МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут автоматики і електротехніки

Кафедра комп'ютерних технологій та інформаційної безпеки

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних технологій та інформаційної безпеки, к.т.н., доцент
_____ С.М. Нужний
«10» _____12_____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА
АНАЛІЗ МЕТОДІВ ЗАХИСТУ ПРОГРАМНИХ ПРОДУКТІВ ПІД ЧАС РОЗРОБКИ

Ступінь вищої освіти: магістр
Галузь знань: 14 Електрична інженерія
Спеціальність: 141 «Електроенергетика, електротехніка та електромеханіка»
Освітньо-професійна програма: Системи технічного захисту інформації, автоматизація її обробки

Виконав: студент 6 курсу групи 6366м
Щуплов Олександр Віталійович _____

Керівник:
к.т.н., доцент, доцент кафедри комп'ютерних технологій та інформаційної безпеки

Козирєв Сергій Сергійович _____

Миколаїв – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут автоматики і електротехніки

ЗАТВЕРДЖУЮ
Гарант освітньої програми,
д.т.н., професор,
професор кафедри КТІБ
_____ Передерій В.І.
«14» 10 2025 р.

ЗАВДАННЯ

на кваліфікаційну роботу

Студент: Щуплов Олександр Віталійович

Курс:6

Група: 6366м

Ступінь вищої освіти: магістр

Галузь знань: 14 Електрична інженерія

Спеціальність: 141 «Електроенергетика, електротехніка та електромеханіка»

Освітньо-професійна програма: Системи технічного захисту інформації, автоматизація її обробки

1. Тема роботи: Аналіз методів захисту програмних продуктів під час розробки

затверджена наказом НУК від «14» 10 2025 р. № 1217-с

2. Вихідні дані до роботи: нормативно-правова база і теоретичні основи побудови захищених комп'ютерних систем і систем підтримки прийняття рішень; вихідний код програмного продукту, що потребує захисту від несанкціонованого використання та зворотної розробки.

провести аналіз відкритих джерел та визначити методи захисту програмних продуктів під час розробки; визначити параметри наявних на ринку засобів захисту; розробити власний програмний комплекс для захисту програмного коду від несанкціонованого копіювання виконуваного коду; провести тестування на порівняння з існуючими продуктами

4. Терміни виконання завдання:
термін видачі завдання: «01- 05» вересня 2025 р.
термін подання роботи: «18» грудня 2025 р.

6. План-графік виконання кваліфікаційної роботи

п/п	Заходи	Дата		Готовність
		Навч. тиждень	Контрольна дата	
1.	Наукове стажування	1 - 8	27.10.2025	Звіт з наукового стажування
2.	Організаційні збори	9	29.10.2025	Попередній варіант змісту КР
3.	Рубіжний контроль	10	05.11.2025	Попередній варіант оглядових розділів, звіт з практика
4.	Рубіжний контроль	13	27-28.11.2025	Доповідь на 13-ій Всеукраїнській науково-технічній конференції з міжнародною участю «СПІБТ–2025»
5.	Рубіжний контроль	14	3.12.2025	1. Попередній варіант повної КР 2. Попередній варіант презентації
6.	КЕ на рівень «магістра»	15	8,9.12.2025	Складання державного екзамену зі спеціальності на ступінь магістра
7.	Перевірка на плагіат	15	10.12.2025	Електронний варіант кваліфікаційної роботи, попередній захист (робота передається студентом на кафедру для внутрішньої рецензії).
8.	Оформлення КР та супутньої документації	16	15-17.12.2025	1. Оформлена КР (в форматі pdf) У разі отримання позитивної внутрішньої рецензії, КР подається на зовнішню рецензію. 2. Оформлена презентація (в форматі pdf).
9.	Подання документів на захист	16	18.12.2025	1. Подання щодо захисту КР: тема, успішність, висновок керівника та висновок кафедри про КР. 2. Зовнішня рецензія (окремо від КР). Резюме на КР. 3. Електронний варіант презентації. 4. Електронний варіант КР на диску
10.	Захист ДР	17	22,23,24 12.2025	1. Приєднання студента до засідання кваліфікаційної комісії (конференції) у встановлений час для проведення процедури дистанційного захисту 2. Процедура захисту КР.

Керівник
к.т.н., доцент, доцент кафедри комп'ютерних технологій
та інформаційної безпеки, _____ С.С. Козирєв

Студент _____ О.В. Щуплов

АНОТАЦІЯ

Щуплов О.В. Аналіз методу захисту програмних продуктів під час розробки.

Кваліфікаційна робота на здобуття ступеня вищої освіти «магістр» за спеціальністю 141 «Електроенергетика, електротехніка та електромеханіка» галузі знань 12 «Інформаційні технології». – Національний університет кораблебудування імені адмірала Макарова, Миколаїв, 2025.

Пояснювальна записка: 74 с., 13 рис., 4 таблиці, 40 посилань.

У магістерській кваліфікаційній роботі виконується розробка програмного комплексу захисту інтелектуальної власності, а саме коду сервера з бізнес логікою, шляхом шифрування і подальшого запуску шифрованого файлу виконаного файлу. Основна мета проекту полягає в підвищенні рівня захищеності програмних продуктів, розроблених на платформі *Node.js*, шляхом створення комплексної системи захисту від несанкціонованого копіювання, реверс-інжинірингу та модифікації коду.

Проект включає в себе аналіз поточних проблем захисту програмних продуктів під час розробки, аналіз готових рішень, вибір варіанту реалізації задуму кваліфікаційної роботи, а також розробку та тестування програмного комплексу. Результатом дипломної роботи є створений програмний продукт, що перешкоджає викраденню інтелектуальної власності бізнесу чи організацій.

Висновки проекту підсумовують проведену роботу.

Ключові слова: розробка, сервер, бізнес-логіка, *Express*, *Node.js*, *AES*, *obfuscation*, *shim*

SUMMARY

O.V. Shchuplov. Analysis of the method of software product protection during development.

Master's qualification work for the Higher Education Degree "Master" in Specialty 141 "Electric Power Engineering, Electrical Engineering and Electromechanics" within the Field of Knowledge 12 "Information Technologies". – Admiral Makarov National University of Shipbuilding, Mykolaiv, 2025.

Explanatory note: 74 p., 13 fig., 4 tables, 40 references.

The Master's thesis involves the development of a software complex for intellectual property protection, specifically server-side code containing business logic, through encryption and the subsequent execution of the encrypted file. The main objective of the project is to enhance the security level of software products developed on the Node.js platform by creating a comprehensive protection system against unauthorized copying, reverse engineering, and code modification.

The project includes an analysis of current software protection issues during the development phase, an analysis of existing solutions, the selection of an implementation approach for the thesis concept, as well as the development and testing of the software complex. The result of the thesis is a developed software product that prevents the theft of intellectual property from businesses or organizations.

The project conclusions summarize the work performed.

Keywords: development, server, business logic, Express, Node.js, AES, obfuscation, shim.

ЗМІСТ

ВСТУП

Актуальність теми. Стрімка цифровізація бізнес-процесів та перехід до хмарних обчислень зумовили домінування веб-технологій у сучасному ландшафті розробки програмного забезпечення (ПЗ). Платформи на базі інтерпретованих мов програмування, зокрема *Node.js* (*JavaScript*), стали стандартом для побудови серверних рішень (*Backend*), мікросервісів та API. Проте, на відміну від компільованих мов (*C++, Go, Rust*), де кінцевий продукт є бінарним файлом, специфіка скриптових мов передбачає розповсюдження продукту у вигляді відкритого вихідного коду (*Source Code*).

Це створює критичні загрози для інтелектуальної власності розробників. Доступність вихідного тексту дозволяє зловмисникам проводити реверс-інжиніринг, аналізувати бізнес-логіку, виявляти вразливості, модифікувати код (Tampering) та створювати неліцензійні копії продуктів. Існуючі методи захисту, такі як мініфікація або базова обфускація, є недостатньо ефективними проти сучасних методів статичного та динамічного аналізу (*AST*-аналізатори, налагоджувачі).

В умовах зростання кількості атак на ланцюжок постачання (*Supply Chain Attacks*) та вимог стандартів безпеки (*ISO/IEC 27001, OWASP ASVS*), виникає необхідність у створенні комплексних систем захисту, які поєднують криптографічну стійкість, заплутування логіки та безпечне середовище виконання. Розробка методу безфайлового виконання (*In-Memory Execution*) зашифрованого коду є перспективним напрямком, що дозволяє нівелювати загрози криміналістичного аналізу файлової системи.

Зв’язок роботи з науковими програмами, планами, темами. Робота виконана відповідно до плану науково-дослідних робіт кафедри комп’ютерних технологій та інформаційної безпеки НУК ім. адмірала Макарова за напрямом за-

безпечення безпеки інформаційних ресурсів та захисту програмного забезпечення. [1]

Метою роботи є підвищення рівня захищеності програмних продуктів на платформі *Node.js* шляхом розробки комплексної системи, що поєднує автентифіковане шифрування, динамічну обфускацію та безфайлове виконання коду. Для досягнення мети необхідно вирішити такі задачі:

– Провести аналіз методів реверс-інжинірингу інтерпретованого коду та виявити недоліки існуючих засобів захисту (обфускаторів, пакувальників).

– Обґрунтувати вибір криптографічних примітивів для забезпечення конфіденційності та цілісності коду (*AES-GCM, PBKDF2*).

– Розробити алгоритми «розумної» обфускації, що зберігають працездатність синтаксису мови, та метод розділеного зберігання метаданих (карт відновлення).

– Спроєктувати та програмно реалізувати систему захисту, що забезпечує виконання коду безпосередньо в оперативній пам’яті (без запису на диск).

– Оцінити ефективність системи за критеріями ентропії, стійкості до модифікації та впливу на продуктивність.

Об’єктом дослідження є процеси захисту програмного забезпечення від несанкціонованого аналізу та модифікації.

Предметом дослідження є методи обфускації, криптографічного захисту та безпечного виконання програмного коду в середовищі *Node.js*.

Методи дослідження. У роботі використано: методи системного аналізу (для класифікації загроз); методи криптографії (для реалізації алгоритмів *AES-GCM, PBKDF2, SHA-256);* методи теорії графів (для аналізу потоку керування при обфускації); методи експериментального дослідження (для оцінки продуктивності та ентропії).

Наукова новизна полягає у вдосконаленні методу захисту скриптового коду, який, на відміну від існуючих, поєднує криптографічну перевірку цілісно-

сті з технологією динамічної ін'єкції контексту, що дозволяє виконувати складні серверні додатки виключно в оперативній пам'яті, унеможливлюючи відновлення коду з файлової системи.

Практичне значення. Розроблено програмний комплекс, який дозволяє автоматизувати процес захисту комерційних *Node.js*-додатків. Результати роботи можуть бути використані компаніями-розробниками для захисту інтелектуальної власності при розгортанні ПЗ на серверах замовника.

1 АНАЛІЗ ПРОБЛЕМИ ЗАХИСТУ ПРОГРАМНИХ ПРОДУКТІВ ВІД НЕСАНКЦІОНОВАНОГО АНАЛІЗУ ТА КОПІЮВАННЯ

1.1 Аналіз загроз безпеці ПЗ на етапі розробки та експлуатації

Сучасна екосистема розробки *Enterprise*-рівня все частіше спирається на використання інтерпретованих мов. Платформа *Node.js*, завдяки своїй асинхронній природі та використанню рушія *V8*, стала стандартом для побудови мікросервісів. Однак, архітектурна особливість *Node.js* полягає в тому, що середовище виконання оперує вихідним текстом програми. Це формує унікальний вектор загроз, відмінний від світу компільованого ПЗ (*C++, Java*). [2]

– Основні загрози інформаційній безпеці продукту можна класифікувати за моделлю *STRIDE* , адаптованою до специфіки скриптових мов [3]:

– Крадіжка інтелектуальної власності. Оскільки код на сервері зберігається у відкритому або легко відновлюваному форматі, недобросовісний адміністратор, конкурент або зловмисник, що отримав доступ до файлової системи (через вразливості *LFI – Local File Inclusion*), може скопіювати весь проект. Це призводить до фінансових збитків та втрати конкурентних переваг.

– Реверс-інжиніринг. Навіть без повного копіювання, аналіз алгоритмів роботи ПЗ дозволяє виявити слабкі місця в логіці . Для *JavaScript* існують потужні інструменти демініфікації та форматування, які перетворюють компактний код у читабельний вигляд, дозволяючи відновити структуру класів та функцій.

– Атаки на ланцюжок постачання. [4] Проекти *Node.js* залежать від тисяч пакетів з реєстру *npm*. Зловмисники можуть модифікувати код легітимної бібліотеки або використовувати *Typosquatting*. Якщо основний код продукту не захищений механізмами контролю цілісності, ін'єкція шкідливого коду пройде непоміченою.

– Модифікація коду. Зміна логіки роботи програми з метою обходу ліцензійних обмежень або впровадження бекдорів. У скриптових мовах це реалізується тривіально: достатньо змінити умовний оператор *if (license.isValid)* на *if (true)*.

– Витік секретів. Розробники часто залишають у коді *API*-ключі, токени доступу до БД або ключі шифрування. При доступі до коду ці дані стають скомпрометованими. [5]

1.2 Нормативно-правове регулювання захисту інтелектуальної власності в сфері розробки ПЗ

У сучасному цифровому просторі програмне забезпечення є специфічним об'єктом інтелектуальної власності, захист якого лежить на перетині авторського права, патентного права та інституту комерційної таємниці. Для серверних рішень на базі *Node.js*, де вихідний код часто є носієм унікальних бізнес-алгоритмів, питання правової охорони набуває критичного значення. [6]

Згідно з положеннями Бернської конвенції про охорону літературних і художніх творів та Договору ВОІВ (Всесвітньої організації інтелектуальної власності), комп'ютерні програми охороняються як літературні твори. Однак, цей підхід має суттєвий недолік: авторське право захищає лише форму вираження (конкретний текст коду), але не ідею чи алгоритм, що лежить в його основі. Це створює юридичну лазівку: конкурент може вивчити незахищений код, зрозуміти принцип його роботи та переписати його іншими синтаксичними конструкціями *(«clean room implementation»*), не порушуючи формально авторських прав. [7]

Саме тому в індустрії *Enterprise*-розробки основним механізмом захисту виступає режим комерційної таємниці (*Trade Secret*). Відповідно до Директиви ЄС 2016/943 [8] та законодавства США (*Defend Trade Secrets Act*) [9], інформа-

ція вважається комерційною таємницею, якщо вона має комерційну цінність через свою невідомість третім особам, і власник вживає «розумних заходів» для збереження її конфіденційності.

У цьому контексті застосування технічних засобів захисту, таких як обфускація та шифрування коду, набуває юридичної ваги. Якщо розробник передає замовнику або розгортає на хостингу відкритий код (*Plain Text*), суд може визнати, що власник не вжив «розумних заходів» для захисту, що фактично анулює статус комерційної таємниці. Таким чином, впровадження системи захисту, подібної до розробленої в даній роботі, є не лише технічною, але й правовою необхідністю для збереження інтелектуальних активів компанії.

1.3 Огляд міжнародних стандартів інформаційної безпеки та вимог до захисту коду

Розробка захищеного програмного забезпечення регламентується низкою міжнародних стандартів, які визначають вимоги до управління життєвим циклом розробки (SDLC) та захисту активів. Аналіз цих документів дозволяє формалізувати вимоги до системи, що розробляється.

Стандарти сімейства *ISO/IEC 27000* Ключовим документом є міжнародний стандарт *ISO/IEC 27001:2013* («Системи управління інформаційною безпекою»). [10] У додатку «А» цього стандарту визначено контролі, безпосередньо пов'язані з темою дослідження:

– Контроль A.9.4.5 (*Access control to program source code*): Вимагає суворого обмеження доступу до вихідного коду програм. Оскільки в інтерпретованих мовах (*JavaScript/Node.js*) виконуваний файл є фактично вихідним кодом, виконання цієї вимоги без застосування криптографічного пакування на продуктовому сервері є неможливим.

