Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних систем та мереж
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: Обфускаційні методи захисту програмного коду в комп'ютерних системах

Виконав(ла): студент(ка) VI курсу, групи Сім-61
спеціальності 123 «Комп'ютерна інженерія»
(шифр і назва спеціальності)

(підпис) Карплюк В. І. (прізвище та ініціали)

Керівник (підпис) Ясній О. П. (прізвище та ініціали)

Нормоконтроль (підпис) Луцик Н. С. (прізвище та ініціали)

Завідувач кафедри (підпис) Осухівська Г. М. (прізвище та ініціали)

Рецензент (підпис) Скоренький Ю. Л. (прізвище та ініціали)

Тернопіль 2020

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра Кафедра комп'ютерних систем та мереж
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Осухівська Г.М.
(підпис) (прізвище та ініціали)
« » 20___ р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Магістр
(назва освітнього ступеня)

за спеціальністю 123 «Комп'ютерна інженерія»
(шифр і назва спеціальності)

студенту Карплюку Володимиру Ігоровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Обфускаційні методи захисту програмного коду в комп'ютерних системах

Керівник роботи Ясній Олег Петрович, докт. техн. наук, проф.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «*28*» *вересня* 2020 року № *4/7 -687*

2. Термін подання студентом завершеної роботи

3. Вихідні дані до роботи Документація та модуль програми Esprima у вигляді окремого класу для парсингу синтаксису JavaScript коду

4. Зміст роботи (перелік питань, які потрібно розробити)
Аналіз уже існуючих технологій обфускації програмного коду. Захист на апаратному та програмному рівнях. Оцінка якості обфускації. Дослідження методів обфускації на різних рівнях. Розробка алгоритмів обфускатора. Реалізація власного програмного продукту для обфускації вихідного коду на мові програмування JavaScript.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Тема та актуальність. 2. Мета та задачі. 3. Таблиця характеристик заплутуючих перетворень. 4. UML діаграма прецедентів та діаграма потоків даних роботи обфускатора з файлом. 5. Блок-схема алгоритму роботи програми. 6. Результат обфускації та виконання. 7. Порівняння режимів обфускації. 8. Висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності	Стадник І. Я.		
Охорона праці	Осухівська Г. М.		

7. Дата видачі завдання 01.10.2020

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Затвердження теми кваліфікаційної роботи		
2	Аналіз літературних джерел		
3	Обґрунтування актуальності досліджень		
4	Оформлення розділу «Аналітичний огляд в області дослідження»		
5	Аналіз предмета дослідження і предметної області		
6	Огляд та дослідження існуючих алгоритмів обфускації		
7	Оформлення розділу «Теоретична частина»		
8	Розробка алгоритму обфускатора та програмна реалізація		
9	Оформлення розділу «Опис програмної реалізації»		
10	Оформлення розділу «Охорона праці та безпека в надзвичайних ситуаціях		
11	Захист кваліфікаційної роботи магістра		

Студент _____ Карплюк В. І.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Ясній О. П.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Обфускаційні методи захисту програмного коду в комп’ютерних системах // Кваліфікаційна робота магістра // Карплюк Володимир Ігорович // Тернопільський національний технічний університет, кафедра комп'ютерних систем та мереж, комп’ютерна інженерія, група СІм-61 // Тернопіль, 2020 // С. – 83, рис. – 27, бібліогр. – 32, табл. – 1.

Кваліфікаційну роботу магістра присвячено розробці обфускатора для мови програмування JavaScript. Знайдено баланс між рівнем обфускації та необхідною продуктивністю. Проведені дослідження допомогли підвищити рівень захищеності інформації та програмних продуктів власними методами обфускації. Здійснено аналіз готових рішень.

У роботі проведено оцінку вже існуючих методів та засобів обфускації, знайдено оптимальні комбінації заплутуючих перетворень. Доведено ефективність використання розробленого обфускатора. Застосування розроблених обфускаційних технологій дасть змогу покращити такі показники розробленого ПЗ як: стійкість, надійність, безпечність.

Ключові слова: обфускація, ПЗ, заплутування, JavaScript, надлишковість, методи обфускації.

ANNOTATION

Obfuscation methods of program code protection in computer systems // Master's qualification work // Karplyuk Volodymyr Ihorovych // Ternopil National Technical University, Department of Computer Systems and Networks, Computer Engineering, SIM-61 Group // Ternopil, 2020 / / P. - 83, fig. - 27, bibliogr. - 32, table. - 1.

The master's thesis is devoted to the development of an obfuscator for the JavaScript programming language. A balance was found between the level of obfuscation and the required productivity. The research helped to increase the level of security of information and software products by their own methods of obfuscation. The analysis of ready decisions is carried out.

The evaluation of already existing methods and means of obfuscation is carried out in the work, the optimal combinations of confusing transformations are found. The efficiency of using the developed obfuscator is proved. The application of the developed obfuscation technologies will allow to improve such indicators of the developed software as: stability, reliability, safety.

Keywords: obfuscation, software, confusion, JavaScript, redundancy, obfuscation methods.

ЗМІСТ

ВСТУП

Сучасний світ важко уявити без уже звичних кожному різноманітних Інтернет-технологій та мультимедіа. Вони стали невід'ємною частиною життя не лише окремо взятих людей, але й суспільства в цілому.

В останнє десятиліття все більша частина усіх бізнес-проектів містить такі програмні продукти як хмарні обчислення, тисячі нових веб-сервісів, що пропонують свої розробки за допомогою мережі Інтернет (Google Docs, Google Compute, Dropbox, Telegram та інші). Захист будь-яких розробок, що лежать в основі схожих сервісів, стає одним із основних питань розробників.

Абсолютна більшість сучасних веб-проєктів користується мовою програмування JavaScript або іншими мовами програмування, котрі базуються на ній (TypeScript, NodeJS та інші). Усі кінцеві пристрої – настільні комп'ютери, планшети, ігрові консолі або ж навіть смартфони – містять інтерпретатори мови JavaScript, котра, завдяки тому, стала найпоширенішою мовою програмування в історії. Таким чином, JavaScript – основа більшості веб-сервісів як у frontend, так і в backend.

На ранніх етапах розробки JavaScript головно забезпечував інтерактивність простих веб-сторінок, що додавало нових можливостей, а також значно поліпшувало взаємодію з кінцевим користувачем. Сьогодні, користуючись такою мовою програмування, можна створювати все, що заманеться (від frontend- та backend розробки сайту до кросплатформних мобільних додатків, та навіть програмувати мікроконтролери).

Однак такий стрімкий розвиток спричинив збільшення різного роду отримання і несанкціонованого використання вихідного коду, унікальних алгоритмів та виконуваних модулів програмних продуктів.

Задача розробки найбільш ефективного та недорогого методу захисту вихідного коду програмних продуктів досить актуальна на сьогодні. Велика кількість розробників зацікавлена у захисті своєї інтелектуальної власності та своїх продуктів від несанкціонованих змін та реверс-інженерії.

Існує чимало ідей та методів захисту програмних продуктів (використання машинного коду, застосування різного роду шифрувань, виконання основного коду на своїх серверах), проте саме обфускація є мабуть найпростішим, найдешевшим та актуальним розв'язанням задач безпеки та захисту.

Неофіційна мета обфускації полягає у тому, щоб перетворити вихідний код програмного продукту у код, максимально незрозумілий для аналізу людиною та реверс-інженерії без зміни логіки та функціональності. Хоча у 2001 році доведено, що процес автоматичної обфускації неможливий, нею активно користуються у більшості проєктів. Витрати розробників на захист своїх продуктів за допомогою обфускації беззаперечно окупаються і компенсують потенційні збитки, до котрих призводить нелегальне копіювання та цілеспрямовані зміни в логіці продуктів.

З іншого боку, обфускацією можуть скористатися автори вірусів та інших небезпечних модулів коду для маскування власної мети. Унікальні алгоритми обфускації та прийоми перетворення коду значно сповільнюють виявлення потенційно небезпечного коду антивірусними програмами, тому значної уваги потребує і процес деобфускації.

Об'єкт дослідження – розробка програмного продукту, написаного для мови програмування JavaScript.

Актуальне завдання дослідження – пошук балансу між рівнем обфускації та необхідною продуктивністю у тому чи іншому проекті. При правильній реалізації процесів обфускації отримують безпечний, продуктивний та захищений від аналізу зі сторони зловмисників програмний код.

Мета дослідження: підвищити рівень захищеності інформації та програмних продуктів на мові JavaScript власними методами обфускації.

Для досягнення поставленої мети, потрібно розв'язати наступні задачі та завдання:

1. Здійснити аналіз готових рішень (обфускаторів, моделей, алгоритмів) та виявити їх переваги та недоліки;

2. Оцінивши вже існуючі методи, перейти до розробки власних методів обфускації програмних продуктів на мові програмування JavaScript, користуючись необхідною комбінацією заплутуючих перетворень.

3. Практично перевірити розроблений алгоритм заплутування, здійснивши експериментальні дослідження, що дозволить оцінити здійснену роботу.

Наукова новизна – розробка нового алгоритму заплутування програмного коду на мові програмування JavaScript. Розроблена методика вважатиметься оптимальною, оскільки її реалізовано на основі різноманітних методів обфускації у різних пропорціях, а також з модифікованою логікою. Саме тому запропонований алгоритм стійкий до методів автоматичної обфускації.

Технічна значимість такої розробки полягає в тому, що вона дозовлить досягнути вищого рівня захищеності інформації та вихідних кодів на базі обфускації, збільшить структурованість методів обфускації та аналізу.

Практична значимість роботи полягає в фактично миттєвому підвищенні рівня захисту інформації та програмних продуктів на базі розробленого обфускатора. Створені методики дозволять значно мінімізувати зусилля та заощадити фінанси розробників, спрямовані на захист проектів, замінити неефективні методи заплутування на нові та підняти на новий більш складний рівень мінімальні зусилля зловмисника для аналізу та деобфускації програмних продуктів. Тобто, у геометричній прогресії знизяться затрати часу та людських ресурсів на захист необхідних ділянок коду чи інформації. Отриманими результатами можна скористатися практично всюди, починаючи від простих веб-проектів та сервісів, закінчуючи великими підприємствами та організаціями, де наявна проблема захисту програмних продуктів від нелегального поширення та різного роду модифікацій.

Структура роботи: робота складається з пояснювальної записки та графічної частини. Пояснювальна записка складається із вступу, чотирьох розділів, висновків, списку використаних джерел та додатку. Обсяг роботи: пояснювальна записка – 81 аркуш формату А4, графічна – аркушів А1.

Публікації: результати дослідження апробовано на IX Міжнародній науково-технічній конференції молодих учених та студентів «Актуальні задачі сучасних технологій» м. Тернопіль 25-26 листопада 2020 року та VIII науково-технічна конференція «Інформаційні моделі, системи та технології» м. Тернопіль. 9-10 грудня 2020 року.

РОЗДІЛ 1
АНАЛІТИЧНИЙ ОГЛЯД В ОБЛАСТІ ДОСЛІДЖЕНЬ

Перші згадки про обфускацію зафіксовано у 1976 році в роботах Діффі та Геллмана. У вигляді, схожому на сучасний, обфускацію представлено у 1997 році в роботі Лоу, Томборсона та Коллберга. Формальне поняття «обфускація» з'явилося у 2001 році. Воно трактувалось наступним чином: результуюча програма, що отримана на виході обфускатора, повинна містити не більше інформації, ніж просто «чорна скринька» (буде деталізовано далі у роботі), котра імітує вхідну/вихідну поведінку початкової програми. Іншими словами, не повинно бути жодної різниці у логіці вихідної та заплутаної програм, або ж веб-сервісом, де перша виступає в якості алгоритму. Такий принцип обфускації отримав назву «обфускація чорної скриньки» (від англ. «black box obfuscation»).

У 2013 році з'явилось ще одне уявлення про обфускатор та запропоновано конструкцію для нього. Новоутворений вид обфускатора стали називати «обфускація нерозрізненості» (від англ. «indistinguishability obfuscation»). Ідея методу наступна: припустимо, якщо існує два різних програмних продукти, але з ідентичною функціональністю, то їх обфускації не стануть відрізнятись одна від іншої. Тобто, якщо наші програми `P1` та `P2` такі, що для будь-якого входу `x`, `P1(x) = P2(x)`, а `Z` – слугуватиме офускатором нерозрізненості, який на вхід отримує вихідну програму `P1` та у результаті перетворень генерує новоутворену програму `Z(P)`, то відрізнити `Z(P1)` та `Z(P2)` буде неможливо. Так само і неможливо буде визначити `Z(P)` це результат заплутування `Z(P1)` чи `Z(P2)`.

Рішення на базі сервіс-орієнтованої архітектури дозволяють скористатися перевагами широко поширених сервісів і значно спростити затрати на старт проєкту, а також налагодити взаємодію бізнес-процесів різних організацій. Програмні продукти в межах одного веб-проекту можуть

охоплювати декілька обчислювальних пристроїв, операційних систем та підприємств чи організацій, саме тому їх доволі складно контролювати, керувати ними та забезпечити високу продуктивність усіх процесів.

Потреба в захисті програмного забезпечення виникає в багатьох областях. Безпека даних охоплює конфіденційність і цілісність даних (під час передачі та зберігання). Однак не завжди кінцевим користувачам можна довіряти. Для кращого розуміння області досліджень необхідно детальніше розглянути зв'язок між керуванням веб-проектами, веб-сервісами та розподіленими обчислювальними системами, а також основними принципами роботи з користувачами.

На сьогоднішній день веб-сервіси – найпопулярніша парадигма хмарних обчислень та послуг. Спираючись на сервіс-орієнтовані архітектури на базі веб-сервісів, організації та підприємства можуть доволі швидко та гнучко розв'язувати задачі інтеграції як на внутрішньому, так і на міжкорпоративному рівнях. Очевидно, що коректність роботи та захищеність таких сторонніх сервісів та послуг повинні бути на високому рівні, оскільки вони відіграють важливу роль у житті їх кінцевих користувачів.

Принципи, які лежать в основі веб-проєктів та веб-сервісів доволі прості:

- розробник, відповідальний за сервіс, встановлює формат запитів та відповідей;
- користувачеві надають певні права та можливості;
- налагоджують чітку та обмежену взаємодію між діями користувача та відповіддю сервісу;
- певний рівень безпеки та конфіденційності інформації підтримують, розгортаючи найбільш вразливі та важливі частини коду на власних серверах, котрі недоступні звичайному користувачеві.

На жаль, у сучасному світі для гарантування безпеки та захисту від несанкціонованих змін, вище перерахованих принципів уже недостатньо. Програмні продукти потребують значно ефективнішого захисту зі сторони

розробників. Навіть з ідеально відлагодженою логікою функціональність розроблених послуг не завжди може бути гарантована чи обмежена при запуску програмних продуктів у незахищених середовищах.

Для захисту своїх розробок програмісти часто користуються власними алгоритмами шифрування та надають права доступу до окремих функцій чітко визначеним користувачам. В ідеалі це забезпечує конфіденційність та захист інформації як користувача, так і сервісу надання послуг. Однак можливі ситуації, коли команда розробників пише складний код на мові JavaScript, при чому його потрібно здати у короткі терміни та додати до основного проєкту. Припустимо, що функціональність «свіжого» коду унікальна та немає аналогів на ринку, але на розробку різноманітних методів захисту від копіювання та небажаних змін зі сторони користувачів часу не лишилось і виникає питання: «Як в короткі терміни захистити алгоритм?».

1.1 Основні методи захисту програмного коду веб-проєктів на мові програмування JavaScript

Очевидно, що у веб-проектах JavaScript виконується безпосередньо на стороні браузера клієнта. Увесь алгоритм, що передається від сервера до кінцевого користувача, наявний у вихідному вигляді у останнього. Маючи перелік усіх функцій та можливих методів взаємодії з веб-проєктом значно спрощується аналіз потенційних вразливостей та зловживань ними.

Припустимо, що розробники це передбачили і застосували методи шифрування для вдосконалення безпеки алгоритму, однак сучасні хакери справляються і з такими завданнями, знаючи основні принципи браузера та кодування. Тому можна спробувати максимально ускладнити роботу хакерові, значно заплутавши вихідний код, котрий поступає у браузер клієнтові, що у свою чергу значно перешкоджатиме його аналізу та змінам.

Тобто, веб–розробники мають декілька варіантів захисту коду:

- користуватися власними унікальними крипторами;

- користуватися веб-сокетами із криптованими повідомленнями;
- здійснити обфускацію.

Власні криптори перетворять дані або ділянки коду в нечитабельний вигляд. Популярним прикладом слугує формат base64 (позиційна система числення з основою 64). Перетворивши необхідний код у транспортний формат (base64), до результату необхідно додати ключ із набору символів, без якого дешифрування буде неможливим. Здійснивши дешифрування, увесь отриманий текст можна зробити виконуваним за допомогою функції `eval()`. Основна проблема крипторів полягає в тому, що знаючи алгоритми їх роботи, та маючи ключ дешифрування, зловмисник відразу отримає весь шифр у вихідному вигляді. Допомогти розв'язати таку задачу можуть обфускатори.

Заплутувачі змінюють вихідний код програми за допомогою різноманітних змін: від вставлення незрозумілих символів та перейменування змінних до кардинальних змін у графі керування та вставок псевдокоду, поєднаного із заплутуванням готових функцій. Зрозуміло, що після таких змін обсяги коду значно зростуть.

Застосування власних обфускаторів значно ускладнює подальше відлагодження коду не тільки зі сторони зловмисників, а й зі сторони розробників. Тому то, до певної міри, незворотній процес, оскільки обфускація може зачепити граф керування, що однозначно вплине на логіку функціонування вихідних функцій, проте результат роботи буде такий же, як і у незаплутаного коду.

Обфускацію можна застосовувати як метод, що дозволить у майбутньому довести авторство, накладавши певні приховані водяні знаки або інші особливості при перетворенні, притаманні тільки автору . Такі перетворення безсумнівно дозволять пройти експертизу коду та підтвердити авторство. Проте, щоб таке відбулось, автор повинен помістити відповідні методи у власний обфускатор. Такий принцип захисту програмного продукту дозволить розробнику відстежувати нелегальне розповсюдження його

алгоритму. Виявивши наведений вище факт зловживання правовласник може захищати свої права у встановленому законом порядку.

Загрозу становить не лише нелегальне поширення програмних продуктів, а й нелегальні модифікації зі сторони хакерів. Відбувається це наступним чином: зловмисник отримує копію програмного продукту (або ж вихідного коду), здійснює аналіз вихідного коду, вносить в нього модифікації, після чого запускає (якщо це JavaScript) або ж перекомпільовує (у випадку мов програмування з байт-кодом) новоутворений код та отримує будь-які переваги у порівнянні з іншими користувачами. Наприклад, такі модифікації можуть привести до обходу обов'язкових механізмів перевірок (ліцензійний ключ, прив'язка до комп'ютеру, відкриття платного функціоналу та інших), внесення змін до баз даних сервісу (доступ до даних інших користувачів, зміна балансу на рахунку, здійснення інших операцій недоступних користувачам). Тобто такі дії дозволять не лише отримати певні привілеї, а й здатні нанести непоправну шкоду цілому проєктові. Отже, захист вихідного коду програмних продуктів – пріоритет у сьогоднішніх реаліях.

Заплутування як метод захисту перетворює роботу з так званою «білою скринькою» на роботу з «чорною скринькою». Під «білою скринькою» мають на увазі роботу із частиною коду або проєктом в цілому, робота і функціонал яких повністю детерміновані. Під «чорною скринькою» – робота лише з доступним функціоналом та функціями, принцип роботи та алгоритм яких взагалі невідомий, а відома лише реакція методів на вхідні дані у певні періоди часу. Важко не погодитись, що розробник надасть перевагу другому варіанту, передаючи код кінцевому клієнту.

Обфускація – один із методів безпечного кодування, рекомендованих OWASP (онлайн спільнотою, яка працює в галузі безпеки веб-застосунків), проте ще не таким популярним серед більшості розробників. Причиною цьому слугує те, що при надмірній обфускації вихідного коду страждає продуктивність алгоритму.

Однак, якщо казати про безпеку онлайн-банкінгу чи маркету криптовалют, найважливішим стане безпека даних та користувача, додаткові витрати часу на очікування необхідної дії перейдуть на другий план.

Актуальне завдання дослідження – пошук балансу між рівнем обфускації та необхідною продуктивністю у тому чи іншому проєкті. Правильно реалізувавши процеси обфускації, отримують безпечний, продуктивний та захищений від аналізу зі сторони зловмисників програмний код.

Однак наполегливий зловмисник може витратити багато часу на аналіз та перевірку заплутаного коду, виявити необхідну функціональність, відкинути псевдокод, внести зміни і досягти бажаного результату в злочинній меті. Саме з цієї причини розробники поєднують обфускацію з іншими підходами, що мають на меті оновлення методів захисту, оновлення версій коду, зміну моделей шифрування, переїзд баз даних і так далі. Усе це сильно обмежить хакера у часі, що значно вдосконалює механізми захисту.

1.2 Вклад вчених в обфускаційні методи захисту коду

На сьогоднішній день відомо багато програм для обфускаційних перетворень, таких як: ThinApp, .NET Reactor, CilSecure, C# Source Code Obfuscator та ін. Більшість виконує лише такі перетворення як зміна порядку виклику даних/функцій, генерація нових імен змінних, вставка псевдокоду, проте на сьогоднішній день цього вже недостатньо для забезпечення надійного захисту коду обфускаційними методами. Ефективність обфускації можуть забезпечити вище перераховані механізми, але уже в поєднанні з перетвореннями основних елементів коду, користуючись методами клонування, додаючи мертвий код, змінюючи тотожності та модифікуючи граф управління (детальний опис кожного з них наведено далі у роботі).

Провівши аналіз існуючих робіт, можна зазначити, що обфускаційні методи можна розділити на декілька видів (категорій). У роботах Крістіана

Коллберга, Кларка Томборсона та Дугласа Лоу методи заплутування коду поділено за чотирма критеріями: обфускація даних, трасування, керування та запобіжні перетворення. Проте автори не врахували використання міток у коді програми, реструктурування даних масивів, габаритне оголошення змінних. Використання перерахованих методів заплутування ускладнить нечитабельність, аналіз алгоритму та процес деобфускації в цілому [12].

