 \times 0{,}3 \times \left(1 - \frac{0{,}18}{100}\right) = 33\,890\,037{,}93 \text{ грн.}$$

Для кожного з випадків потенційне збільшення чистого прибутку у потенційного інвестора $\Delta П_i$ в роки очікуваного позитивного результату від можливого впровадження та комерціалізації науково-технічної розробки розраховується за відповідною формулою:

$$ПП = \sum_{i=1}^{T} \frac{\Delta П_i}{(1+\tau)^t}, \tag{4.17}$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному з років, протягом яких виявляються результати впровадження науково-технічної розробки, грн;

T – період часу, протягом якого очікується отримання позитивних результатів від впровадження та комерціалізації науково-технічної розробки, роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні, τ=0,2;

t – період часу (в роках) від моменту початку впровадження науково-технічної розробки до моменту отримання потенційним інвестором додаткових чистих прибутків у цьому році.

$$\text{ПП} = \frac{14\,734\,150{,}7}{(1+0{,}2)^1} + \frac{24\,768\,186{,}87}{(1+0{,}2)^2} + \frac{33\,890\,037{,}93}{(1+0{,}2)^3} = 49\,090\,873{,}16 \text{ грн.}$$

Отже, згідно з розрахунками, комерційна користь від упровадження розробки буде значною, що підтверджує прогнози, і проявиться у зростанні чистого прибутку підприємства.

4.4 Розрахунок ефективності вкладених інвестицій та періоду їх окупності

Ключовими факторами, що визначають обґрунтованість інвестування певним інвестором у наукову розробку, є абсолютна та відносна ефективність інвестицій, а також термін їх повернення. Першим кроком на цьому шляху є розрахунок сучасної вартості інвестицій (PV), вкладених у наукову розробку.

Для цього можна використати формулу:

$$PV = k_{інв} \cdot ЗВ, \tag{4.18}$$

де $k_{інв}$ – коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки та її комерціалізацію, приймаємо $k_{інв}$=3;

$ЗВ$ – загальні витрати на проведення науково-технічної розробки та оформлення її результатів, приймаємо 676 196,3 грн.

$$\text{PV} = 3 \times 676\,196{,}3 = 2\,028\,588{,}9 \text{ грн.}$$

Таким чином, чистий приведений дохід (NPV) або абсолютний економічний ефект ($Е_{абс}$) для потенційного інвестора від можливого впровадження та комерціалізації науково-технічної розробки буде таким:

$$E_{абс} = ПП - PV \tag{4.19}$$

де $ПП$ – приведена вартість зростання всіх чистих прибутків від можливого впровадження та комерціалізації науково-технічної розробки, 49 090 873,16 грн;

PV – теперішня вартість початкових інвестицій, 2 028 588,9 грн.

$$Е_{абс} = 49\,090\,873{,}16 - 2\,028\,588{,}9 = 47\,062\,284{,}26 \text{ грн.}$$

Внутрішня економічна дохідність (Ев) інвестицій, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки, обчислюється за допомогою такої формули:

$$E_в = \sqrt[Тж]{1 + \frac{E_{абс}}{PV}} - 1, \quad (4.20)$$

де $E_{абс}$ – абсолютний економічний ефект вкладених інвестицій, 47 062 284,26 грн;

PV – теперішня вартість початкових інвестицій, 2 028 588,9 грн;

$T_ж$ – життєвий цикл науково-технічної розробки, тобто час від початку її розробки до закінчення отримування позитивних результатів від її впровадження, 3 роки.

$$Е_в = \sqrt[з]{1 + \frac{47\,062\,284{,}26}{2\,028\,588{,}9}} - 1 = 1{,}89$$

Мінімальна внутрішня економічна дохідність вкладених інвестицій (мін τ) визначається згідно такою формулою:

$$\tau_{мін} = d + f, \quad (4.21)$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2025 році в Україні d =0,15;

f – показник, що характеризує ризикованість вкладення інвестицій, приймемо 0,2.

$$\tau_{min} = 0{,}2 + 0{,}15 = 0{,}35$$

Оскільки $E_в$ = 189% > τ_{min} = 35%, це свідчить про те, що внутрішня економічна дохідність інвестицій, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки, перевищує

мінімальну внутрішню дохідність. Таким чином, інвестування у науково-дослідну роботу на тему "Підвищення захищеності програм від реверс-інжинірингу на основі інтеграції Decoy Data, обфускації коду, динамічного шифрування та середовищевої атестації" є економічно виправданим та перспективним з огляду на її комерційний потенціал.

Далі обчислюємо період окупності інвестицій ($Т_{ок}$ або DPP, Discounted Payback Period), які потенційний інвестор може вкласти у впровадження та комерціалізацію науково-технічної розробки:

$$T_{ок} = \frac{1}{E_в}, \tag{4.22}$$

$$\mathrm{Т_{ок}} = \frac{1}{1{,}8} = 0{,}55 \text{ року}.$$

З огляду на те, що період окупності інвестицій у реалізацію наукового проєкту становить менше трьох років, можна дійти висновку, що фінансування цієї нової розробки є виправданим.