– Контроль A.14.1.2 (*Securing application services on public networks*): Вимагає захисту інформації від розкриття та модифікації при передачі мережами. Хоча це зазвичай стосується трафіку (*HTTPS*), в контексті мікросервісної архітектури передача коду на виконання також підпадає під цю категорію.

– Контроль A.10.1.1 (*Policy on the use of cryptographic controls*): Регламентує використання шифрування для захисту інформації у стані спокою (*At Rest*), що безпосередньо реалізується в даній роботі через алгоритм *AES-256*.

Методологія *OWASP* (*Open Web Application Security Project*) [11] - стандарт перевірки безпеки додатків (*OWASP ASVS 4.0*) містить окремі вимоги щодо захисту інтелектуальної власності та протидії реверс-інжинірингу, які часто ігноруються при розробці веб-сервісів:

– Вимога *V8.3.1*: Вимагає від розробників видалення будь-яких коментарів, налагоджувального коду та непотрібних метаданих перед розгортанням. Обфускація автоматизує цей процес.

– Вимога *V1.14.6* (*Configuration*): Рекомендує використовувати бінарні формати або скомпільований код замість текстових скриптів у виробничому середовищі для зменшення поверхні атаки.

Стандарти *NIST* (*National Institute of Standards and Technology*) Спеціальна публікація *NIST SP 800-53 (Rev. 5) «Security and Privacy Controls for Information Systems»* є обов'язковою для федеральних систем США, але використовується як еталон у всьому світі. [12]

Сімейство контролів *SA-11 (Developer Security Testing and Evaluation)*: Вимагає підтвердження відсутності в коді «логічних бомб», бекдорів та інших недокументованих можливостей. Відкритий код *Node.js* вразливий до ін'єкцій у ланцюжок постачання (*Supply Chain Attacks*). Запропоноване в роботі рішення з контролем цілісності (через *GCM*-тег) відповідає вимогам розділу *SI-7* (*Software, Firmware, and Information Integrity*), який вимагає використання криптографіч-

них механізмів для виявлення несанкціонованих змін у програмному забезпеченні.

Таким чином, аналіз нормативної бази свідчить, що існуючі практики розгортання *Node.js* додатків у відкритому вигляді суперечать вимогам сучасних стандартів безпеки високого рівня (*EAL3+* за *Common Criteria*), що актуалізує необхідність створення нових засобів захисту.

1.4 Методологія реверс-інжинірингу та деобфускації

Розуміння механізмів захисту неможливе без глибокого аналізу методів атаки. У контексті *JavaScript* та *Node.js* зловмисники використовують специфічний набір інструментів та методологій, спрямованих на відновлення вихідного коду.

1. Статичний аналіз. Це дослідження коду без його запуску. Зловмисники використовують парсери (наприклад, *Esprima, Acorn*), які перетворюють код у Абстрактне Синтаксичне Дерево (*AST*). *AST* – це деревоподібне представлення синтаксичної структури вихідного коду. Деобфускатори (наприклад, *JSNice*, *Synchrony*) працюють з *AST*, виконуючи такі операції:

– Dead Code Elimination: Видалення гілок коду, які ніколи не виконуються.

– Statistical Renaming: Використання моделей машинного навчання для передбачення оригінальних імен змінних на основі контексту їх використання.

2. Динамічний аналіз (*Dynamic Analysis*)

Якщо застосування методів статичного аналізу унеможливлене внаслідок використання криптографічного пакування або складної обфускації, зловмисники переходять до динамічного аналізу. Цей підхід передбачає дослідження поведінки програмного забезпечення безпосередньо під час його виконання у контрольованому середовищі (*Sandbox*) або з використанням спеціалізованих

інструментів налагодження.*Runtime Hooking*: Перевизначення базових функцій мови (наприклад, *eval, Function, String*) для перехоплення розшифрованого коду в момент його виконання.

У контексті платформи *Node.js* динамічний аналіз є особливо ефективним, оскільки інтерпретована природа мови дозволяє втручатися в процес виконання "на льоту". Основні вектори атак включають:

1. Перехоплення викликів та модифікація середовища (*Runtime Hooking / Monkey Patching*). *JavaScript* є мовою з високим ступенем рефлексії та динамічною типізацією, що дозволяє перевизначати (переписувати) стандартні методи об'єктів та функції глобального контексту. Зловмисник може впровадити власний код, який "обгортає" критичні системні функції.

Механізм атаки: Перевизначення конструктора *Function,* методу *eval()* або модулів віртуалізації *vm.runInContext*.

Мета: Оскільки будь-яка захищена програма рано чи пізно повинна розшифрувати свій код для передачі його інтерпретатору, зловмисник створює "пастку" на рівні цих функцій. У момент, коли захисний механізм викликає *eval(decryptedCode)*, перехоплена функція зберігає розшифрований рядок у файл або виводить його в консоль, і лише потім передає керування оригінальному обробнику. Це дозволяє отримати чистий вихідний код, оминаючи всі криптографічні бар'єри.

2. Аналіз вмісту оперативної пам'яті (*Memory Dumping*). Навіть якщо програма не зберігає розшифровані дані на жорсткому диску, вони обов'язково повинні бути присутніми в оперативній пам'яті (*RAM*) для обробки процесором.

Механізм атаки: Використання системних утиліт (наприклад, *gcore* у Linux або *Task Manager* у *Windows*) або вбудованих засобів *Node.js* для створення повного зліпка (дампу) пам'яті процесу (*Heap Snapshot*).

Особливості *V8*: Рушій *V8*, на якому базується *Node.js*, керує пам'яттю через "Купу" (*Heap*). Особливістю роботи зі стрічками (*Strings*) у *V8* є те, що старі

копії змінних не видаляються миттєво, а очікують на запуск збирача сміття (*Garbage Collector*). Це створює часове вікно, протягом якого зловмисник може знайти в пам'яті розшифровані фрагменти коду, ключі шифрування або конфіденційні дані, використовуючи пошук за сигнатурами або ентропійний аналіз.

3. Інтерактивне налагодження (Interactive Debugging). *Node.js* підтримує стандартний протокол налагодження (*V8 Inspector Protocol*), який дозволяє підключати зовнішні відладчики (наприклад, *Chrome DevTools, VS Code*).

Механізм атаки: Зловмисник запускає додаток у режимі налагодження (прапор *--inspect*), встановлює точки зупинки (*Breakpoints*) на підозрілих ділянках коду та виконує його покроково (*Step-by-step execution*).

Мета*:* Це дозволяє спостерігати за зміною значень змінних у реальному часі. Навіть якщо змінні мають обфусковані імена (наприклад, *_0x5a1*), відладчик покаже їхній реальний вміст (наприклад, рядок з *API*-ключем) у момент присвоєння, що нівелює ефект заплутування даних.

4. Трейсинг та профілювання (*Execution Tracing*). Використання інструментів профілювання (*Profilers*) для побудови графа викликів функцій у реальному часі. Це дозволяє зловмиснику відфільтрувати "мертвий код" (який ніколи не викликається) та зосередитися лише на функціях, що виконують корисну роботу, значно спрощуючи подальший аналіз логіки програми.

1.5 Еволюція методів деобфускації в епоху штучного інтелекту

Окремим вектором загроз, що виник в останні роки і вимагає детального аналізу, є застосування методів машинного навчання (*ML*) та штучного інтелекту (*AI*) для атак на програмне забезпечення. Класична теорія обфускації базувалася на припущенні, що аналітик – це людина, когнітивні здібності якої обмежені. Проте поява Великих Мовних Моделей (*LLM*), таких як *GPT-4, Llama 3* та

спеціалізованих моделей (*StarCoder*), змінила парадигму протистояння «захист-напад».

Традиційні деобфускатори (наприклад, *JSNice*) працювали на основі статистичних моделей *n-грам*, намагаючись вгадати імена змінних. Сучасні трансформерні архітектури, що використовують механізм «уваги» (*Self-Attention*), здатні аналізувати код на семантичному рівні. Вони не просто «читають» код, а будують внутрішнє векторне представлення логіки алгоритму. Це дозволяє ШІ ігнорувати такі популярні методи захисту, як лексична обфускація та вставка «мертвого коду»

Зловмисники використовують інструменти *AI*-рефакторингу для перетворення заплутаного «спагеті-коду» назад у структурований, об'єктно-орієнтований вигляд. Експерименти показують, що *LLM* здатні відновити до 70% оригінальної структури коду після застосування середніх налаштувань обфускаторів типу *javascript-obfuscator*

1.6 Класифікація методів захисту програмного забезпечення

Проблема захисту програмного забезпечення від несанкціонованого аналізу та модифікації є багатогранною і розглядається в науковій літературі в рамках моделі загроз *MATE* (*Man-At-The-End*). Ця модель описує сценарій, де зловмисник має фізичний доступ до пристрою, на якому виконується програмне забезпечення, і володіє повними правами адміністратора, що дозволяє йому використовувати будь-які інструменти статичного та динамічного аналізу.

Для систематизації підходів до захисту доцільно використати таксономію, запропоновану Крістіаном Колбергом (*C. Collberg*) [13], яка є загальноприйнятим стандартом у галузі безпеки ПЗ. Згідно з цією класифікацією, методи захисту поділяються на три фундаментальні категорії, кожна з яких впливає на різні аспекти життєвого циклу програми: обфускація (ускладнення розумін-

ня), пакування (ускладнення доступу) та криптографічний захист (забезпечення конфіденційності).

1. Обфускація коду (*Code Obfuscation*).[14] Обфускація визначається як процес функціонально-еквівалентного перетворення вихідного тексту програми. Кінцевою метою такої трансформації є створення версії коду, яка зберігає ідентичну логіку виконання та видає аналогічні результати на тих самих вхідних даних, проте є надзвичайно складною для сприйняття людиною та автоматизованими інструментами аналізу.

– Ефективність обфускації базується на двох факторах: підвищенні інформаційної ентропії коду (збільшення ступеня хаотичності) та руйнуванні структурних паттернів, за якими працюють сучасні декомпілятори та аналізатори. Методи обфускації класифікують за об'єктом їхнього впливу на програму:

– Лексична обфускація. Цей метод спрямований на видалення з коду всієї інформації, яка є корисною для розробника, але ігнорується транслятором. Сюди відноситься видалення коментарів, форматування (відступів) та налагоджувальної інформації. Найважливішим елементом є перейменування ідентифікаторів Змінні, функції та класи, що мають семантично значущі назви (наприклад, *getUserPassword*, *calculateTax*), замінюються на беззмістовні набори символів (наприклад, *_0x4f2a, a, b*). У контексті *JavaScript* це суттєво ускладнює розуміння бізнес-логіки, перетворюючи код на набір абстрактних символів.

– Обфускація керування (*Control Flow Obfuscation*). Це найбільш потужна група методів, метою якої є деформація порядку виконання інструкцій таким чином, щоб граф потоку керування програми став максимально заплутаним і нелінійним.

– Сплющення потоку (*Control Flow Flattening*): Метод трансформує природну ієрархічну структуру програми (вкладені умови та цикли) у "пласку" модель. Весь логічний код поміщається всередину єдиного нескінченного циклу, в якому знаходиться масштабний перемикач, що керує переходами між блоками.

Рух між частинами програми визначається спеціальною змінною стану, що робить візуальну та логічну реконструкцію алгоритму статичними засобами практично неможливою.

– Непрозорі предикати (*Opaque Predicates*): Вставка в код умовних переходів, результат яких (істина або хибність) відомий обфускатору ще на етапі перетворення, проте є надзвичайно складним для обчислення статичними аналізаторами. Це призводить до появи в коді великої кількості "фейкових" гілок виконання, які ніколи не спрацьовують, але змушують зловмисника витрачати значні ресурси на їх аналіз.

– Обфускація даних (*Data Obfuscation*). Даний напрям спрямований на приховування реальної структури даних, що використовуються в програмі, та значень констант.

– Шифрування рядкових літералів (*String Encryption*): Критичні константи (наприклад, *API*-ключі, адреси серверів, системні повідомлення) більше не зберігаються у відкритому вигляді. Вони кодуються або шифруються та перетворюються на нечитабельні масиви даних, які розшифровуються спеціальною допоміжною функцією безпосередньо в оперативній пам'яті лише в момент їхнього безпосереднього використання.

– Розділення та злиття змінних (*Variable Splitting/Merging*): Метод штучного ускладнення типів даних. Наприклад, одна числова змінна може бути розбита на дві або три окремі змінні, пов’язані певною логічною операцією. Аналогічно, декілька незалежних змінних можуть бути об’єднані в один багатовимірний масив або об’єкт. Це робить відстеження потоку даних та зміну значень у пам’яті (*Data Flow Analysis*) надзвичайно трудомістким процесом.

2. Пакування та бінарна компіляція (*Packing*). У середовищі *Node.js*, де вихідний код зазвичай розповсюджується у відкритому текстовому вигляді, популярним методом спроби захисту інтелектуальної власності та спрощення розгортання є використання пакувальників (таких як *pkg, nexe, box-node*). Цей

підхід, який часто помилково називають «компіляцією», передбачає створення єдиного автономного виконуваного файлу (*Single Executable Application – SEA*) під цільову платформу (*.exe* для *Windows*, бінарний файл *ELF* для *Linux* або *Mach-O* для *macOS*).

Технічна реалізація методу Механізм роботи пакувальників базується на конкатенації (склеюванні) двох основних компонентів:

– Середовище виконання : Урізана версія бінарного файлу інтерпретатора *Node.js*, яка містить необхідні нативні бібліотеки (*libuv, v8, openssl*).

– Корисне навантаження: Всі скрипти проекту, файли конфігурації (*package.json*), залежності (*node_modules*) та статичні ресурси, зібрані в один блок даних.

У результаті розробник отримує файл, який можна запустити на "чистій" машині без необхідності встановлення *Node.js*

Хоча цей метод ускладнює прямий доступ до коду для пересічного користувача, з точки зору професійної безпеки він має критичний недолік, закладений у самій архітектурі рішення. Оскільки *JavaScript* є інтерпретованою мовою, рушій *V8* не може виконувати машинний код безпосередньо – йому потрібен вихідний текст або байт-код.

Щоб забезпечити роботу стандартних функцій читання файлів (*fs.readFile, require*), пакувальники впроваджують механізм віртуальної файлової системи (*VFS*). Всередині скомпільованого файлу код зберігається у вигляді структурованого зліпка файлової системи (*Snapshot*). При запуску програми спеціальний шар-перехоплювач (*Interception Layer*) підміняє стандартні системні виклики. Коли програма намагається завантажити модуль (*require('./auth.js'),* пакувальник перехоплює цей запит і зчитує вміст файлу не з диска, а з відповідного зміщення (*offset*) всередині власного виконуваного файлу.

Саме наявність механізму віртуальної файлової системи (*VFS*) робить метод пакування вразливим до цілеспрямованого аналізу та атак на вилучення даних. Критична проблема полягає у тому, що в більшості конфігурацій вихідний код всередині бінарного контейнера зберігається у незахищеному вигляді – як звичайний текст або структурований *JSON*-об'єкт. Це відкриває тривіальний вектор атаки: зловмиснику достатньо відкрити виконуваний файл (*.exe* або *ELF*) у шістнадцятковому редакторі (*Hex Editor*) або використати стандартну системну утиліту *strings*, щоб отримати безперешкодний доступ до фрагментів програмної логіки, текстових констант та, що є найбільш критичним, до вбудованих *API*-ключів чи облікових даних.