Українські вчені також не залишили питання обфускації без уваги. Сучасні обфускаційні методи захисту програмного коду проаналізували та досліджували: Ірина Степаненко, Василь Кінзерявий, Іван Лозінський. Метою їх дослідження стали: аналіз та розробка узагальненої класифікації обфускаційних методів захисту програмного коду (рис. 1.1), що дозволить у майбутньому розробляти надійні алгоритми захисту програмного коду від деобфускації.

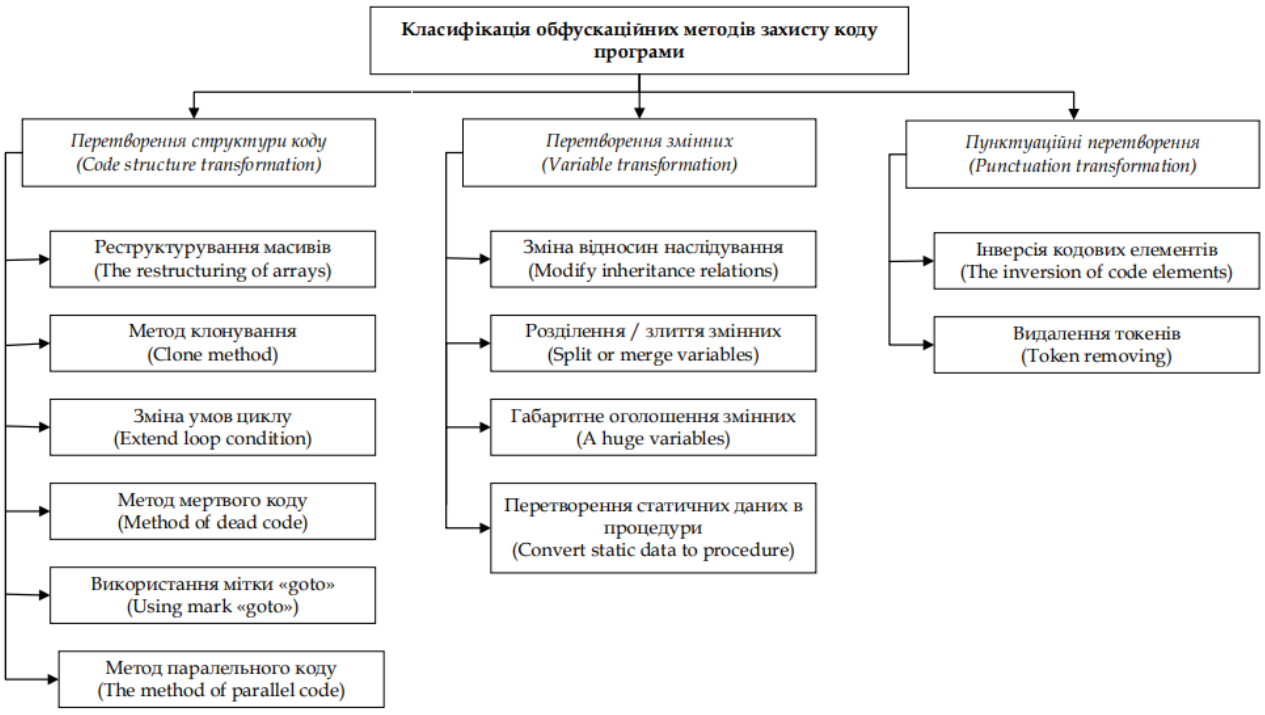


Рис. 1.1. Запропонована класифікація обфускаційних методів захисту коду програми українськими вченими [13]

Вчені Арізонського університету запропонували класифікувати методи на основі свого підходу до таких елементів як: вхідний-вихідний потоки даних, трасування спрощення, управління потоками структури коду. Дана класифікація спрямована на перетворення структури коду, перенаправлення потоку даних, порушення логічного порядку виконання операцій.

Корейські вчені Ілсун Ю та Кенгбін Юім створили класифікацію методів заплутування коду, котра базується на тих методах, котрими користуються у процесі розробки шкідливого та потенційно небезпечного програмного забезпечення. До них відносять: метод мертвого коду, реєстрація перерозподілу, реорганізація підпрограм, заміна інструкцій, метод перестановки коду, метод об'єднання. Однак не враховано пунктуаційні перетворення програмного коду, метод клонування, зміни умов циклу та використання міток у програмі [21].

1.3 Евристичний метод аналізу обфускованого шкідливого коду

Більше трьох десятиліть назад стало очевидним, що програмне забезпечення потребує захисту від шкідливого коду. У 1988, тестуючи нешкідливий вірус-хробак Морріса, його було випадково випущено в Інтернет, що спричинило відмову від обслуговування тисяч машин Unix. Саме тоді програмісти раптово усвідомили проблему захисту від шкідливого коду, який здатний самопоширюватися. Особливої уваги потребували критичні комп'ютерні системи, такі як мережеві сервери. З тих пір почали розробляти антивіруси та брандмауери.

Спочатку методи обфускації спрямовували на захист інтелектуальної власності розробників програмного забезпечення та забезпечення механізмів цілісності та захисту інформації, однак вони набули широкого застосування серед авторів шкідливих та потенційно небезпечних програм, щоб приховати їх алгоритми від антивірусного сканування.

Обфускувавши шкідливий програмний продукт, зловмисник отримує усі плюси від перетворення та заплутування коду. Очевидно, що для ефективного виявлення та усунення шкідливих програм важливо знати їх алгоритм, а тому й самі методи застосованого обфускування.

За допомогою евристичного аналізу можна знайти нові потенційні загрози, які не були виявлено раніше засобами на основі відомих сигнатур, приміром, новими шаблонами шкідливих програм або ж їх новими типами та модифікаціями. Евристичне сканування застосовує алгоритми та шаблони для аналізу програмного забезпечення на підозрілі дії. Спираючись на рівень незворотності обфускування, програма захисту може попередити користувача про виявлення такого факту та вказати на те, що програма підозріла та нерекомендована для запуску (наприклад ступінь замаскованості більше 50%) через надто приховану функціональність.

Почастішали випадки копіювання вихідного коду широко поширених програмних продуктів та створення їх копій , із додаванням шкідливого коду, який паралельно виконує потрібні зловмиснику дії. За рахунок автоматичної деобфускації отриманий код можна проаналізувати на наявність подібних модифікацій та визначити його шкідливість.

Частина антивірусних сканерів уже здатна запускати небезпечний код у віртуальному середовищі «пісочниця» (від англ. «sandbox»), де шкідливий код розпочинає свою діяльність, яку можна повністю відстежити. Розшифрувавши зловмисні дії можна значно прискорити деобфускацію та автоматично виявляти такі програмні продукти. Розуміючи це, зловмисники постійно вдосконалюють механізми обфускації та оновлюють свої продукти, щоб подовжити термін їх прихованого життя та зменшити ефективність механізмів виявлення.

Тобто, антивірусні продукти на основі традиційних методів обфускації стають дуже вразливими для новітніх розробок заплутуючих методів зі сторони зловмисників. Тому значної уваги потребує новий підхід до виявлення шкідливих програмних засобів. Евристичні методи –

перспективний підхід до виявлення заплутаних шкідливих програм. Однак такі механізми потребують повністю ізольованих середовищ для виявлення на кшталт «пісочниця».

1.4 Захист на апаратному та програмному рівнях

До основних переваг захисту на програмному рівні можна віднести: гнучкість, низьку вартість, сумісність з уже існуючими системами. Проте великим недоліком виступає обмежена міцність. Із розвитком нових методів обходу безпеки додатків, спеціалістам доводиться вдосконалювати та винаходити нові методи підтримки належного безпеки своїх додатків. Враховуючи стрімкий ріст обчислювальної потужності процесорів, а також збільшення об'ємів пам'яті, методи захисту та атаки дуже стрімко розвиваються.

Локальне програмне забезпечення та конфіденційні дані можуть бути захищені за допомогою аутентифікації з шифруванням. При кожному запуску програмного забезпечування воно розшифровуватиме необхідні для роботи блоки на льоту. Проте така методика може гарантувати абсолютну безпеку лише у випадку, коли шифрування та дешифрування виконується на спеціально призначеному криптографічному співпроцесорі або подібному обладнанні. У такому випадку зловмисник не зможе отримати алгоритм шифрування та витягнути ключ дешифрування (система працюватиме на основі «чорної скриньки» і доступними будуть лише ввід/вивід). Очевидно, що у більшості клієнтів такі апаратні модулі відсутні, а тому ефективність даної методики стрімко знижується, тому що зловмисник зможе перехопити вихідний код модуля шифрування, як тільки той поступить у чистому вигляді в ОЗП або процесор. З схожими принципами та проблемами стикається спосіб аутентифікації користувача.

Можливим розв'язком таких задач може стати окрема апаратна смарт-картка або апаратний маркер із обов'язковим ключем для роботи захищеного

програмного продукту, котрі виконуватимуть процеси шифрування та аутентифікації користувача на основі «чорної скриньки». Однак і такі засоби не набули популярності, оскільки такі апаратні додатки часто губляться та зазнають фізичних пошкоджень, а для їх заміни виробнику необхідно створювати та передавати новий, або повністю видавати нову копію програмного продукту з необхідними апаратними додатками. Зловживання зі сторони користувачів неминучі.

Тобто, недоліки апаратних методів захисту – підвищення вартості, несумісність з деякими уже існуючими системами, складнощі при передачі таких програмних продуктів, модернізації, технічному обслуговуванні та видачі. Отже, апаратні засоби мають значно меншу гнучкість, ніж програмні. Якщо механізм захисту зламали або пошкодили (втрата апаратного додатку), апаратне забезпечення обов’язково повинно бути оновлено. Здійснення подібних дій є дуже затратним. Тому програмні механізми захисту за допомогою обфускації повинні бути максимально апаратно та платформо незалежними для гарантування гнучкості.

1.5 Висновки до розділу

В процесі огляду літературних джерел проаналізовано доцільність обфускації на сьогоднішній день, зокрема у сфері веб-програмування. Відзначено її позитивні та негативні наслідки. Розглянуто та оцінено розробки вчених, наведено можливі методи їх поліпшення.

Запропоновано виявляти обфускований шкідливий код евристичними методами аналізу. З’ясовано переваги захисту програмних продуктів на програмному рівні за допомогою обфускації над апаратними методиками захисту.

РОЗДІЛ 2
ТЕОРЕТИЧНА ЧАСТИНА

Найбільший вклад у розвиток обфускації здійснено у роботах Колберга С. [1], Варнавського Н.П. [2; 21], Мадоу М. [4; 5], Делла Педра М. [6; 7], Курмангалєєва Ш. Ф. [8]. Методики обфускації, запропоновані ними, можна приймати як основні.

Обфускація дає можливість не тільки розв'язати задачу незаконної модифікації вихідного коду, а й інші, такі як накладення водяних знаків (від англ. «watermark»), прив'язка до домену веб-проекту і створення програмних відбитків пальців (від англ. «fingerprinting»).

Такі технології дозволяють не лише захистити вихідний код від можливих модифікацій, а й поліпшити методи захисту авторських прав та створити докази нелегального користування. Для кращого розуміння наведемо приклад обфускації з прихованим водяним знаком. Очевидно, що логіка заплутувача, якою скористався автор уже є унікальною та немає аналогів на ринку, а присутність власного водяного знаку стане незаперечним фактом підтвердження авторства та нелегального використання чужого програмного продукту. Цей же метод може використовуватись і для виявлення порушників (злочинців, що модифікують код програмного продукту). Розробник може встановити певні умови, в залежності від яких водяні марки у кожного користувача будуть унікальними та відомими першому. Тобто програма матиме ідентичний функціонал, однак можливо розрізнити, хто саме є порушником режиму захисту інформації або ж модифікацію коду, автор завжди зможе визначити, хто саме є потенційним зловмисником. Кожен з алгоритмів, спрямованих на захист від незаконної модифікації, може оцінюватись окремо, різноманітними способами, а також без прив'язки його до конкретної мети (тому, що їх може бути декілька).

Для оцінки ефективності застосування методів обфускації, до конкретного вихідного коду, методи оцінки прийнято поділяти на дві групи:

емпіричні та аналітичні. Як емпіричні, так і аналітичні методи оцінки можуть бути чисельними та експертними (на основі технічної експертизи).

2.1 Оцінка за трьома критеріями

Більшість аналітичних експертних методів ґрунтуються на трьох величинах, що характеризують, наскільки ефективний той чи інший процес обфускації: стійкість, еластичність, вартість перетворення. Передбачається, що людина може деобфускувати будь-який обфускований код, тому оцінка еластичності проводиться на автоматизованих деобфускаторах. Даний підхід добре розглянуто в роботах Нікольської К. Ю. і Хлестова А. Д. [27].

1. Стійкість. Вказує на ступінь складності проведення реверсної інженерії над заплутаним кодом. Складність деобфускації для оцінки даного критерію можна визначити різними способами. Ручна деобфускація – експертна оцінка отриманого коду. У випадку автоматичної деобфускації може оцінюватися складність розроблених алгоритмів деобфускації.

2. Еластичність. Вказує на кількісні взаємозв’язки одного чинника (застосування конкретного методу обфускації) з іншим (кількість програмних продуктів, здатних виконати подібну зворотну операцію). В якості оцінки даної властивості часом вдаються до перерахування способів і/або програмних продуктів, що дозволяють здійснити зворотне перетворення. Дана величина характеризує кількість готових зворотних автоматизованих програмних продуктів і відомих методів деобфускації, котрі дозволять виконати повну деобфускацію програмного продукту. В рамках стійкості розглядають виключно кількісну оцінку варіативності зворотних перетворень. Даною оцінкою можна скористатися для захисту програмних продуктів методами, які слабо автоматизовано або вивчено.

3. Вартість перетворення. Дозволяє оцінити, наскільки більше системних ресурсів потрібно на виконання заплутаного коду, ніж для

виконання вихідного коду програми. Необхідні ресурси з оцінюють на основі складності алгоритмів обфускації.

Тобто, такі критерії допоможуть визначити експертну оцінку конкретного методу обфускації на певній ділянці вихідного коду. Такий підхід дозволяє поліпшити методи аналізу захищеності програмного продукту на основі обфускації. Дослідження у даному напрямі спрямовані на систематизацію і подальше вдосконалення. Скориставшись такою методикою, розробники, які мають справу з аналізом, матимуть змогу з легкістю систематизувати отримані результати, мінімізувати часо- та трудовитрати на створення власних критеріїв оцінки. Метод оцінки за трьома параметрами слугує незамінним помічником, коли інші методи незастосовні. Прикладом таких ситуацій можуть слугувати заплутування найбільш вразливих та значущих ділянок вихідного коду великого програмного продукту. Також дана методика допоможе при аналізі ефективності нестандартних алгоритмів заплутування або незвичних предметних областей. Отже, метод оцінки на основі трьох параметрів – найефективніший за експертної оцінки програмного продукту, а також спрямований на мінімізацію часо- та трудовитрат на заходи з оцінки його захищеності.

2.2 Оцінка методів на основі циклічної складності та розгалужень потоку управління

Розглянемо критерії ефективності методів обфускації, заснованих на метриках складності програмних продуктів. Метрикою називають таке відображення, яке ставить у відповідність кожному програмному продуктові певне число. Застосовують такі метрики:

1. Метрика LC розміру процедури. Найпростіша метрика складності коду. Що більший обсяг інструкцій процедури, то вища оцінка її складності.

2. Метрика YC складності циклічної структури. ЇЇ визначають як потужність транзитивного замикання відношення досяжності у графі потоку керування конкретної процедури.

3. Метрика DC складності потоку даних конкретної процедури. Оцінка складності залежностей даних у процедурі. Значення обчислюють за кількістю дуг у графі, що призводять до процедури.

4. Метрика MC ускладнення програми внаслідок заплутування. Розраховують за наступним принципом: методи оцінюють на основі їх циклічної складності, розгалуження потоку управління за допомогою критеріїв ефективності методів обфускації, що в свою чергу базуються на метриках складності графу потоку управління програм.

$$MC = \frac{YC(\text{після заплутування})}{YC(\text{до заплутування})} + \frac{DC(\text{після заплутування})}{DC(\text{до заплутування})} \qquad (1)$$

Найліпшими властивостями володіє перетворення підвищення опосередкованості, введення диспетчера, вставка непрозорих предикатів. Стійкість таких перетворень до аналізу зростає при застосуванні одразу декількох методів заплутування. Значно ускладнити аналіз заплутаного коду можна, перемішуючи та зв’язуючи потоки даних, що утворюються внаслідок заплутувань, та потоків даних вихідної програми.

Іншими словами, що більша метрика ускладнення програмного продукту, то складніше його аналізувати та зрозуміти алгоритм у результаті навантаження інформаційних та керуючих зв’язків. Зведену таблицю метрики MC для різних методів обфускації наведено у табл. 2. 1).

В результаті дослідження визначено, що найліпшими заплутуючими параметрами володіють перетворення, що ґрунтуються на підвищенні опосередкованості, введенні диспетчера та вставці непрозорих предикатів.

Таблиця 2.1

Таблиця характеристик заплутуючих перетворень

Перетворення	МС
Непрозорі предикати	14,4
Посилення опосередкованості	13
Відкрита вставка процедур	2,6
Виділення процедур	2,7
Додання диспетчеру	6,1
Локалізація змінних	2,7
Розгортання циклів	3,1
Мертвий код	3,9
Недосяжний код	3,6
Внесення тотожностей	4,8
Дублювання коду	2,7
Зміна області дії змінних	2,3
Клонування базових блоків	4,9
Переплетення процедур	1,5

Отже, метрика на основі циклічної складності та розгалужень потоку керування найбільш ефективна для оцінки складних методів обфускації. Розробка розгалужень потоку керування – найперспективніша технологія, враховуючи сучасні засоби редагування та перетворення коду. Змінні можна легко замінити глобально в додатку, тоді як потік управління аналізують з урахуванням багатопотокових обчислень, багаторазових викликів та залежностей, які змінюються за настання певних умов. Застосовуючи таку метрику ,необхідно повністю аналізувати характеристики новоутвореного графа керування, тому ним не можна скористатися без спеціалізованих програмних засобів, а також необхідності наявності кваліфікованого спеціаліста в області захисту інформації. Якщо такою метрикою користуватися систематично, то наявність експерта, котрий дозволить отримати результати оцінки, обов’язкова лише за першої оцінки, а вся подальша робота зводитиметься до застосування уже готових метрик та даної методики.

2.3 Емпіричні методи оцінки обфускації

Емпіричні методи ґрунтуються на статистичних даних, одержаних у результаті дослідження. Для проведення одного такого дослідження обов'язкова вже група людей (близько знайомих з реверсною інженерією), частина вихідного коду програми, що підлягає захисту, та набір алгоритмів обфускації. Результат такого коду – мінімальний час, необхідний групі людей для вивчення усіх фрагментів коду, котрі перетворено одним з алгоритмів обфускації. Замість групи людей можна застосувати автоматично деобфускацію, однак у такому випадку знадобиться група експертів для аналізу результату. Така необхідність обґрунтовується тим, що аналізувати необхідно новий код програми, що пройшла деобфускацію, оскільки він відрізнятиметься від вихідного коду, котрий пройшов обфускацію. Технічна експертиза новоутвореної програми повинна встановити рівність двох програм та ідентичність алгоритмів.

Обфускація обмежена наступними трьома основними вимогами: збереження функціональності програми (її реакції на вхідні дані), стійкості, дозволяються незначні зміни розміру програми (зміна структури, додавання псевдокоду та інші). Щодо стійкості, то програму вважають стійкою, коли зловмисник, маючи тексти коду обфускованої програми, здійснивши будь-які маніпуляції та експерименти, може отримати лише ту інформацію, яку міг би отримати, виконавши можливі функції програми без доступу до вихідного коду.

Для точного математичного визначення поняття обфускації програм в моделі «чорного ящику» потрібно ввести ще декілька понять. У літературних джерелах можна знайти два чітких та загальноприйнятих визначення поняття «обчислювальна програма». У першому випадку, такі програми представляють у вигляді машини Тюрінга, а в другому другого – у вигляді логічних схем. Якщо складність обчислень та розмір програм оцінюють з точністю до поліноміальних перетворень, такі два поняття еквівалентні . Щоб

показати їх відмінності, обмежимося представленням програм у вигляді машини Тюрінга. У криптографії в якості суперника часто застосовують ймовірнісні алгоритми, котрі забезпечують обчислення, застосовуючи випадкові величини. Обчислення обмежені у часі та поліноміально залежать від розміру даних, котрі підлягають обробці. Тобто, модель суперник – машина Тюрінга але, уже з ймовірностями та обмеженнями у часі. У даній теорії також застосовують машини Тюрінга з оракулом (будь-які функції та предикати), які розглядають як допоміжні.

У результаті досліджень встановлено, що існує множина функцій, яка не піддаються обфускації у моделі «чорної скриньки». Тобто, задача обфускації програм не має простих розв'язків та потребує детальнішого дослідження.

2.4 Заплутування JavaScript коду на прикладі калькулятора

Припустимо, у нас є розроблений калькулятор вартості послуг із великою кількістю взаємопов'язаних параметрів. Розробка такого продукту є непростою, тому алгоритм роботи потрібно заплутати для захисту від копіювання та подальших модифікацій з метою використання на сторонніх веб-проєктах.

Як вже відомо, будь-який код можна розшифрувати, проте для цього необхідно немало часу, детальний аналіз та перебір усіх можливих варіантів заплутування на кожній стадії розшифрування, аж поки не знайдено правильний. Нелегкий процес розшифрування може відштовхнути зловмисників від цього процесу, оскільки абсолютна більшість з них після декількох невдалих спроб відкине ідею деобфускації та направить свої сили на пошук відкритих аналогів або ж на розробку власного продукту.

У результаті перетворень, описаних нижче, отримано вихідний код алгоритму роботи калькулятора у наступному вигляді (рис. 2.1), з логікою, ідентичною до вихідного варіанту.

Рис. 2.1. Приклад заплутаного JavaScript алгоритму калькулятора

Просунуті спеціалісти з реверс-інженерії одразу візуально відмітять схожість із кодуванням base64, однак спроби декодування будуть марними. Заплутати у такий вигляд можна все, що завгодно (скрипт, параметри, зображення), тому можна здійснити ще декілька несправжніх заплутувань схожого типу, аби ще більше ввести в оману зловмисника.

Тепер перейдемо безпосередньо до заплутувань алгоритму калькулятора. У коді можна створити функцію `glob()`, якій на вхід поступає зашифрована стрічка. Для прикладу `glob=function(s){sfd(rty(s.substring(-~[])))`. Всередині такої функції бачимо використання інших `sfd()` та `rty()`, які в свою чергу також заплутані: `sfd=this["\x65\x76\x61\x6C"]; rty=this["\x61\x74\x6F\x62"]`.

