Міністерство освіти і науки України
Державний вищий навчальний заклад
«Національний гірничий університет»

Інститут електроенергетики
Факультет інформаційних технологій
Кафедра безпеки інформації та телекомунікацій

ЗАТВЕРДЖЕНО:
завідувач кафедри
безпеки інформації та телекомунікацій
_____ Бабенко Т.В.

«_____»_____20___року

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

спеціальності *125 Кібербезпека*
(код і назва спеціальності)

студенту **125м-16-1** ***Повіренний Юрій Станіславович***
(група) (прізвище ім'я по-батькові)

Тема дипломної роботи ***Методика захисту критично важливого програмного коду від дослідження***

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора Державного ВНЗ «НГУ» від _____№_____

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень ***методи захисту програмного забезпечення від аналізу***

Предмет досліджень ***методика обфускації критично важливого коду програми***

Мета НДР ***підвищення якості захисту програмних продуктів від дослідження та модифікації***

Вихідні дані для проведення роботи ***Існуючі методи обфускації програм***

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна ***дослідження визначається тим, що в ній запропоновано метод обфускації коду з шифруванням адресу масива невизначених покажчиків з їх подальшою розшифровкою при обчисленні окремих виразів критичного коду програми.***

Практична цінність ***роботи полягає в розробці методики, алгоритму і програми для його реалізації, які дозволяють ефективно вирішити завдання захисту програмного забезпечення від аналізу***

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати здійснених досліджень можуть бути використані при розробці комерційних програмних продуктів

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок-кінець)**
Огляд джерел за темою та напрямом досліджень	18.09.17-06.10.17
Методи досліджень	09.10.17-21.11.17
Результати досліджень	22.11.17-15.12.17
Виконання економічного розділу	18.12.17-28.12.17
Оформлення пояснювальної записки	29.12.17-15.01.18

6 РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект ***економія досягається завдяки розробці методу обфускації, а не придбанню готової програмної продукції***

Соціальний ефект _____

7 ДОДАТКОВІ ВИМОГИ

Завдання видав _____ Школа Н.І.
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ Повіренний Ю.С.
(підпис) (прізвище, ініціали)

Дата видачі завдання: 01.09.17р.
Термін подання дипломної роботи до ДЕК 16.01.18

РЕФЕРАТ

Пояснювальна записка: __с., __ рис., __табл., __ додатків, __ джерел.

Об’єкт дослідження: методи захисту програмного забезпечення від аналізу і злому.

Мета роботи: підвищення якості захисту програмних продуктів від дослідження і модифікації.

Методи дослідження: методи теорії компіляторів, поняття і методи об'єктно-орієнтованого програмування і криптографії.

У спеціальній частині запропонований метод обфускації коду з шифруванням адрес масиву невизначених покажчиків з їх подальшою розшифровкою при обчисленні окремих виразів критичного коду програми. Даний метод посилює заплутуючий ефект за рахунок спільного застосування раніше відомих і окремо використаних способів обфускації.

В економічному розділі проаналізовані витрати на захист програм методом обфускації при використанні існуючих систем обфускації і при використанні запропонованого методу обфускації.

Практичне значення роботи полягає в розробці методики, алгоритму і програми для його реалізації, які дозволяють ефективно вирішувати завдання захисту програмного забезпечення від аналізу.

Результати здійснених у дипломній роботі досліджень можуть бути використані при розробці комерційних програмних продуктів.

Наукова новизна дослідження визначається тим, що в ній запропоновано метод обфускації коду з шифруванням адресу масива невизначених покажчиків з їх подальшою розшифровкою при обчисленні окремих виразів критичного коду програми.

ОБФУСКАЦІЯ, ЗАПУТУВАННЯ КОДУ, ЗЛОМ ПРОГРАМ, ДЕЗАССЕМБЛІРОВАННЯ

РЕФЕРАТ

Объяснительная записка: __С., _ рис., __табл., __ приложений,__ лит.

Объект исследования: методы защиты программного обеспечения от анализа и взлома.

Цель работи: повышение качества защиты программных продуктов от исследования и модификации.