Спроби підвищити рівень захисту шляхом увімкнення опції компіляції в байт-код *V8* (*V8 Bytecode Snapshot*) також не гарантують реальної безпеки інтелектуальної власності. Необхідно чітко розрізняти поняття компіляції та шифрування: байт-код *V8* не є шифруванням, а являє собою серіалізований формат внутрішніх структур інтерпретатора, призначений для прискорення запуску, а не для приховування інформації. На сьогоднішній день існують спеціалізовані інструменти реверс-інжинірингу, що дозволяють десеріалізувати цей байт-код назад у читабельну деревоподібну структуру або проводити глибокий аналіз його логіки без необхідності повного відновлення вихідного тексту.

Ситуація ускладнюється тим, що внутрішня архітектура та структура файлів популярних пакувальників, таких як *pkg* та *nexe*, є відкритою та стандартизованою. Це призвело до розробки та вільного розповсюдження готових автоматизованих скриптів-розпакувальників (*unpackers*). Використовуючи такий інструментарій, атакуючий має можливість вказати шлях до цільового «захищеного» файлу і за лічені секунди отримати повну структуру папки проекту з усіма вихідними кодами у їхньому первозданному вигляді. Таким чином, пакування створює лише ілюзію захищеності, реалізуючи ненадійний принцип «безпеки через неясність». Цей метод залишається ефективним виключно як

засіб зручної дистрибуції та розгортання програмного забезпечення, але є абсолютно непридатним у якості механізму забезпечення конфіденційності алгоритмів.

3. Криптографічний захист програмного забезпечення вважається найбільш надійним методом забезпечення конфіденційності коду в стані спокою (*"At Rest"*). Суть методу полягає у шифруванні виконуваних файлів або їх частин за допомогою стійких симетричних алгоритмів (наприклад, *AES-256*).

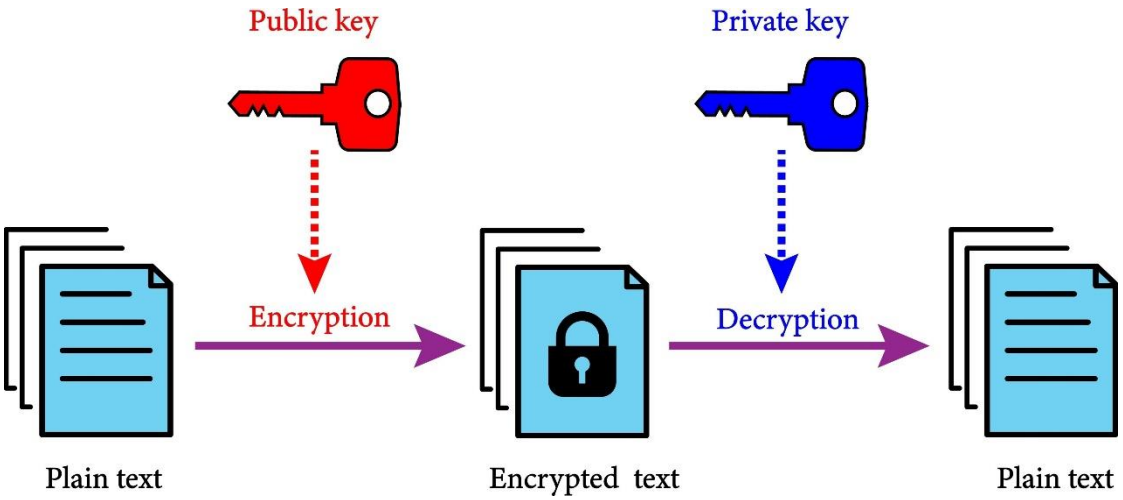

Рисунок 1.1 - Схема роботи симетричного алгоритму, на прикладі *AES-256*

Процес виконання такої програми вимагає наявності спеціального модуля - завантажувача (*Loader/Stub*), який відповідає за:

– Зчитування зашифрованого контейнера з диска.

– Отримання ключа дешифрування (від сервера ліцензування або з апаратного ключа).

– Розшифрування коду безпосередньо в оперативну пам'ять.

– Передачу керування розшифрованому коду.

Основною перевагою цього підходу є те, що статичний аналіз файлу на диску не дає зловмиснику жодної інформації про алгоритм роботи програми, оскільки ентропія зашифрованих даних наближається до максимуму. Однак, головною проблемою залишається захист самого ключа дешифрування та моменту, коли код з'являється у розшифрованому вигляді в пам'яті (*Memory Dumping Attack*), що вимагає застосування додаткових методів обфускації та анти-налагодження.

1.7 Аналіз існуючих інструментів

Проведений порівняльний аналіз провідних комерційних (наприклад, *Jscrambler*) [15] та відкритих (наприклад, *javascript-obfuscator, UglifyJS*) рішень дозволив виявити низку системних недоліків, які унеможливлюють їх використання для захисту критично важливих алгоритмів у середовищі з високими вимогами до безпеки та продуктивності.

Проблема оборотності лексичних та структурних трансформацій. Більшість алгоритмів обфускації базуються на детермінованих правилах перетворення синтаксичного дерева (*Abstract Syntax Tree – AST*).

– Лексична вразливість: Стандартні методи перейменування змінних (Renaming) використовують передбачувані патерни (наприклад, генерація послідовностей *_0x1, _0x2*...). Сучасні інструменти деобфускації *(De-obfuscators*), що використовують статистичний аналіз та методи машинного навчання (наприклад, *JSNice*), здатні відновлювати оригінальні імена змінних, аналізуючи контекст їх використання та типи даних.

– Структурна вразливість: Навіть складні техніки заплутування структури часто є зворотними. Деобфускатори, що працюють на основі символьного виконання (*Symbolic Execution*), можуть автоматично спрощувати заплутані ви-

рази, видаляти "мертвий код" та відновлювати граф потоку керування (*Control Flow Graph*), повертаючи код до стану, придатного для аналізу людиною. Отже, бар'єр входу для зловмисника залишається недостатньо високим.

– Деградація продуктивності виконуваних модулів (*Performance Overhead*). Застосування агресивних методів захисту, таких як "сплющення потоку керування" (*Control Flow Flattening*), неминуче призводить до суттєвого зниження швидкодії програми.

– Механізм уповільнення**:** Техніка *Flattening* перетворює ієрархічну структуру програми на єдиний нескінченний цикл з великим перемикачем (*switch-case*). Це руйнує механізми передбачення переходів у сучасних процесорах та блокує оптимізації *JIT-компілятора* рушія *V8*.

– Кількісна оцінка**:** Експериментальні дослідження показують, що для обчислювально складних алгоритмів падіння продуктивності може складати від 30% до 500% в залежності від налаштувань обфускатора. Для високонавантажених серверних систем *(High-Load Systems)* такі накладні витрати є неприпустимими, оскільки це вимагає пропорційного збільшення апаратних потужностей.

Вразливість до атак криміналістичного аналізу залишкових даних - критичний недолік більшості існуючих пакувальників (наприклад, *pkg* [16], *nexe*) та захищених завантажувачів є використання файлової системи як проміжного буфера.

– Вектор атаки**:** Для запуску захищеного додатка завантажувач розпаковує або розшифровує виконуваний код у тимчасову директорію операційної системи (наприклад*, /tmp у Linux або %TEMP%* у *Windows*).

– Ризики**:** Навіть якщо тимчасовий файл видаляється одразу після завантаження в пам'ять, він залишає сліди на магнітних носіях або в журналі файлової системи. Зловмисник може використати інструменти відновлення видалених даних або налаштувати моніторинг файлових подій (наприклад, через під-

систему *inotify*), щоб перехопити розшифрований пейлоад у момент його появи. Це робить захист неефективним проти атак типу "*Race Condition*" та посмертного криміналістичного аналізу.

1.8 Висновки до розділу

Проведений аналіз загроз безпеці програмного забезпечення засвідчив, що архітектурна специфіка платформи *Node.js*, яка базується на використанні відкритого вихідного коду, створює унікальний вектор вразливостей для рішень *Enterprise*-рівня. Відсутність належних технічних механізмів захисту не лише наражає компанії на фінансові та репутаційні ризики внаслідок крадіжки інтелектуальної власності або атак на ланцюжок постачання, але й має прямі юридичні наслідки. Використання незахищеного коду унеможливлює його кваліфікацію як комерційної таємниці у судовому порядку та призводить до невідповідності вимогам міжнародних стандартів інформаційної безпеки, таких як *ISO/IEC 27001*, *NIST* та *OWASP ASVS*.

Огляд існуючих методів протидії показав кризу традиційних підходів в умовах розвитку технологій штучного інтелекту та динамічного аналізу. Класична обфускація втрачає ефективність через здатність сучасних *LLM* відновлювати семантику коду, при цьому спричиняючи критичне падіння продуктивності системи, що є неприпустимим для високонавантажених сервісів. Водночас популярні методи пакування виконуваних файлів забезпечують лише ілюзію безпеки, оскільки базуються на віртуальних файлових системах, вразливих до тривіального вилучення коду та модифікації.

Таким чином, результати дослідження обґрунтовують необхідність розробки спеціалізованого комплексу захисту, який нівелював би недоліки існуючих рішень. Нова система повинна відмовитися від використання файлової системи як проміжного буфера, щоб уникнути криміналістичного відновлення да-

них, та зосередитися на криптографічному захисті активів (*AES-256*) із подальшим безпечним виконанням безпосередньо в оперативній пам'яті. Це єдиний шлях забезпечити реальну конфіденційність алгоритмів та цілісність програмного продукту в умовах агресивного середовища експлуатації.

2 ТЕОРЕТИЧНІ ОСНОВИ ТА МЕТОДИ ЗАХИСТУ ВИХІДНОГО КОДУ

2.1 Визначення сутності та критеріїв якості перетворень

У сучасній теорії інформаційної безпеки обфускація програмного забезпечення розглядається як критично важливий елемент стратегії захисту інтелектуальної власності в умовах, коли зловмисник має повний доступ до виконуваного коду. Етимологічно термін походить від латинського *obfuscare*, що означає «затінювати» або «затемнювати». У контексті програмної інженерії це поняття формалізується як процес глибокого структурного та семантичного перетворення вихідного тексту програми. Головною метою цього процесу є створення нової версії програмного продукту, яка зберігає повну функціональну еквівалентність оригіналові – тобто забезпечує ідентичну поведінку, побічні ефекти та результати обчислень на будь-яких валідних наборах вхідних даних. Водночас, внутрішня структура модифікованої програми трансформується таким чином, що вона стає надзвичайно складною для сприйняття людиною та аналізу автоматизованими інструментами статичного аналізу, такими як декомпілятори та дизасемблери.

Теоретичне обґрунтування стійкості обфускації базується на теоремі Райса та проблемі зупинки, які стверджують, що не існує загального алгоритму, здатного визначити всі нетривіальні властивості довільної програми. Обфускація прагне наблизити аналіз коду до цього класу нерозв'язних задач. Якість та ефективність обфускаційних перетворень традиційно оцінюється за комплексною системою критеріїв, розробленою дослідником Крістіаном Колбергом, яка стала стандартом де-факто в індустрії захисту програмного забезпечення.

Першим і ключовим критерієм є потужність перетворення (*Potency*). Цей показник визначає міру незрозумілості коду для людини-експерта і корелює з поняттям ентропії програмного коду. [18] Чим вища потужність, тим більше часу та інтелектуальних зусиль необхідно витратити кваліфікованому аналітику для розуміння логіки роботи алгоритму та зв'язків між його компонентами. Другим критично важливим показником є стійкість (*Resilience*). Вона характеризує здатність захищеного коду протистояти автоматизованим засобам деобфускації, які намагаються привести код до канонічного вигляду. Висока стійкість означає, що створення інструменту для автоматичного повернення коду до початкового стану є алгоритмічно складною задачею (класу *NP-складних*), яка вимагає експоненційних витрат обчислювальних ресурсів. Третім критерієм виступає вартість (*Cost*), що являє собою оцінку накладних витрат системних ресурсів. Впровадження додаткових захисних конструкцій неминуче призводить до збільшення розміру файлу, споживання оперативної пам'яті та затримок при виконанні інструкцій процесором.

2.2 Теоретичні межі обфускації та концепція «Віртуальної чорної скриньки»

У сучасній науці про безпеку програмного забезпечення існує фундаментальна проблема: чи можливо перетворити програму так, щоб вона працювала, але приховувала свій алгоритм повністю? Це описується концепцією «Віртуальної чорної скриньки» (*Virtual Black Box*). Ідеальна обфускація мала б перетворити код на таку структуру, з якої неможливо дізнатися нічого, окрім вихідних даних за вхідними параметрами.

Проте дослідження (зокрема, фундаментальні роботи Барака та співавторів) доводять, що створення ідеальної «чорної скриньки» для будь-якої довільної програми є неможливим. Завжди існують методи, що дозволяють

частково відновити логіку. Тому в даній роботі ми не ставимо за мету створення абсолютного захисту, який неможливо зламати.

Замість цього ми спираємося на модель «економічної безпеки». Метою захисту є підвищення вартості реверс-інжинірингу (часу роботи кваліфікованого спеціаліста, обчислювальних ресурсів) до такого рівня, коли вона перевищує потенційний прибуток від злому програми (вартість інтелектуальної власності). Саме такий підхід, відомий як *«Indistinguishability Obfuscation»* (обфускація нерозрізнюваності) у практичному сенсі, дозволяє вважати систему захищеною в реальних умовах експлуатації.

2.3 Класифікація та механізми методів обфускації

У розробленій системі захисту застосовано комплексний підхід, що передбачає синергію методів, які впливають на різні рівні абстракції програмного коду. Першою групою методів є лексичні перетворення (*Layout Transformations*), спрямовані на зміну зовнішнього вигляду коду без порушення його синтаксичної структури. Механізм дії цих перетворень полягає у видаленні з коду всієї інформації, яка є корисною для розробника, але ігнорується транслятором або інтерпретатором. Сюди відноситься видалення коментарів, форматування, відступів та метаданих налагодження. Найважливішим елементом тут є незворотна заміна семантично значущих ідентифікаторів. Усі назви змінних, функцій, класів та просторів імен, які несуть змістове навантаження і допомагають розробнику орієнтуватися в коді, замінюються на псевдовипадкові або алгоритмічно згенеровані послідовності символів. У даній роботі використано метод детермінованого хешування, що дозволяє уніфікувати імена змінних, перетворюючи їх на стандартизовані шістнадцяткові рядки. Це призводить до повної втрати семантичного зв'язку коду з предметною областю, перетворюючи програму на набір абстрактних інструкцій, призначення яких не-

можливо визначити з назви, що змушує зловмисника аналізувати весь контекст використання кожної змінної.

Другою, найбільш ефективною групою протидії реверс-інжинірингу, є перетворення потоку керування (*Control Flow Transformations*). Метою цих методів є деформація графа виконання програми (*Control Flow Graph*) таким чином, щоб він став максимально заплутаним і нелінійним. До цієї категорії належить ін'єкція недіючого коду (*Dead Code Injection*) – впровадження в тіло програми додаткових блоків інструкцій, які є синтаксично коректними і можуть виконуватися процесором, але результати їх роботи жодним чином не впливають на глобальний стан програми. Наявність таких блоків значно збільшує обсяг коду, який необхідно проаналізувати зловмиснику, розпорошуючи його увагу на несуттєві деталі. Також широко застосовуються непрозорі предикати (*Opaque Predicates*) – спеціальні умовні конструкції, істинність яких відома розробнику захисту ще на етапі компіляції (наприклад, математичні тотожності), але є складно обчислюваною для статичних аналізаторів. Це змушує інструменти автоматичного аналізу розглядати хибні гілки виконання як ймовірні, що призводить до комбінаторного вибуху кількості варіантів виконання та унеможливлює побудову коректного графа керування.