Такий синтаксис коду уже відіб’є бажання у багатьох. Розглянемо такі перетворення детальніше.

Спочатку в футері сайту потрібно вказати шлях до обфускованого скрипту: `<?$filebase64='TNTU_decryption'.base64_encode(file_get_contents ('/src/example.js'));?>`

Таким чином, вказано шлях до файлу `example.js` та ключ дешифрування `'TNTU_decryption'`, який може бути будь-якою варіацією латиниці та чисел. Додавання такого ключа сприяє захисту від дешифрування

через загальновідомі методи вставки коду в `alert()` або пропускання його через base64 декодер. З додатковими символами код не буде працювати. Після чого у `glob()` необхідно вказати отриманий скрипт. Аналогічним способом потрібно оголосити `sfd()` та `rty()`.

Іншими словами, основна функція glob() приймає на вхід параметр `s`, що передається в функцію `substring()` із закодованим параметром `-~[]` (у JS це спосіб подання одиниці), довжина якого дорівнює довжині ключа дешифрування. То лише найпростіший спосіб шифрування. Вдосконалити заплутування можна, розбивши кодування на більше функцій, частину значень сховати в файлах `cookie`, `sessionID`, домені веб-проєкту, або ж в обфускованих функціях у інших JavaScript файлах. Число можливих варіацій необмежене.

Отримавши результат, на наступному кроці відправляємо його у функцію `rty()`, яка при розкодовуванні має вигляд: `ƒ atob() { [native code] }`. Тобто, стрічка, закодована з використанням алгоритму base64,декодується,. Тепер відправляємо отримані значення у `sdf()` – `ƒ eval() { [native code] }`. Описана раніше функція eval() виконає інструкції скрипту, отримані з тексту, який вже заплутано. На такому простому прикладі показано лише запуск інструкцій заплутаного алгоритму, які теж потрібно деобфускувати та відловити можливі взаємозв'язки з іншими скриптами для налагодження вихідної логіки продукту.

Функції підлягають шифруванню наборами символів за допомогою тексту в форматі hex (hexadecimal), який підтримує усі відомі символи. Ускладнити заплутування можна багатьма способами у різних варіаціях, зрозуміти які складно та часозатратно.

2.5 Дослідження методів обфускації

Заплутування коду – практика, що перетворює код на нерозбірливий або, принаймні, важкий для розуміння. Процес обфускації коду включає в себе перетворення коду програми в такий код, який важко зрозуміти, змінюючи зовнішній вигляд коду, зберігаючи при цьому специфікацію «чорної скриньки» програми. Обфускацію, котра перетворює програму, можна розглядати як окремий спосіб кодувати дані. Подальший аналіз показує, що існує багато схожостей між обфускацією і криптографією, але все ж таки ці дві технології не еквівалентні [22].

Методика обфускції коду приховує дані, методи управління, синтаксис вихідного варіанту реалізації програмного забезпечення. У результаті отримують нову заплутану реалізацію того ж алгоритму.

На рис. 2.2 представлено таксономію обфускації. Перший рівень ієрархії розбиває існуючі методи обфускації на чотири рівні, в залежності від кінцевої мети:

1. Елементи коду. Заплутування окремих фрагментів коду (включає перетворення форматування, графу потоку управління, даних, заплутуючих методів та класів).
2. Програмні компоненти. Стосується всього програмного компоненту (такого, як бібліотека Java або форматів виконуваних файлів на кшталт ELF).
3. Міжкомпонентний рівень. Зосереджується на інтерфейсах (приміром, JNI) між різними компонентами проєкту.
4. Прикладний рівень. Унікальні методи заплутування, котрі застосовують для конкретних видів додатків (шифрування «білої скриньки» для систем DRM, шифрування нейронних мереж та ін.).

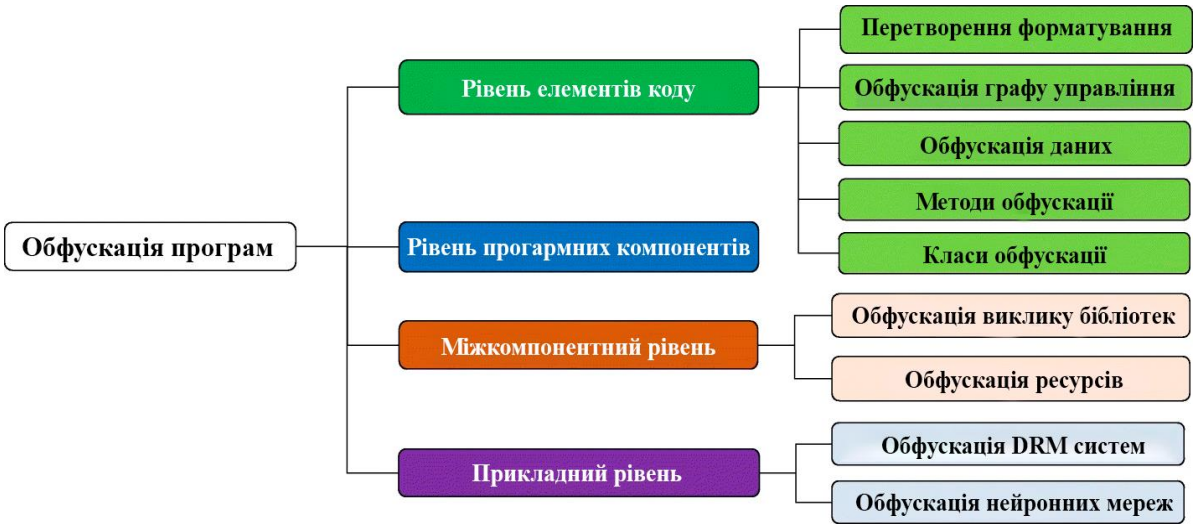


Рис. 2.2. Таксономія обфускації [6]

Головне призначення таксономії – удосконалити захист внаслідок ідентифікації різних програмних компонент та можливих заплутуючих перетворень. Наведемо простий приклад: якщо розробники зацікавлені у захисті лексичних інструкцій алгоритму, то вони можуть обрати лексичну обфускацію; коли виникла потреба захисту виклику певних функцій між JAVA або .NET та машинним кодом – необхідно обрати міжкомпонентний рівень. Таким чином, наведена вище таксономія унікальна для всіх мов та форм.

Важливо відмітити, що стратегії обфускації різних рівнів ортогональні одна до одної. Це може допомогти розробникам знайти необхідну стратегію, залежно від особливостей цільового програмного забезпечення. Очевидно, що вони можуть обрати одразу декілька методів заплутування, попередньо оцінивши їх ефективність у своєму проєкті стосовно вартості, продуктивності та стійкості (описано вище).

Запропонована таксономія відрізняється від розроблених раніше (наприклад, Коллберга 1997р, Шрітвізер 2016) [1; 8]. Вона орієнтована на програмні пакети, котрі складаються з гетерогенних компонент.

2.5.1 Рівень елементів коду. У цьому пункті розглянемо можливі методи обфускації для конкретних елементів коду (рис. 2.3). Цей рівень охоплює більшість публікацій в області обфускації програмних продуктів. В залежності від того, на які елементи націлено заплутування, дану класифікацію можна розділити на п’ять підкатегорій, розглянутих на попередньому рисунку.

Розглянемо класифікацію детальніше:

1. Перетворення форматування. Здійснює перетворення коду та інструкцій, не змінюючи синтаксис вихідного варіанту. Розрізняють чотири стратегії заплутування:

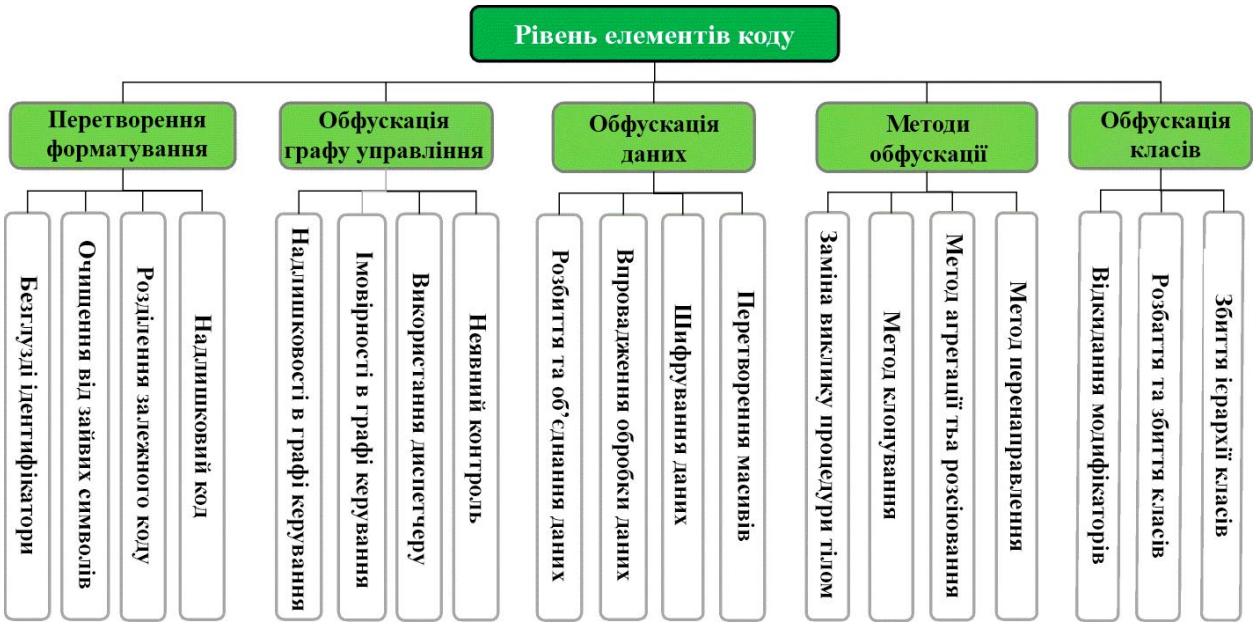


Рис. 2.3. Ієрархія рівня елементів коду [6]

- безглузді ідентифікатори – лексична обфускація, котра перетворює значущі ідентифікатори в безглузді, як правило, на односимвольні. У більшості мов програмування використання осмислених та однотипних правил іменування змінних та процедур потрібне в якості хорошої практики програмування. Деякі імена перейдуть і у випущений додаток (у JAVA усі імена зберігаються в байт-коді, а у мові C/C++ зберігаються в двійкових файлах). Очевидно, що значущі ідентифікатори значно спростять аналіз

реверсної інженерії, тому їх потрібно перетворити. Щоб максимізувати заплутування Чан і Ян ще в 2004 запропонували користуватися не лише односимвольними, а й одних і тими ж іменами для об'єктів різних типів та локально переоголошувати. Такий підхід прийнято за схему обфускації для JAVA програм за замовчуванням.

- Очищення від зайвих символів. Така стратегія видаляє надлишкову символьну інформацію з релізних версій програмного забезпечення, таку як символи відладки для більшості програм. Крім того існує й інша надлишкова символьна інформація для конкретних форматів програм (mapping або файли ELF, що містять табличні пари ідентифікаторів та адрес). Застосовуючи стандартні параметри компіляції вихідних кодів на мові C/C++, згенеровані двійкові файли міститимуть небажані таблиці символів. Їх можна очистити, скориставшись інструментом strip у Linux. Ще одним аналогічним прикладом може слугувати Android smali. За замовчуванням символьні коди smali містять надлишкову для кінцевого користувача інформацію, що розпочинається з .рядок та .source, наявність яких ніяк не впливає на працездатність програмного продукту, тому її можна видалити з метою обфускації.

- Розділення залежного коду. Програму значно простіше аналізувати, коли усі необхідні логічні зв'язки та залежності залишаються у вихідному вигляді (так, як зручно програмісту). Тому, для того, аби ускладнити читання коду, їх необхідно фізично розділити. Застосовувати такий підхід можна як до збірки з обфускуванням, так і до вихідного коду. На практиці популярно користуватися стрибками в алгоритмі програми. Тобто, можливо перетасувати логіку алгоритму таким чином, що для правильного його виконання необхідно обов'язково скористатися стрибками goto для відновлення вихідного потоку керування.

- Надлишковий код. Така стратегія додає непотрібні інструкції, які не є функціонально залежними. Наприклад, для двійкових файлів додають інструкції no-operation (0x00). Однак, цим ми не обмежені, можливо також додавати непотрібні методи або ж фейкові інструкції у коди Android smali.

Надлишковий код змінює сигнатуру програмного продукту, тому, фактично, цим також активно користуються для статичного розпізнавання шкідливих додатків.

Перетворення форматування найменш залежне від проблем сумісності або логічних помилок, оскільки воно майже не змінює синтаксис вихідного коду продукту. Тому такий підхід активно застосовують розробники на практиці. Крім того, така методика часто допомагає зменшити розмір вихідного продукту, оскільки замінює довгі назви на односимвольні, що також приваблює веб-розробників.

Важливо розуміти, що такі перетворення є обмеженими, проте мають дуже ефективну рису незворотних перетворень, що забезпечує стійкість до деобфускації. Прикладом обмежень для мови JAVA можуть слугувати деякі відомості з вихідного форматування, такі як ідентифікатори методів із JAVA SDK. Залишки такої інформації частково спрощують роботу зловмисникам.

2. Обфускація графу управління. Збільшення складності читання програмних продуктів шляхом перетворення елементів керування. Досягається за допомогою:

- Надлишковостей в графі керування. Надлишковості додають до керуючих потоків, які свідомо вносяться у програмний код. Такі вставки виконуватись не будуть, тому впливу на зниження продуктивності такі зміни не несуть. Однак це може ускладнити програму. Прикладом може слугувати McCabe complexity, складність вихідного алгоритму якого обмежується обчисленням числа ребер на графі потоку управління мінус число вузлів, а потім подвійним доданням пов’язаних компонентів. Щоб збільшити складність читання такого алгоритму можна ввести як нові ребра, так і вузли зв’язуючих компонентів. Щоб гарантувати недосяжність надлишковостей керуючих потоків, Коллберг запропонував застосовувати непрозорі предикати. Вони відповідають за визначення непрозорого прогнозу як предиката, результат якого відомий впродовж обфускації, але котрий доволі складно отримати , скориставшись статичним підходом до аналізу

програмного коду. Тобто, непрозорий предикат може бути постійно істинним (P^T), постійно хибним (P^F) або контекстно-залежним ($P^?$). Такі непрозорі предикати створюють, користуючись наступним схемами: програмування, чисельні та контекстуальні. Розглянемо детальніше кожну з них:

а) Схеми програмування. Оскільки змагальний аналіз програм – серйозна загроза для непрозорих предикатів, можливо застосовувати складні завдання для їх аналізу. Коллберг разом із співавторами запропонували дві класичні задачі: аналіз вказівників та розпаралелювання програм. Зазвичай аналіз вказівників зводиться до того, чи можуть два вказівники вказувати на одну адресу. Статичний аналіз для розв'язання завдання аналізу вказівників іноді може бути надто складним або практично не привести до розв'язку. Тому розробники стараються застосовувати стійкі та ефективні непрозорі прогнози, користуючись заздалегідь розробленими завданнями аналізу вказівників на динамічні об'єкти. Розпаралелювання програм створює аналогічну проблему, якою займався Коллберг та запропонував застосовувати такий підхід для вдосконалення складності аналізу вказівників шляхом одночасного оновлення вказівників.

б) Числові схеми. Створюють непрозорі предикати математичних виразів. Для прикладу, вираз $7x^2 - 1 \neq y^2$ постійно істинний для усіх цілих чисел x та y. Для створення фіктивних керуючих потоків можна використовувати такі непрозорі предикати. На рис. 2.4 (б) зображено саме такий випадок створення фіктивного керуючого потоку, який ніколи не буде виконаним. Проте користуватися таким методом потрібно рідко, щоб зловмисники мали небагато шансів виявити таке заплутування. Тому важливо розробити такий алгоритм, котрий визначатиме на основі заданих ймовірностей, чи застосовувати такий підхід чи ні, а якщо потреба існує, то яким саме згенерованим унікальним непрозорим предикатом заплутувач скористається.

```
int a, b;
…
if(7*a*a – 1 != b*b){
  //always true
  originalCodes();
}else{
  bogusCodes();
}
```

(a) **Непрозора константа**

```
int x; //for any x>0
…
while(x>1){
  if(x%2==1)
    x=x*3+1;
  else x=x/2;
  if(x==1)//always reachable
    originalCodes();
}
```

(b) **Гіпотеза Коллатза**

```
int *p = &x;
int *q = &x;
if((*p)%2 == 0){
  y = x+1;
}else{
  y = x+1;
  y = y+2;
}
```

```
if((*q)%2 == 0){
  y = y+3;
  x = y+3;
}else{
  x = y+3;
}
```

(c) **Динамічний непрозорий предикат**

Рис. 2.4. Числові схеми [6]

Інший підхід із значно вищим ступенем безпеки полягає у застосовуванні функцій криптографії (наприклад, геш-функцій, гомоморфного шифрування). Тобто, можна замінити предикат, щоб приховати вибір. Варто знати, що такий підхід досить популярний для обфускації шкідливого програмного коду від динамічного аналізу, тому його актуальність сумнівна, а також призводить до значного обчислювального навантаження для виконання алгоритму.

в) Контекстуальні схеми. Ними користуються для складання непрозорих предикатів ($P^?$). Предикати повинні мати деякі детерміновані властивості, щоб ними можна було скористатися при заплутуванні. Для прикладу, Drape разом з командою у 2009 році запропонували користуватися такими непрозорими предикатами, котрі інваріантні за контекстуального обмеження. Нехай непрозорий предикат $x\ mod 3 == 1$ завжди істинний, якщо $x\ mod 3$ дорівнює одиниці, то збільшуємо *x* на одиницю, в іншому випадку $x = x + 3$. Можна користуватися динамічними непрозорими предикатами, які містять послідовність корельованих предикатів, так що результат оцінки кожного предикату може варіюватись. Однак, поведінка програми стане недетермінованою. На рис. 2.4 (c) наведено саме таке приклад, де, незалежно від того, як задано змінні $*P$ та $*q$, програма еквівалентна $y = x + 3$, $x = y + 3$.

- Ймовірності в графі керування. Фіктивні керуючі потоки цілком здатні створити проблеми для статичного аналізу програмного продукту, однак

вразливі до динамічного аналізу, оскільки мають неактивний статус. Стратегія боротьби із загрозою зводиться до реплікації керуючих потоків з тією ж семантикою, але уже іншим синтаксисом. При отриманні одного й того ж сигналу на вхід, алгоритм може вести себе по різному в різний час виконання, що є перевагою при боротьбі з типом атак бічними каналами. Тобто, дана стратегія аналогічна фіктивним керуючим потокам із контекстуальними непрозорими предикатами, проте має іншу природу. Контекстуальні непрозорі предикати створюють мертві шляхи, не вводячи надлишковий код.

- Використання диспетчера. Керування, котре ґрунтується на диспетчері, визначає декілька блоків коду, котрі виконуватимуться під час роботи алгоритму. Такі керуючі елементи сприяють заплутуванню потоку управління через можливість приховати вихідні керуючі потоки від статичного аналізу програм. Найпопулярніший підхід обфускації на основі диспетчера – вирівнювання керуючого потоку, котре перетворює коди в дрібніші та складніші. Прикладом може слугувати перетворення циклу while в іншу форму за допомогою оператора switch-case (рис. 2.5 с). На першому кроці перетворення вводять оператори if-then-goto (рис. 2.5а-б). На другому кроці, їх змінюють за допомогою switch-case. Такі перетворення неявно реалізують вихідну семантику програмного коду внаслідок керування потоком даних. Порядок виконання блоків такого коду визначається змінною динамічно, тому взнати керуючі потоки без виконання алгоритму неможливо. Cappaert і Preneel [7] у 2010 році формалізували вирівнювання керуючого потоку з диспетчером, котрий керує наступним блоком коду, що підлягає виконанню. Після завершення виконання такого блоку керування знову повертається у руки диспетчера. Можна впровадити завдання аналізу вказівників, щоб підвищити стійкість вирівнювання потоку до статичного аналізу. Стійкість значно зросте при одночасному поєднанні з фіктивними блоками коду.

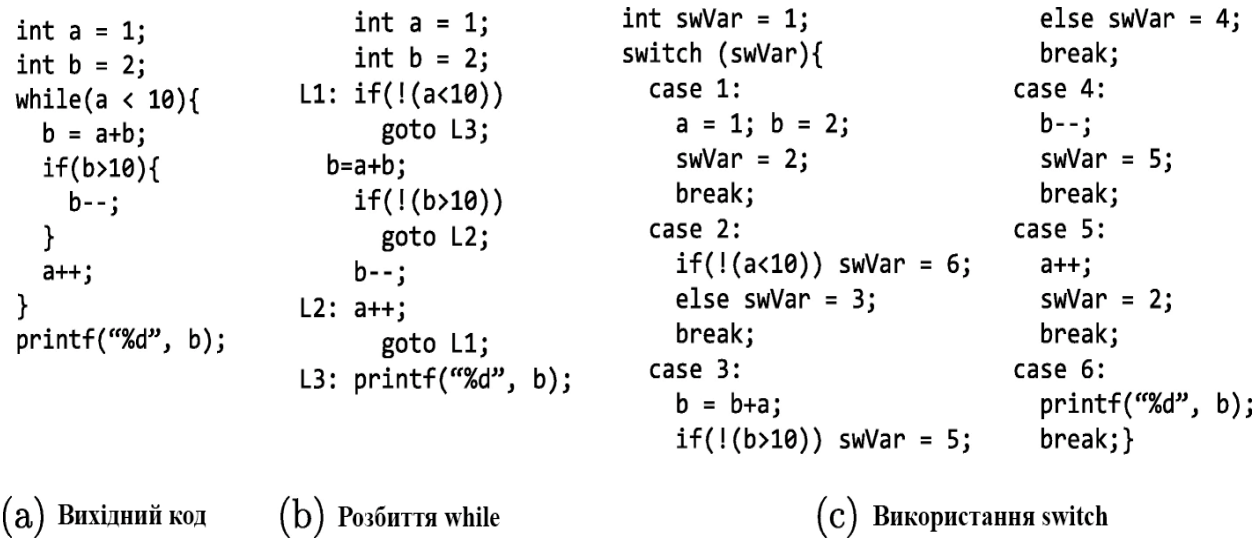


Рис. 2.5. Розбиття while за допомогою goto та switch [6]

Аналогічні вирівнювання керуючого потоку можна забезпечити для операторів try-catch, while-do, continue, break. Механізм ґрунтується на абстрактному синтаксичному дереві та забезпечується фіксованим шаблоном компоновки. Для блоків коду, що підлягають заплутуванню, в зовнішньому циклі створюють оператор while та switch-case відповідно вже у середині циклу. Однак, така структура не здатна заплутати зловмисників, оскільки вони відразу можуть визначити, котрий блок коду виконуватиметься наступним. Такий підхід можна поліпшити заміною прямих вказівників на заплутані (приміром, користуючись if-else або певними обчислювальними функцій).