Методы исследования: методы теории компиляторов, понятия и методы объектно-ориентированного программирования и криптографии.

В специальной части предложен метод обфускации кода с шифрованием адресов массива неопределенных указателей с их последующей расшифровкой при вычислении отдельных выражений критического кода программы. Данный метод усиливает запутывающий эффект за счет совместного применения ранее известных и раздельно используемых способов обфускации.

В экономическом разделе проанализированы затраты на защиту программ методом обфускации при использовании существующих систем обфускации и при использовании предложенного метода обфускации .

Практическое значение работы состоит в разработке методики, алгоритма и программы для его реализации, которые позволяют эффективно решать задачи защиты программного обеспечения от анализа.

Результаты проведенных в дипломной работе исследований могут быть использованные при разработке коммерческих программных.

Результаты проведенного в дипломной работе исследования могут быть использованы при разработке коммерческих программных продуктов.

Научная новизна исследования определяется тем, что в ней предложен метод обфускации кода с шифрованием адресов массива неопределенных указателей с их последующей расшифровкой при вычислении отдельных выражений критического кода программы.

ОБФУСКАЦИЯ, ЗАПУТЫВАНИЕ КОДА, ВЗЛОМ ПРОГРАММ,ДЕЗАСЕМ.

ABSTRACT

Explanatory note: __C., __ figure, __table, __ annexes, __literature.

Object of research: methods of protecting software from analysis and hacking.

The purpose of the work: improving the quality of protection of software products from research and modification.

Methods of research: methods of compiler theory, concepts and methods of object-oriented programming and cryptography. In the special part, a method of obfuscating the code with encryption of the addresses of the array of indeterminate pointers with their subsequent decoding is proposed when calculating individual expressions of the critical program code. This method enhances the confusing effect through the joint application of previously known and separately used obfuscation methods.

In the economic section, the costs of program protection by the obfuscation method were analyzed using the existing obfuscation systems and using the proposed obfuscation method.

The practical value of the work is to develop a methodology, algorithm and program for its realization, which allow to effectively solve problems of software protection from analysis.

The results of research carried out in the thesis can be used in the development of commercial software.

The scientific novelty of the research is determined by the fact that it proposes a method of obfuscating the code with the encryption of addresses of an array of indeterminate pointers with their subsequent decoding in the calculation of individual expressions of the critical code of the program.

FURNISHING, CODE RETURN, SOFTWARE, DEASABLE.

ЗМІСТ

с.

ВСТУП

Інтенсивний розвиток інформатизації суспільства викликає потребу в захисті такої інтелектуальної власності, як програмні продукти (ПП).

Розробка найбільш ефективного методу захисту для того чи іншого програмного продукту стає однією з важливих задач більшості програмістів, які займаються розробкою спеціалізованого платного програмного забезпечення (ПО). Це дозволяє їм продавати свою інтелектуальну працю, і виключити можливості його нелегального використання серед споживачів, тобто користувач не зможе використовувати оригінальну, ліцензійну копію певної програми попередньо не купивши, не заплативши грошей її розробнику.

В даний час захист коду програм приділяється велика увага, тому що найчастіше вихідні тексти програм становлять комерційну таємницю. Крім того, знання внутрішнього устрою деяких систем може допомогти зловмисникам організувати атаку на користувачів цієї системи, використовуючи при цьому помилки, допущені при створенні продукту.

Існують два основних способи захисту інтелектуальної власності, і отже, самих програмних продуктів: юридичний і технічний, який реалізується шляхом включення в ПП, будь-якого з наявних методів захисту, який заборонятиме його нелегальне використання. У порівнянні з юридичним способом захисту ПП, він є найбільш поширеним, оскільки він практичний, і порівняно не дорогий в реалізації (далі, буде приводитися саме його опис).

У більшості випадків для обходу захисту, зломникові потрібно вивчити принцип роботи її коду, і то, як вона взаємодіє з самої захищається програмою, цей процес вивчення називається процесом реверсивної (зворотної) інженерії.

Обфускація («obfuscation» - заплутування), це один з методів захисту програмного коду, який дозволяє ускладнити процес реверсивної інженерії

коду захищається програмного продукту.