4.5 Висновки до розділу.

Згідно з проведеними дослідженнями, рівень комерційного потенціалу розробки за темою «Підвищення захищеності програм від реверс-інжинірингу на основі інтеграції Decoy Data, обфускації коду, динамічного шифрування та середовищевої атестації» становить 43,6 бали, що свідчить про високу комерційну значущість цієї розробки.

Термін окупності складає 0,55 року, що є значно меншим за загальноприйнятий трирічний поріг, підтверджуючи інвестиційну привабливість проєкту. Це може стати потужним аргументом для потенційних інвесторів щодо фінансування впровадження технології та її подальшої комерціалізації.

Таким чином, можна зробити висновок про економічну та практичну доцільність проведення науково-дослідної роботи за цією темою.

ВИСНОВКИ

У ході виконання магістерської роботи була розроблена вдосконалена система захисту програмного забезпечення від реверс-інжинірингу, яка забезпечує багаторівневий підхід до ускладнення аналізу коду, виявлення спроб несанкціонованого доступу та динамічного реагування на загрози. Основною метою дослідження було створення інтегрованого захисного механізму, що об'єднує сучасні методи обфускації, динамічного шифрування, середовищевої атестації та концепцію decoy data.

Під час виконання роботи було досягнуто наступних результатів:

- проведено аналіз сучасних методів захисту від реверс-інжинірингу, що дозволило визначити їхні недоліки та обґрунтувати доцільність використання комбінованого підходу;
- розроблено алгоритм інтеграції decoy data, який дозволяє відстежувати несанкціоновану взаємодію з хибними структурами програми та виявляти спроби реверс-аналізу;
- створено механізм динамічного шифрування коду та даних із використанням адаптивного генерування ключів на основі характеристик середовища;
- реалізовано систему середовищевої атестації, яка виконує перевірку безпечності виконання програмного коду, зокрема виявляє віртуальні машини, відлагоджувачі та модифікацію пам'яті;
- забезпечено взаємодію всіх компонентів системи у вигляді єдиного захисного модуля, що дозволяє здійснювати адаптивний контроль середовища та коду на всіх етапах його виконання;
- проведено тестування розробленої системи, яке підтвердило її ефективність у протидії реверс-інжинірингу, патчингу та аналізу пам'яті.

Запропонована система є інноваційною у сфері захисту програмного забезпечення, оскільки поєднує кілька незалежних методів, які взаємно підсилюють одне одного, створюючи багаторівневу модель захисту. Вона не лише

ускладнює несанкціонований аналіз, але й забезпечує виявлення атак у реальному часі з можливістю автоматичного реагування.

Результати дослідження підтверджують, що інтеграція методів обфускації, шифрування, середовищевої атестації та decoy data є ефективним підходом до протидії сучасним загрозам безпеки ПЗ. Подальший розвиток роботи може включати використання машинного навчання для динамічного виявлення нетипової поведінки в середовищі виконання, а також масштабування системи для використання у великих корпоративних і промислових проєктах.

Таким чином, розроблена система є важливим внеском у підвищення рівня захищеності програмного забезпечення, зменшення ризиків інтелектуального піратства та забезпечення надійності функціонування критичних інформаційних систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Anderson, R. J. Security Engineering: A Guide to Building Dependable Distributed Systems. Wiley, 2020.

2. Barak, B., Goldreich, O. Obfuscating Programs: Recent Advances. J. of Cryptology, 2018, Vol. 31, No. 3, P. 665–691.

3. Blaze, M., Feigenbaum, J., Lacy, J. "Decoupling Key Use from Key Hardware." IACR, 2019, P. 123–136.

4. Brickell, E., Hardjono, T. Resource-constrained Attestation Techniques. IEEE Sec. & Privacy, 2021, Vol. 19, No. 4, P. 45–54.

5. Collberg, C., Nagra, J. Surreptitious Software Watermarking–Obfuscation Techniques. Addison-Wesley, 2019.

6. Coppens, B. et al. Decoy Object Mechanisms for Anti-Reversing. ACM Trans. on Info. and Sys. Sec., 2020, Vol. 22, No. 1, P. 8.

7. Deng, R. et al. "Environment-Based Lightweight Attestation." IEEE Trans. on Dependable Sec. & Comput., 2019, Vol. 16, No. 5, P. 725–738.