Очевидно, що застосування диспетчера не обмежене вирівнюванням керуючого потоку. У 2003 році запропоновано можливість заплутування двійкових файлів функцією галуження, котра направляє блоки для виконання на основі інформації стеку. Аналогом такого підходу можуть слугувати розгалужені функції, котрі дозволяють заплутувати об'єктно-орієнтовані програми, де визначено єдиний підхід до виклику методів з певним пулом об'єктів. Для вдосконалення заплутуючих властивостей таких підходів можна приховати керуючу інформацію в іншому автономному процесі за допомогою встановлених міжпроцесних комунікацій.

Отже, обфускація на основі диспетчера є беззаперечно стійкою до статичного аналізу, оскільки вона приховує граф потоку управління в програмі. Однак обійти його можна за допомогою спеціальних динамічних або гібридних методів аналізу.

- Неявний контроль Така стратегія має на меті перетворити явні інструкції керування в неявні. Такий підхід перешкоджає реверс інженерам звертатись до правильних керуючих потоків. Якщо взяти для прикладу асемблер, то керуючі інструкції асемблерних кодів (jmp, jne) можна реалізувати різними комбінаціями mov та інших можливих інструкцій з тією ж семантикою керування. Важливо знати, що усі підходи до заплутування керуючого потоку зосереджуються на перетвореннях у синтаксичному рівні, заплутування якого нечасто розглядають. Незважаючи не те, що вони вдосконалюють стійкість до аналізу коду зловмисниками, їх заплутуюча ефективність щодо семантичного захисту не до кінця зрозуміла.

3. Обфускація даних. Існуючі методи обробки даних зосереджено на загальних типах даних, таких як рядки, цілі числа, масиви. Можна перетворити так дані, скориставшись розбиттям, збиттям, кодуванням, процедуризацією та ін. Основні з них:

- Розбиття та об’єднання даних. Розподіляють інформацію однієї змінної на декілька нових. Наприклад, булеву змінну можна розбити на дві, згенерувати логічну залежність та отримати ті ж вихідні дані, що й з однією. Аналогічно можна й об’єднати змінні.

- Впровадження обробки даних. Даний підхід заміняє статичні дані викликами процедур. Коллберг запропонував замінити усі рядки в програмному коді функцією, яка буде генерувати їх, наділяючи спеціальними значеннями параметрів для покращення заплутування.

- Шифрування даних. Дані кодують шифрами або ж математичними функціями. Прикладом може слугувати кодування рядків афінними (або іншими) шифрами, поєднане із дискретними логарифмами для упакування

слів. Визначені числа можна закодовувати операції виключного «або», а згодом за необхідності розшифровувати їх під час виконання на льоту.

- Перетворення масивів. Важко не погодитись, що масив – одна з структур даних, котрою найчастіше користуються. Маніпуляцій з такими структурами можна здійснити доволі багато: згладжування, згортання, розбиття, злиття та інші. Індексацію масивів можна перетворити за допомогою функцій та процедур.

4. Методи обфускації. Очевидно, що можна вигадати дуже багато методів, проте необхідно розглянути основні з них:

- Заміна виклику процедури тілом. Методом називають незалежну процедуру, котру можуть викликати інструкції програми. Можна замінити вихідний процедурний виклик безпосередньо самим тілом функції. Аналогічно можна витягнути послідовність інструкцій та абстрагувати метод. Хорошою практикою вважають заплутування початкової абстракції процедур.

- Метод клонування. Якщо конкретний метод часто викликають у програмі, то можна спробувати створити копії цього методу і випадковим чином викликати одну з цих копій. Щоб це не кидалось в очі зловмиснику, потрібно унікально заплутати кожну з копій шляхом різних обфускуючих перетворень.

- Метод агрегації та розсіювання. Ідеологія така ж, як і у заплутуванні даних, тільки уже з методами. Можна здійснити розбиття конкретного метода на декілька дрібних із заплутаними залежностями, або ж, навпаки, об’єднати декілька нерелевантних методів в один.

- Метод перенаправлення. Ідея підходу полягає у створенні прихованих перенаправляючих методів, що перешкоджатиме зворотній інженерії. Тобто, можна створити перенаправлення як публічні статичні методи, але з випадковими ідентифікаторами. На один метод посилатиметься декілька таких новоутворених перенаправлень. Така ідея стає у пригоді, коли замінити оригінальну назву методу неможливо (залежність від імпорту і т.д.).

5. Обфускація класів. Ідеологія заплутування класів близька до методів обфускації. Основна відмінність полягає в тому, що класи наявні не у всіх мовах програмування, а лише у об'єктно-орієнтованих (JAVA, .NET), тому їх виділено в окрему категорію. Розглянемо основні стратегії заплутування класів:

- Відкидання модифікаторів. Оскільки ООП містять модифікатори (public, private), котріобмежують доступ до елементів класу, або ж до класу в цілому, то відкидання таких модифікаторів значно допомагає реалізувати заплутування класів, проте усі члени класу стають загальнодоступними.
- Розбиття та збиття класів. Суть полягає у заплутуванні при проектуванні класів розробником. При об'єднанні класів можливо переносити локальні змінні або частини інструкцій в інший клас. Аналогічно при розбитті можна виносити частину змінних або інструкцій із одного класу в інші.
- Збиття ієрархії класів. У ООП часто користуються таким інструментом як інтерфейс. Як у методі перенаправлення, можна створювати щось схоже для класів з інтерфейсами, проте в цьому методі розриватимемо видимі зв'язки наслідування між класами та інтерфейсами. Якщо дозволити кожній гілці в ієрархії користуватися одним і тим же інтерфейсом, ієрархія згладиться.

2.5.2 Рівень програмних компонентів. Тепер розглянемо методи обфускації, котрі не мають справи з конкретним синтаксисом коду або його елементами. Такі методи включають у себе перетворення коду, запобігання декомпіляції, диверсифікацію та обфускацію на рівні віртуальної машини (рис. 2.6). Дослідимо кожен з них:

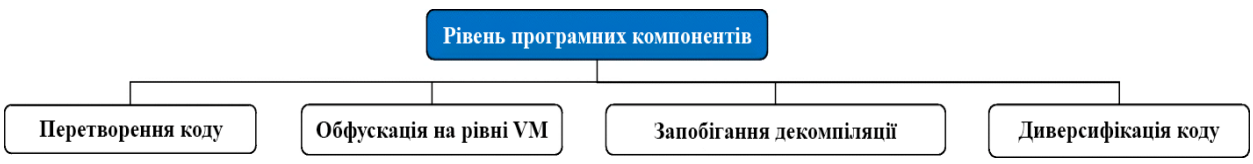


Рис. 2.6. Рівень програмних компонентів [6]

1) Перетворення коду. Мета – заплутати проміжний код, що також заплутує трансляцію програм (на мові С) так, що двійкові файли важче зрозуміти. Така методика матиме ліпший результат, якщо інструкції на високому рівні також заплутано.

2) Обфускація на рівні віртуальної машини. Досить популярний метод, котрим доволі часто користуються на практиці. Він перетворює вихідні машинні інструкції в код операцій, призначений лише для конкретної віртуальної машини. Спрощена віртуальна машина вбудовується у код програми для інтерпретації коду операцій у режимі реального часу. У результаті вихідний код такого продукту буде невеликим завантажувачем, що ініціює віртуальну машину. Станом на сьогодні існують десятки інструментів для таких задач (VMProtectFootnote2, ThemidaFootnote4).

Оскільки така методика обфускації популярна, то деобфускація такого типу заплутувань теж не залишилась без уваги. З метою поліпшити способи захисту встановлено, що статичні підходи ефективні для декодування, а саме для знаходження зв'язків між кодом операцій та асемблерним кодом, і запропоновано динамічний підхід, котрий користується різними відображеннями для різних блоків коду. Також у ході досліджень виявлено, що шляхи виконання сучасних підходів на основі віртуальних машин детерміновані для одного й того ж вхідного сигналу, що може бути вразливим місцем. Таку задачу можна розв'язати ,поєднавши декілька віртуальних машин із різними наборами команд в одну, а саме, планування вибору шляху зробити перетворити на недетерміноване (додати в нього схему планування з випадковим вибором шляху).

3) Запобігання декомпіляції. Ідея методу полягає у так званому піднятті планки заплутаності для зловмисників, щоб навіть у разі зламу та отримання вихідного коду, останній був у максимально нечитабельному форматі. Це буде корисно для таких мов програмування як JAVA, .NET, C. При стандартних методах декомпіляції із машинного коду зловмисник стикнеться з перешкодами у вигляді помилок декомпіляції. Дієво вставляти

незакінчені інструкції у код програми після безумовних стрибків. Такі інструкції надлишкові, проте дизасемблер не зможе впоратись із такою ситуацією, оскільки виникнуть проблеми поділу незакінчених інструкцій. Вдосконалити заплутування можливо за допомогою допоміжної обфускації керуючого графа. Іншим підходом, котрий запобігає декомпіляції може стати підміна іменування змінних на імена, зарезервовані середовищами розробки. Програма залишиться цілком працездатною, оскільки перевірка здійснюється лише зовнішніми інтерфейсами при компіляції у середовищі розробки. Після таких перетворень проблеми з декомпіляцією неминучі.

4) Диверсифікація коду. Розглянуті вище методи вносили різні заплутування до конкретного програмного компоненту, в той час як диверсифікація коду базується на одночасній генерації одразу декількох обфускованих версій компонента. Це може запобігти масштабним атакам на однорідне програмне забезпечення, оскільки додає декілька додаткових бар'єрів для зловмисників. Однак таким методом користуються розробники шкідливого ПЗ для створення копій із різними сигнатурами, що перешкоджає виявленню антивірусами.

2.5.3 Міжкомпонентний рівень. Сучасний програмний комплекс зазвичай містить компоненти, написані розробниками та запозичені з інших, уже готових бібліотек (рис. 2.7).

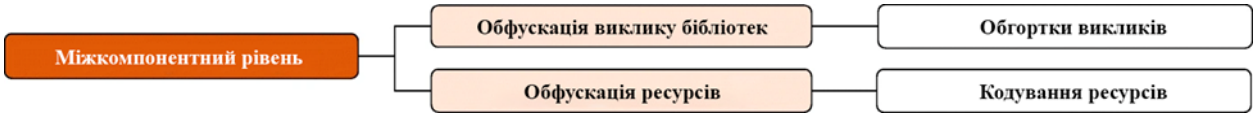


Рис. 2.7. Міжкомпонентний рівень [6]

Така можливість запозичення значно прискорює розробку ПЗ, проте створює проблеми для якісного заплутування. Зокрема, розробники, запозичивши функції з бібліотеки, не можуть змінювати ідентифікатори функцій, а така інформація може значно прискорити процес аналізу та

деобфускації програми. Для розв'язання такої задачі запропоновано заміняти стандартні виклики функцій менш очевидними закономірностями. Виклики можна заплутати, пропустивши їх через внутрішній проксі сервер, який є обфускованим класом, що здійснює обгортку функцій. Для обфускації команд через Windows PowerShell можна створити безглузду стрічку, і за допомогою стрічкових операторів Windows здійснювати перетворення такого тексту в допустиму команду в режимі реального часу. В останні роки популярності набувають засоби шифрування файлів ресурсів програмних пакетів та реалізація їх розшифрування під час виконання.

2.5.4 Прикладний рівень. Розглянуті вище методи не пов'язані з функціональністю програмного забезпечення. У цьому пункті розглянуто декілька методів обфускації, котрі призначено для ПЗ з певним функціоналом, зокрема для DRM систем та нейронних мереж (рис. 2.8). Така гілка може розширюватись при виявленні нових методів обфускації додатків.

Рис. 2.8. Прикладний рівень [6]

DRM системи контролюють доступ користувачів до мультимедійних файлів. Найчастіше контент шифрують., а ключі дешифрування приховують, особливо, коли ПЗ запускають у незахищеному середовищі та зловмисники мають повний доступ до дешифрування. Шифрування «білої скриньки» – метод обфускації, котрий зможе протистояти схожим атакам.

На високому рівні така методика попередньо оцінює усі операції, пов'язані з ключами, та заплутує їх. Прикладом може слугувати алгоритм DES, що містить 16 раундів функцій Фейстеля. Кожна функція аналізує відкритий текст, користуючись ключем, а потім впроваджує таблицю пошуку та перестановок для отримання вихідних даних. Додатково можна впровадити шифрування результатів кожного раунду на мережевому рівні.

Нейронні мережі стали новою парадигмою у програмуванні та значно змінилися в останнє десятиліття. Дослідження показали, що саме структура нейронних мереж – критичне місцем моделей глибокого навчання. Тому така структурна інформація – ключова інтелектуальна власність для такого типу ПЗ.

Для заплутування моделей глибокого навчання запропоновано метод обфускації на основі моделювання. Такий метод здійснює перегонку знань уже добре навчених моделей глибокого навчання до моделей неглибоких мереж. В результаті точність у обох буде однакова, проте зловмисники не зможуть дізнатись усіх налаштувань як імітації так і навчання таких мереж.

2.6 Висновки до розділу

Отже, обфускація дає можливість розв’язати задачу незаконних модифікацій програмних продуктів. Для визначення ефективності застосування методів обфускації проаналізовано найефективніший універсальний метод експертної оцінки за трьома критеріями (стійкість, еластичність, вартість перетворення), досліджено доцільність використання методів на основі циклічної складності та розгалужень потоку управління, а також описано емпіричні методи оцінки.

Досліджено усі найпопулярніші методи обфускації на різних рівнях, у залежності від кінцевої мети. Описано шляхи їх вдосконалення та поєднання для досягнення найкращого результату.

РОЗДІЛ 3
ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

На сьогодні існують різні програмні продукти, котрі пропонують обфускацію вихідного коду, проте ефективні алгоритми обфускації реалізовано тільки у платних продуктах, безкоштовні ж працюють здебільшого на рівні мініфікації коду (зменшення розміру вихідного коду внаслідок скорочення імен змінних і функцій, видалення символів форматування коду) та алгоритми їх роботи уже відомі.

Після аналізу існуючих обфускаторів прийнято рішення реалізувати комбінації ефективних алгоритмів обфускації.

Мета роботи – підвищити рівень захищеності інформації та програмних продуктів, написаних на мові JavaScript, власними методами обфускації з подальшим об'єднанням їх в комплексний алгоритм заплутуючих перетворень з вибором режимів обфускації.

3.1 Проектування обфускатора

Після аналізу доступних готових засобів обфускації для мови JavaScript, побудовано діаграму прецедентів для проектованого обфускатора (рис. 3.1).

Діаграма прецедентів UML –діаграма, котра показує зв'язок між так званими «акторами» і діями (прецедентами), котрі вони можуть виконувати. Така діаграма дозволяє точно описати повну функціональність програмної системи.

Як видно з рис. 3.1, користувач може завантажувати файл з вихідним кодом, зберігати обфускований код в файл, запускати обфускацію, а також вибирати між наступними режимами обфускації:

- crush (спрощений та обчислювально легкий);
- reverse (складний з використанням інверсій);

- zero (складний із застосуванням випадковостей).

Всі перераховані вище режими обфускації застосовують логічні перетворення в програмно заданому порядку, який неможливо змінити, тобто є етапами. Це дозволяє максимально ефективно заплутати вихідний код, оскільки з'являються зв'язки ще й між самими заплутуючими методами для конкретного режиму.

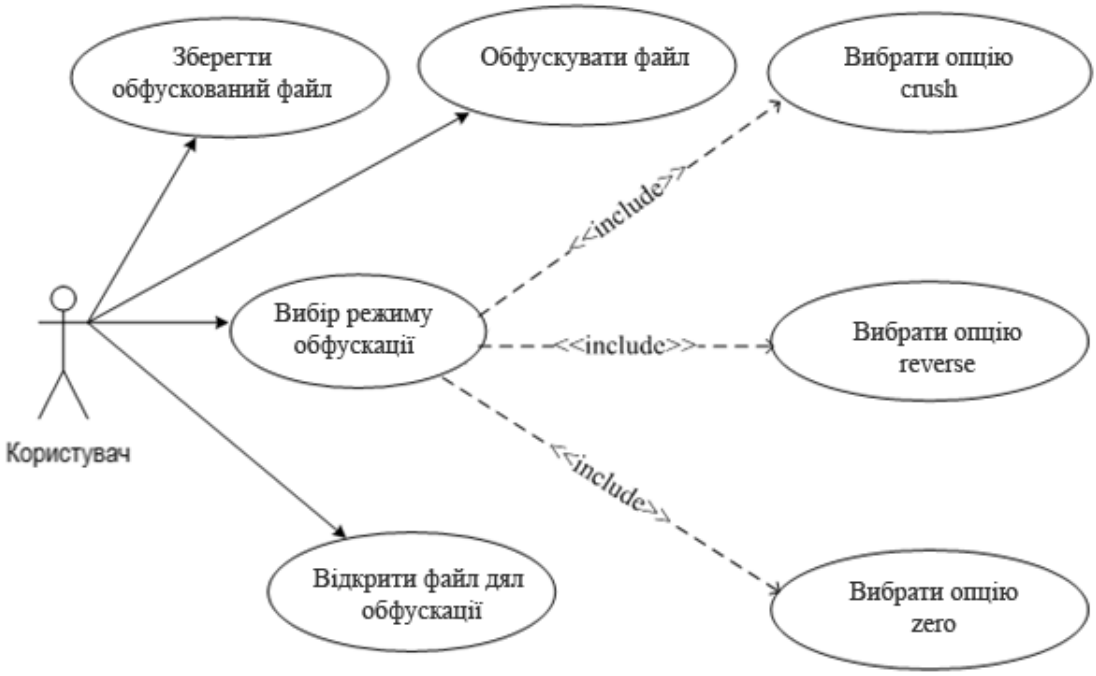


Рис. 3.1. UML діаграма прецедентів

Розроблений додаток функціонуватиме на простій html сторінці, де міститиметься головний робочий клас заплутувача, котрий заплутуватиме вхідний код на мові програмування JavaScript. У свою чергу, у залежності від обраних режимів, обфускатор застосовуватиме відповідні алгоритми заплутувань. Після обфускації програма виводитиме вихідний файл, котрий міститиме обфускований JavaScript код, еквівалентний вихідному.

На основі спроектованої діаграми прецедентів, побудовано діаграму потоків даних обфускатора, наведену на рис. 3.2.

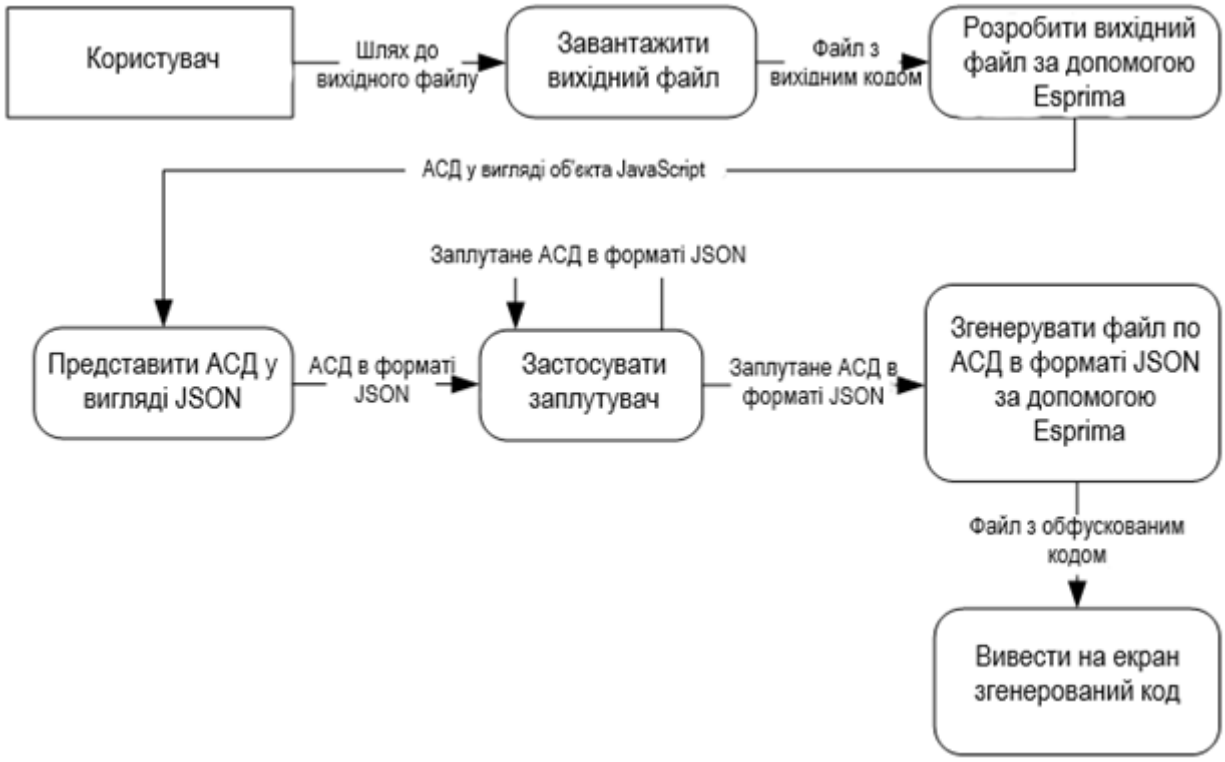


Рис. 3.2. Діаграма потоків даних роботи обфускатора з файлом

Користувач обирає файл .js (або вставляє у відповідне поле), що підлягає заплутуванню, аналізатор Esprima розбиває отриманий код до вигляду абстрактного синтаксичного дерева. Після чого АСД уже перетворюють в JSON та відправляють на заплутування, відповідно до обраної опції (crush, reverse, zero). Отримане заплутане АСД у форматі JSON перетворюють назад у JavaScript код та виводять його у відповідне поле на сторінці.