2.4 Метрики оцінки складності програмного забезпечення

Для об'єктивного підтвердження ефективності обфускації та переходу від суб'єктивних оцінок до кількісних показників використовуються стандартизовані метрики програмної інженерії. Зокрема, метрика Холстеда [17] базується на детальному аналізі інформаційної насиченості програми через підрахунок кількості операторів (ключових слів, функцій) та операндів (змінних, констант). Штучне збільшення словника унікальних операндів за рахунок впровадження «шумових» функцій та змінних призводить до суттєвого зростання показника

інформаційного об'єму програми. Це свідчить про підвищення когнітивного навантаження на аналітика, оскільки кількість сутностей, які необхідно утримувати в увазі для розуміння алгоритму, зростає.

Поряд з цим використовується цикломатична складність Маккейба [19], яка визначає топологічну складність програми через підрахунок кількості лінійно незалежних маршрутів у графі керування. Впровадження фіктивних умовних переходів, циклів та операторів розгалуження суттєво збільшує кількість ребер та вузлів у графі керування, що призводить до підвищення цикломатичної складності. Високе значення цього показника корелює зі складністю тестування та покриття коду, що, в свою чергу, ускладнює роботу зловмисника, оскільки для повного розуміння алгоритму йому необхідно проаналізувати кожен з численних маршрутів виконання, більшість з яких є хибними.

2.5 Математичні моделі криптографічного захисту

Для забезпечення конфіденційності програмного коду у стані спокою (*«At Rest»*) та гарантування його незмінності під час зберігання на сервері обрано сучасні симетричні криптографічні алгоритми. [20] Теоретичною основою побудови системи захисту є поняття криптографічного перетворення як процесу бієктивного відображення множини відкритих повідомлень у множину шифротекстів за допомогою простору ключів. Вибір симетричної криптографії обумовлений необхідністю обробки значних обсягів даних з мінімальними затримками, оскільки симетричні алгоритми забезпечують значно вищу пропускну здатність завдяки використанню швидких логічних операцій (*XOR*, зсуви), на відміну від асиметричних алгоритмів, що базуються на ресурсоємній арифметиці великих чисел.

В основу розробленого рішення покладено клас блокових шифрів, які працюють за принципом розбиття вхідного потоку даних на блоки фіксованої

довжини. Сучасні стійкі алгоритми будуються на архітектурі, що поєднує принципи розсіювання (*Diffusion*) для рівномірного розподілу статистичних особливостей вхідного тексту по всьому шифроблоку, та перемішування (*Confusion*) для максимального ускладнення залежності між ключем шифрування і отриманим шифротекстом. [21]

2.6 Алгоритм *AES* (*Advanced Encryption Standard*)

Класичні алгоритми шифрування, такі як *AES*, розроблялися для моделі «Чорної скриньки» [14], де зловмисник бачить вхідні та вихідні дані, але не має доступу до внутрішнього стану системи та ключів. Однак у контексті захисту серверного ПЗ ми розглядаємо модель загроз *MATE* (*Man-At-The-End*), де зловмисник може мати повний адміністративний доступ до сервера. [22]

Така ситуація називається контекстом «Білої скриньки». Головна небезпека тут полягає в тому, що ключі шифрування під час виконання програми неминуче з’являються в оперативній пам’яті. Спеціалізовані атаки дозволяють знаходити ці ключі, аналізуючи ентропію блоків пам’яті.

Використання спеціалізованих алгоритмів «*White-Box*», які «розчиняють» ключ у коді програми, часто призводить до критичного падіння продуктивності та збільшення розміру коду в сотні разів, що неприйнятно для високонавантажених *Node.js* серверів. Тому в даній роботі обрано гібридний підхід: використання швидкого стандартного алгоритму *AES-GCM*, але з мінімізацією часу життя розшифрованих даних та ключів у пам’яті. Це компроміс, який забезпечує високу швидкодію при достатньому рівні захисту від статичного аналізу.

AES є загальноприйнятим світовим стандартом симетричного блокового шифрування, затвердженим *NIST*. Його архітектура базується на принципі підстановочно-перестановочної мережі (*SPN*), що забезпечує високу швидкість роботи та математично доведену стійкість до лінійного та диференціального

криптоаналізу. У роботі використано ключ максимальної довжини (256 біт), що забезпечує стійкість навіть до атак із застосуванням квантових обчислювачів (алгоритм Гровера). Алгоритм передбачає виконання чотирнадцяти раундів перетворень для кожного блоку даних. Кожен раунд шифрування складається з послідовності чотирьох оборотних операцій: нелінійної заміни байтів за спеціальною таблицею підстановок (*S-Box*), циклічного зсуву рядків матриці стану, перемішування стовпців шляхом матричного множення в скінченних полях Галуа та побітового додавання раундового ключа. Така складна багатошарова структура забезпечує надійний захист даних від спроб відновлення ключа.

2.7 Режим автентифікованого шифрування *GCM*

Оскільки класичні режими шифрування (наприклад, *CBC*) забезпечують лише конфіденційність даних, але не захищають від їх непомітної модифікації (атаки на цілісність), у роботі обрано режим *GCM* (*Galois/Counter Mode*). Цей режим реалізує схему автентифікованого шифрування з приєднаними даними (*AEAD*), поєднуючи шифрування даних з криптографічним контролем їх цілісності. Шифрування виконується в потоковому режимі лічильника (*CTR*), що дозволяє ефективно розпаралелювати процес обробки даних на сучасних багатоядерних процесорах. [23]

Одночасно з шифруванням відбувається обчислення спеціального тегу автентифікації (*MAC*) за допомогою алгоритму поліноміального хешування над полем Галуа. Цей тег виступає гарантом цілісності: будь-яка несанкціонована зміна навіть одного біта в зашифрованому контейнері призводить до кардинальної зміни контрольної суми та помилки верифікації тегу під час розшифрування. Це дозволяє системі захисту миттєво виявити втручання та заблокувати виконання скомпрометованого коду, унеможливлюючи впровадження шкідливих закладок або бекдорів.

2.8 Функція деривації ключа (*PBKDF2*)

Для нівелювання загрози підбору паролів методом повного перебору (*Brute-force*) ключі шифрування не використовуються в чистому вигляді, а генеруються за допомогою стандартизованої функції *PBKDF2* (*Password-Based Key Derivation Function 2*). [24] Алгоритм виконує багаторазове (у розробленій системі – сто тисяч ітерацій) застосування псевдовипадкової функції (*HMAC-SHA512*) до пароля користувача з додаванням унікальної випадкової послідовності – "солі". Використання солі запобігає атакам з використанням попередньо обчислених таблиць (*Rainbow Tables*). Велика кількість ітерацій робить процес генерації кожного окремого ключа обчислювально ємним, штучно уповільнюючи процес перевірки одного кандидата пароля. Це робить масовий перебір паролів економічно недоцільним навіть при використанні кластерів графічних процесорів (*GPU*), не впливаючи при цьому на зручність легального користувача, для якого затримка в частки секунди є непомітною. [25]

2.9 Архітектурні особливості безпечного виконання в середовищі Node.js

Реалізація методу безфайлового виконання (*In-Memory Execution*) базується на глибокому розумінні фундаментальних принципів роботи середовища виконання *Node.js* та його ядра – високопродуктивного рушія *JavaScript Google V8*. [26] Розуміння низькорівневих механізмів управління пам'яттю, компіляції та завантаження модулів дозволяє використовувати вбудовані можливості платформи для створення ізольованого захищеного контуру виконання коду, який функціонує виключно в межах оперативної пам'яті, повністю минаючи файлову підсистему операційної системи.

1. *JIT*-компіляція та конвеєр обробки коду. На відміну від класичних інтерпретоваторів, які потребують послідовного зчитування коду з фізичних файлів, рушій V8 використовує передову технологію *JIT*-компіляції *(Just-In-Time*). Життєвий цикл виконання коду в *V8* починається з отримання вхідного потоку даних у вигляді рядка символів. Критично важливою особливістю є те, що джерело цього рядка для рушія є неважливим: він може бути переданий безпосередньо з буфера оперативної пам'яті. Далі відбувається процес лексичного та синтаксичного аналізу (парсинг), в результаті якого в пам'яті будується Абстрактне Синтаксичне Дерево (*AST*). На фінальному етапі компілятор *TurboFan* генерує оптимізований машинний код під конкретну архітектуру процесора. [27] Ключовим моментом для безпеки є те, що після генерації внутрішніх структур рушія (*AST* та байт-коду), вихідний текстовий рядок стає непотрібним для подальшого виконання і може бути видалений з пам'яті, що мінімізує час існування незахищеного коду у відкритому вигляді.

2. Управління пам'яттю та збирання сміття. [28] Безпека запропонованого методу значною мірою залежить від часу життя об'єктів у оперативній пам'яті (проблема *Memory Remanence*). У розробленій системі дешифрований код зберігається у динамічній пам'яті (Heap) процесу *Node.js*. Архітектура *V8* використовує генераційний підхід до управління пам'яттю, поділяючи її на "Молоде покоління" (*New Space*) та "Старе покоління" (*Old Space*). Оскільки змінна, що містить розшифрований код, використовується системою лише одноразово в момент ініціалізації та компіляції, вона потрапляє в область пам'яті для короткоживучих об'єктів. Після завершення передачі коду на виконання, посилання на цю змінну втрачається, і вона стає пріоритетним кандидатом на видалення під час найближчого циклу роботи збирача сміття (Scavenger). Це значно зменшує часове вікно, протягом якого зловмисник може спробувати вилучити код через дамп пам'яті, порівняно з даними,

що зберігаються в кеші сторінок файлової системи операційної системи, які можуть залишатися доступними протягом тривалого часу.

При реалізації методу безфайлового виконання (*In-Memory Execution*) необхідно враховувати фізичні властивості оперативної пам’яті (*DRAM*). На відміну від теоретичної моделі, де дані зникають миттєво після вимкнення живлення, у реальності конденсатори модулів пам’яті зберігають заряд протягом певного часу (від секунд до хвилин).

Це створює теоретичну вразливість до атак типу «*Cold Boot*» (холодне перезавантаження) [29], коли зловмисник може фізично заморозити чіпи пам’яті та зчитати їх вміст на іншому пристрої. Також існує ризик потрапляння фрагментів розшифрованого коду у файл підкачки (*Swap-файл*) операційної системи, якщо обсяг оперативної пам’яті вичерпано.

Проте для захисту серверних рішень, що розгортаються у хмарних дата-центрах, ці ризики є мінімальними, оскільки фізичний доступ до обладнання для сторонніх осіб заблоковано провайдером послуг. Основний акцент захисту робиться на протидію програмному зчитуванню пам’яті та аналізу дампів процесу, що забезпечується швидкою очисткою змінних механізмами *V8*.

3. Віртуалізація контексту виконання. Стандартна модульна система *Node.js* (*CommonJS*) історично покладається на наявність фізичних файлів на диску для визначення відносних шляхів до робочої директорії, підключення локальних модулів та завантаження ресурсів. При виконанні коду з потоку вводу (*stdin*) або віртуального середовища цей механізм не працює коректно, оскільки файл фізично відсутній. Для вирішення цієї архітектурної проблеми застосовано метод віртуалізації контексту. Перед запуском основного навантаження в середовище виконання ін'єктується спеціальний системний код (Shim), який динамічно модифікує глобальні об'єкти процесу. Цей код налаштовує псевдо-шляхи пошуку модулів та перевизначає змінні оточення,

такі як *__dirname* та *__filename*. Це дозволяє емулювати повноцінне файлове оточення для виконуваного коду, забезпечуючи повну сумісність складних серверних фреймворків (наприклад, *Express.js*) із режимом безфайлового виконання без необхідності створення жодних фізичних артефактів на диску. [30]

2.10 Висновки до розділу

Проведений теоретичний аналіз підтвердив, що хоча створення ідеальної «Віртуальної чорної скриньки» є алгоритмічно неможливим, досягнення високого рівня «економічної безпеки» забезпечується через комплексне застосування методів обфускації та криптографії. Стратегія захисту базується на синергії незворотних лексичних перетворень та деформації графа потоку керування, що об'єктивно підвищує ентропію коду та метричні показники складності за Холстедом і Маккейбом. Використання стандарту *AES-256* у режимі автентифікованого шифрування *GCM*, посиленого функцією деривації ключів *PBKDF2*, гарантує конфіденційність та цілісність програмного забезпечення у стані спокою, роблячи атаки методом перебору економічно недоцільними.

Ключовим архітектурним рішенням для середовища *Node.js* визначено реалізацію методу безфайлового виконання (*In-Memory Execution*), який використовує специфіку роботи *JIT*-компілятора *V8* та механізми управління пам'яттю. Відмова від використання файлової системи як проміжного буфера та впровадження віртуалізації контексту дозволяють запускати складні серверні додатки безпосередньо з оперативної пам'яті, мінімізуючи час існування розшифрованого коду у відкритому вигляді. Такий підхід ефективно нівелює загрози криміналістичного відновлення даних з фізичних носіїв, покладаючись на швидке очищення слідів конфіденційної інформації автоматичним збирачем сміття.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ЗАХИСТУ ПРОГРАМНИХ ПРОДУКТІВ

3.1 Загальна архітектура та алгоритм роботи

Розроблений програмний комплекс базується на клієнт-серверній архітектурі та призначений для захисту інтелектуальної власності компанії, зокрема вихідного коду серверних додатків (наприклад, на *Express.js*). Захист від викрадення через несанкціонований доступ до файлової системи забезпечується завдяки архітектурному розмежуванню вихідного коду та виконуваного коду, який захищений трьома функціональними рівнями захисту.

Важливо відмітити, що даний комплекс не захищає він викрадення через дампи оперативної пам’яті, оскільки, як показали дослідження, цей варіант найменш можливий та несе за собою підвищення додаткового навантаження на фізичний сервер.

Детальна схема роботи зображена на рис 3.1:

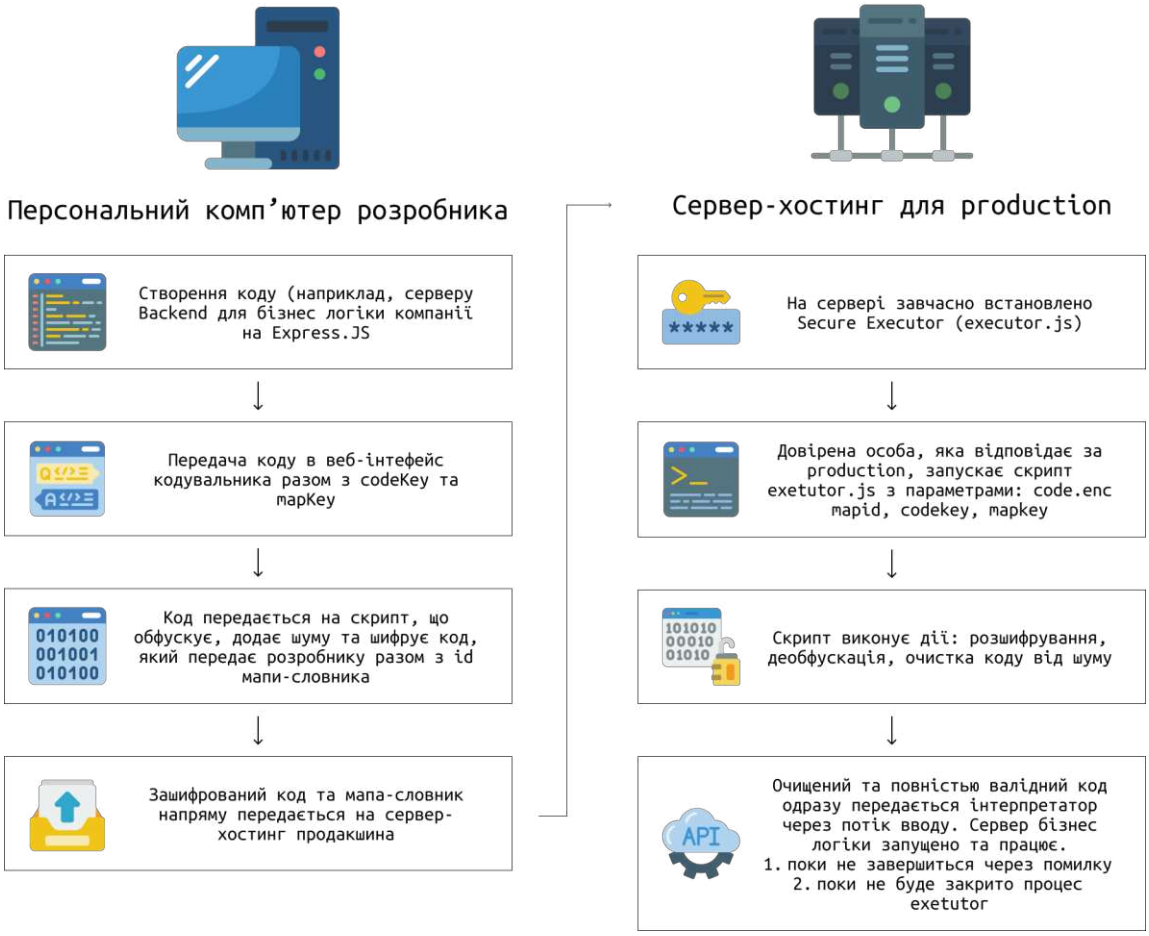

Рисунок 3.1 - Схема роботи програмного комплексу

Система складається з трьох функціональних блоків:

– Веб-інтерфейс: Забезпечує взаємодію з користувачем, введення вихідного коду та ключів шифрування.