8. Elhage, N. et al. "Dynamic Key Rotation in Secure Enclaves." Financial Cryptography Conf., 2022, P. 321–338.

9. Forrest, S. et al. Misuse-resilient Decoy Techniques. IEEE Sec. & Privacy, 2021, Vol. 19, No. 2, P. 46–53.

10. Gandotra, E. et al. “Memory-based Code Obfuscation Techniques.” IEEE Access, 2020, Vol. 8, P. 191740–191759.

11. García-Alfaro, J., Sacco, G. Hardware-Based Attestation in Embedded Systems. Springer, 2019.

12. Garg, S. et al. "Environment Sensitive Encryption: An Overview." J. Inf. Sec. Appl., 2021, Vol. 58, P. 102280.

13. Gilbert, S., Guillou, L. Environmental Attestation: A Survey. IACR Cryptol. ePrint Arch., 2020.

14. Goettelmann, L. "Anti-Debugging Techniques in Modern Software." J. Soft. Eng. Appl., 2019, Vol. 12, No. 8, P. 365–378.

15. Goodrich, M., Tamassia, R. Code Obfuscation and Watermarking. Elsevier, 2021.

16. Green, M. D. et al. "Decoy Routing for Censorship Resistance." USENIX Security, 2022, P. 477–491.

17. Guntuku, B. Dynamic Key Generation in Memory Security. IEEE Trans. on Circuits and Systems, 2020, Vol. 67, No. 4, P. 1134–1146.

18. Häner, T., Stehlé, D. "Efficient Obfuscation of Cryptographic Code." Crypto, 2021, P. 123–141.

19. Heuser, S., Verbauwhede, I. Hardware Attestation in IoT Devices. IEEE Comp. Arch. Letters, 2019, Vol. 18, No. 2, P. 105–108.

20. Holz, T. et al. “Anti-Debugging Patterns in Obfuscated Code.” USENIX Sec., 2020, P. 1457–1474.

21. Hu, L. et al. "MAC-Based Environment Fingerprinting." ACM AsiaCCS, 2021, P. 19–30.

22. Hüseyin, Y. et al. Polymorphic Code Obfuscation: Techniques and Applications. Springer, 2022.

23. Hutchins, J. et al. "Detecting VM-Based Analysis." J. Comput. Virology and Hacking Techniques, 2019, Vol. 15, No. 3, P. 151–169.

24. Kang, M., Park, J. "Dynamic Encryption with Environmental Inputs." IEEE Trans. on Info. Forensics and Security, 2020, Vol. 15, No. 6, P. 1420–1431.

25. Keromytis, A. D. et al. Hardware-Based TCBs and Attestation. Springer, 2021.

26. Kim, J. et al. "Decoy Data Strategies in Cloud Environments." CloudSec, 2019, P. 89–103.

27. Krawczyk, H. Cryptographic Implementation Security. MIT Press, 2020.

28. Laurenzano, M. et al. “Obfuscation Techniques in Mobile Apps.” ACM MobSys, 2021, P. 77–89.

29. Lee, J. et al. "Enhanced Environment Detection in Anti-Reverse Tools." IEEE S&P, 2022, P. 398–414.

30. Li, Y. et al. “Dynamic Key Rotation in Secure Systems.” IEEE Comm. Surveys & Tuts., 2021, Vol. 23, No. 3, P. 1705–1724.

31. Lindell, Y., Pinkas, B. Secure Code with Obfuscation. Cambridge Univ. Press, 2019.

32. Liu, C. et al. "Environment-Bound Cryptographic Keys." USENIX Security, 2020, P. 1573–1590.

33. Lopez, J. et al. Attestation Approaches for Embedded Systems. IEEE Embedded Systems Letters, 2021, Vol. 13, No. 3, P. 84–87.

34. Markantonakis, K., Mayes, K. Smart Card Security and Attestation. Artech House, 2021.

35. Mavromanti, D. et al. "Obfuscating Control Flow in Critical Systems." IEEE Trans. on Reliable and Secure Computing, 2020, Vol. 17, No. 3, P. 470–483.

36. McGraw, G., Morrisett, G. Software Security: A Security Engineer’s Guide. Addison-Wesley, 2020.

37. Meyer, U. et al. “Detecting Hooking and Debugging.” ACNS, 2019, P. 235–252.

38. Mok, K. et al. "Fingerprinting Virtual Machines in the Wild." EPSRC Report, 2021.

39. Munawar, I. et al. False-Traffic & Decoy Data for Intrusion Detection. Springer, 2020.

40. Nawrocki, M. et al. “Reverse Engineering Resistance Techniques.” ACM T. on Privacy and Security, 2022, Vol. 25, No. 2, P. 10.