3.2 Аналіз середовища розробки та застосованих технологій

Основне середовище розробки програмного продукту – Visual Studio Code. Елементи інтерфейсу створено на базі і основних методів html та css. Перетворення синтаксису JavaScript у АСД (абстрактне синтаксичне дерево) забезпечив аналізатор Esprima, а обернений процес генерації – Escodegen.

3.2.1 Visual Studio Code. VS Code – безкоштовне повноцінне середовище розробки програмних продуктів з багатим функціоналом (рис. 3.3). Відмінно підходить для веб-продуктів. Повністю підтримує більшість мов програмування, включаючи node.js, TypeScript та JavaScript.

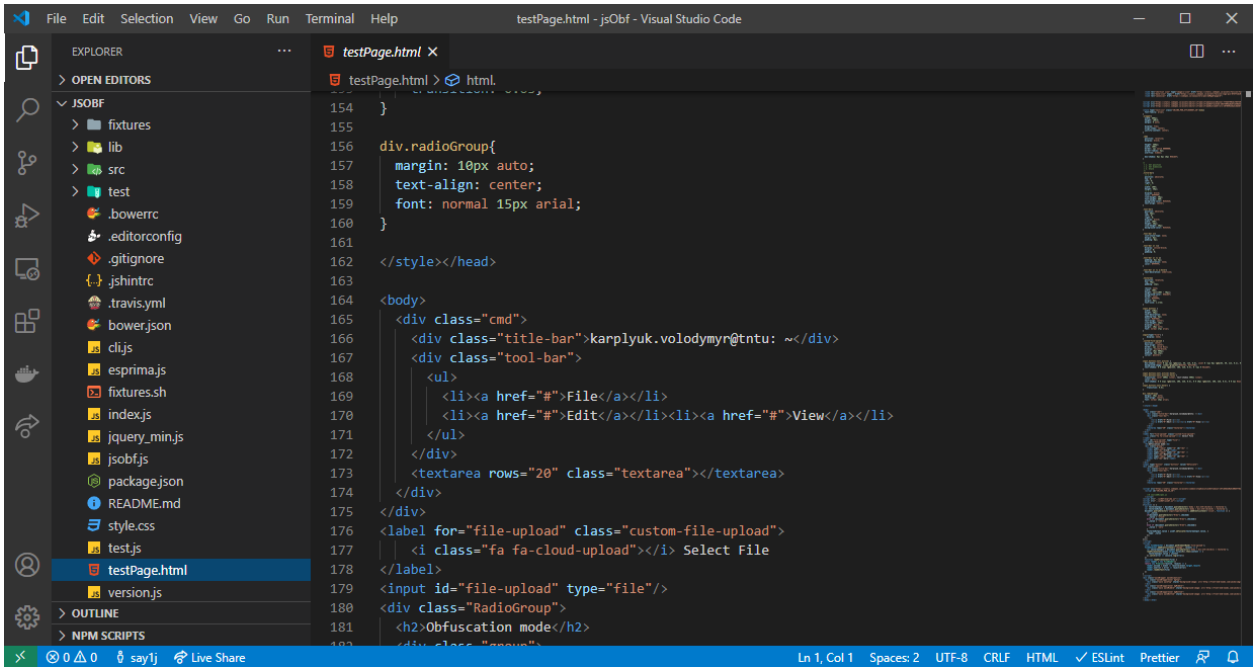


Рис. 3.3. Середовище розробки VS Code

Реалізовано інструментарій для роботи з Git, технології автодоповнень IntelliSense, підсвітку синтаксису, підтримку терміналів, рефакторинг. Підтримує одночасну роботу одразу з декількома проєктами (у різних вікнах). Містить безліч додатків як до текстового редактору коду, так і до самого середовища. Наявна вбудована та широко функціональна відладка коду. VS Code створено компанією Microsoft для операційних систем Windows, Linux, MacOS. Таке середовище розробки легше від основного аналогу Microsoft Visual Studio.

Отже, Visual Studio Code – потужне, просте у та достатньо гнучке середовище розробки, котре можна розширити за допомогою уже існуючих або власних плагінів.

3.2.2 Аналізатор Esprima. Обфускаційні перетворення вихідного коду програмного продукту можна провести, отримавши абстрактне синтаксичне дерево, котре структурно представляє програму без елементів конкретного синтаксису. Таку задачу відмінно розв'язує бібліотека від JQuery Esprima.

АСД можна зобразити графічно. Вузли такого дерева позначають іменами операцій або операндів, а ребра відображають відповідні відношення «батько-син» між такими вузлами.

Такий абстрактний синтаксис мови програмування визначає лише значущі компоненти конструкцій. Тобто, АСД єдине незалежно від конструкції (перестановки операндів або префіксного/постфіксного/ інфіксного запису виразу). У гілках дерева зберігається інформація про тип вузла, його властивості та синів.

На мові програмування JavaScript існує бібліотека Esprima, яка перетворює вихідний код програми на АСД у вигляді JSON.

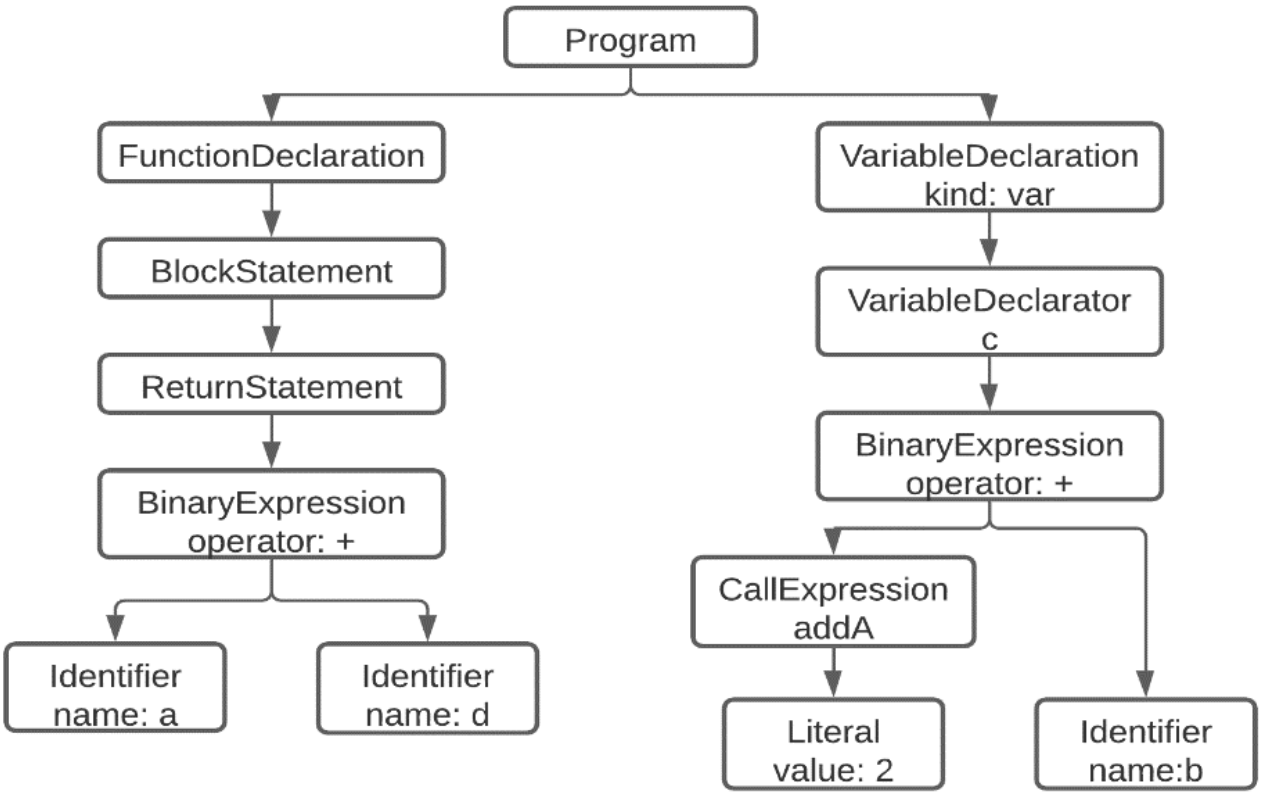


Рис. 3.4. Графічний приклад АСД бібліотеки Esprima

Розглянемо простий приклад такого перетворення, отриманого за допомогою даної бібліотеки:

```
Function addA(d) {
  return a + d
}
var c = addA(2) + b
```

Припустимо, визначено просту функцію додавання двох чисел, котрою обчислють значення с. АСД такого коду зображено на рис. 3.4.

АСД у вигляді об’єкту JSON на прикладі змінної x із значенням ‘x’ наведено на Рис. 3.5.

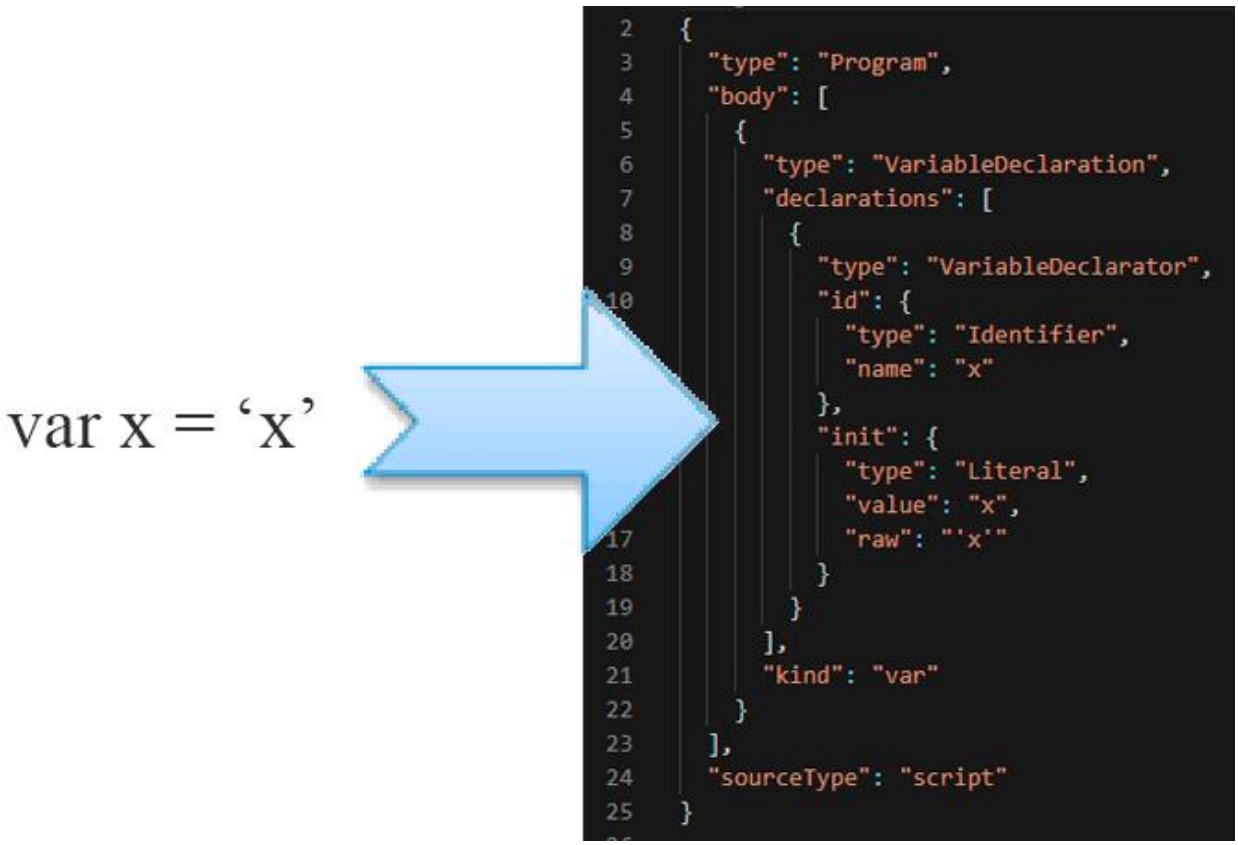


Рис. 3.5. АСД у вигляді JSON

Обернену генерацію коду за АСД можна виконати засобами тієї ж Esprima або його аналогу Escodegen. Оскільки при тестуванні обох систем результати виявились практично ідентичними, користувалися тільки Esprima, щоб не навантажувати програму зайвими бібліотеками.

3.3 Розробка алгоритмів роботи обфускатора

Необхідно відзначити, що основним об'єднуючим алгоритмом програми є такий обраний (із трьох) алгоритм, в якому об'єднуються всі встановлені обфускаційні перетворення та вихідний код перетворюється у незрозумілий для людини (рис. 3.6).

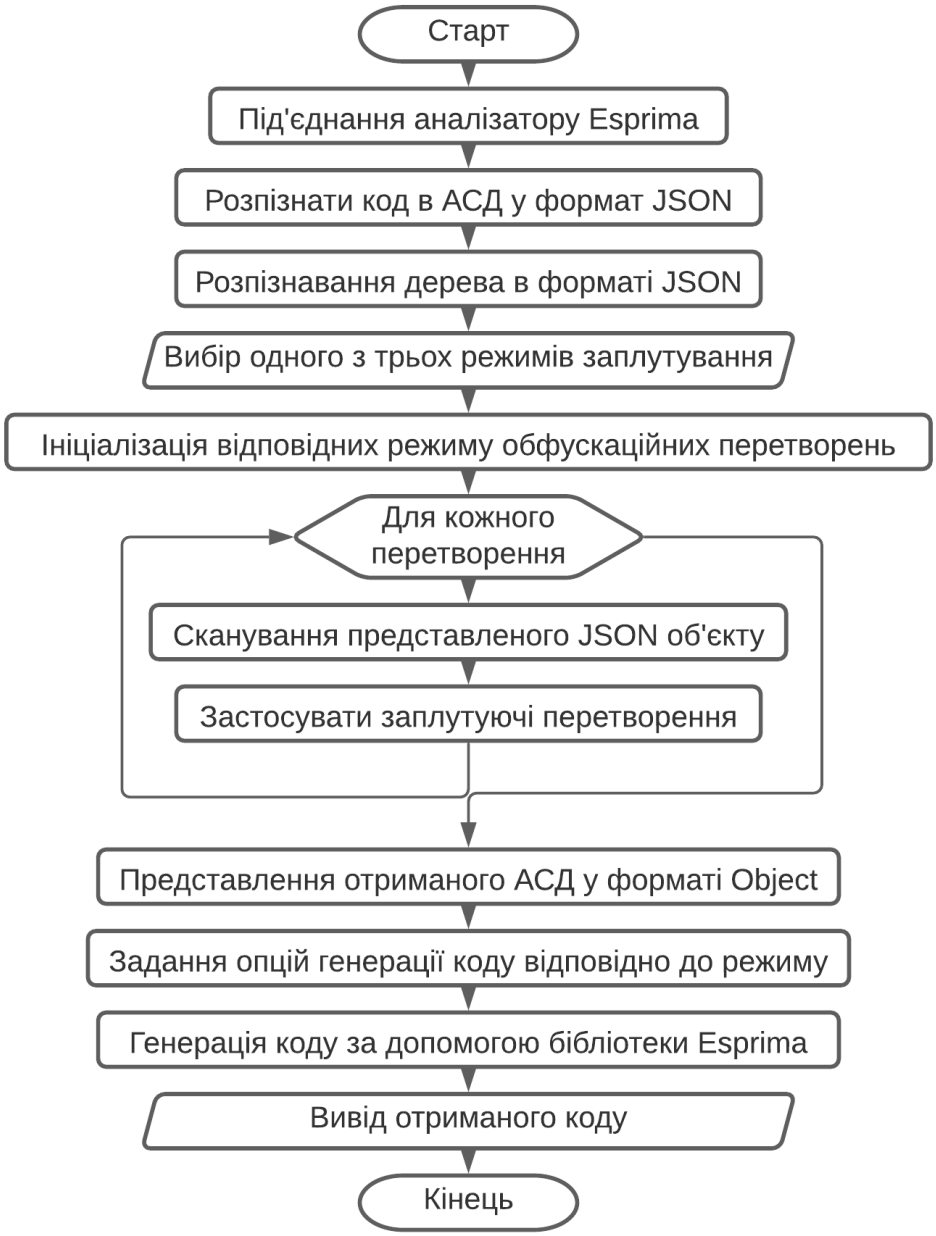


Рис. 3.6. Блок-схема алгоритму роботи програми

Розглянемо усі три режими обфускаційних перетворень та алгоритми заплутувань, котрими користуються у різних пропорціях.

3.3.1 Алгоритм логічного перетворення. Такий алгоритм перетворює логічні вирази. Приміром, один із режимів його роботи інвертує порівняння таким чином, щоб замість вихідного оператора застосовували заперечення протилежного оператора з заперечувальними операндами (див. рис. 3.7). На схемі алгоритму вказано тільки частину логічних операторів.

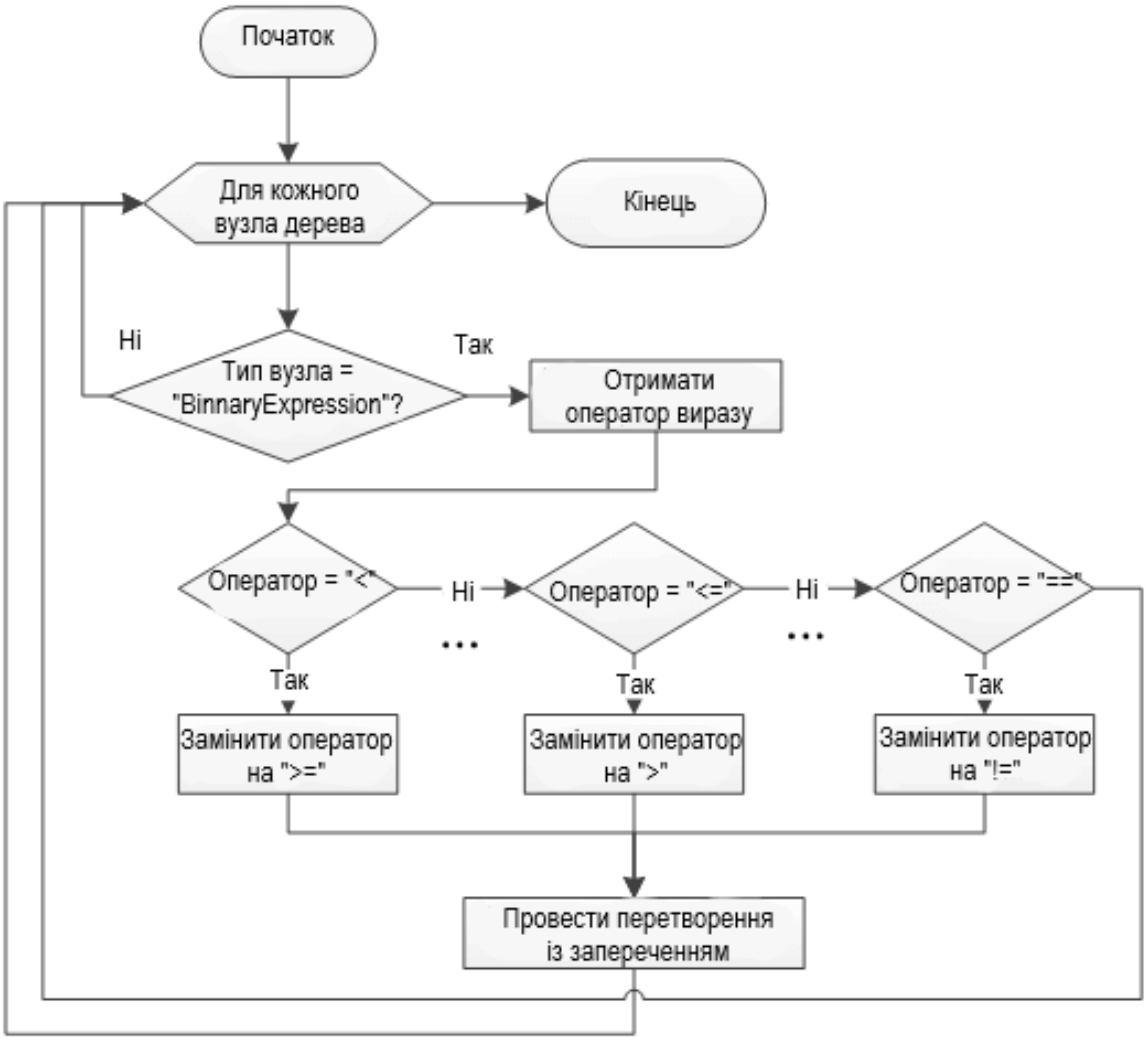


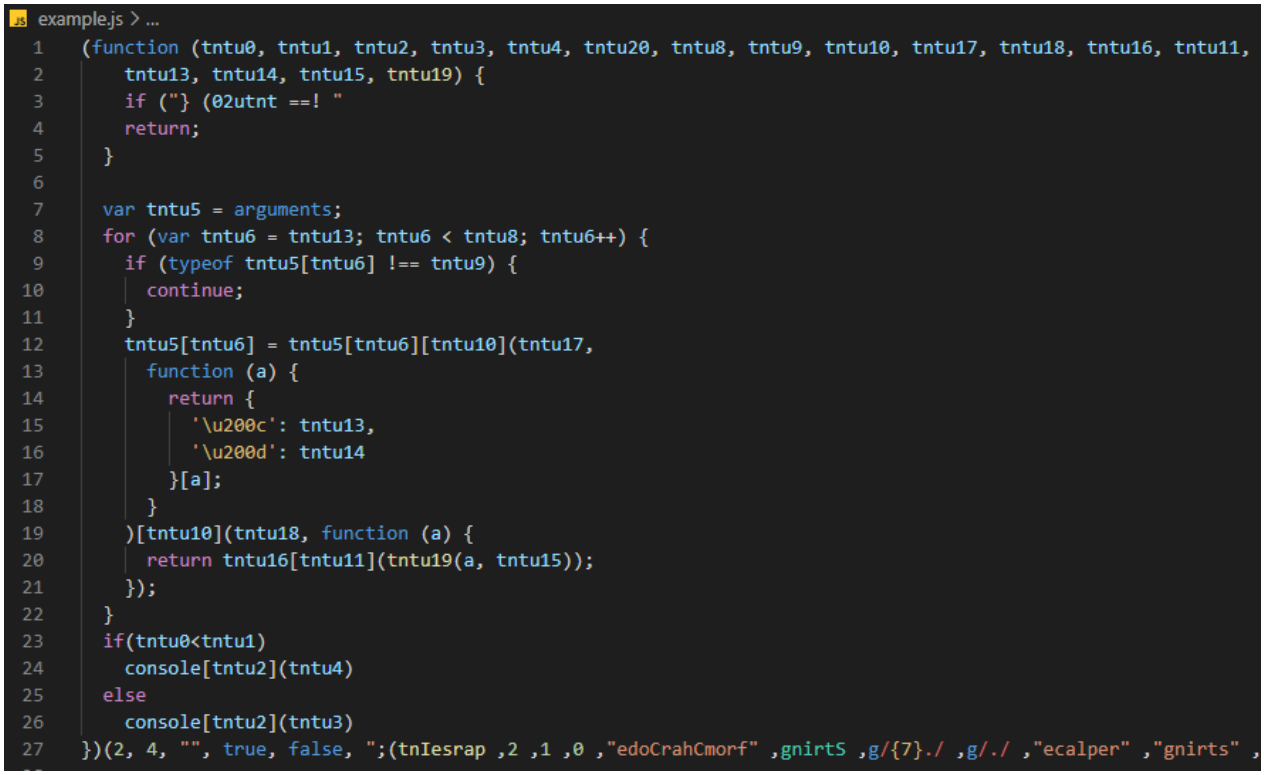
Рис. 3.7. Алгоритм інверсного перетворення логічних виразів

Роботу алгоритму з логічними перетвореннями показано на прикладі логічного виразу виводу булевого значення на консоль:

```
if(2>4)
```

```
    console.log(true)
else
    console.log(false)
```

Обфускувавши наведений вище код обфускаційним режимом reverse, отримаємо надлишковий код із випадково згенерованими логічними операторами та відповідними їм діями, а також бажаний результат із запереченням потрібного логічного оператора та перетворенням булевих операндів на 0, 1 та надлишкову 2 в коді програми (рис. 3.8).



```
example.js > ...
(function (tntu0, tntu1, tntu2, tntu3, tntu4, tntu20, tntu8, tntu9, tntu10, tntu17, tntu18, tntu16, tntu11,
    tntu13, tntu14, tntu15, tntu19) {
    if ("} (02utnt ==! "
    return;
  }

  var tntu5 = arguments;
  for (var tntu6 = tntu13; tntu6 < tntu8; tntu6++) {
    if (typeof tntu5[tntu6] !== tntu9) {
      continue;
    }
    tntu5[tntu6] = tntu5[tntu6][tntu10](tntu17,
      function (a) {
        return {
          '\u200c': tntu13,
          '\u200d': tntu14
        }[a];
      }
    )[tntu10](tntu18, function (a) {
      return tntu16[tntu11](tntu19(a, tntu15));
    });
  }
  if(tntu0<tntu1)
    console[tntu2](tntu4)
  else
    console[tntu2](tntu3)
})(2, 4, "", true, false, ";(tnIesrap ,2 ,1 ,0 ,"edoCrahCmorf" ,gnirtS ,g/{7}./ ,g/./ ,"ecalper" ,"gnirts" ,
```

Рис. 3.8. Результат логічних перетворень

Варто зауважити, що можна користуватися й іншими логічними перетвореннями, застосовуючи логічні перетворення, проте в JavaScript результатом логічного виразу може бути не тільки `true` або `false`, а й об'єкт. Тому більш складні формули перетворень можуть призвести до помилок.