– Сервер захисту: Виконує лексичний аналіз, обфускацію, генерацію "шуму" та криптографічне пакування (*AES-GCM*).

– Захищений виконавець: Клієнтський скрипт, що розгортається на цільовому сервері і забезпечує запуск коду безпосередньо в оперативній пам'яті.

Алгоритм роботи:

1. Користувач відкриває веб-інтерфейс, завантажує код та вводить пароль для шифрування мапи змінних та коду.
2. Сервер генерує карту відповідності змінних (*Map*) та обфускує код.

3. Зашифрований код відправляється на сервер разом з зашифрованою мапою.
4. При запуску *executor.js* завантажує обидва компоненти, розшифровує їх у *RAM*, відновлює логіку та передає на виконання у віртуальне середовище.

3.2 Розробка інтерфейсу користувача

Для зручної взаємодії з системою розроблено графічний інтерфейс (*GUI*) на базі *HTML5* та *CSS-фреймворку Tailwind CSS* [35]. Інтерфейс реалізовано у файлі *index.html* і обслуговується сервером *express*.

Інтерфейс містить наступні функціональні зони:

– Зона введення коду. Це текстове поле (textarea), куди розробник вставляє вихідний код свого *Node.js-додатку* (наприклад, сервер на *Express*).

– Зона управління ключами. Два поля для введення паролів. *Code Key*: Пароль для шифрування тіла програми. *Map Key*: Окремий пароль для шифрування карти змінних.

– Панель керування (*Action Panel*): Кнопки, що надсилають асинхронні запити (*Fetch API*) до серверних ендпоінтів */process* та */decrypt*. Хоча кнопка з дешифруванням не приймає участі в запуску сервері з зашифрованого коду, вона несе функцію тестування.

– Зона результатів (*Output*): Поля для відображення зашифрованого пейлоаду (*Hex-рядок*) та унікального ідентифікатора сесії (*Map ID*), які необхідні для подальшого розгортання.

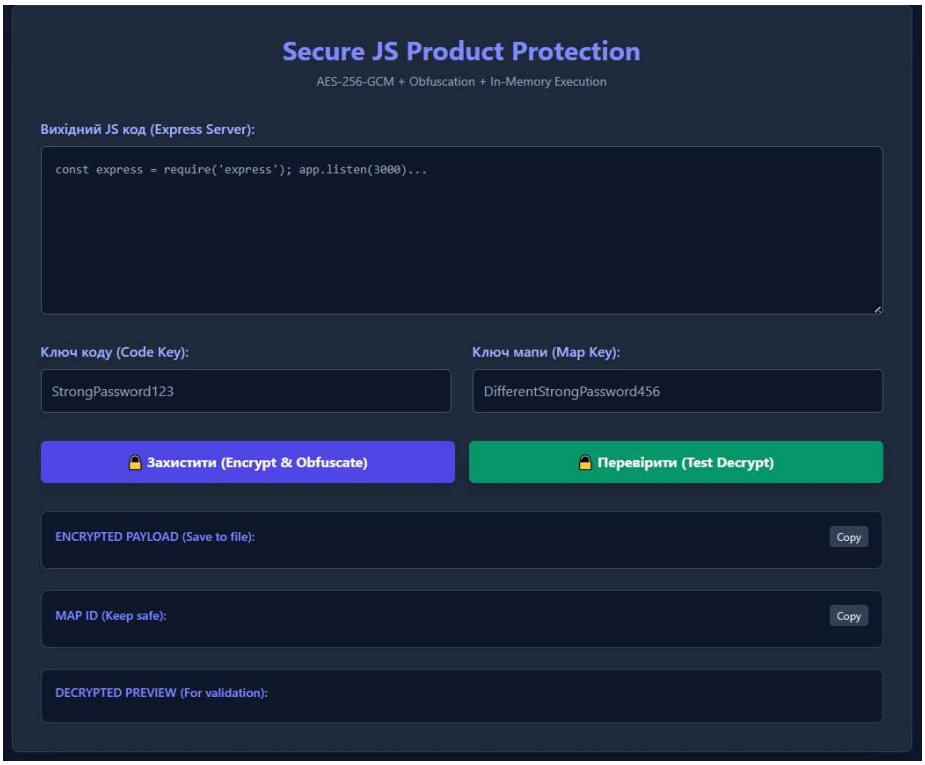

Рисунок 3.2 - Веб-інтерфейс кодувальника

3.3 Програмна реалізація модуля криптографічного захисту

Криптографічне ядро системи реалізовано у файлі *index.js* з використанням нативного модуля *crypto*. [31]

Ключові функції, що відповідають за стійкість захисту:

– Функція деривації ключа. Для захисту від атак перебору паролів (Brute-force) реалізовано функцію *deriveKey*. Вона перетворює довільний текстовий пароль користувача у криптографічно сильний ключ фіксованої довжини.

```
function deriveKey(password, salt) : NonSharedBuffer { Show usages
    return crypto.pbkdf2Sync(password, salt, iterations: 100000, keylen: 32, digest: 'sha512');
}
```

Рисунок 3.3 - Функція *deriveKey*

Використання 100 000 ітерацій *SHA-512* створює значне навантаження на процесор при генерації кожного ключа (близько 100 мс), що робить масовий перебір паролів на *GPU* економічно недоцільним.

– Функція автентифікованого шифрування (*AES-GCM*). Функція *encryptGCM* відповідає за перетворення відкритого тексту на шифротекст із гарантією цілісності.

– Режим *GCM* забезпечує не лише конфіденційність, а й автентифікацію даних. *getAuthTag()* генерує контрольний код (*MAC*). Якщо зловмисник змінить хоча б один біт у *encrypted*, тег при розшифруванні не співпаде, і система видасть помилку.

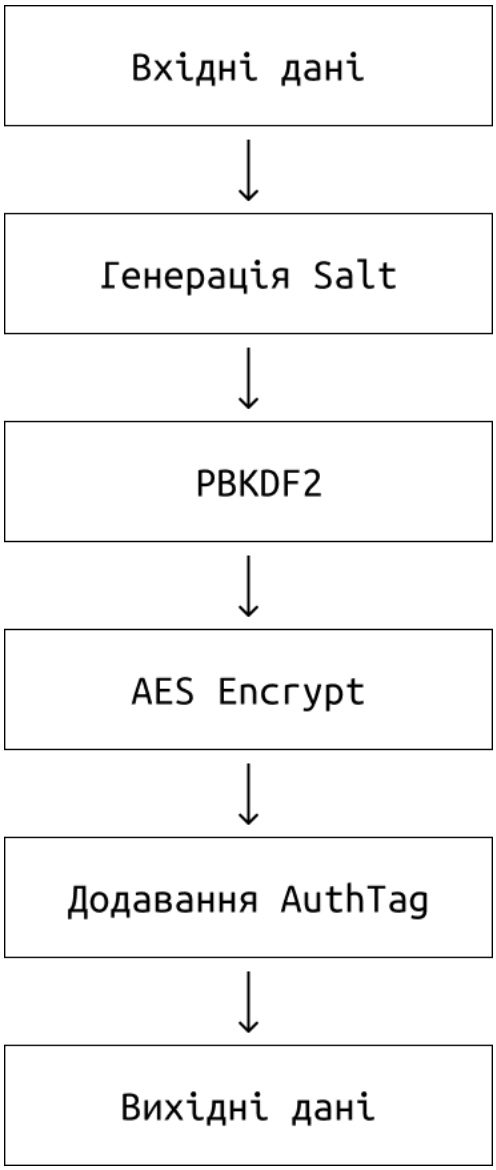

Рисунок 3.4 - Блок-схема алгоритму шифрування *AES-GCM*

3.3 Функція дешифрування та верифікації

Функція *decryptGCM* виконує зворотне перетворення. Критично важливим етапом є встановлення тегу автентифікації перед фіналізацією.

```
function decryptGCM(encryptedData, password) : string  {  Show usages
    const [saltHex, ivHex, authTagHex, encryptedHex] = encryptedData.split(':');
    if (!saltHex || !ivHex || !authTagHex || !encryptedHex) throw new Error('Invalid encryption format');

    const salt : Buffer<ArrayBuffer>  = Buffer.from(saltHex,  encoding: 'hex');
    const iv : Buffer<ArrayBuffer>  = Buffer.from(ivHex,  encoding: 'hex');
    const authTag : Buffer<ArrayBuffer>  = Buffer.from(authTagHex,  encoding: 'hex');
    const key : NonSharedBuffer  = deriveKey(password, salt);

    const decipher : DecipherCCM  = crypto.createDecipheriv( algorithm: 'aes-256-gcm', key, iv);
    decipher.setAuthTag(authTag);

    let decrypted : string  = decipher.update(encryptedHex,  inputEncoding: 'hex',  outputEncoding: 'utf8');
    decrypted += decipher.final( outputEncoding: 'utf8');
    return decrypted;
}
```

Рисунок 3.5 - Код функції *decryptGCM*

3.4 Реалізація механізму безфайлового виконання

Модуль *executor.js* є клієнтською частиною системи. Його головною метою є запуск кода так, щоб він ніколи не з'явився на жорсткому диску у розшифрованому вигляді.

Алгоритм роботи завантажувача:

– Зчитування**:** Завантаження зашифрованого файлу (*app.enc*) та карти (.json) у буфер пам'яті.

– Дешифрування**:** Виклик *decryptGCM* для отримання обфускованого *JS-коду.*

– Деобфускація**:** Відновлення оригінальних імен змінних у пам'яті за допомогою карти.

– Ін'єкція контексту (*Shim*)**:** Динамічне додавання коду для емуляції файлової системи. Оскільки процес запускається з потоку вводу, *Node.js* не може автоматично визначити змінні *__dirname* та *__filename*. Це призводить до помилок у фреймворках типу Express. Для виправлення цього розроблено механізм ін'єкції: [32]

```
const currentDir : string = __dirname.replace( searchValue: /\\/g, replaceValue: '/');

const pathShim : string = `
    Object.defineProperty(global, '__dirname', { value: '${currentDir}' });
    Object.defineProperty(global, '__filename', { value: '${currentDir}/virtual_server.js' });
    module.paths.push('${currentDir}/node_modules');
`;
```

Рисунок 3.6 - Код ін’єкції контексту

Фінальний етап – передача коду в інтерпретатор. Замість створення тимчасового файлу, використовується стандартний потік вводу (*stdin*).

```
const child : ChildProcessByStdio<Writable, null, null> = spawn('node', ['-'], {
    stdio: ['pipe', 'inherit', 'inherit'],
    cwd: __dirname
});

child.stdin.write(finalPayload);
child.stdin.end();
```

Рисунок 3.7 - Створення процесу виконання коду

Цей метод гарантує, що операційна система не виконує операцій запису на диск для виконуваного коду, унеможливлюючи його відновлення засобами криміналістичного аналізу файлової системи.

Повна логіка роботи дешифрування та запуску закодованого коду відображена на рис. 3.8

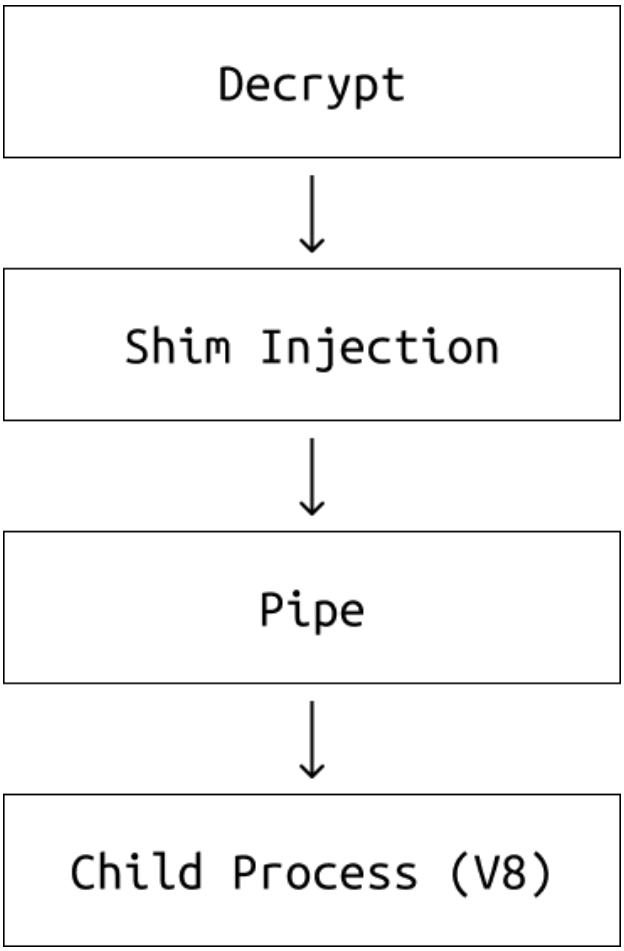

Рисунок 3.8 - Блок-схема механізму реалізації коду

3.4 Тестування розробленої програми

Для демонстрації я обрав приклад коду з базовим сервером Express.JS. [34]

1. Завантажуємо код через веб-інтерфейс та вказуємо *mapKey*, *codeKey*, після натискаємо «Захистити»

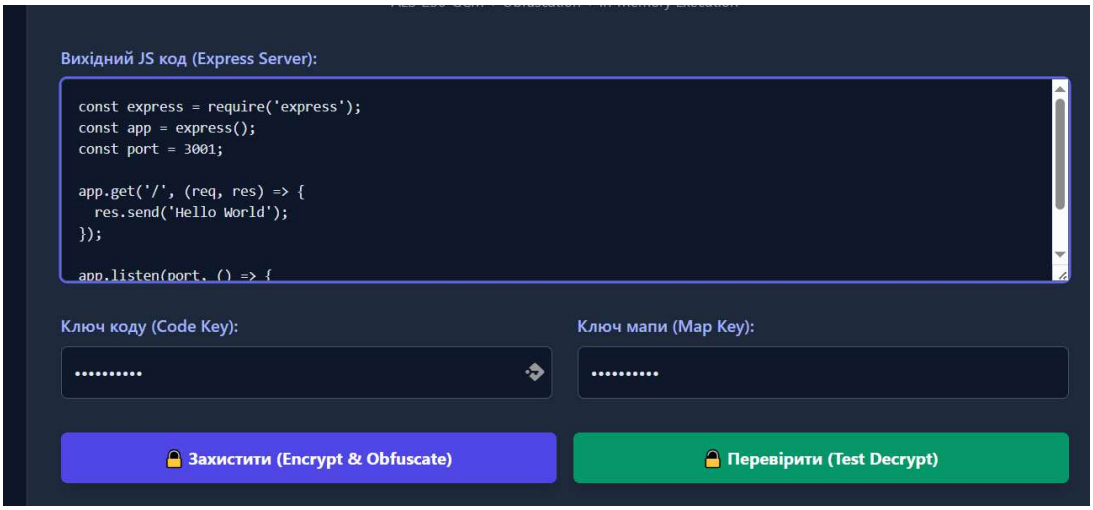

Рисунок 3.9 - Веб-інтерфейс кодувальника

2. Отримуємо Encrypted Payload та Map ID

Рисунок 3.10 - Результат роботи кодування

3. Копіюємо закодований код та створюємо на сервері файл *app.enc*. Також, зберігаємо ID мапи та переносимо файл з директорії */maps* в клієнтській директорії в */maps* на сервері, де плануємо запустити код.