Суть процесу обфускації полягає в тому, щоб заплутати програмний код і усунути більшість логічних зв'язків в ньому, тобто трансформувати його так, щоб він був дуже важкий для вивчення і модифікації сторонніми особами (будь то зломники, або програмісти які збираються дізнатися унікальний алгоритм роботи захищається програми).

Дана обставина визначає актуальність розробки методик обфускації коду і даних програм, що забезпечують стійкість щодо засобів їх аналізу та модифікації.

Метою роботи є підвищення якості захисту програмних продуктів від дослідження і модифікації.

Об'єктом дослідження є методи захисту програмного забезпечення від аналізу.

Предметом дослідження є методика обфускації критично важливого коду програм.

Методи дослідження. Для вирішення поставленого завдання в даній роботі використовувалися методи теорії компіляторів, поняття і методи об'єктно-орієнтованого програмування і криптографії.

Достовірність і обґрунтованість результатів магістерської роботи забезпечується застосуванням коректних вихідних даних, апробованих методів обфускації, перевіркою несуперечності висновків і результатами аналізу тестових програм.

Наукова новизна роботи визначається тим, що в ній запропоновано метод обфускації коду з шифруванням адреса масиву невизначених покажчиків з їх подальшою розшифровкою при обчисленні окремих виразів критичного коду програми. Даний метод посилює заплутування ефекту за рахунок спільного застосування раніше відомих і окремо використовуваних способів обфускації.

Практичне значення магістерської роботи полягає в тому, що

розроблена методика, алгоритм і програмна його реалізація дозволяють ефективно вирішувати завдання захисту програмного забезпечення від аналізу. Розроблена методика вимагає наявності вихідних текстів захисних програм і може застосовуватися програмістами, що не володіють специфічними знаннями та навичками в області захисту програмного забезпечення.

РОЗДІЛ 1. ХАРАКТЕРИСТИКА ОБ'ЄКТА ДОСЛІДЖЕННЯ

1.1 Визначення процесу обфускації

Розробка найбільш ефективного методу захисту для того чи іншого програмного продукту, в нинішній час, стає однією з важливих задач більшості програмістів.

Існують два основних способи захисту програмних продуктів (ПП):

1) Юридичний. Даний спосіб захисту полягає у створенні певних актів, відповідно до закону, які будуть охороняти інтелектуальну власність від нелегального використання. Даний спосіб охоплює патентування, оформлення авторських прав на інтелектуальну власність і так далі.

2) Технічний. Реалізується шляхом включення в ПП, будь-якого з наявних методів захисту, який заборонятиме його нелегальне використання. У порівнянні з юридичним способом захисту ПП, він є найбільш поширеним, оскільки практичний, і порівняно не дорогий в реалізації. Одним з технічних методів захисту програм є обфускація.

У більшості випадків для обходу захисту, зломникові потрібно вивчити принцип роботи її коду, і те, як вона взаємодіє з самою захищеною програмою, цей процес вивчення називається процесом реверсивної (зворотної) інженерії. Цей процес часто залежить від властивостей людської психіки, тому використання цих властивостей дозволяє знизити ефективність самого процесу реверсивної інженерії.

Обфускація («obfuscation» - заплутування) це один з методів захисту програмного коду, який дозволяє ускладнити процес реверсивної інженерії коду захисного програмного продукту. Обфускація може застосовуватися не тільки для захисту ПП, вона має більш широке застосування, наприклад вона, може бути використана творцями вірусів, для захисту їх творінь і так далі.

Суть процесу обфускації полягає в тому, щоб заплутати програмний

код і усунути більшість логічних зв'язків в ньому, тобто трансформувати його так, щоб він був дуже важкий для вивчення і модифікації сторонніми особами (будь то зломники, або програмісти які збираються дізнатися унікальний алгоритм роботи захисної програми). З цього випливає, що обфускація одна не призначена для забезпечення найбільш повного та ефективного захисту програмних продуктів, так як вона не надає можливості запобіганню нелегального використання програмного продукту. Тому обфускацію зазвичай використовують разом з одним з існуючих методів захисту (шифрування програмного коду і так далі), це дозволяє значно підвищити рівень захисту ПП в цілому. Процес обфускації як метод захисту, можна вважати порівняно новим (перші статті, присвячені обфускації, як методу захисту коду програмних продуктів, з'явилися приблизно десять років тому), і перспективним.