Програмна реалізація наступна:

1. Сканування JSON об’єкта на наявність логічних операторів та літералів (рис. 3.9).

```
function recordJObj(obj, name) {
    let range;
    if (obj.type === 'Literal') {
      range = obj.range;
    } else {
      range = obj.property.range;
    }
    if (ranges[range]) {
      return;
    }
    ranges[range] = true;
    obj.$name = name;
    memberExpressions.push(obj);
    if (!propertys[name]) {
      propertys[name] = identFrom(guid++);
      names.push(propertys[name]);
      expressions.push(name);
    }
}
function scanJObj(obj, parentKey) {
    if (!obj) {
      return;
    }
    if (parentKey === 'range') {
      return;
    }
    if (obj.type === 'MemberExpression') {
      if (obj.property.type === 'Identifier' && !obj.computed) {
        recordJObj(obj, JSON.stringify(obj.property.name));
      }
    }
    if (obj.type === 'Literal') {
      if (parentKey === 'expression') {
        return;
      }
      if (/^(["']|\d)/.test(obj.raw)) {
        if (parentKey !== 'key') {
          recordJObj(obj, JSON.stringify(eval(obj.raw)));
        }
      } else {
        recordJObj(obj, obj.raw);
      }
      return;
    }
    for (let key in obj) {
      if (typeof obj[key] === 'object') {
        scanJObj(obj[key], key);
      }
    }
```

Рис. 3.9. Сканування на наявність логічних операторів та запис об’єкта

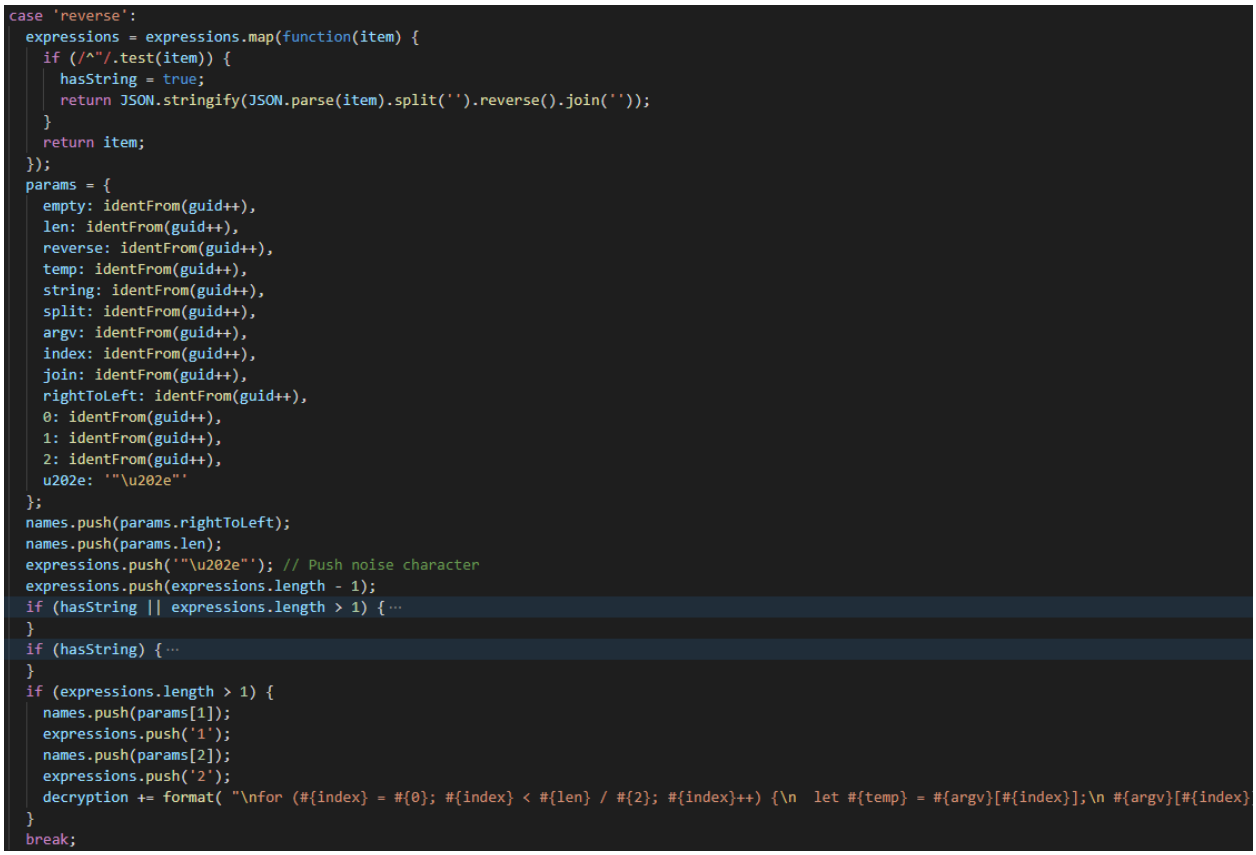
2. Запис JSON об’єкта, що підлягає перетворенню (див. рис. 3.9), попередньо (за присутності) відзначивши відповідне логічне перетворення за допомогою `expressions.push(name)`. У такому разі, застосувавши заплутування, обфускатор здійснить необхідні для режиму дії з раніше оголошеними логічними виразами.

3. Сортування згенерованих JSON об’єктів (рис. 3.10) та стиснення. Іншими словами, підготовка до встановленого режиму заплутування.

```
115     memberExpressions.sort(function(first, second) {
116       if (first.type === 'Literal') {
117         first = first.range[1];
118       } else {
119         first = first.property.range[1];
120       }
121       if (second.type === 'Literal') {
122         second = second.range[1];
123       } else {
124         second = second.property.range[1];
125       }
126       return second - first;
127     }).forEach(function(obj) {
128       if (obj.type === 'Literal') {
129         let begin = code.slice(0, obj.range[0]);
130         let end = code.slice(obj.range[1]);
131         code = begin;
132         if (/[^\s~!%^&*()_+\-={}\[\]|:";'<>,.?]$/.test(begin)) { // compress
133           code += ' ';
134         }
135         code += propertys[obj.$name];
136         if (/^[^\s~!%^&*()_+\-={}\[\]|:";'<>,.?]/.test(end)) {
137           code += ' ';
138         }
139         code += end;
140       } else { // if (obj.type === 'MemberExpression') {
141         code = code.slice(0, obj.property.range[0]).replace(/\.\s*$/, '') +
142           '[' + propertys[obj.$name] + ']' +
143           code.slice(obj.property.range[1]);
144       }
145     });
```

Рис. 3.10. Сортування та компресія JSON об’єктів

4. Застосування оголошених перетворень (рис. 3.11). Відповідно до обраного режиму заплутування, встановлюються параметри логічних перетворень, які здійснюються циклічно по кожному з них. Очевидно, що саме тіло циклу також зазнає детермінованих перетворень, що значно покращує захищеність коду від реверс інженерії. Режим reverse передбачає описане зворотне перетворення, а також вставляє рейковий несправжній цикл з перебором та випадковими логічними умовами. Режим zero зберігає логічні оператори, проте має інші надлишкові зв’язки та умови. Якщо розглядати crush, то логічні умови зберігаються, проте саме тіло зазнає змін.



```
case 'reverse':
  expressions = expressions.map(function(item) {
    if (/^"/.test(item)) {
      hasString = true;
      return JSON.stringify(JSON.parse(item).split('').reverse().join(''));
    }
    return item;
  });
  params = {
    empty: identFrom(guid++),
    len: identFrom(guid++),
    reverse: identFrom(guid++),
    temp: identFrom(guid++),
    string: identFrom(guid++),
    split: identFrom(guid++),
    argv: identFrom(guid++),
    index: identFrom(guid++),
    join: identFrom(guid++),
    rightToLeft: identFrom(guid++),
    0: identFrom(guid++),
    1: identFrom(guid++),
    2: identFrom(guid++),
    u202e: '"\u202e"'
  };
  names.push(params.rightToLeft);
  names.push(params.len);
  expressions.push('"\u202e"'); // Push noise character
  expressions.push(expressions.length - 1);
  if (hasString || expressions.length > 1) {···
  }
  if (hasString) {···
  }
  if (expressions.length > 1) {
    names.push(params[1]);
    expressions.push('1');
    names.push(params[2]);
    expressions.push('2');
    decryption += format( "\nfor (#{index} = #{0}; #{index} < #{len} / #{2}; #{index}++) {\n  let #{temp} = #{argv}[#{index}];\n #{argv}[#{index}
  }
  break;
```

Рис. 3.11. Застосування логічних перетворень

Отже, реалізацію логічних перетворень можна вважати успішною та ефективною, оскільки впроваджено три різні варіанти заплутувань. Для забезпечення кращого захисту від зловмисників різні функціональні блоки програми можна заплутати різними режимами, що значно ускладнить аналіз коду та виявлення закономірностей на його ділянках.

3.3.2 Алгоритм скорочення констант та змінних. За стандартом ECMA Script 5.0 у мові JavaScript нема констант, які б забезпечували саму мову програмування, тому програма аналізує змінні, які не змінюють свого значення в вихідному коді програми користувача.

Перетворення складається з наступних етапів:

- знайти можливі константи в коді;
- виключити мінливі змінні;

- виключити змінні, чиє значення не є простим, тобто не є числом, рядком або булевих значенням;
- за можливості замінити константи на їх значення.

Такий алгоритм передбачає застосовування свого роду геш-таблиць, що містять дані про константи та їх значення. Спочатку здійснюється перебір усіх вузлів АСД і у місцях, де вузол містить константу, її ім’я та значення вносять у геш-таблицю (рис. 3.12).

Для запобігання можливих логічних помилок при підстановці значень у місцях використання цих констант встановлено перевірку на наявність інкременту чи декременту. При їх присутності такі константи видаляють з таблиці, оскільки вказані операції не можна застосовувати безпосередньо до чисел. Аналогічна ситуація виникає зі змінними, проте, у їх випадку можлива зміна значенньь при прогонці коду, тому, коли код виконується, змінну заміняють у геш-таблиці. Коли ініціалізують складне значення, то змінну повністю видаляють з геш–таблиці та користуються нею у заданому вигляді.

```
43      let names = [];
44      if (!propertys[name]) {
45          if(!identFrom.contains(Math)){
46              propertys[name] = identFrom(guid++);
47              names.push(propertys[name]);
48              expressions.push(name);
49          }
50          else{
51              names.unshift(propertys[name])
52          }
53        }
54      let items = expressions.slice().reverse();
55      items.forEach(function(item, index) {
56      propertys[item] = names[index];
57      });
58    >  params = {···
73      };
74      names.push(params.rightToLeft);
75      names.push(params.len);
76      names.unshift(params.rightToLeft);
77      expressions.unshift('"\u202e"');
78      decryption += format( "\nif (#{u202e} !== #{rightToLeft}) {\n  return;\n}\n     ", params);
```

Рис. 3.12. Реалізація роботи зі змінними та константами

У результаті таких перетворень отримано бажаний результат. Логіка алгоритму працює, як планувалось (рис. 3.13).

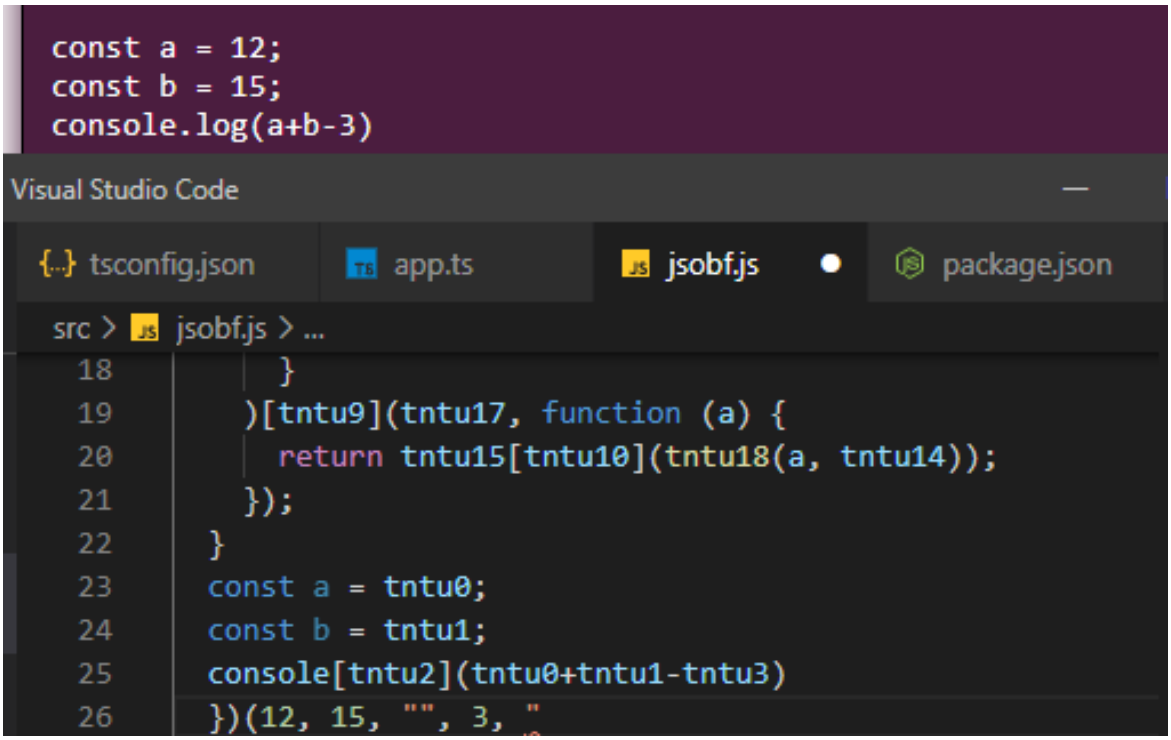


Рис. 3.13. Результат обфускації змінних та констант

Для тестування на вхід подано примітивну функцію, що виводить на консоль результат математичних дій із константами. Константами не скористалися, а у вираз підставлено ідентичне їм значення із геш-таблиці.

3.3.3 Алгоритм кодування чисел. Такий алгоритм дуже важливий для програмної системи, оскільки більша частина вихідного коду в будь-якій мові програмування користується величезною кількістю чисел для лічильників різного роду, для циклів і для числових констант.

Ідея такого алгоритму полягає в тому, щоб замінити числа деяким набором арифметичних виразів, і після цього представити числа в 16-ій системі числення. Детальний алгоритм наведено на рис. 3.14. Для поліпшення такого заплутування масив з усіма числами подаватиметься справа наліво у режимі crush разом з інвертуванням самого порядку подання числа, у режимі

reverse – випадково для кожного числа, а у режимі zero – у звичайному (зліва направо) порядку.

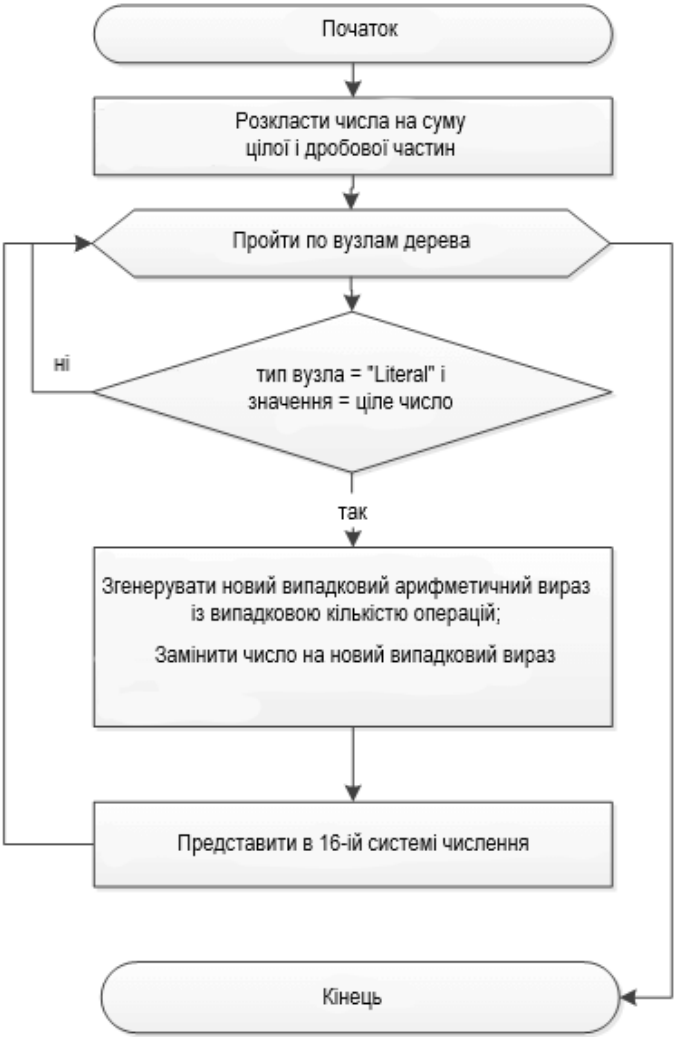


Рис. 3.14. Алгоритм кодування чисел

Очевидно, що зловмиснику знадобиться багато часу, щоб зрозуміти, за яким принципом відбуваються зміни з числами.

3.3.4 Алгоритм кодування стрічок. Такий алгоритм також важливий у роботі програмної системи. Майже у всіх програмах користуються будь-якими стрічковими константами, приміром, повідомленнями про помилки або текстові елементи інтерфейсу. Для, того щоб важче було розібрати вихідний код програми, необхідно перетворити і приховати явне використання таких стрічкових констант.

У даному перетворенні рядки подають у вигляді конкатенації викликів різних функцій, а також за допомогою власного алгоритму кодування та декодування символів (рис. 3.15), котрі здатні практично приховати від людини не тільки наявність стрічки, а й її вміст. Розбиття стрічкових констант та їх кодування відбувається кожного разу випадково, забезпечуючи різний вихідний код на виході обфускатора.

```
expressions = expressions.map(function(item) {
    if (/^"/.test(item)) {
      hasString = true;
      return JSON.stringify(JSON.parse(item).split('').reverse().join(''));
    }
    return item;
  });
params = {
    empty: identFrom(guid++),
    len: identFrom(guid++),
    reverse: identFrom(guid++),
    temp: identFrom(guid++),
    string: identFrom(guid++),
    split: identFrom(guid++),
    argv: identFrom(guid++),
    index: identFrom(guid++),
    join: identFrom(guid++),
    rightToLeft: identFrom(guid++),
    0: identFrom(guid++),
    1: identFrom(guid++),
    2: identFrom(guid++),
    u202e: '"\u202e"'
};
if (hasString) {
    names.push(params.string);
    expressions.push('"string"');
    names.push(params.split);
    expressions.push('"split"');
    names.push(params.reverse);
    expressions.push('"reverse"');
    names.push(params.join);
    expressions.push('"join"');
    decryption += format( "\nfor (#{index} = #{0}; #{index} < #{len}; #{index}++) {\n  if (typeof #{argv
  }
```

Рис. 3.15. Програмна реалізація обфускації стрічок

При застосуванні розроблених перетворень до стрічкових констант отримаємо надскладну форму представлення стрічки на виході обфускатора, котра не міститиме ні кирилиці, ні латиниці. Візуально у коді вона представлятиме невидимий набір символів. Приклад такого перетворення подано у загальному тестуванні алгоритму (рис. 3.18).