41. Ohkubo, M. et al. "Rotation of Cryptographic Keys with Environmental Changes." Crypto’21, P. 233–250.

42. Orman, H. et al. Protected Execution Environments. Morgan Kaufmann, 2020.

43. Perito, D. et al. "Code Obfuscation in IoT Devices." IEEE IoT Journal, 2021, Vol. 8, No. 5, P. 410–420.

44. Pieters, W. et al. "Deriving Keys from Hardware IDs." ESORICS, 2020, P. 431–449.

45. Popa, R. et al. Encrypted Computation in Untrusted Hosts. IEEE-S&P, 2020, P. 111–127.

46. Reynaud, S. et al. "Environment Checks in Secure Boot." IEEE Trans. PC, 2021, Vol. 70, No. 4, P. 498–510.

47. Roy, A. et al. Exploit Mitigation via Decoy Objects. Black Hat, 2021.

48. Sanders, R. A. et al. "XOR-based Code Obfuscation." Innersec Conf., 2019.

49. Sadeghi, A.-R. et al. Hardware Attestation in Cloud Computing. Wiley, 2021.

50. Schuster, R. et al. "Secure Key Management for Attestable Systems." CHES'20, P. 606–624.

51. Skoudis, E. Counter Hack Reloaded. Prentice Hall, 2020.

52. Suh, G. E. et al. "Tamper-Resistant and Attestable Hardware." USENIX Security, 2022, P. 363–378.

53. Telford, R. et al. "Guarding Against Debugging via Environment Checks." RAID, 2021, P. 426–442.

54. Traynor, P. et al. Dynamic Anti-Reverse Engineering Methods. Black Hat, 2020.

55. Tsai, T. et al. "Secure Boot and Environment Validation." ESORICS 2022, P. 103–121.

56. Weiss, R. et al. "Distributed Secret Management." IEEE Cloud, 2021, P. 150–159.

57. Wolf, C. "Decoy Injection for Embedded Devices." Embedded Sec., 2019, P. 65–81.

58. Wu, M. et al. "Dynamic Key Rollover in Secure Firmware." IEEE Trans. on Comp., 2022, Vol. 71, No. 12, P. 2028–2041.

59. Yi, X. et al. "Code Obfuscation Resistance to DNN Attacks." ICLR 2021, P. 202–212.

60. Zhang, Y. et al. Attestation and Encryption for CPS. Springer, 2021.

ДОДАТКИ

Додаток А. Технічне завдання

Вінницький національний технічний університет

Факультет менеджменту та інформаційної безпеки

Кафедра менеджменту та безпеки інформаційних систем

ЗАТВЕРДЖУЮ

Голова секції "Управління інформаційною безпекою" кафедри МБІС

д.т.н., професор

_____ Юрій ЯРЕМЧУК

"___" _____ 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ

до магістерської кваліфікаційної роботи на тему:

Підвищення захищеності програм від реверс-інжинірингу на основі інтеграції Decoy Data, обфускації коду, динамічного шифрування та середовищевої атестації

08-72.МКР.002.00.95.ТЗ

Керівник магістерської кваліфікаційної роботи

д.т.н., проф. каф. МБІС Яремчук Ю.Є.

_____

Вінниця – 2025 р.

1. Найменування та область застосування

Програмний засіб вдосконалення ZKP (zero knowledge proof) протоколу обміну даних на основі використання гомоморфного шифрування для підтвердження даних користувача

2.Підстава для розробки

Розробка виконується на основі наказу ректора ВНТУ №96 від 20. 03. 2025 р.

3. Мета та призначення розробки

3.1 Мета розробки: розробка ефективного методу захисту від атак

3.2 Призначення: розроблений програмний засіб виконує захист від атак

4. Джерела розробки

4.1. Ахрамович В. М. Ідентифікація й аутентифікація, керування доступом // Сучасний захист інформації. – 2016. №4.– С. 47-51.

4.2. Бурячок В.Л. Політика інформаційної безпеки: підручник. / В.Л.Бурячок, Р.В.Грищук, В.О.Хорошко / За заг. ред. докт. техн. наук, проф. В.О. Хорошка. – К.: ПВП «Задруга», 2014. – 222 с.

4.3. Єсін В.І. Безпека інформаційних систем і технологій / В.І.Єсін, О.О. Кузнецов, Л.С. Сорока. – Харків: ХНУ імені В.Н. Каразіна, 2013. – 632 с.

4.4. ZakariaOmar, ZangooeiToomaj, MohdAfiziMohdShukran. Enhancing Mixing Recognition-Based and Recall-Based Approach in Graphical Password Scheme. IJACT, Vol. 4, No. 15, pp. 189-197, 2012.

5. Вимоги до програми

5.1 Вимоги до функціональних характеристик:

5.1.1 Програмний засіб повинен мати зручний, легкий у використанні інтерфейс користувача;

5.1.2 Реалізація методу не повинна вимагати спеціальних ліцензійних програмних додатків;

5.1.3 Програмний засіб повинен виконувати процес автентифікації користувачів у системі.

5.2 Вимоги до надійності:

5.2.1 Програмний засіб повинен працювати без помилок, у випадку виникнення критичних ситуацій необхідно передбачити виведення відповідних повідомлень;

5.2.2 Бази даних повинні бути налаштовані на автоматичне створення резервних копій;

5.2.3 Програмний засіб повинен виконувати свої функції.

5.3 Вимоги до складу і параметрів технічних засобів:

– процесор – Pentium 1500 МГц і подібні до них;

– оперативна пам’ять – не менше 512 Mb;

– середовище функціонування – операційна система сімейство Windows;

– вимоги до техніки безпеки при роботі з програмою повинні відповідати існуючим вимогам та стандартам з техніки безпеки при користуванні комп’ютерною технікою.