Обфускація відповідає принципу економічної доцільності, так як її використання не сильно, збільшує вартість програмного продукту, і дозволяє при цьому знизити втрати від піратства, і зменшити можливість плагіату в результаті крадіжки унікального алгоритму роботи захисного програмного продукту. Кінцевою метою обфускації є ускладнення декомпіляції, налагодження або вивчення програм з метою виявлення функціональності, а також запобігання обходу засоби захисту авторських прав.

1.1.1 Вимоги до процесу обфускації

Нехай "TR" буде трансформируючим процесом, який перетворює вихідний код програми PR1 в інший вихідний код PR2. Процесс трансформації "TR" буде вважатися процесом обфускації якщо, будуть задоволені такі вимоги:

- Код програми "PR2" в внаслідок трансформації буде істотно

відрізнятися від коду програми "PR1", але при цьому він буде виконувати ті ж функції що і код програми "PR1", а також буде працездатним.

• Вивчення принципу роботи, тобто процес реверсивної інженерії, програми "PR2" буде більш складним, трудомістким, і буде займати більше часу, ніж програми "PR1".

• При кожному процесі трансформації одного і того ж коду програми "PR1", код програм "PR2" будуть різні.

• Створення програми детрансформіруючої програму "PR2" в її найбільш схожий первісний вигляд, буде неефективно.

Іншими словами заплутаною або обфусцірованою називається програма, яка на всіх допустимих для вихідної програми вхідних даних, видає той же самий результат, що і оригінальна програма, але більш важка для аналізу, розуміння і модифікації. Заплутана програма виходить в результаті застосування до вихідної незапутанної програми маскуючих заплутуючих перетворень.

Вихідний код програми модифікується програмою - обфускатор (рисунок 1.1) і в залежності від параметрів обфускатор перетворює його в інший вид. Далі модифікований код компілюється в виконуючий код. Основною властивістю обфускатора є те, що при різних параметрах обфускації виконуючий код різний, але результат роботи будь-якої обфусцірованої програми завжди один і той же.