3.3.5 Алгоритм перейменування змінних. Таке перетворення вихідного коду найменш витратнее з точки зору ресурсів , однак одне з найбільш ефективних алгоритмів обфускації з точки зору кінцевого заплутування.

Головна особливість такого алгоритму в тому, щоб представити імена змінних і функцій у вигляді незрозумілих для людини наборів символів. Алгоритм детально розглянуто на рис. 3.16.

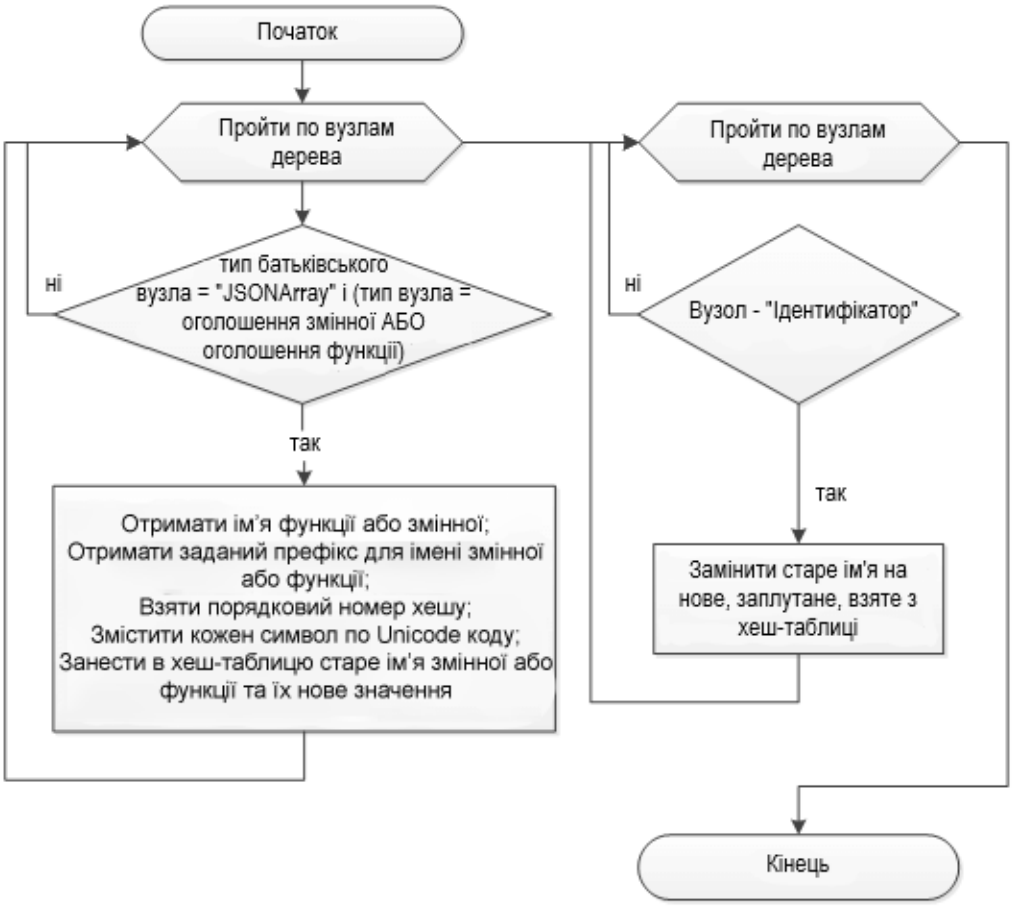


Рис. 3.16. Алгоритм перейменування змінних

У результаті алгоритму, імена усіх змінних та функцій матимуть вигляд наперед заданого префікса та порядкового номера із геш-таблиці.

Програмна реалізація має наступний вигляд:

```
let prefix = useOptions.prefix || 'tntu';
function identFrom(_index) {
  return prefix + _index;
}
```

3.4 Представлення та тестування розробленого обфускатора

Після розробки алгоритмів, їх реалізовано на мові JavaScipt та об'єднано у головний алгоритм обфускації з трьома різними режимами.

На рис. 3.17 показано веб-інтерфейс розробленого обфускатора. На ньому присутні текстові поля для вводу та виводу програмного коду, вибір можливих режимів обфускації, вибір файлу, а також кнопка старту обфускації.

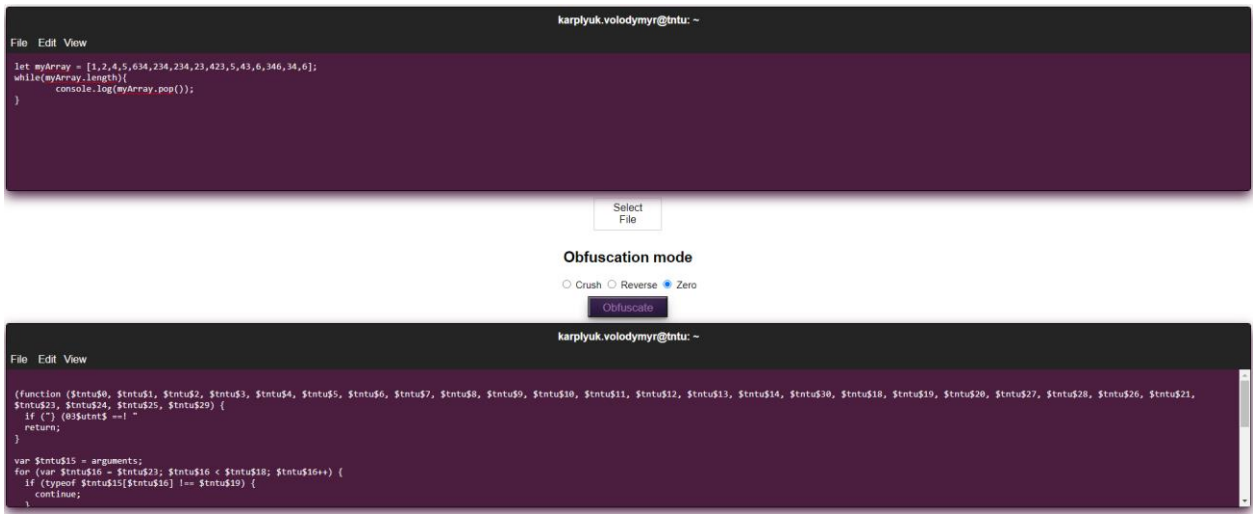


Рис. 3.17. Інтерфейс обфускатора

Як показало тестування, код набуває бажаних значних заплутуючих перетворень, при чому функціональність залишається ідентичною вихідному програмному продукту. Суб'єктивний висновок – складність розуміння вихідного коду збільшилася у 3 рази.

Приклад комплексного перетворення із застосуванням усіх алгоритмів, а також підтвердження збереження вихідної функціональності показано на рис. 3.18.

В ході розробки власного обфускатора для мови JavaScript проаналізовано існуючі обфускатори. Після їх детального вивчення, виявлено властиві їм недоліки, враховуючи які, вирішено розробити і реалізувати власні ефективні алгоритми обфускації.

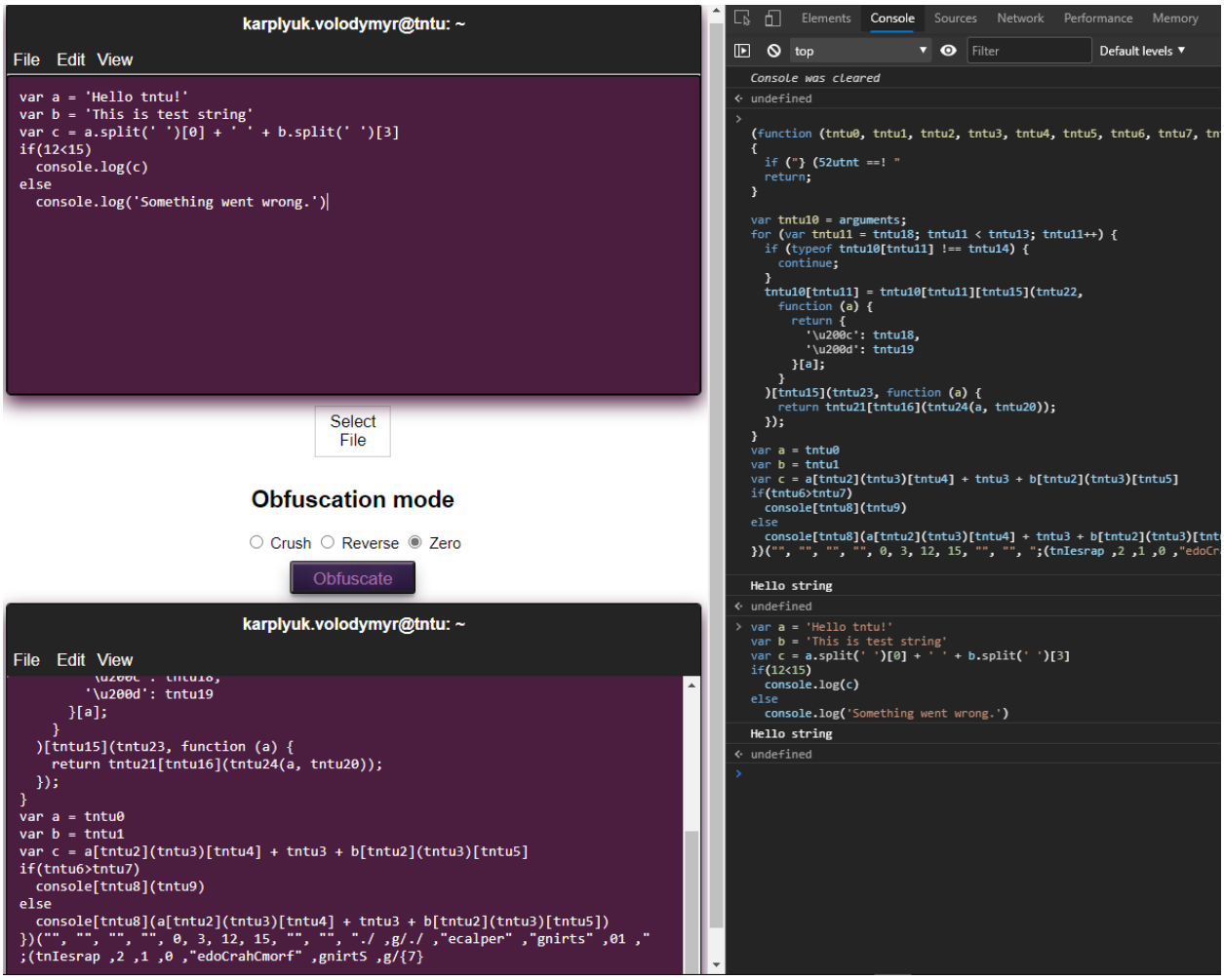


Рис. 3.18. Результат обфускації та виконання

Отже, розробка відповідає усім поставленим вимогам та перетворює код, зберігаючи бажану функціональність.

3.5 Обґрунтування доцільності розробки

Науковою новизною виступить розробка нового алгоритму заплутування програмного коду на мові програмування JavaScript. Розроблена методика вважатиметься оптимальною, оскільки при її написанні скористалися та перевірялися різноманітні методи обфускації у різних частках, а також з модифікованою логікою. Тому, розроблений алгоритм стійкий до методів автоматичної обфускації.

Технічна значимість такої розробки полягає в тому, що вона переведе захищеність інформації та вихідних кодів на вищий рівень, збільшить структурованість методів, а також методів захисту на базі обфускації.

Практична значимість даної роботи полягає в фактично миттєвому підвищенні рівня захисту інформації та програмних продуктів на базі розробленого обфускатора. Застосовуючи створені методики, можна значно мінімізувати зусилля та фінанси розробників, котрі займаються безпекою проєктів, замінити неефективні методи заплутування на нові та підняти на новий (більш складний) рівень мінімальні зусилля зловмисника для аналізу та деобфускації програмних продуктів. У геометричній прогресії знизяться затрати часу та людських ресурсів на захист необхідних ділянок коду чи інформації. Отриманими результатами можна скористатися майже всюди, починаючи від простих веб-проектів та сервісів, закінчуючи великими підприємствами та організаціями, де спостерігається проблема захисту програмних продуктів від нелегального поширення та різного роду модифікацій.

3.6 Висновки до розділу

Отже, отриманий код виявився стійким до відомих та популярних на сьогоднішній день автоматичних деобфускаторів. Швидкодія обфускатора залишилась практично на тому ж рівні, а з режимом crush у деяких випадках навіть краща, що може допомогти веб-проєктам, націленим на мобільних користувачів. Режими reverse та crush довзоляють заплутати найважливіші ділянки коду, оскільки додають пов’язаний надлишковий код, змінюють стрічки та виконують логічні перетворення, за можливості змінюють форму операторів, проте потребують більше обчислювальних ресурсів же для того ж функціоналу. Алгоритми заплутувань реалізовано у повному обсязі і вони не вносять змін у функціональні властивості вихідного продукту.

РОЗДІЛ 4
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

Охорона праці – система правових, соціально-економічних, організаційно-технічних, санітарно-гігієнічних і лікувально-профілактичних заходів та засобів, спрямованих на збереження життя, здоров'я і працездатності людини у процесі трудової діяльності.

Організація охорони праці на об’єктах промисловості цілісна система прав, обов’язків та повноважень суб’єктів виробничого процесу, процедур, спрямованих на дотримання безпечного рівня виробництва, правил та нормативних вимог, які регулюють питання найманої праці.

4.1 Охорона праці

Будь-яка взаємодія із розробленим програмним продуктом буде проводить за допомогою ЕОМ, згідно з ДСанПІН 3.3.2.007-98 (Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин) умови користування ЕОМ:

- площа на одне робоче місце має становити не менше ніж 6,0 м^2, а об'єм не менше ніж 20,0 м^3;
- приміщення для роботи з ВДТ (візуальними дисплейними терміналами) повинні мати природне та штучне освітлення відповідно до СНиП II-4-79;
- природне освітлення має здійснюватись через світлові прорізи, орієнтовані переважно на північ чи північний схід і забезпечувати коефіцієнт природною освітленості (КПО) не нижче ніж 1,5%. Розраховується КПО за методикою, викладеною в СНиП II-4-79;
- виробничі приміщення для роботи з ВДТ (операторські, диспетчерські) не повинні межувати з приміщеннями, в яких рівні шуму і вібрації перевищують допустимі значення (виробничі цехи, майстерні тощо) за СН 3223-85, СН 3044-84, ГР 2411-81;

- приміщення для роботи з ВДТ мають бути обладнані системами опалення, кондиціонування повітря, або припливно-витяжною вентиляцією відповідно до СНиП 2.04.05-91. Нормовані параметри мікроклімату, іонного складу повітря, вмісту шкідливих речовин мають відповідати вимогам СН 4088-86, СН 2152-80;

- виробничі приміщення можуть обладнуватись шафами для зберігання документів, магнітних дисків, полицями, стелажами, тумбами тощо з урахуванням вимог до площі приміщень;

- конструкція робочого місця користувача ЕОМ і ПЕОМ з ВДТ має забезпечити підтримання оптимальної робочої пози;

- робочі місця з ВДТ слід так розташовувати відносно світових прорізів, щоб природнє світло падало збоку переважно зліва;

- при розміщенні робочих столів з ВДТ слід дотримувати такі відстані між бічними поверхнями ВДТ 1,2 м, відстань від тильної поверхні одного ВДТ до екрана іншого ВДТ 2,5 м;

- конструкція робочого столу має відповідати сучасним вимогам ергономіки і забезпечувати оптимальне розміщення на робочій поверхні використовуваного обладнання (дисплея, клавіатури, принтера) і документів;

- висота робочої поверхні робочого столу з ВДТ має регулюватися в межах 680...800 мм, а ширина і глибина забезпечувати можливість виконання операцій у зоні досяжності моторного поля (рекомендовані розміри: 600...1400 мм, глибина 800..1000 мм);

- робочий стіл повинен мати простір для ніг заввишки не менше ніж 600 мм, завширшки не менше ніж 500 мм, завглибшки (на рівні колін) не менше ніж 450 мм, на рівні простягнутої ноги від 650 мм;

- робочий стілець має бути підйомно-поворотним, регульованим за висотою, з кутом і нахилу сидіння та спинки і за відстанню від спинки до переднього краю сидіння поверхня сидіння має бути плоскою, передній край заокругленим. Регулювання за кожним із параметрів має здійснюватися незалежно, легко і надійно фіксуватися. Шаг регулювання елементів стільця

має становити: для лінійних розмірів -15...20 мм, для кутових 2...5 град. Зусилля регулювання має не перевищувати 20 Н;

- висота поверхні сидіння має регулюватися в межах 400...500 мм, а ширина і глибина становити не менше ніж 400 мм. Кут нахилу сидіння до 15 град. вперед і до 5 град. назад;
- висота спинки стільця має становити (300+-20) мм, ширина не менше ніж 380 мм, радіус кривизни горизонтальної площини 400 мм. Кут нахилу спинки має регулюватися в межах 1...30 град. від вертикального положення. Відстань від спинки до переднього краю сидіння має регулюватися в межах 260...400 мм;
- для зниження статичного напруження м'язів верхніх кінцівок слід використовувати стаціонарні або змінні підлокітники завдовжки не менше ніж 250 мм, завширшки 50...70 мм, що регулюються за висотою над сидінням у межах 230...260 мм і відстанню між підлокітниками в межах 350...500 мм;
- поверхність сидіння і спинки стільця має бути напівм'якою з нековзним, повітронепроникним покриттям, що легко чиститься і не електризується;
- робоче місце має бути обладнане підставкою для ніг завширшки не менше ніж 300 мм, завглибшки не менше ніж 400 мм, що регулюється за висотою в межах до 150 мм і за кутом нахилу опорної поверхні підставки до 20 град. Підставка повинна мати рифлену поверхню і бортик по передньому краю заввишки 10 мм;
- екран ВДТ має розташовуватися на оптимальній відстані від очей користувача, що становить 600...700 мм, але не ближче ніж за 600 мм з урахуванням розміру літерно-цифрових знаків і символів;
- розташування екрана ВДТ має забезпечувати зручність зорового спостереження у вертикальній площині під кутом +30 град. до нормальної лінії погляду працюючого;
- клавіатуру слід розташовувати на поверхні столу на відстані 100...300 мм від краю, звернутого до працюючого. У конструкції клавіатури має

передбачатися опорний пристрій (виготовлений із матеріалу з високим коефіцієнтом тертя, що перешкоджає мимовільному її зсуву), який дає змогу змінювати кут нахилу поверхні клавіатури у межах 5...15 град;

- висота середнього рядка клавіш має не перевищувати 30 мм. Поверхня клавіатури має бути матовою з коефіцієнтом відбиття 0,4;
- розташування пристрою введення виведення інформації має забезпечувати добру видимість екрана ВДТ, зручність ручного керування в зоні досяжності моторного поля і за висотою 900...1300 мм, за шириною 400...500 мм;
- для забезпечення захисту і досягнення нормованих рівнів комп'ютерного випромінювання необхідно застосовувати екранні фільтри, локальні світлофільтри (засобів індивідуального захисту очей) та інші засоби захисту, що пройшли випробування в акредитованих лабораторіях і мають щорічний гігієнічний сертифікат;
- при оснащеності робочого місця з ВДТ лазерним принтером параметри лазерного випромінювання повинні відповідати вимогам СанПиН N 5804-91.

4.2 Організація цивільного захисту на об'єктах промисловості та виконання заходів щодо запобігання виникненню надзвичайних ситуацій техногенного походження

Цивільний захист об´єкта - система організаційних, інженерно-технічних, санітарно- гігієнічних, протиепідеміологічних та інших заходів, що здійснюються керівництвом об'єкта господарської діяльності з метою запобігання та ліквідації надзвичайних ситуацій, які загрожують життю та здоров'ю людей у мирний та воєнний час. Цивільний захист об´єкта здійснюється відповідно до Конституції України, з вимогами Кодексу Цивільний захисту України, забезпечується з урахуванням особливостей, визначених Законом України «Про основи національної безпеки України», іншими законами та нормативно – правовими актами КМ України, Державної

служби надзвичайних ситуацій (ДСНС), територіальних управлінь надзвичайних ситуацій, наказами та розпорядженнями керівника об´єкта.

Підприємства для захисту від надзвичайних ситуацій техногенного та природного характеру:

- планують і здійснюють заходи для захисту працівників, об'єктів господарювання та довкілля;
- підтримують у готовності до застосування сили й засоби із запобігання надзвичайним ситуаціям та ліквідації їх наслідків;
- створюють та підтримують матеріальні резерви для запобігання і ліквідації надзвичайних ситуацій;
- забезпечують, щоб працівникам своєчасно повідомляли про загрозу або виникнення надзвичайних ситуацій.

Начальник ЦЗ повинен вживати організаційних, інженерно-технічних, санітарно-гігієнічних та інших заходів. У начальника ЦЗ підприємства мають бути такі документи:

- функціональні обов'язки керівного складу органів управління, начальників служб і командирів формувань;
- календарний план основних заходів ЦЗ;
- план реагування на надзвичайні ситуації;
- інструкція щодо дій персоналу суб'єкта господарювання у разі загрози або виникнення надзвичайних ситуацій;
- проєкти наказів — у разі ймовірних надзвичайних ситуацій;
- схема (план) підприємства;
- список телефонів і позивних посадових осіб;
- схема оповіщення і зв'язку;
- довідкові дані основних показників ЦЗ підприємства;
- робочий зошит;
- інші документи залежно від характеристик об'єкта та найімовірніших надзвичайних ситуацій.

Відповідно до планів реагування завдання у разі надзвичайних ситуацій об’єктового рівня реалізують, як правило, сили ЦЗ підприємства.

На об’єктах підвищеної небезпеки створюють автоматизовані системи раннього виявлення загрози надзвичайних ситуацій та оповіщення про їх виникнення працівників і населення, яке потрапляє до зони можливого ураження.

Виконання заходів щодо запобігання виникненню надзвичайних ситуацій техногенного походження:

1. Забезпечення техногенної безпеки на об’єктах здійснюється на випадок:

- наявності будівель та споруд з порушенням умов експлуатації;
- наявності об’єктів з критичним станом виробничих фондів та порушенням умов експлуатації;
- можливості впливу сторонніх (зовнішніх) факторів (природних, терористичних, соціальних тощо) на діяльність та безпеку об’єкта;
- виникнення надзвичайних ситуацій техногенного характеру (порушення умов експлуатації) на небезпечних об’єктах, ядерних установках.