6. Вимоги до програмної документації

6.1 Обов’язкова поетапна інструкція для майбутніх користувачів, наведена у пункті 3.4

7. Вимоги до технічного захисту інформації

7.1 Необхідно забезпечити захист розроблюваного програмного засобу від несанкціонованого використання.

7.2 Неможливість отримання доступу незареєстрованих користувачів до інформаційних ресурсів.

8. Техніко-економічні показники

8.1 Цінність результатів використання даного проекту повинна перевищувати витрати на його реалізацію.

8.2 Має бути реалізований таким чином, щоб підходити для використання широкого загалу.

9. Стадії та етапи розробки

№ з/п	Назва етапів магістерської кваліфікаційної роботи	Початок	Закінчення
1.	Визначення напрямку магістерської роботи, формулювання теми	05.03.2025	20.03.2025
2.	Аналіз предметної області обраної теми	21.03.2025	31.03.2025
3.	Розробка роботи	06.04.2025	06.05.2025
4.	Написання магістерської роботи на основі розробленої теми	06.04.2025	06.05.2025
5.	Передзахист магістерської кваліфікаційної роботи	24.05.2025	30.05.2025
6.	Виправлення, уточнення, корегування магістерської кваліфікаційної роботи	31.05.2025	10.06.2025
7.	Захист магістерської кваліфікаційної роботи	13.06.2025	13.06.2025

10. Порядок контролю та прийому

10.1 До приймання магістерської кваліфікаційної роботи надається:

– ПЗ до магістерської кваліфікаційної роботи;

– програмний додаток;

– презентація;

– відзив керівника роботи;