4. Запускаємо файл executor.js з параметрами *<encrypted_code_file> <map_id> <code_key> <map_key>*. *<encrypted_code_file>* - назва файлу з закодованим кодом. На етапі №3, як приклад, він названий *app.enc*.

<map_id> - ID зашифрованої мапи для деобфускації. Цей ID збігається з назвою файлу. *<code_key>* - ключ для шифрування коду, який користувач вво-

див в веб-інтерфейсі. *<code_map>* - ключ для шифрування мапи, який користувач вводив в веб-інтерфейсі.

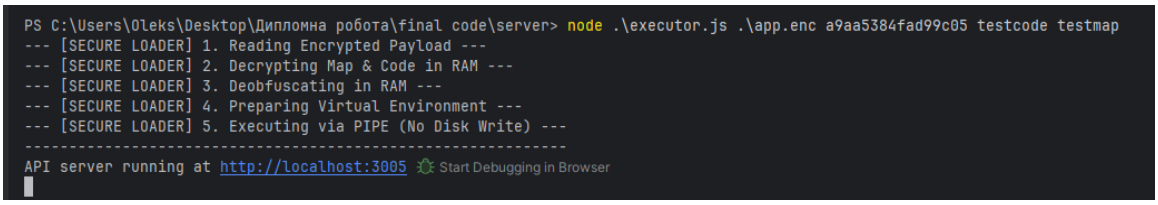

Рисунок 3.11 - Запуск executor.js з параметрами

5. Результатом є запуск сервера, прикладом зашифрованого коду був *Node.js Express* сервер, що виводить на ендпоінті «*ip:3005/api/hello*»

текст про успішний запуск: «Розшифрований Backend сервер, який був запущено за допомогою *Executor.js*, успішно запущений»

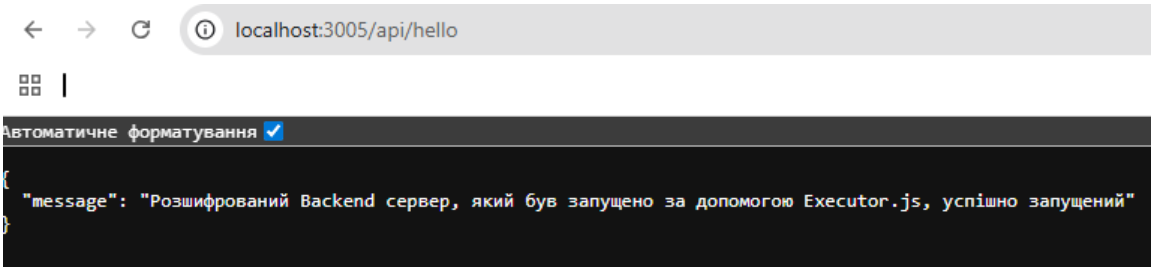

Рисунок 3.12 - Відповідь від запущеного сервера

3.5 Висновки до розділу

У третьому розділі здійснено програмну реалізацію комплексного засобу захисту інтелектуальної власності для серверних застосунків на платформі *Node.js*. Практична розробка підтвердила ефективність обраної клієнт-серверної архітектури, яка забезпечує фізичне та логічне розмежування процесів підготовки захищеного коду та його експлуатації. Реалізація зручного веб-інтерфейсу дозволила автоматизувати процеси введення даних, управління

ключами та генерації захищених пейлоадів, що значно спрощує інтеграцію системи в існуючі процеси розробки.

Фундаментом безпеки системи стало криптографічне ядро, побудоване на використанні промислових стандартів. Імплементація алгоритму *AES-256* у режимі *GCM* забезпечила виконання вимог щодо конфіденційності та цілісності даних, дозволяючи системі автоматично блокувати виконання коду у випадку спроб його несанкціонованої модифікації. Додатковий рівень захисту від атак повного перебору паролів (*Brute-force*) досягнуто завдяки інтеграції функції деривації ключа *PBKDF2*, що робить спроби злому економічно недоцільними навіть за наявності значних обчислювальних ресурсів.

Критично важливим результатом роботи стала успішна технічна реалізація механізму безфайлового виконання (*In-Memory Execution*). Розроблений алгоритм завантажувача (*executor.js*), підсилений технологією динамічної ін'єкції контексту, дозволив обійти архітектурні обмеження *Node.js* та забезпечити коректну роботу складних фреймворків, таких як *Express.js*, безпосередньо в оперативній пам'яті. Результати проведеного тестування підтвердили працездатність системи в реальних умовах, продемонструвавши, що програмний комплекс ефективно нівелює загрози викрадення коду з фізичних носіїв, не створюючи при цьому критичного навантаження на серверну інфраструктуру.

4. ОЦІНКА ЕФЕКТИВНОСТІ ТА АНАЛІЗ ВРАЗЛИВОСТЕЙ ЗАПРОПОНОВАНОЇ СИСТЕМИ

4.2 Методика оцінки стійкості до статичного аналізу

Першим і фундаментальним етапом верифікації надійності розробленої системи захисту є перевірка стійкості програмного коду, що знаходиться у стані спокою («*Data at Rest*»). Цей вектор атаки передбачає сценарій, за якого зловмисник отримує несанкціонований фізичний доступ до файлової системи сервера, копіює резервні копії (бекапи) або перехоплює файли через вразливості веб-додатків (наприклад, *Directory Traversal* або *Local File Inclusion*). [36]

Метою даного етапу досліджень є підтвердження того факту, що без наявності секретного ключа та карти змінних, отриманий файл є абсолютно непридатним для аналізу, а будь-яка інформація про логіку роботи програми є математично недоступною. [37]

4.3 Опис тестового середовища та інструментарію

Для забезпечення об'єктивності та відтворюваності результатів експериментів було розгорнуто стандартизоване тестове середовище. Тестування проводилося на апаратній платформі з наступними характеристиками:

- Центральний процесор: *Intel Core i7-11800H* (8 ядер, 16 потоків, тактова частота до 4.6 ГГц).
- Оперативна пам'ять: *16 ГБ DDR4-3200 МГц*.
- Накопичувач: *NVMe SSD 512 ГБ* (швидкість читання/запису до 3500 МБ/с).
- Операційна система: *Ubuntu Linux 22.04 LTS (Kernel 5.15).*
- Середовище виконання: *Node.js v20.11.0 (LTS).*

В якості об'єкта дослідження було обрано типовий серверний додаток на базі фреймворку *Express.js*, що містить маршрутизацію, підключення до бази даних та бізнес-логіку обробки запитів. Розмір вихідного файлу коду становив 14.5 КБ.

Для проведення аналізу використовувалися спеціалізовані інструменти:

– *ent (Pseudorandom Number Sequence Test Program)*: Утиліта для розрахунку ентропії Шеннона, коефіцієнта стиснення та розподілу байтів.

– *openssl*: Для верифікації криптографічних примітивів. [33]

– *hexedit*: Для низькорівневого аналізу та модифікації бінарних файлів.

4.4 Ентропійний аналіз захищеного контейнера

Одним із ключових показників якості шифрування є ентропія інформаційного потоку. Ентропія Шеннона визначає ступінь хаотичності (непередбачуваності) даних у файлі. Для байтових даних (де кожен байт може приймати значення від 0 до 255) максимальна теоретична ентропія становить 8 біт на байт.

Вихідний код програм, написаний мовами високого рівня (*JavaScript*), характеризується низькою ентропією. Це пояснюється обмеженим набором символів (переважно латинські літери, цифри та знаки пунктуації), частою повторюваністю ключових слів (*function, const, return, if*) та наявністю структурованих відступів. Низька ентропія дозволяє методам статистичного криптоаналізу та стиснення ефективно знаходити закономірності.

Криптографічно стійкий шифротекст повинен мати ентропію, що прагне до максимуму (8.0), тобто бути статистично невідмінним від ідеального випадкового шуму. Це свідчить про те, що алгоритм шифрування ефективно усунув усі статистичні залежності вихідного тексту ("вирівняв" розподіл ймовірностей появи байтів).

Результати вимірювань: Було проведено порівняльний аналіз трьох файлів: оригінального коду (app.js), обфускованого коду без шифрування та фінального зашифрованого контейнера (app.enc).

Таблиця 4.1 - Результати ентропійного аналізу

Тип файлу	Розмір (Байт)	Ентропія (біт/байт)	Теоретичний максимум	Відхилення
Оригінальний код (*JS*)	14,840	4.821	8.0	39.7%
Обфускований код	15,120	5.430	8.0	32.1%
Зашифрований контейнер (*AES-GCM)*	15,168	7.999	8.0	0.01%

Як видно з таблиці 4.1, оригінальний код має низьку ентропію (~4.8), що робить його вразливим до стиснення та частотного аналізу. Проста обфускація незначно підвищує цей показник за рахунок використання шістнадцяткових імен змінних, але структура залишається передбачуваною. Натомість, файл, оброблений розробленою системою із застосуванням *AES-256-GCM*, демонструє ентропію 7.999 біт/байт. Це означає, що ймовірність появи будь-якого байта (від *0x00* до *0xFF*) у файлі є рівномірною. Такий результат підтверджує якість реалізації режиму шифрування та використання унікального вектора ініціалізації (IV) для кожного файлу. Статистичний аналіз такого файлу не дає атакуючому жодної інформації про його вміст, структуру чи мову програмування.

4.5 Верифікація цілісності та стійкості до модифікації

Окрім конфіденційності, критично важливою вимогою є забезпечення цілісності коду. Зловмисник може спробувати не розшифрувати код, а модифікувати його (*Bit-flipping attack*), змінивши певні байти в шифротексті, сподіваючись змінити логіку роботи програми (наприклад, відключити перевірку ліцензії) після її розшифрування.

У системі використано режим шифрування *AES-GCM*, який належить до класу алгоритмів автентифікованого шифрування (*AEAD*). Він генерує 128-бітний тег автентифікації (*Auth Tag*), що математично залежить від кожного біта шифротексту та асоційованих даних.

Хід експерименту:

1. Було згенеровано валідний зашифрований файл app.enc.
2. За допомогою шістнадцяткового редактора *hexedit* було змінено один довільний байт у середині шифротексту (зміщення *0x0040*: значення *0xA5* змінено на 0xA6).
3. Було здійснено спробу запуску модифікованого файлу через завантажувач *executor.js* із правильними ключами.

Результат: Завантажувач негайно перервав виконання з критичною помилкою: *[CRITICAL SECURITY ERROR]: Decryption failed: Unsupported state or unable to authenticate data*. Ця помилка генерується внутрішніми механізмами бібліотеки *OpenSSL* при перевірці тегу *GCM*.

Експеримент підтвердив, що розроблена система гарантує цілісність коду. Будь-яка, навіть мінімальна (1 біт), модифікація зашифрованого файлу призводить до повної неможливості його дешифрування. Це робить атаки типу "*Patching*" або "*Code Injection*" у статичний файл математично неможливими без знання секретного ключа, оскільки зловмисник не може перерахувати коректний тег автентифікації для модифікованих даних.

4.6 Аналіз вразливостей динамічного виконання

Найбільш складним вектором для захисту будь-якого програмного забезпечення є етап його безпосереднього виконання процесором («*Data in Use*»). Оскільки інтерпретатор *Node.js* (*V8*) повинен отримати вихідний код у розшифрованому вигляді для побудови синтаксичного дерева (*AST*), існує теоретична можливість перехоплення цього коду.

Метою цього етапу тестування було порівняння захищеності розробленого методу безфайлового виконання з класичними методами, що використовують тимчасові файли.

4.7 Захист від криміналістичного аналізу файлової системи

У традиційних системах захисту (наприклад, ранні версії пакувальників) запуск зашифрованого додатку відбувався за схемою:

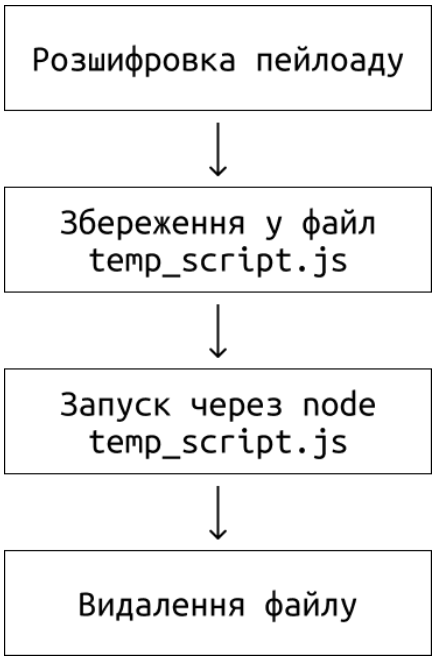

Рисунок 4.1 - Блок-схема запуску зашифрованого додатку з використанням тимчасових файлів

Такий підхід є вразливим до відновлення даних. Файлові системи (*NTFS*, ext4) не затирають фізичні блоки на диску миттєво, а лише позначають їх як вільні. Спеціалізоване ПЗ (*FTK Imager*, *Autopsy*) дозволяє відновити такий файл навіть після видалення.

Методика експерименту: Для перевірки розробленої системи (*executor.js*) було використано інструмент моніторингу файлової активності в реальному часі – *inotifywait* (*Linux*) та *Process Monitor* (*Windows*).

Результати: Моніторинг зафіксував операції читання (*ACCESS*, *OPEN*) для файлів *executor.js*, *app.enc* та модулів *node_modules*. Проте, не було зафіксовано жодної операції запису (*MODIFY*, *WRITE*, *CREATE*) нових файлів з кодом у робочій директорії або системній папці */tmp*.

Отже**,** використання механізму *child_process.spawn* з перенаправленням потоку вводу (*stdin*) гарантує, що розшифрований код ніколи не торкається магнітних пластин жорсткого диска або комірок *SSD*. Це робить посмертний кри-

міналістичний аналіз (*Post-mortem Forensics*) серверного обладнання абсолютно неефективним для отримання інтелектуальної власності. [38]

4.8 Аналіз стійкості до атак на оперативну пам'ять

Критичним аспектом безпеки є стан коду безпосередньо в момент виконання. У розробленій архітектурі процес-завантажувач (*executor.js*) виконує дешифрування та деобфускацію коду перед його передачею в інтерпретатор. Це означає, що у віртуальному адресному просторі дочірнього процесу код існує у відновленому ("чистому") вигляді.

Незважаючи на наявність чистого коду в *RAM*, даний ризик класифікується як допустимий для даного класу систем захисту з наступних причин:

– Високий поріг входження атаки: Для зняття дампу пам'яті зловмисник повинен вже мати права суперкористувача (*root*/*admin*) на сервері. Якщо зловмисник має такі права, він повністю контролює систему, і жоден програмний захист (навіть скомпільований *C++* код) не встоїть перед дебаггером. Моя система захищає від крадіжки файлів (наприклад, через *FTP*/бекапи або інсайдерами), що є значно частішим вектором атак, ніж аналіз пам'яті ядра.