2. Суб’єкти господарювання у випадках, визначених Кодексом, забезпечують техногенну безпеку шляхом:

- виконання вимог Кодексу, цих Правил, норм і стандартів стосовно забезпечення техногенної безпеки, а також приписів, розпоряджень і постанов, що відповідно до законодавства видаються посадовими особами центрального органу виконавчої влади, що реалізує державну політику у сфері цивільного захисту;
- розроблення організаційно-розпорядчих документів щодо забезпечення техногенної безпеки, здійснення постійного контролю за їх дотриманням;
- забезпечення відповідно до законодавства своїх працівників засобами колективного та індивідуального захисту;
- навчання працівників діям у надзвичайних ситуаціях;

- організації розроблення інженерно-технічних заходів цивільного захисту під час будівництва (реконструкції, технічного переоснащення) на об’єктах, включених до переліку об’єктів, що належать суб’єктам господарювання, проектування яких здійснюється з урахуванням вимог інженерно-технічних заходів цивільного захисту, затвердженого постановою Кабінету Міністрів України від 09 січня 2014 року № 6;

- навчання працівників порядку укриття в захисних спорудах цивільного захисту та навчання персоналу з обслуговування захисних споруд цивільного захисту їх утриманню відповідно до Вимог з питань використання та обліку фонду захисних споруд цивільного захисту, затверджених наказом Міністерства внутрішніх справ України від 09 липня 2018 року № 579, зареєстрованих у Міністерстві юстиції України 30 липня 2018 року за № 879/32331;

- проведення об’єктових тренувань і навчань з питань цивільного захисту з урахуванням вимог техногенної безпеки.

3. Забезпечення техногенної безпеки на небезпечних об’єктах у випадках, визначених Кодексом, здійснюється шляхом:

- оцінки ризиків виникнення надзвичайних ситуацій техногенного характеру на підпорядкованих небезпечних об’єктах відповідної галузі;

- інформування органів влади, сил цивільного захисту про основні загрози на небезпечних об’єктах з метою вжиття ними ефективних заходів захисту населення, промислових і сільськогосподарських об’єктів від надзвичайних ситуацій техногенного характеру;

- інформування органів управління та сил цивільного захисту про аварійні та небезпечні ситуації, розвиток яких призвів або міг призвести до аварії і завдати шкоди життю та здоров’ю населення і навколишньому середовищу;

- розміщення інформації про заходи безпеки та поведінку населення на випадок виникнення надзвичайної ситуації техногенного характеру на офіційних веб-сайтах, інформаційних стендах та в засобах масової інформації;

- організації заходів із захисту своїх працівників від шкідливого впливу надзвичайних ситуацій техногенного характеру;
- утворення об’єктових формувань та спеціалізованих служб цивільного захисту, створення необхідної для їх функціонування матеріально-технічної бази, забезпечення готовності таких формувань до дій за призначенням;
- здійснення навчання працівників правилам техногенної безпеки;
- підтримання готовності створених диспетчерських служб;
- розроблення аварійних планів об’єктів, де здійснюється практична діяльність, пов’язана з радіаційними або радіаційно-ядерними технологіями;
- розроблення планів локалізації та ліквідації аварій;
- декларування безпеки об’єктів підвищеної небезпеки;
- створення, експлуатації і технічного обслуговування автоматизованих систем раннього виявлення загрози виникнення надзвичайних ситуацій та оповіщення населення у разі їх виникнення;
- створення та утримання в робочому стані засобів зв’язку, аварійно-рятувальної техніки та обладнання і використання їх за призначенням;
- аварійно-рятувального обслуговування небезпечних об’єктів;
- створення об’єктового матеріального резерву для запобігання і ліквідації наслідків надзвичайних ситуацій та проведення невідкладних відновлювальних робіт;
- фінансування витрат у порядку та обсягах, необхідних для повного і якісного забезпечення вимог техногенної безпеки;
- розроблення заходів щодо забезпечення техногенної безпеки з урахуванням досягнень науки і техніки, позитивного досвіду із зазначеного питання;
- виконання інших завдань і заходів техногенної безпеки з урахуванням вимог чинного законодавства України.

4. На території небезпечних об’єктів, які розташовані в міській зоні, допускається утримувати небезпечні хімічні речовини (далі - НХР) в обсягах,

передбачених проектними нормами зберігання (накопичення) та технологічними регламентами.

5. На майданчиках (у будівлях) небезпечних об’єктів, розташованих у заміській зоні, кількість НХР не може перевищувати 2 тис. тонн у разі зберігання на одному майданчику (у будівлі), якщо ці майданчики (будівлі) розташовані на відстані не менше ніж 500 м один від одного.

Загальна кількість НХР на всіх майданчиках (у будівлях) небезпечних об’єктів, розташованих на відстані менше ніж 500 метрів один від одного, не може перевищувати 2 тис. тонн.

6. Запаси НХР на базових складах у заміських зонах не повинні перевищувати норм, установлених для таких складів.

7. Керівники об’єктів, що за характером своєї діяльності не належать до небезпечних об’єктів, але за прогнозом можуть опинитись у зонах надзвичайних ситуацій, повинні:

- ураховувати можливу небезпеку, що може виникнути на їх територіях у разі виникнення надзвичайних ситуацій техногенного характеру на небезпечних об’єктах і територіях, та вживати заходів щодо забезпечення безпеки працівників об’єктів з урахуванням Кодексу та цих Правил;
- взаємодіяти з керівництвом небезпечних об’єктів і відповідними органами влади для отримання інформації та оповіщення про небезпеку, що може впливати на діяльність об’єкта;
- організовувати навчання працівників діям у надзвичайних ситуаціях техногенного характеру.

Висновки: в цьому пункті проаналізовано основні питання охорони праці та безпеки охорони праці в надзвичайних ситуаціях, висвітлено основні аспекти та питання організації цивільного захисту на об’єктах промисловості та виконання заходів щодо запобігання виникненню надзвичайних ситуацій техногенного походження, які загрожують життю та здоров'ю людей у мирний та воєнний час.

ВИСНОВКИ

Отже, під час розробки нового алгоритму заплутування програмного коду на мові програмування JavaScript досліджено та встановлено оптимальний баланс між рівнем обфускації та необхідною продуктивністю для різних видів проектів.

У результаті обробки вихідного коду запропонованим обфускатором на виході отримуємо безпечний, продуктивний та захищений від аналізу зі сторони зловмисників функціонально ідентичний код.

Досягнуто основної мети, а саме підвищено рівень захищеності інформації та вихідного коду програмних продуктів на мові JavaScript за допомогою власних алгоритмів обфускації.

Розроблену методику обфускації можна вважати оптимальною, оскільки при її створенні застосовано, перевірено та практично вдосконалено різноманітні методи обфускації, що забезпечило стійкість власного алгоритму до методів автоматичної деобфускації.

Запропонований заплутувач коду дозволить досягнути вищого рівня захищеності інформації та вихідних кодів, мінімізуючи зусилля та фінанси розробників ПЗ, котрі працюють над безпекою коду.

СПИСОК ЛІТЕРАТУРИ

1. Collberg C. Taxonomy of Obfuscating Transformations / C. Collberg, C. Thomborson, D. Low. // Technical Report 148: Univ. of Auckland. – 1997.

2. Collberg C. A taxonomy of obfuscating transformations / C. Collberg, C. Thomborson, D. Low. – Department of Computer Science, The University of Auckland, New Zealand, 1997. – 36 p.

3. Ilsun Y. Malware obfuscation techniques: A brief survey / Y. Ilsun, Y. Kangbin. – Broadband: Wireless Computing, Communication and Applications (BWCCA), 2010. – (Department of Computer Science, The University of Auckland).

4. Schrittwieser S. Protecting software through obfuscation: can it keep pace with progress in code analysis [Електронний ресурс] / S. Schrittwieser, S. Katzenbeisser, J. Kinder // ACM Comput Surv. – 2016. – Режим доступу до ресурсу: https://doi.org/10.1145/2886012.

5. Modern obfuscation methods for secure coding / [I. Stepanenko, V. Kinzeryavyy, A. Nagi та ін.]. // Ukrainian Scientific Journal of Information Security. – 2016. – №22. – P. 32–37.

6. Xu H. Layered obfuscation: a taxonomy of software obfuscation techniques for layered security [Електронний ресурс] / H. Xu, Y. Zhou, J. Ming // Cybersecur 3. – 2020. – Режим доступу до ресурсу: https://doi.org/10.1186/s42400-020-00049-3.

7. Cappaert J. A general model for hiding control flow [Електронний ресурс] / J. Cappaert, B. Preneel // ACM Workshop on Digital Rights Management. – 2010. – Режим доступу до ресурсу: https://doi.org/10.1145/1866870.1866877.

8. Barak B. Hopes, fears, and software obfuscation [Електронний ресурс] / B. Barak // Commun ACM. – 2016. – Режим доступу до ресурсу: https://doi.org/10.1145/2757276.

9. Barak B. On the (im) possibility of obfuscating programs [Електронний ресурс] / B. Barak, O. Goldreich, R. Impagliazzo // Annual International Cryptology

Conference. – 2001. – Режим доступу до ресурсу: https://doi.org/10.1007/3-540-44647-8_1.

10. Barrantes E. G. Randomized instruction set emulation [Електронний ресурс] / E. G. Barrantes, D. H. Ackley, S. Forrest // ACM Trans Inf Syst Secur. – 2005. – Режим доступу до ресурсу: https://doi.org/10.1145/1053283.1053286.

11. Barrington D. A. Bounded-width polynomial-size branching programs recognize exactly those languages [Електронний ресурс] / D. A. Barrington // NC1 In: STOC. – 1986. – Режим доступу до ресурсу: https://doi.org/10.1145/12130.12131.

12. Варновский Н. П. О применении методов деобфускации программ для обнаружения сложных компьютерных вирусов / Н. П. Варновский, В. А. Захаров, Н. Н. Кузюрин. // Известия Таганрогского радиотехнического университета. – 2006. – №6. – С. 18–20.

13. Bichsel B. Statistical deobfuscation of android applications [Електронний ресурс] / B. Bichsel, V. Raychev, P. Tsankov // CCS. – 2016. – Режим доступу до ресурсу: https://doi.org/10.1145/2976749.2978422.

14. Cheng X. Dynopvm: Vm-based software obfuscation with dynamic opcode mapping [Електронний ресурс] / X. Cheng, Y. Lin, D. Gao // International Conference on Applied Cryptography and Network Security. – 2019. – Режим доступу до ресурсу: https://doi.org/10.1007/978-3-030-21568-2_8.

15. Collberg C. Toward digital asset protection [Електронний ресурс] / C. Collberg, J. Davidson, R. Giacobazzi // IEEE Intell. – 2011. – Режим доступу до ресурсу: https://doi.org/10.1109/mis.2011.106.

16. Crane S. Thwarting cache side-channel attacks through dynamic software diversity [Електронний ресурс] / S. Crane, A. Homescu, S. Brunthaler // NDSS. – 2015. – Режим доступу до ресурсу: https://doi.org/10.14722/ndss.2015.23264.

17. Crane S. J. It’s a TRaP: table randomization and protection against function-reuse attacks [Електронний ресурс] / S. J. Crane, S. Volckaert, F. Schuster // CCS. – 2015. – Режим доступу до ресурсу: https://doi.org/10.1145/2810103.2813682.

18. Kovacheva A. Efficient code obfuscation for android [Електронний ресурс] / A. Kovacheva // International Conference on Advances in Information Technology. – 2013. – Режим доступу до ресурсу: https://doi.org/10.1007/978-3-319-03783-7_10.

19. Kuang K. Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling [Електронний ресурс] / K. Kuang, Z. Tang, X. Gong // Comput Secur. – 2018. – Режим доступу до ресурсу: https://doi.org/10.1016/j.cose.2018.01.008.

20. Larsen P. Sok: Automated software diversity [Електронний ресурс] / P. Larsen, A. Homescu, S. Brunthaler // IEEE Symposium on Security and Privacy. – 2014. – Режим доступу до ресурсу: https://doi.org/10.1109/sp.2014.25.

21. Варновский Н. П. Современное состояние исследований в области обфускации программ: определения стойкости обфускации / Н. П. Варновский, В. А. Захаров, Н. Н. Кузюрин. // Труды Института системного программирования РАН. – 2013. – №26. – С. 167–198.

22. Madou M. On the effectiveness of source code transformations for binary obfuscation / M. Madou, B. Anckaert, B. de Sutter. // Proceedings of the International Conference on Software Engineering Research and Practice. – 2006.

23. Madou M. Software protection through dynamic code mutation / M. Madou, B. Anckaert, P. Moseley. // Proceedings of the 6-th international conference on Information Security Applications. – 2006. – P. 194–206.

24. Della Preda M. Semantic-based code obfuscation by abstract interpretation / M. Della Preda, G. Giacobazzi. // Journal of Computer Security. – 2009. – №17. – P. 855–908.

25. Della Preda M. Modelling Metamorphism by Abstract Interpretation / M. Della Preda, G. Giacobazzi, S. Debray. // Proceedings of the 17th International Static Analysis Symposium (SAS'10). Lecture Notes in Computer Science. – 2010. – №6337. – P. 218–235.

26. Карплюк В. Захист програмного забезпечення на апаратному та програмному рівнях / В. Карплюк, О. Ясній // Матеріали VIII науково-технічної конференції ТНТУ ім. І. Пулюя, 9-10 грудня 2020 року — Т. : ТНТУ, 2020 — С. 105. — (Секція: Комп'ютерні системи та мережі).

27. Карплюк В. Методи обфускації програмного коду в комп'ютерних системах / В. Карплюк, О. Ясній // Матеріали IX наукової конференції ТНТУ ім. І. Пулюя, 25-26 листопада 2020 року — Т. : ТНТУ, 2020 — С. 77. — (Секція: Комп'ютерні системи та мережі).

28. Курмангалеев Ш. Ф. Описание подхода к разработке обфусцирующего компилятора / Ш. Ф. Курмангалеев, В. П. Корчагин, Р. А. Матевосян. // Труды Института системного программирования РАН. – 2012. – №23. – С. 67–76.

29. Методичні вказівки до виконання підрозділу "Охорона праці" в кваліфікаційних роботах магістрів спеціальності 123 «Комп'ютерна інженерія» /Укл.: Осухівська Г.М. – Тернопіль: ТНТУ імені Івана Пулюя, 2020. – 22 с.

30. Тишко Н.І. Розробка інтерактивного середовища соціалізації мовою програмування Java Script, фреймворк Meteor JS : автореферат дипломної роботи магістра за спеціальністю «121 — інженерія програмного забезпечення»/ Н.І. Тишко. — Тернопіль: ТНТУ, 2019.

31. Радчук М. І. Розробка веб-застосунку для візуалізації та аналізу мережі друзів за допомогою графів : дипломна робота магістра за спеціальністю «122 — комп'ютерні науки» / М. І. Радчук. — Тернопіль: ТНТУ, 2020. — 146 с.

32. Радчук В. Огляд методу обфускації коду, як протидія дизасемблюванню / В. Радчук, Н. Шингера // Матеріали XVIII наукової конференції ТНТУ ім. І. Пулюя, 29-30 жовтня 2014 року — Т. : ТНТУ, 2014 — С. 65-66. — (Секція: Інформаційні технології).

Додаток А. Тези конференцій

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Тернопільський національний технічний університет імені Івана Пулюя (Україна)
Національна академія наук України
Університет імені П'єра і Марії Кюрі (Франція)
Маріборский університет (Словения)
Технічний університет у Кошице (Словаччина)
Вільнюський технічний університет ім. Гедимінаса (Литва)
Шяуляйська державна колегія (Литва)
Жешувський політехнічний університет ім. Лукасевича (Польща)
Білоруський національний технічний університет (Республіка Білорусь)
Міжнародний університет цивільної авіації (Марокко)
Національний університет біоресурсів і природокористування України (Україна)
Наукове товариство ім. Шевченка
ГО «Асоціація випускників Тернопільського національного технічного університету імені Івана Пулюя»

АКТУАЛЬНІ ЗАДАЧІ СУЧАСНИХ ТЕХНОЛОГІЙ

Збірник

тез доповідей

Том ІІ

ІХ Міжнародної науково-технічної конференції молодих учених та студентів

25-26 листопада 2020 року

УКРАЇНА
ТЕРНОПІЛЬ – 2020

УДК 004.4
О.П. Ясній, докт. техн. наук, проф., В.І. Карплюк
Тернопільський національний технічний університет імені Івана Пулюя, Україна

МЕТОДИ ОБФУСКАЦІЇ ПРОГРАМНОГО КОДУ В КОМП'ЮТЕРНИХ СИСТЕМАХ

O.P. Yasniy, Dr. Sc., Prof., V.I. Karplyuk

OBFUSCATION METHODS OF SOFTWARE CODE PROTECTION IN COMPUTER SYSTEMS

Програміст витрачає багато зусиль, налагоджуючи код, однак то тільки мала частина роботи. Після того, як проект створено, налагоджено та розгорнуто, потрібно запобігти спробам сторонньої особи скопіювати вихідний код і скористатися ним.

Сьогодні зловмисники мають велику кількість програм для злому – від простих шкідливих програм до складних інструментів реверсної інженерії. Дизасемблери, декомпілятори та інші інструменти дозволяють хакерам отримувати доступ та аналізувати вихідний код програми. Очевидно, що за допомогою такої інформації хакери можуть зловживати програмним забезпеченням різними способами: витягувати конфіденційну інформацію, додавати шкідливий код, наносити різного роду збитки клієнтам або ж цілим проектам, клонувати програмні продукти.

У результаті обфускації вихідний код навмисно ускладнюють для того, щоб запобігти реверсній інженерії. При цьому функціональність обфускованого коду еквівалентна вихідному.

Мета обфускації – заплутати програмний код і усунути більшість логічних зв'язків у ньому, тобто перетворити код так, щоб його було важко вивчити і модифікувати стороннім особам. Обфускацію здійснюють наступними методами:

- лексична обфускація (перейменування імен змінних та методів);
- обфускація даних (маскування структур даних під такі, якими вони не є);
- обфускація графа потоку керування (заміна виконуваної логіки недетермінованою та додавання випадкового зайвого заплутаного коду).

Проаналізувавши існуючі методи обфускації програмного коду, написано удосконалений обфускатор для мови JavaScript, котру застосовують у веб-проектах.

Розроблено три унікальних режими роботи обфускатора, кожен із яких обфускує код у різних частках того чи іншого етапу:

- зміна розміру графу потоку керування функції (клонування базових блоків, зміна структури циклів);
- руйнування вихідної структури графу потоку керування функції (додавання до логіки зайвих зв’язків, розбиття блоків на менші, створення нових блоків для наступного етапу);
- генерація додаткового коду (заповнення новоутворених порожніх блоків інструкціями, що ніяк не впливають на виконання основної логіки, додавання мертвого коду в інші блоки програми);
- поєднання мертвого коду із основною програмою за допомогою зайвих логічних зв’язків та математичних тотожностей і нерівностей.

Обфускувавши важливі частини вихідного коду запропонованим інструментом, отриманий код стає значно складнішим для реверсної інженерії. Зловмиснику знадобиться набагато більше часу, щоб зрозуміти, що саме замасковано у коді.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ПУЛЮЯ

М А Т Е Р І А Л И

VIII НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ

«ІНФОРМАЦІЙНІ МОДЕЛІ, СИСТЕМИ ТА ТЕХНОЛОГІЇ»

9–10 грудня 2020 року

ТЕРНОПІЛЬ
2020

УДК 004.4
О.П. Ясній., докт. техн. наук, проф., В.І. Карплюк
(Тернопільський національний технічний університет імені Івана Пулюя)

ЗАХИСТ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА АПАРАТНОМУ ТА ПРОГРАМНОМУ РІВНЯХ

UDC 004.4
O.P. Yasniy, Dr. Sc., Prof., V.I. Karplyuk

SOFTWARE PROTECTION AT HARDWARE AND SOFTWARE LEVELS

Перевагами захисту на програмному рівні є: гнучкість, порівняно низька вартість, сумісність із уже готовими системами. Однак великий недолік такого підходу – обмежена міцність. Із вдосконаленням існуючих та розвитком нових потужних методів, котрі дозволяють обійти безпеку додатків, фахівцям доводиться винаходити нові методики захисту своїх програмних продуктів. Враховуючи збільшення об'ємів віртуальної та накопичувальної пам'яті, а також беззаперечного приросту потужності процесорів, розробки методів захисту зі сторони програмістів та атак зі сторони реверс інженерів значно пришвидшилися.

Локальні конфіденційні дані, а також програмне забезпечення можна захистити за допомогою зашифрованої аутентифікації. При кожному запуску додатку здійснюватиметься захищена ідентифікація користувача, а усі необхідні для поточної робочої сесії блоки коду розшифровуватимуться у режимі реального часу обчислювальними ресурсами кінцевого користувача. Однак така методика захисту зможе гарантувати абсолютну безпеку лише в тому випадку, коли процеси шифрування та дешифрування виконуватимуться на окремому криптографічному співпроцесорі або спеціальному зовнішньому обладнанні. У такому випадку зловмисник не зможе отримати унікальний алгоритм шифрування та витягнути необхідний йому ключ дешифрування, оскільки доступними будуть тільки передбачені функції вводу та результати виводу (в якості реакції програмного продукту). Очевидно, що далеко не всі кінцеві клієнти зможуть забезпечити своє середовище такими апаратними модулями, тому ефективність такої методики низька. Аутентифікація стикається з подібними проблемами захисту.

Апаратними засобами, котрі дозволяють розв'язати таких задачі, можуть бути смарт-картки, які відіграють роль носія ключа доступу до продукту, а також виконуватимуть процеси шифрування/дешифрування. Проте такий підхід також не є популярним, оскільки такі апаратні додатки часто зазнають фізичних пошкоджень і потребують заміни зі сторони виробника. Також трапляються зловживання, пов'язані із вигаданим пошкодженням/втратою таких засобів, а для їх заміни виробнику потрібно виготовити та передати новий.

Отже, недоліками апаратних методів захисту є: неминуче підвищення вартості, несумісність із певними видами уже готових систем, модернізація, складність передачі, технічне обслуговування. Якщо хоча б один з апаратних засобів зламають зловмисники, заміні підлягатимуть усі, а саме тому гнучкість апаратних пристроїв низька у порівнянні з програмними підходами. Тому програмні механізми захисту можна доповнити апаратно та платформо незалежними методами обфускації, що забезпечить їх гнучкість та результативність.