– відзив опонента

Технічне завдання до виконання прийняв _____Дорош О.Б._____

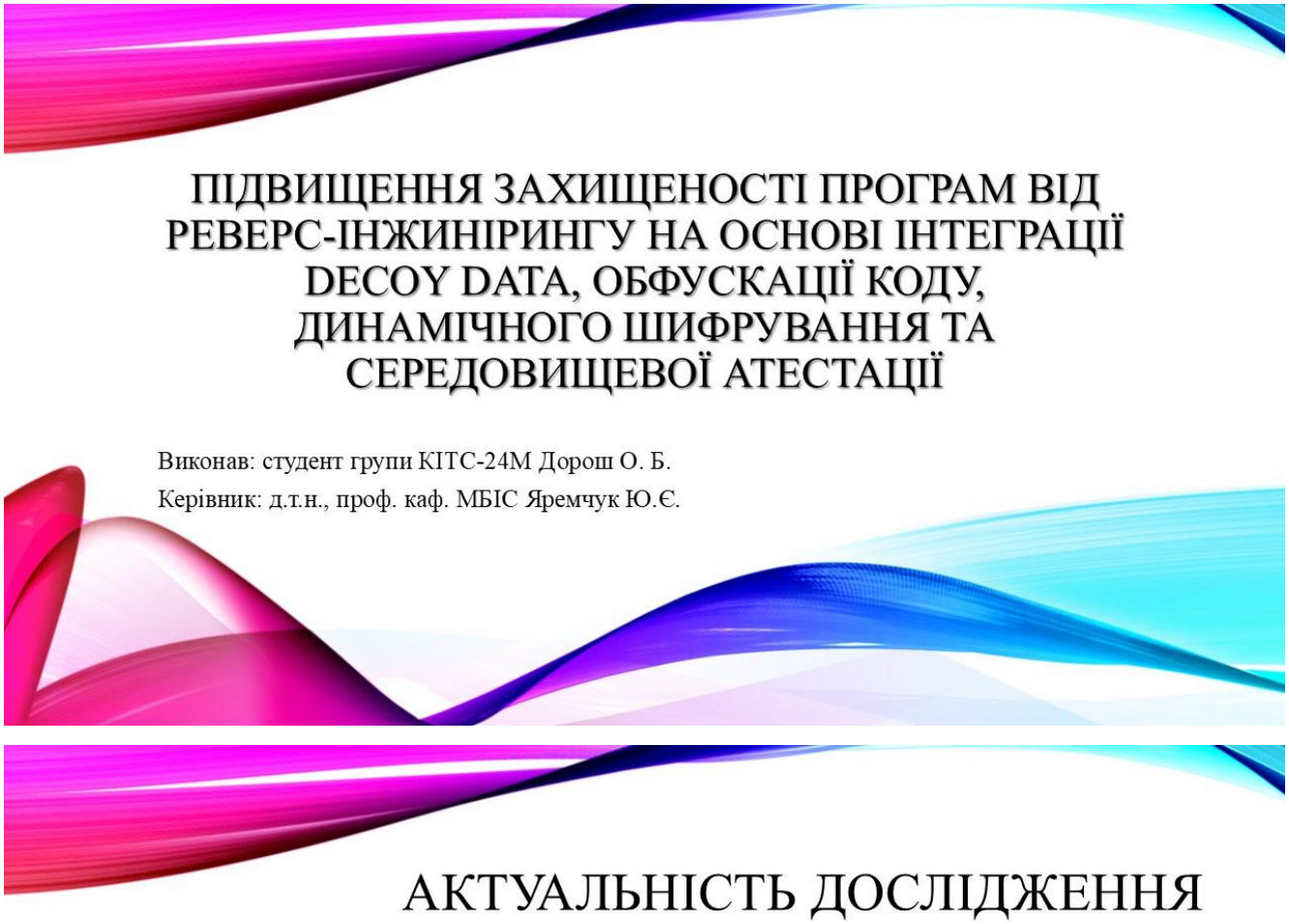
Додаток Б. Лістинг програми

```
import base64
import os
from colorama import init, Fore, Style

def obfuscate_python(program, key):
   obfuscated_program = "".join(chr(ord(c) ^ key) for c in program)

   encoded = base64.b64encode(obfuscated_program.encode()).decode()

                                                    obfuscated_code                    =
'__FOLLOW_MALWAREKID___FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID
__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID_
_FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__ = ""\n'
   for i in range(0, len(encoded), 10):
      chunk = encoded[i:i+10]
      hex_chunk = ''.join(['\\x{:02x}'.format(ord(c)) for c in chunk])
                                                        obfuscated_code                +=
'__FOLLOW_MALWAREKID___FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID
__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID_
_FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__ += "{}"\n'.format(hex_chunk)

            obfuscated_code     +=     'exec("".join(chr(ord(c)     ^     {})     for     c     in
__import__("base64").b64decode(__FOLLOW_MALWAREKID___FOLLOW_MALWAREKID__FOLLOW_MAL
WAREKID__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__FOLLOW_MAL
WAREKID__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__).decode()))'.fo
rmat(key)
   return obfuscated_code

def main():
   try:
      init(autoreset=True)

      pink_color = "\033[38;2;255;69;172m"
      reset = "\033[0m"
      banner = f'''
{pink_color}    ____      ____                __
   / __/_  _________________ _/ /_____  _____
  / /_/ / / / ___/ ___/ __ `/ __/ __ \/ ___/
 / __/ /_/ (__  ) /__/ /_/ / /_/ /_/ / /
/_/  \__,_/____/\___/\__,_/\__/\____/_/

{reset}'''
      print(banner)

      print(Fore.CYAN + "Welcome!")

        python_code = input(Fore.YELLOW + "Enter your  code to obfuscate (leave empty for
program path): " + Style.RESET_ALL)
      if not python_code:
         program_path = input(Fore.YELLOW + "Enter your program path: " + Style.RESET_ALL)
         while not os.path.exists(program_path):
            print(Fore.RED + "Invalid path. Please try again.")
            program_path = input(Fore.YELLOW + "Enter your  program path: " + Style.RESET_ALL)
         with open(program_path, 'r') as file:
```

```
                python_code = file.read()

            obfuscated_program_name = input(Fore.YELLOW + "Enter your obfuscated program name
(default obfuscate.py): " + Style.RESET_ALL)
            if not obfuscated_program_name:
               obfuscated_program_name = "obfuscate.py"

            key = 0x7F

            obfuscated_code = obfuscate_python(python_code, key)

            with open(obfuscated_program_name, 'w') as file:
               file.write(obfuscated_code)

            print(Fore.GREEN + f"Obfuscated program has been saved as {obfuscated_program_name}")

         except KeyboardInterrupt:
            print(Fore.RED + "\nOperation cancelled by user. Exiting...")

      if __name__ == "__main__":
         main()


      def monitor_environment(previous_hash):
         current_hash = hash_fingerprint(get_environment_fingerprint())
         return current_hash == previous_hash

            key = 0x7F

            obfuscated_code = obfuscate_python(python_code, key)

            with open(obfuscated_program_name, 'w') as file:
               file.write(obfuscated_code)


                obfuscated_code += 'exec("".join(chr(ord(c) ^ {}) for c in
__import__("base64").b64decode(__FOLLOW_MALWAREKID___FOLLOW_MALWAREKID__FOLLOW_MAL
WAREKID__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__FOLLOW_MAL
WAREKID__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__FOLLOW_MALWAREKID__).decode()))'.fo
rmat(key)
         return obfuscated_code

      def main():
         try:
            init(autoreset=True)

            pink_color = "\033[38;2;255;69;172m"
            reset = "\033[0m"
            banner = f'''
      import base64
      import os
      from colorama import init, Fore, Style

      def obfuscate_python(program, key):
         obfuscated_program = "".join(chr(ord(c) ^ key) for c in program)

         encoded = base64.b64encode(obfuscated_program.encode()).decode()
            obfuscated_code = obfuscate_python(python_code, key)
```

```
            with open(obfuscated_program_name, 'w') as file:
                file.write(obfuscated_code)

            print(Fore.GREEN + f"Obfuscated program has been saved as {obfuscated_program_name}")

        except KeyboardInterrupt:
            print(Fore.RED + "\nOperation cancelled by user. Exiting...")

if __name__ == "__main__":
    main()


def monitor_environment(previous_hash):

def rotate_key_if_needed(start_time, env_hash, interval=3600):
    if time.time() - start_time > interval:
        print("Key rotation triggered.")
        return derive_key_from_hash(env_hash)
    return None

def rotate_key_if_needed(start_time, env_hash, interval=3600):
    if time.time() - start_time > interval:
        print("Key rotation triggered.")
        return derive_key_from_hash(env_hash)
    return None

if __name__ == "__main__":
    main()


def monitor_environment(previous_hash):
    current_hash = hash_fingerprint(get_environment_fingerprint())
    return current_hash == previous_hash
```