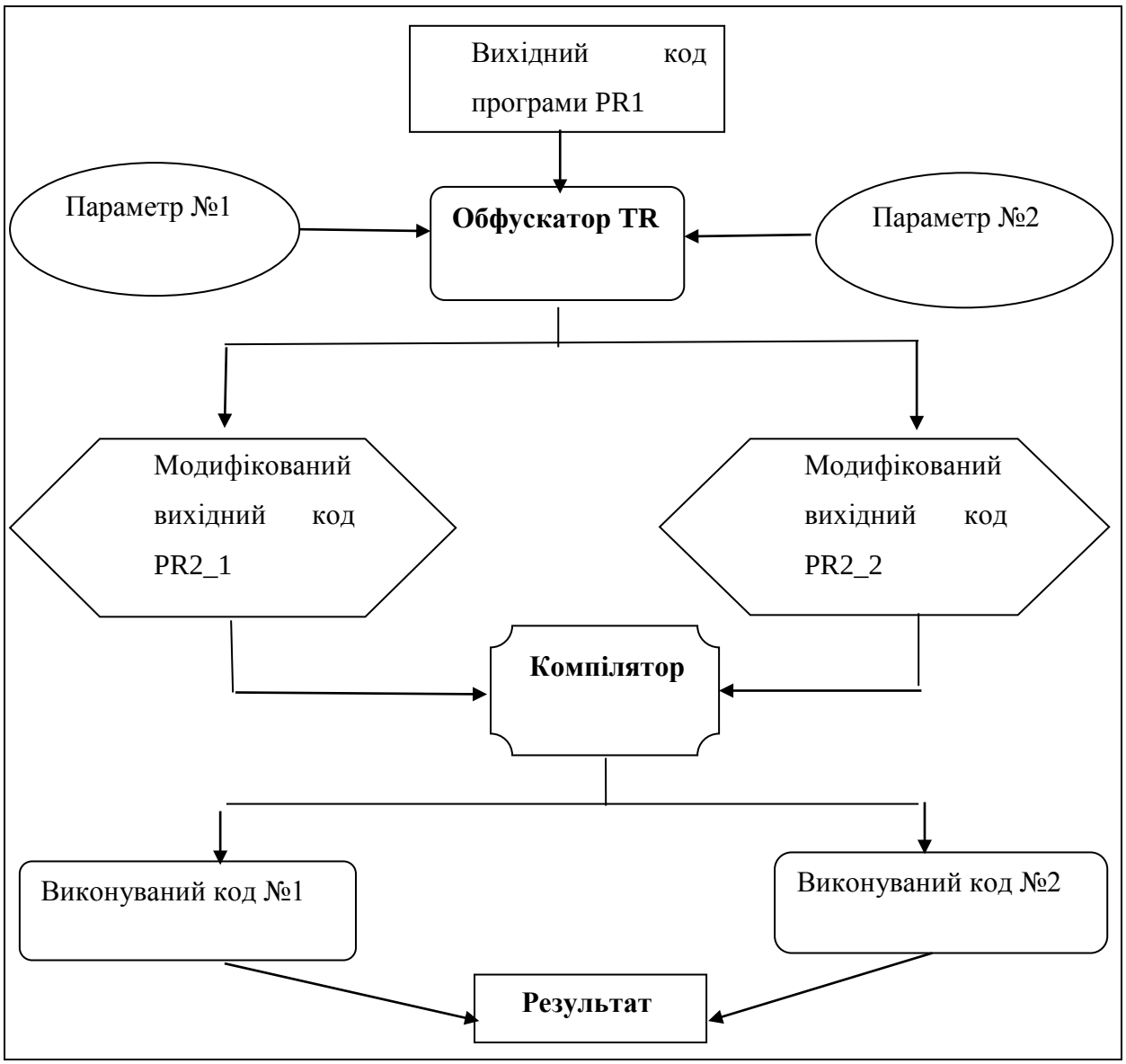


Рисунок 1.1 - Принцип роботи обфускатора

1.1.2 Задачі обфускації

Завдання заплутування і аналізу заплутаних програм мають три аспекти:

• Теоретичний, що включає в себе розробку нових алгоритмів перетворення графа потоку керування або трансформації даних програми, а також теоретичну оцінку складності їх аналізу і розкриття.

• Прикладний аспект включає в себе розробку конкретних методів заплутування (розплутування), тобто найкращих комбінацій алгоритмів, емпіричний порівняльний аналіз різних методів, емпіричний аналіз

стійкості методів.

Третій аспект, *психологічний* поки не піддається формалізації, але не може ігноруватися.

Заплутана програма повинна відповідати таким властивостям:
Заплутування має бути замаскованим. Те, що до програми були застосовані заплутуючі перетворення, не повинно кидатися в очі.

Заплутування не повинно бути регулярним. Регулярна структура заплутаної програми або її фрагмента дозволяє людині відокремити заплутані частини і навіть ідентифікувати алгоритм заплутування.

Застосування стандартних синтаксичних і статичних методів аналізу програм на початковому етапі її аналізу не повинно давати істотних результатів.

Центральним завданням теорії обфускації є наступна задача.

Потрібно придумати алгоритм (написати додаток), якій, отримуючи на вхід деяку програму P, трансформував би її в програму O (P). Природною ідеєю обфускації є складне становище аналізу коду, так щоб неможливо було встановити сенс окремо взятого фрагмента коду. При цьому повинні бути виконані три умови:

• збереження семантики ті ж виходи на тих же входах;

• обмеження на зростання споживаних ресурсів розмір коду, пам'ять і час виконання, потребуючі заплутаною програмою в порівнянні з вихідною;

• секретність збільшення складності програми, приховування внутрішніх параметрів. При цьому на вході і на виході обфускатор програма може бути записана в різних уявленнях.