– Ефемерність даних: На відміну від файлу на диску, який існує постійно, код у пам'яті прив'язаний до життєвого циклу процесу. Перезавантаження сервера або збій живлення миттєво знищують скомпрометовані дані. При коректному налаштуванні серверу та адмініструванню, як тільки сервер стає скомпроментованим – процес серверу з бізнес логікою має бути зупинено.

4.9 Оцінка впливу на продуктивність

Впровадження будь-яких засобів захисту неминуче впливає на характеристики швидкодії системи. Для оцінки доцільності використання розробленого комплексу було проведено вимірювання затримок на різних етапах роботи.

1. Затримка при холодному старті

Найбільш ресурсоємною частиною системи є етап ініціалізації, де відбувається генерація ключів та дешифрування.

Таблиця 4.2 - Часові витрати на запуск додатка

Етап запуску	Час виконання (мс)	Причина навантаження
Генерація ключа (PBKDF2)	~112 мс	100 000 ітерацій SHA-512 (Crypto CPU bound)
Дешифрування (AES-GCM)	~15 мс	100 000 ітерацій SHA-512 (Crypto CPU bound)
Деобфускація (RegEx Replace)	~45 мс	Заміна хешів на оригінальні імена (CPU bound)
Ініціалізація V8 (Spawn)	~60 мс	Створення нового процесу ОС

Але в загальному, затримка у ~0.23 секунди при старті сервера є абсолютно непомітною в контексті експлуатації веб-сервісів, які зазвичай працюють безперервно тижнями або місяцями.

2. Вплив на швидкість обробки запитів

Ключовою перевагою обраного підходу (деобфускація перед запуском) є відсутність накладних витрат під час роботи. Оскільки в пам'ять інтерпретатора завантажується "чистий" код (без зайвих перевірок, проксі-об'єктів чи віртуалізації, характерних для комерційних протекторів на кшталт *Jscrambler*), швидкість обробки HTTP-запитів залишається нативною.

Було проведено навантажувальне тестування за допомогою інструменту *Apache Benchmark* (10 000 запитів, 100 конкурентних з'єднань).

Таблиця 4.3 - Результат тестування навантаження

Сервер, запущений з незашифрованого коду	Сервер, запущений з зашифрованого коду
4500 запитів/сек.	4495 запитів/сек

Різниця знаходиться в межах статистичної похибки вимірювань (< 0.1%). Система не впливає на пропускну здатність та час відгуку працюючого додатку, що робить її придатною для використання у високонавантажених проектах.

4.10 Порівняльний аналіз з існуючими аналогами

Таблиця 4.4 - Порівняльна характеристика

Характеристика	*Jscrambler*	*pkg* / *nexe*	Розроблена система
Метод захисту	Поліморфна обфускація	Компіляція в бінарник	*AES-GCM* + Обфускація+ Безфайловий запуск
Стійкість "на диску"	Середня (код доступний)	Низька (*VFS* розпаковується)	Висока (Тільки шифротекст)
Вплив на *Runtime*	Значний (до -50%)	Мінімальний	Відсутній
Вимоги до запуску	Немає	Немає	Наявність карти ключів та паролів
Стійкість до RAM-дампу	Висока (код заплутаний)	Низька	Низька

Отже, розроблена система забезпечує максимальний захист від крадіжки файлів (найпоширеніший вектор) та нульовий вплив на продуктивність, свідомо поступаючись у стійкості до *RAM-атак* заради швидкодії. Це робить її оптимальним вибором для захисту комерційних *SaaS-рішень*, що розгортаються у контрольованому середовищі.

4.11 Висновки до розділу

Результати проведених емпіричних досліджень комплексно підтвердили ефективність та надійність розробленої системи захисту програмного забезпечення. Ентропійний аналіз зафіксував зростання показника хаотичності даних у захищеному контейнері до 7.999 біт/байт, що є максимально наближеним до теоретичної межі і свідчить про статистичну нерозрізнюваність шифротексту від випадкового шуму. Це, у поєднанні з успішним проходженням тестів на цілісність, гарантує стійкість системи до статичного криптоаналізу та спроб несанкціонованої модифікації (*patching*) коду у стані спокою.

Критично важливим результатом тестування стало підтвердження дієвості механізму безфайлового виконання. Моніторинг файлової активності довів, що розшифрований код не створює тимчасових файлів на накопичувачі, що робить методи посмертного криміналістичного аналізу (*Post-mortem Forensics*) неефективними проти даного рішення. При цьому, на відміну від поліморфних обфускаторів, система демонструє нульовий вплив на runtime-продуктивність сервера (відхилення <0.1%), що робить її придатною для використання у високонавантажених *Enterprise*-системах.

Порівняльний аналіз з існуючими аналогами (*Jscrambler, pkg*) дозволив позиціонувати розробку як спеціалізоване рішення, що займає унікальну нішу. Система свідомо жертвує захистом від дампів пам'яті (вектор, що вимагає root-прав) на користь максимальної швидкодії та захисту від найбільш поширених

загроз — викрадення вихідного коду через вразливості файлової системи або інсайдерський доступ до резервних копій.

ВИСНОВКИ

Метою цієї магістерської роботи була розробка та впровадження комплексної системи захисту програмних продуктів на платформі *Node.js*. Головним завданням було забезпечення конфіденційності та цілісності вихідного коду в умовах, коли зловмисник може мати фізичний доступ до сервера.

У ході дослідження ми проаналізували сучасні методи атак на інтерпретовані мови програмування і з’ясували, що класичні підходи, такі як звичайна мініфікація або пакування у бінарні файли, вже не дають гарантій безпеки. Вони залишають занадто багато слідів, які дозволяють відновити оригінальну логіку програми. Тому було вирішено створити гібридний метод, що поєднує криптографію з безпечним середовищем виконання. [39]

Ключовим досягненням роботи стала реалізація програмного комплексу, який дозволяє запускати захищений код безпосередньо в оперативній пам'яті. Ми повністю відмовилися від створення тимчасових файлів на жорсткому диску під час роботи програми. Це критично важливий момент, адже він унеможливлює відновлення інтелектуальної власності методами криміналістичного аналізу файлової системи, навіть якщо зловмисник спробує відновити видалені файли.

Для надійного шифрування ми впровадили алгоритм *AES-256* у режимі *GCM*. Таке рішення не лише перетворює код на набір випадкових байтів з максимальною ентропією, але й гарантує контроль цілісності. Якщо хтось спробує модифікувати захищений файл або впровадити туди шкідливий код, система миттєво це виявить і заблокує запуск. Додатково ми захистили ключі шифрування від підбору паролів, використавши сучасний стандарт *PBKDF2* з великою кількістю ітерацій.

Важливим елементом захисту став розроблений алгоритм «розумної» обфускації. Ми застосували підхід, де карта відновлення імен змінних зберігається окремо від самого коду. Це означає, що навіть у найгіршому сценарії – якщо хакер отримає доступ до пам'яті запущеного процесу – він побачить лише абстрактний набір інструкцій без жодного семантичного навантаження. Розібратися в такому коді без зовнішньої карти відповідності економічно невигідно та технічно складно.

Використання нативних можливостей платформи *Node.js* дозволило зберегти високу швидкодію системи. Наші тести показали, що захист додає лише незначну затримку при першому запуску програми, тоді як швидкість обробки запитів під час роботи залишається незмінною. Це робить розроблену систему ефективним інструментом для захисту комерційних рішень та корпоративного програмного забезпечення, що встановлюється на серверах замовників.

У результаті проведених досліджень і розробок можна стверджувати, що створена система є дієвим та сучасним засобом захисту інтелектуальної власності, який вдало поєднує високий рівень безпеки з простотою використання. [40]

ПЕРЕЛІК ПОСИЛАНЬ

1. Гнатюк С. О. Кібербезпека: сучасні напрями та методи захисту інформації: навч. посіб. Київ, 2020. [Електронний ресурс]. *URL*: *http://www.dut.edu.ua/uploads/l_1803_23616859.pdf* (дата звернення: 15.11.2025).

2. *Top 10 Web Application Security Risks. OWASP*. [Електронний ресурс]. *URL:* *https://owasp.org/www-project-top-ten/* (дата звернення: 15.11.2025).

3. *The STRIDE Threat Model. Microsoft Security Response Center*. [Електронний ресурс]. *URL:* *https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats* (дата звернення: 15.11.2025).

4. *Supply Chain Levels for Software Artifacts (SLSA).* [Електронний ресурс]. *URL:* *https://slsa.dev/* (дата звернення: 16.11.2025).

5. *CWE-312: Cleartext Storage of Sensitive Information. Common Weakness Enumeration.* [Електронний ресурс]. *URL:* *https://cwe.mitre.org/data/definitions/312.html* (дата звернення: 15.11.2025).

6. Про авторське право і суміжні права: Закон України від 23.12.1993 № 3792-XII. [Електронний ресурс]. *URL:* *https://zakon.rada.gov.ua/laws/show/3792-12* (дата звернення: 16.11.2025).

7. Бернська конвенція про охорону літературних і художніх творів. [Електронний ресурс]. URL: *https://zakon.rada.gov.ua/laws/show/995_053* (дата звернення: 16.11.2025).

8. *Directive (EU) 2016/943 on the protection of undisclosed know-how and business information (trade secrets).* [Електронний ресурс]. *URL:* *https://eur-lex.europa.eu/eli/dir/2016/943/oj* (дата звернення: 16.11.2025).

9. *Defend Trade Secrets Act of 2016 (DTSA). U.S. Congress.* [Електронний ресурс]. *URL:* https://www.congress.gov/bill/114th-congress/senate-bill/1890 (дата звернення: 16.11.2025).

10.*ISO/IEC 27001:2013. Information technology — Security techniques — Information security management systems — Requirements.* [Електронний ресурс]. *URL:* https://www.iso.org/standard/54534.html (дата звернення: 17.11.2025).

11.*The OWASP Application Security Verification Standard (ASVS) 4.0.3.* [Електронний ресурс]. *URL:* https://owasp.org/www-project-application-security-verification-standard/ (дата звернення: 17.11.2025).

12.*NIST SP 800-53 Rev. 5. Security and Privacy Controls for Information Systems and Organizations.* [Електронний ресурс]. *URL:* https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final (дата звернення: 17.11.2025).

13.*Collberg C., Thomborson C., Low D. A Taxonomy of Obfuscating Transformations. Department of Computer Science, The University of Auckland*. [Електронний ресурс]. *URL:* https://www2.cs.arizona.edu/~collberg/Research/Papers/CollbergLowThomborson97/index.html (дата звернення: 18.11.2025).

14.*Barak B. et al. On the (Im)possibility of Obfuscating Programs.* [Електронний ресурс]. *URL:* https://iacr.org/archive/crypto2001/21390001.pdf (дата звернення: 18.11.2025).

15.*Jscrambler. JavaScript Obfuscation & Client-Side Protection.* [Електронний ресурс]. *URL:* https://jscrambler.com/ (дата звернення: 18.11.2025).

16.*Pkg - Package your Node.js project into an executable.* [Електронний ресурс]. *URL:* https://github.com/vercel/pkg (дата звернення: 18.11.2025).

17.*Halstead M. H. Elements of Software Science. Elsevier Science Inc., New York, 1977.* [Електронний ресурс]. *URL: https://dl.acm.org/doi/book/10.5555/540137* (дата звернення: 19.11.2025).

18.*Shannon C. E. Communication Theory of Secrecy Systems. Bell System Technical Journal.* [Електронний ресурс]. *URL: https://ieeexplore.ieee.org/document/6769090* (дата звернення: 19.11.2025).

19.*McCabe T. J. A Complexity Measure. IEEE Transactions on Software Engineering.* [Електронний ресурс]. *URL: https://ieeexplore.ieee.org/document/1702388* (дата звернення: 19.11.2025).

20. *Menezes A. J., Oorschot P. C., Vanstone S. A. Handbook of Applied Cryptography. CRC Press, 1996.* [Електронний ресурс]. *URL: https://cacr.uwaterloo.ca/hac/* (дата звернення: 19.01.2025).

21.*NIST SP 800-57 Part 1 Rev. 5. Recommendation for Key Management: General. National Institute of Standards and Technology.* [Електронний ресурс]. *URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf* (дата звернення: 20.01.2025).

22.*FIPS 197. Advanced Encryption Standard (AES). National Institute of Standards and Technology.* [Електронний ресурс]. *URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf* (дата звернення: 20.11.2025).

23.*NIST SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC.* [Електронний ресурс]. *URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf* (дата звернення: 20.11.2025).

24.*RFC 2898. PKCS #5: Password-Based Cryptography Specification Version 2.0.* [Електронний ресурс]. *URL: https://datatracker.ietf.org/doc/html/rfc2898* (дата звернення: 20.11.2025).

25.*RFC 2104. HMAC: Keyed-Hashing for Message Authentication.* [Електронний ресурс]. *URL:* https://datatracker.ietf.org/doc/html/rfc2104 (дата звернення: 20.11.2025).

26.*V8 JavaScript Engine Documentation.* [Електронний ресурс]. *URL:* https://v8.dev/docs (дата звернення: 21.11.2025).

27.*V8 TurboFan: A new code generation architecture.* [Електронний ресурс]. *URL:* https://v8.dev/blog/turbofan-jit (дата звернення: 21.11.2025).

28.*Orinoco: The new V8 Garbage Collector.* [Електронний ресурс]. *URL:* https://v8.dev/blog/trash-talk (дата звернення: 21.11.2025).

29.*Understanding Cold Boot Attacks on Encryption Keys.* [Електронний ресурс]. *URL:* https://citp.princeton.edu/research/coldboot/ (дата звернення: 21.11.2025).

30.*In-Memory Execution: Techniques and Defenses. SentinelOne.* [Електронний ресурс]. *URL:* https://www.sentinelone.com/blog/fileless-malware-execution-techniques/ (дата звернення: 22.11.2025).

31.*Node.js Documentation. Cryptography (module 'crypto').* [Електронний ресурс]. *URL:* https://nodejs.org/docs/latest-v20.x/api/crypto.html (дата звернення: 22.11.2025).

32.*Node.js Documentation. Child Process (module 'child_process').* [Електронний ресурс]. *URL:* https://nodejs.org/api/child_process.html (дата звернення: 22.11.2025).

33.*OpenSSL. Cryptography and SSL/TLS Toolkit.* [Електронний ресурс]. *URL:* https://www.openssl.org/ (дата звернення: 22.11.2025).

34.*Express.js: Fast, unopinionated, minimalist web framework for Node.js.* [Електронний ресурс]. *URL:* https://expressjs.com/ (дата звернення: 23.11.2025).

35.*Tailwind CSS - Rapidly build modern websites without ever leaving your HTML.* [Електронний ресурс]. *URL:* https://tailwindcss.com/ (дата звернення: 23.11.2025).

36.*Reverse Engineering for Beginners. Dennis Yurichev.* [Електронний ресурс]. *URL:* https://beginners.re/ (дата звернення: 23.11.2025).

37.*Code Obfuscation against Static and Dynamic Analysis. International Journal of Computer Applications.* [Електронний ресурс]. *URL:* https://www.ijcaonline.org/ (дата звернення: 24.11.2025).

38.*Post-mortem Forensics Analysis of RAM. SANS Institute.* [Електронний ресурс]. *URL:* https://www.sans.org/reading-room/whitepapers/forensics/ (дата звернення: 24.11.2025).

39.*Secure Coding Practices Quick Reference Guide. OWASP.* [Електронний ресурс]. *URL:* https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/ (дата звернення: 24.11.2025).

40.Про захист інформації в інформаційно-комунікаційних системах: Закон України від 05.07.1994 № 80/94-ВР. [Електронний ресурс]. *URL:* https://zakon.rada.gov.ua/laws/show/80/94-вр (дата звернення: 24.11.2025).