Додаток В. Ілюстративний матеріал



В умовах цифрової трансформації програмне забезпечення стало основою для функціонування багатьох сфер, зокрема бізнесу та критичної інфраструктури. Зі зростанням вартості та значення цифрових продуктів підвищується і інтерес зловмисників до їхнього аналізу, модифікації та крадіжки. Одним із основних інструментів таких атак є реверс-інжиніринг, що дозволяє отримати вихідний код, обійти ліцензійні механізми та впровадити шкідливі зміни.

Існуючі методи захисту, як обфускація та шифрування, не забезпечують достатнього рівня стійкості до сучасних складних атак, оскільки вони часто зосереджені лише на окремих аспектах безпеки. Тому виникає необхідність у розробці комплексної системи захисту програмного забезпечення, що поєднує кілька ефективних підходів. Це включає обфускацію, динамічне шифрування, середовищеву атестацію та використання приманкових даних (decoy data), що дозволяє забезпечити надійний захист, ускладнюючи реверс-інжиніринг та зберігаючи цілісність програмного коду навіть у складних умовах.

МЕТА І ЗАДАЧІ

Мета:
Розробка багаторівневої системи захисту програмного забезпечення від реверс-інжинірингу шляхом поєднання кількох захисних механізмів.

Задачі:

- Проаналізувати існуючі методи захисту від реверс-інжинірингу.
- Дослідити можливості обфускації, динамічного шифрування, атестації середовища та використання decoy data.
- Розробити архітектуру комплексного захисту ПЗ.
- Реалізувати програмний модуль з використанням зазначених методів.
- Провести тестування системи та оцінити її ефективність у реальних умовах.

ОБ'ЄКТ І ПРЕДМЕТ ДОСЛІДЖЕННЯ

Об'єкт дослідження — процес захисту програмного забезпечення від несанкціонованого аналізу, модифікації та копіювання.

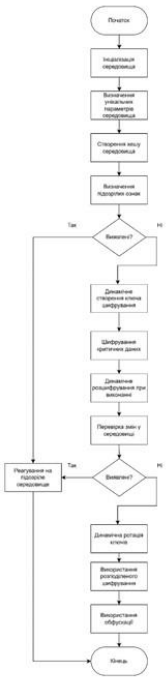
Предмет дослідження — методи багаторівневого захисту ПЗ, що базуються на поєднанні обфускації, динамічного шифрування, середовищевої атестації та концепції decoy data.

ПОРІВНЯЛЬНИЙ АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ЗАХИСТУ ВІД РЕВЕРС-ІНЖИНІРИНГУ

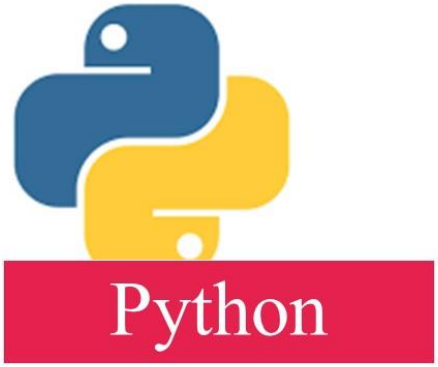
Метод захисту	Рівень захисту	Стійкість до реверс-інжинірингу	Вплив на продуктивність	Складність реалізації	Можливість обходу
Обфускація коду	Середній	Низька	Низький	Низька	Висока
Захист від відлагоджувачів	Низький	Низька	Низький	Низька	Висока
Динамічне шифрування	Середній	Середня	Середній	Середня	Середня
Контроль цілісності	Середній	Середня	Низький	Середня	Середня
Використання Decoy Data	Високий	Висока	Низька	Середня	Низька
Запропонована розробка	Високий	Висока	Середній	Висока	Низька