На даний момент найбільш поширений підхід, що полягає в застосуванні великої кількості простих трюків. Кожен метод окремо легко розплутується, але в сукупності вони роблять завдання аналізу коду (особливо статичного аналізу) досить складною. Така методологія дає дві важливі переваги: вона універсальна (застосовується до всіх програм) і дуже легко програмується. Такі методи спрямовані на утруднення

статичного аналізу. Основним їх недоліком є повна відсутність оцінок на якість заплутаності. Більш того, ці методи взагалі не можуть дати якусь доказову секретність.

Паралельно заплутування програм вивчається [3] із суміжним питанням: як організувати таке середовище, в якій можна було б виконувати програми без побоювання, що їх внутрішня структура буде вивчена і піддана зміни або вилучення даних. Такий підхід до заплутування називається архітектурним. Всі результати, які розглядають захист програм як захист внутрішніх параметрів функції, яку обчислює ця програма, відносяться до алгоритмічному підходу. Явними плюсами є строгість моделі і можливість доводити оцінки на отриману секретність.

1.1.3 Рівні обфускації

Програмний код може бути представлений в подвійному вигляді (послідовність байтів представляють собою так званий машинний код, який виходить після компіляції вихідного коду програми) або початковому вигляді (текст, що містить послідовність інструкцій якоїсь мови програмування, який зрозумілий людині, цей текст після буде схильний до компіляції або інтерпретації на комп'ютері користувача). Процес обфускації може бути здійснений над будь-яким з вище перерахованих видів представлення програмного коду, тому прийнято виділяти такі рівні процесу обфускації:

- нижчий рівень, коли процес обфускації здійснюється над асемблерним кодом програми, або навіть безпосередньо над подвійним файлом програми зберігають машинний код.
- вищий рівень, коли процес обфускації здійснюється над вихідним кодом програми написаному на мові високого рівня.

Здійснення обфускації на нижчому рівні вважається менш комплексним процесом, але у більшості випадків важче реалізувати з

деяких причин. Одна з цих причин полягає в тому, що повинні бути враховані особливості роботи більшості процесорів, так як спосіб обфускації, прийнятний на одній архітектурі, може виявитися неприйнятним на інший.

На сьогоднішній день процес низкорівневої обфускації досліджений мало [2,6]. Більшість існуючих алгоритмів і методів обфускації (включаючи ті які будуть розглянуті нижче) можуть бути застосовані для здійснення процесу обфускації як на нижчому, так і на вищому рівні.

1.1.4 Оцінка процесу обфускації

Існує [2] багато методів визначення ефективності застосування того чи іншого процесу обфускації, до конкретного програмного коду. Ці методи прийнято розділяти на дві групи: аналітичні та емпіричні. Аналітичні методи грунтуються на трьох величинах характеризують наскільки ефективний той чи інший процес обфускації:

1) Стійкість - вказує на ступінь складності здійснення реверсивної інженерії над кодом пройшли процес обфускації.

2) Еластичність - вказує на те наскільки добре даний процес обфускації, захистить програмний код від застосування деобфускаторов.

3) Вартість перетворення - дозволяє оцінити, наскільки більше потрібно системних ресурсів для виконання коду минулого процесу обфускації, ніж для виконання оригінального коду програми.

Їх найбільш ефективно застосовувати при порівнянні різних алгоритмів обфускації, але при цьому вони не можуть дати абсолютної відповіді на питання наскільки ефективним є застосування того чи іншого алгоритму, саме до даного програмного коду. Емпіричні ж методи можуть дати сприятливу відповідь на таке питання, тому що вони ґрунтуються на статистичних даних одержуваних в результаті досліджень. Для проведення одного з таких досліджень потрібна група людей (якнайкраще знайомих, з реверсивної інженерії), фрагмент коду захисної програми, і набір різних

алгоритмів обфускації. Результати такого дослідження будуть включати в себе мінімальну кількість часу, який буде потрібний групі людей, для того щоб вивчити кожен фрагмент коду минулого один з алгоритмів обфускації.

1.2 Короткий опис заплутуючих перетворень

Заплутуючі перетворення можна розділити на кілька груп залежно від того, на трансформацію яких з компонентів програми вони націлені.