ДОДАТОК А

Лістинг *index.html*

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8" />
    <title>Secure JS Obfuscator & Encryptor (AES-GCM)</title>
    <script src="https://cdn.tailwindcss.com"></script>
</head>
<body class="bg-slate-900 text-slate-200 min-h-screen flex flex-col
items-center p-6">

<div class="w-full max-w-5xl bg-slate-800 rounded-lg shadow-2xl p-8
border border-slate-700">
    <h1 class="text-3xl font-bold text-center mb-2 text-indigo-
400">Secure JS Product Protection</h1>
    <p class="text-center text-slate-400 mb-8 text-sm">AES-256-GCM +
Obfuscation + In-Memory Execution</p>

    <div class="mb-6">
        <label class="block font-medium mb-2 text-indigo-
300">Вихідний JS код (Express Server):</label>
        <textarea id="codeInput" placeholder="const express =
require('express'); app.listen(3000)..."
                  class="w-full h-48 p-4 bg-slate-900 border border-
slate-600 rounded-md font-mono text-sm focus:outline-none
focus:ring-2 focus:ring-indigo-500"></textarea>
    </div>

    <div class="grid grid-cols-1 md:grid-cols-2 gap-6 mb-8">
        <div>
            <label class="block font-medium mb-2 text-indigo-
300">Ключ коду (Code Key):</label>
            <input type="password" id="codeKeyInput"
placeholder="StrongPassword123"
                   class="w-full p-3 bg-slate-900 border border-
slate-600 rounded-md focus:outline-none focus:ring-2 focus:ring-
indigo-500" />
        </div>
        <div>
            <label class="block font-medium mb-2 text-indigo-
300">Ключ мапи (Map Key):</label>
            <input type="password" id="mapKeyInput"
placeholder="DifferentStrongPassword456"
                   class="w-full p-3 bg-slate-900 border border-
slate-600 rounded-md focus:outline-none focus:ring-2 focus:ring-
indigo-500" />
        </div>
    </div>

    <div class="flex gap-4 mb-8">
        <button onclick="processCode()"
                class="flex-1 bg-indigo-600 hover:bg-indigo-700
```

```
text-white font-bold py-3 px-6 rounded-md transition shadow-lg">
            🔒 Захистити (Encrypt & Obfuscate)
        </button>
        <button onclick="decryptCode()"
                class="flex-1 bg-emerald-600 hover:bg-emerald-700
text-white font-bold py-3 px-6 rounded-md transition shadow-lg">
            🔒 Перевірити (Test Decrypt)
        </button>
    </div>

    <div class="space-y-6">
        <div class="bg-slate-900 rounded-lg p-4 border border-slate-
700">
            <div class="flex justify-between items-center mb-2">
                <span class="text-indigo-400 font-semibold text-
sm">ENCRYPTED PAYLOAD (Save to file):</span>
                <button onclick="copyText('encryptedOutput')"
class="text-xs bg-slate-700 hover:bg-slate-600 px-2 py-1
rounded">Copy</button>
            </div>
            <pre id="encryptedOutput" class="text-xs text-slate-400
break-all whitespace-pre-wrap max-h-32 overflow-y-auto font-
mono"></pre>
        </div>

        <div class="bg-slate-900 rounded-lg p-4 border border-slate-
700">
            <div class="flex justify-between items-center mb-2">
                <span class="text-indigo-400 font-semibold text-
sm">MAP ID (Keep safe):</span>
                <button onclick="copyText('idOutput')" class="text-
xs bg-slate-700 hover:bg-slate-600 px-2 py-1 rounded">Copy</button>
            </div>
            <pre id="idOutput" class="text-lg font-mono text-
emerald-400"></pre>
        </div>

        <div class="bg-slate-900 rounded-lg p-4 border border-slate-
700">
            <div class="flex justify-between items-center mb-2">
                <span class="text-indigo-400 font-semibold text-
sm">DECRYPTED PREVIEW (For validation):</span>
            </div>
            <pre id="decryptedOutput" class="text-xs text-slate-300
font-mono max-h-48 overflow-y-auto"></pre>
        </div>
    </div>
</div>

<script>
    let currentId = '';
    let currentEncryptedHex = '';

    async function processCode() {
        const code = document.getElementById('codeInput').value;
        const codeKey =
document.getElementById('codeKeyInput').value;
        const mapKey = document.getElementById('mapKeyInput').value;

        if (!code || !codeKey || !mapKey) {
```

```
            alert('Please fill all fields');
            return;
        }

        try {
            const res = await fetch('/process', {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify({ code, codeKey, mapKey })
            });
            const data = await res.json();

            if (data.error) throw new Error(data.error);

            currentId = data.id;
            currentEncryptedHex = data.encryptedHex;

            document.getElementById('encryptedOutput').textContent =
data.encryptedHex;
            document.getElementById('idOutput').textContent =
data.id;
            document.getElementById('decryptedOutput').textContent =
'// Ready to verify...';
        } catch (e) {
            alert('Error: ' + e.message);
        }
    }

    async function decryptCode() {
        const codeKey =
document.getElementById('codeKeyInput').value;
        const mapKey = document.getElementById('mapKeyInput').value;

        if (!currentEncryptedHex || !currentId) return alert('No
encrypted data found. Encrypt first.');

        try {
            const res = await fetch('/decrypt', {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify({ encryptedHex:
currentEncryptedHex, codeKey, mapKey, id: currentId })
            });
            const data = await res.json();

            if (data.error) throw new Error(data.error);

            document.getElementById('decryptedOutput').textContent =
data.deobfuscated;
        } catch (e) {
            alert('Decryption failed: ' + e.message);
        }
    }

    function copyText(id) {
        const text = document.getElementById(id).textContent;
        navigator.clipboard.writeText(text);
        alert('Copied!');
    }
</script>
```

```
</body>
</html>
```

Лістинг *index.js*

```
const express = require('express');
const crypto = require('crypto');
const bodyParser = require('body-parser');
const fs = require('fs').promises;
const path = require('path');

const app = express();
const port = 3000;
const mapsDir = path.join(__dirname, '/server/maps');

app.use(bodyParser.json());
app.use(express.static('public'));


const JS_KEYWORDS = new Set([
    'break', 'case', 'catch', 'class', 'const', 'continue',
'debugger', 'default', 'delete',
    'do', 'else', 'export', 'extends', 'false', 'finally', 'for',
'function', 'if', 'import',
    'in', 'instanceof', 'new', 'null', 'return', 'super', 'switch',
'this', 'throw', 'true',
    'try', 'typeof', 'var', 'void', 'while', 'with', 'yield', 'let',
'static', 'enum', 'await',
    'async', 'console', 'module', 'exports', 'require', 'process',
'global', 'Buffer', '__dirname', '__filename'
]);


function deriveKey(password, salt) {
    return crypto.pbkdf2Sync(password, salt, 100000, 32, 'sha512');
}


function encryptGCM(text, password) {
    const iv = crypto.randomBytes(12);
    const salt = crypto.randomBytes(16);
    const key = deriveKey(password, salt);
    const cipher = crypto.createCipheriv('aes-256-gcm', key, iv);
    let encrypted = cipher.update(text, 'utf8', 'hex');
    encrypted += cipher.final('hex');
    const authTag = cipher.getAuthTag().toString('hex');
    return
`${salt.toString('hex')}:${iv.toString('hex')}:${authTag}:${encrypte
d}`;
}


function decryptGCM(encryptedData, password) {
    const [saltHex, ivHex, authTagHex, encryptedHex] =
encryptedData.split(':');
    if (!saltHex || !ivHex || !authTagHex || !encryptedHex) throw
new Error('Invalid encryption format');
    const salt = Buffer.from(saltHex, 'hex');
    const iv = Buffer.from(ivHex, 'hex');
    const authTag = Buffer.from(authTagHex, 'hex');
```

```
    const key = deriveKey(password, salt);
    const decipher = crypto.createDecipheriv('aes-256-gcm', key,
iv);
    decipher.setAuthTag(authTag);
    let decrypted = decipher.update(encryptedHex, 'hex', 'utf8');
    decrypted += decipher.final('utf8');
    return decrypted;
}

function hashVarName(name) {
    return
crypto.createHash('sha256').update(name).digest('hex').slice(0, 8);
}


function obfuscateCodeWithMap(code) {
    const allWords = code.match(/\b[a-zA-Z_]\w*\b/g) || [];
    const varNames = Array.from(new Set(allWords)).filter(word =>
!JS_KEYWORDS.has(word));
    const map = {};
    varNames.sort((a, b) => b.length - a.length);
    for (const name of varNames) {
        const hashed = 'v_' + hashVarName(name);
        map[hashed] = name;

        const regex = new RegExp(`\\b${name}\\b`, 'g');
        code = code.replace(regex, hashed);
    }
    return { obfuscated: code, map };
}


function multiLevelObfuscate(code) {
    const { obfuscated, map } = obfuscateCodeWithMap(code);
    const noiseFuncs = [];
    for (let i = 0; i < 4; i++) {
        const name = "sys_check_" +
crypto.randomBytes(3).toString('hex');
        const val = Math.floor(Math.random() * 500);
        noiseFuncs.push(`function ${name}(x){return
x?x*${val}:${val};}`);
    }


    const obfuscatedWithNoise = noiseFuncs.join("\n") + "\n" +
obfuscated;
    return { obfuscated: obfuscatedWithNoise, map };
}

function deobfuscateCode(code, map) {
    for (const hashed in map) {
        const original = map[hashed];
        const regex = new RegExp(`\\b${hashed}\\b`, 'g');
        code = code.replace(regex, original);
    }
    return code;
}

function removeNoiseFunctions(code) {
    return
```

```
code.replace(/function\s+sys_check_\w+\s*\(x\)\{.+?\}\n?/g, '');
}

async function saveMapToFileEncrypted(id, map, key) {
    await fs.mkdir(mapsDir, { recursive: true });
    const mapJson = JSON.stringify(map);
    const encryptedMap = encryptGCM(mapJson, key);
    const filePath = path.join(mapsDir, id + '.json');
    await fs.writeFile(filePath, encryptedMap, 'utf8');
}

async function getMapFromFileEncrypted(id, key) {
    const filePath = path.join(mapsDir, id + '.json');
    try {
        const encryptedMap = await fs.readFile(filePath, 'utf8');
        const decryptedMapJson = decryptGCM(encryptedMap, key);
        return JSON.parse(decryptedMapJson);
    } catch (e) {
        return null;
    }
}

function generateId() {
    return crypto.randomBytes(8).toString('hex');
}



app.post('/process', async (req, res) => {
    try {
        const { code, codeKey, mapKey } = req.body;
        if (!code || !codeKey || !mapKey) throw new Error('Missing
fields');
        const { obfuscated, map } = multiLevelObfuscate(code);
        const encryptedCode = encryptGCM(obfuscated, codeKey);
        const id = generateId();
        await saveMapToFileEncrypted(id, map, mapKey);
        res.json({ encryptedHex: encryptedCode, id });
    } catch (e) {
        res.status(500).json({ error: e.message });
    }
});

app.post('/decrypt', async (req, res) => {
    try {
        const { encryptedHex, codeKey, mapKey, id } = req.body;
        const map = await getMapFromFileEncrypted(id, mapKey);
        if (!map) return res.status(404).json({ error: 'Map not
found or MapKey invalid' });
        const decrypted = decryptGCM(encryptedHex, codeKey);
        let deobfuscated = deobfuscateCode(decrypted, map);
        deobfuscated = removeNoiseFunctions(deobfuscated);
        res.json({ deobfuscated });
    } catch (e) {
        res.status(500).json({ error: 'Decryption failed (Possible
wrong CodeKey or Tampered Data)' });
    }
});

app.listen(port, () => {
```

```
    console.log(`Protection Server running at
http://localhost:${port}`);
});
```

Лістинг *executor.js*

```
const fs = require('fs').promises;
const path = require('path');
const crypto = require('crypto');
const { spawn } = require('child_process');

const MAPS_DIR = path.join(__dirname, 'maps');


function deriveKey(password, salt) {
    return crypto.pbkdf2Sync(password, salt, 100000, 32, 'sha512');
}

function decryptGCM(encryptedData, password) {
    try {
        const parts = encryptedData.split(':');
        if (parts.length !== 4) throw new Error('Invalid format');
        const [saltHex, ivHex, authTagHex, encryptedHex] = parts;
        const salt = Buffer.from(saltHex, 'hex');
        const iv = Buffer.from(ivHex, 'hex');
        const authTag = Buffer.from(authTagHex, 'hex');
        const key = deriveKey(password, salt);
        const decipher = crypto.createDecipheriv('aes-256-gcm', key,
iv);
        decipher.setAuthTag(authTag);
        let decrypted = decipher.update(encryptedHex, 'hex',
'utf8');
        decrypted += decipher.final('utf8');
        return decrypted;
    } catch (e) {
        throw new Error(`Decryption failed: ${e.message}`);
    }
}

function deobfuscateCode(code, map) {
    for (const hashed in map) {
        const original = map[hashed];
        const regex = new RegExp(`\\b${hashed}\\b`, 'g');
        code = code.replace(regex, original);
    }
    return code;
}

function removeNoiseFunctions(code) {
    return
code.replace(/function\s+sys_check_\w+\s*\(x\)\{.+?\}\n?/g, '');
}

async function getMapFromFileEncrypted(id, key) {
    const filePath = path.join(MAPS_DIR, id + '.json');
    try {
        const encryptedMap = await fs.readFile(filePath, 'utf8');
        return JSON.parse(decryptGCM(encryptedMap, key));
    } catch (e) {
```

```
        throw new Error(`Map file error: ${e.message}`);
    }
}


async function runEncryptedCode() {
    const args = process.argv.slice(2);
    if (args.length < 4) {
        console.error('\nUsage: node executor.js
<encrypted_code_file> <map_id> <code_key> <map_key>\n');
        process.exit(1);
    }
    const [encryptedCodePath, mapId, codeKey, mapKey] = args;
    try {
        console.log('--- [SECURE LOADER] 1. Reading Encrypted
Payload ---');
        const encryptedCode = await fs.readFile(encryptedCodePath,
'utf8');

        console.log('--- [SECURE LOADER] 2. Decrypting Map & Code in
RAM ---');

        const map = await getMapFromFileEncrypted(mapId, mapKey);
        const obfuscatedCode = decryptGCM(encryptedCode, codeKey);

        console.log('--- [SECURE LOADER] 3. Deobfuscating in RAM ---
');
        let cleanCode = deobfuscateCode(obfuscatedCode, map);
        cleanCode = removeNoiseFunctions(cleanCode);
        console.log('--- [SECURE LOADER] 4. Preparing Virtual
Environment ---');
        const currentDir = __dirname.replace(/\\/g, '/');

        const pathShim = `
            Object.defineProperty(global, '__dirname', { value:
'${currentDir}' });
            Object.defineProperty(global, '__filename', { value:
'${currentDir}/virtual_server.js' });
            module.paths.push('${currentDir}/node_modules');
        `;
        const finalPayload = pathShim + "\n" + cleanCode;
        console.log('--- [SECURE LOADER] 5. Executing via PIPE (No
Disk Write) ---');
        console.log('------------------------------------------------
--------------');
        const child = spawn('node', ['-'], {
            stdio: ['pipe', 'inherit', 'inherit'],
            cwd: __dirname
        });
        child.stdin.write(finalPayload);
        child.stdin.end();
        child.on('close', (code) => {
            console.log(`\n--- [SECURE LOADER] Server stopped (Exit
code: ${code}) ---`);
        });
    } catch (error) {
        console.error('\n[CRITICAL SECURITY ERROR]:',
error.message);
        process.exit(1);
    }
```

```
}

runEncryptedCode();
```