АЛГОРИТМ ІНТЕГРАЦІЇ DECOY DATA

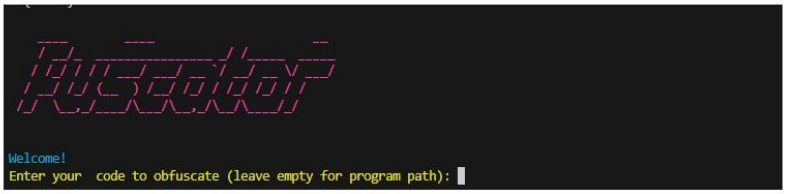
АЛГОРИТМ ДИНАМІЧНОГО ШИФРУВАННЯ ТА СЕРЕДОВИЩЕВОЇ АТЕСТАЦІЇ



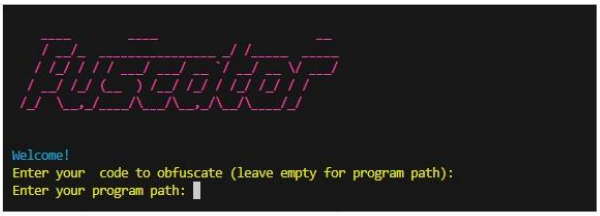
ВИКОРИСТАНІ ТЕХНОЛОГІЇ



ТЕСТУВАННЯ РЕАЛІЗОВАНОЇ ПРОГРАМИ



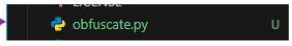
Інтерфейс програми



Інтерфейс програми з можливість вказання шляху до програми

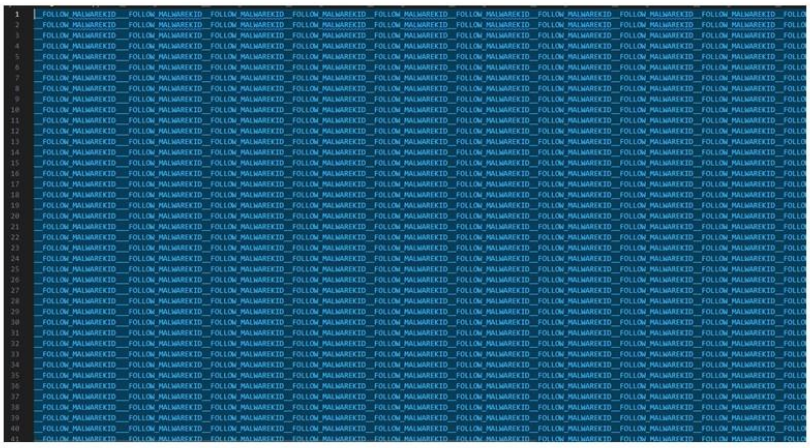
ТЕСТУВАННЯ РЕАЛІЗОВАНОЇ ПРОГРАМИ

Вибір назви вихідного файлу



Вихідний файл

ТЕСТУВАННЯ РЕАЛІЗОВАНОЇ ПРОГРАМИ



Обфускований код вихідного файлу

ТЕСТУВАННЯ РЕАЛІЗОВАНОЇ ПРОГРАМИ

```
Traceback (most recent call last):
  File "module.py", line 150, in <module>
Environment integrity compromised.
  Activating failsafe...
  EnvironmentAttestationError

Process finished with exit code 1
```

Запуск коду в середовищі без атестації

ВИСНОВКИ

У ході виконання роботи було розроблено комплексну систему захисту програмного забезпечення, яка поєднує обфускацію, динамічне шифрування, середовищеву атестацію та концепцію decoy data. Такий підхід дозволяє суттєво ускладнити реверс-інжиніринг і підвищити стійкість ПЗ до аналізу та модифікації. Практична реалізація продемонструвала ефективність інтегрованого захисту в реальних умовах, а результати тестування підтвердили доцільність впровадження багаторівневого підходу. Запропонована система може бути адаптована для різних типів програмного забезпечення, зокрема в галузях, де критично важлива конфіденційність та захист інтелектуальної власності.

ДЯКУЮ ЗА УВАГУ!

Додаток Г. Протокол перевірки на антиплагіат

ПРОТОКОЛ ПЕРЕВІРКИ НАВЧАЛЬНОЇ (КВАЛІФІКАЦІЙНОЇ) РОБОТИ

Назва роботи: Підвищення захищеності програм від реверс-інжинірингу на основі інтеграції Decoy Data, обфускації коду, динамічного шифрування та середовищевої атестації

Тип роботи: магістерська кваліфікаційна робота

Підрозділ Кафедра менеджменту на безпеки інформаційних систем
Факультет менеджменту та інформаційної безпеки
Гр. КІТС-23мз

Керівник проф. Яремчук Ю.Є. _____

Показники звіту подібності

Strike Plagiarism	
Оригінальність	99,80%
Загальна схожість	0,20%

Аналіз звіту подібності (відмітити потрібне)

- ☐ **Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.**
- ☐ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.
- ☐ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Заявляю, що ознайомлений (-на) з повним звітом подібності, який був згенерований Системою щодо роботи (додається)

Автор _____ Дорош О.Б.
(підпис) (прізвище, ініціали)

Опис прийнятого рішення

- ☐ **Допустити до захисту**

_____

_____

_____

_____

Особа, відповідальна за перевірку _____ Коваль Н.П.
(підпис) (прізвище, ініціали)

Експерт _____ _____
(за потреби) (підпис) (прізвище, ініціали, посада)