1) Перетворення форматування, які змінюють тільки зовнішній вигляд програми. До цієї групи належать перетворення, що видаляють коментарі, відступи в тексті програми або перейменовувати ідентифікатори.

2) Перетворення структур даних, які змінюють структури даних, з якими працює програма. До цієї групи належать, наприклад, перетворення, що змінює ієрархію спадкування класів в програмі, або перетворення, що об'єднує скалярні змінні одного типу в масив.

3) Перетворення потоку управління програми, які змінюють структуру її графа потоку управління, такі як розгортка циклів, виділення фрагментів коду в процедури, і інші.

4) Непрозорі предикати .

5) Превентивні перетворення, націлені проти певних методів декомпіляції програм або використовують помилки в певних інструментальних засобах декомпіляцію.

1.2.1 Перетворення форматування

До перетворенням форматування відносяться видалення коментарів, переформатування програми, видалення отладочної інформації, зміна імен ідентифікаторів.

Видалення коментарів і переформатування програми застосовні, коли заплутування виконується на рівні вихідного коду програми. Ці

перетворення не вимагають тільки лексичного аналізу програми. Хоча видалення коментарів - одностороннє перетворення, їх відсутність не утруднює сильно зворотну інженерію програми, так як при зворотній інженерії наявність хороших коментарів до коду програми є скоріше винятком, ніж правилом.

При переформатування програми вихідне форматування втрачається безповоротно, але програма завжди може бути переформатована з використанням будь-якого інструменту для автоматичного форматування програм (наприклад, indent для програм на Сі). Видалення налагоджувальної інформації може бути застосовано, коли заплутування виконується на рівні об'єктної програми. Видалення налагоджувальної інформації призводить до того, що імена локальних змінних стають невідновні.

Зміна імен локальних змінних вимагає семантичного аналізу (прив'язки імен) в межах однієї функції. Зміна імен всіх змінних і функцій програми крім повної прив'язки імен в кожній одиниці компіляції вимагає аналізу міжмодульних зв'язків.

Імена, певні в програмі і не використовуються в зовнішніх бібліотеках, можуть бути змінені довільним, але узгодженим у всіх одиницях компіляції чином, в той час як імена бібліотечних змінних і функцій змінюватися не можуть. Дане перетворення може замінювати імена на короткі автоматично генеруються імена (наприклад, всі змінні програми отримають ім'я v <номер> відповідно до їх деяким порядковим номером). З іншого боку, імена змінних можуть бути замінені на довгі, але безглузді (випадкові) ідентифікатори в розрахунку на те, що довгі імена гірше сприймаються людиною.

1.2. 2 Перетворення потоку управління

Перетворення потоку управління змінюють граф потоку керування однієї функції. Вони можуть призводити до створення в програмі нових функцій.

Коротка характеристика методів наведена нижче.

• Відкрита вставка функцій полягає в тому, що тіло функції підставляється в точку виклику функції. Дане перетворення є стандартним для оптимізують компіляторів. Це перетворення одностороннє, тобто за перетвореною програмі автоматично відновити вставлені функції неможливо.

• Винос групи операторів. Дане перетворення є зворотним до попереднього і добре доповнює його. Деяка група операторів вихідної програми виділяється в окрему функцію. При необхідності створюються формальні параметри. Перетворення може бути легко звернуто компілятором, який (як було сказано вище) може підставляти тіла функцій в точки їх виклику.

• Усунення бібліотечних викликів. Більшість програм істотно використовують стандартні бібліотеки. Оскільки зміст бібліотечних функцій добре відома, такі виклики можуть дати корисну інформацію при зворотної інженерії програм. У багатьох випадках можна обійти цю обставину, просто використовуючи в програмі власні версії стандартних бібліотек.

• Клонування функцій. При зворотної інженерії функцій в першу чергу вивчається сигнатура функції, а також те, як ця функція використовується, в яких місцях програми, з якими параметрами, і в якому оточенні викликається. Аналіз контексту використання функції можна ускладнити, якщо кожен виклик деякої функції буде виглядати як інший виклик, кожен раз нової функції. Може бути створено кілька клонів функції, і до кожного з клонів буде застосований різний набір заплутуючих перетворень.

1.2.3 Перетворення потоку управління

Перетворення потоку управління змінюють граф потоку керування однієї функції. Вони можуть призводити до створення в програмі нових функцій. Коротка характеристика методів наведена нижче.

• Відкрита вставка функцій полягає в тому, що тіло функції підставляється в точку виклику функції. Дане перетворення є стандартним для оптимізації компіляторів. Це перетворення одностороннє, тобто за перетвореною програмою автоматично відновити вставлені функції неможливо.

• Винос групи операторів. Дане перетворення є зворотним до попереднього і добре доповнює його. Деяка група операторів вихідної програми виділяється в окрему функцію. При необхідності створюються формальні параметри. Перетворення може бути легко звернуто компілятором, який (як було сказано вище) може підставляти тіла функцій в точки їх виклику.

• Усунення бібліотечних викликів. Більшість програм істотно використовують стандартні бібліотеки. Оскільки зміст бібліотечних функцій добре відомі, такі виклики можуть дати корисну інформацію при зворотної інженерії програм. У багатьох випадках можна обійти цю обставину, просто використовуючи в програмі власні версії стандартних бібліотек.

• Клонування функцій. При зворотній інженерії функцій в першу чергу вивчається сигнатура функції, а також те, як ця функція використовується, в яких місцях програми, з якими параметрами, і в якому оточенні викликається. Аналіз контексту використання функції можна ускладнити, якщо кожен виклик деякої функції буде виглядати як виклик якийсь інший, кожен раз нової функції. Може бути створено кілька клонів

функції, і до кожного з клонів буде застосований різний набір заплутують перетворень.

• Розгортка циклів. Розгортка циклів застосовується в оптимізації компіляторів для прискорення роботи циклів або їх розпаралелювання. Розгортка циклів полягає в тому, що тіло циклу розмножується два або більше разів, умова виходу з циклу і оператор збільшення лічильника відповідним чином модифікуються. Якщо кількість повторень циклу відомо в момент компіляції, цикл може бути розгорнутий повністю.

• Розкладання циклів. Розклад циклів полягає в тому, що цикл з складним тілом розбивається на кілька окремих циклів з простими тілами і з тим же простором і терірованія.

• Реструктуризація графа потоку керування. Структура графа потоку управління, наявність в графі потоку управління характерних шаблонів для циклів, умовних операторів і так далі. Дає цінну інформацію при аналізі програми. Наприклад, по повторюваним конструкціям графа потоку керування можна легко встановити, що над функцією було виконано перетворення розгортки циклів, а далі можна запустити спеціальні інструменти, які проаналізують розгорнуті ітерації циклу для виділення індуктивних змінних і згортки циклу.

• Локалізація змінних в базовому блоці. Це перетворення локалізує використання змінних одним базовим блоком. Для кожного заплутаного базового блоку функції створюється свій набір змінних. Все використання локальних і глобальних змінних в вихідному базовому блоці замінюються на використання відповідних нових змінних. Щоб забезпечити правильну роботу програми між базовими блоками вставляються так звані сполучні базові блоки, завдання яких скопіювати вихідні змінні попереднього базового блоку в вхідні змінні наступного базового блоку.

• Розширення області дії змінних. Це перетворення за змістом назад попереднього. Це перетворення намагається збільшити час життя змінних настільки, наскільки можна.

• Вставка помилкових циклів. В коді заплутаної програми відбувається пошук ділянок коду, за структурою нагадують одну ітерацію циклу. Далі в початок ділянки або в його кінець (в залежності від різновиду вставляється цикл) вставляється базовий блок з умовним переходом в протилежний кінець виділеної ділянки коду. Умовний перехід містить в собі непрозорий предикат, який і маскує лише одне виконання циклу. В якості відповідної ділянки коду розглядається ділянка з одним входом і виходом.

• Переплетення циклів. Алгоритм заплутування для всіх циклів функції додає «фиктивні» ребра з одного циклу в інший. Як видно на рис. 2, метод створює ребра які входять в цикл, так і виходять з нього.

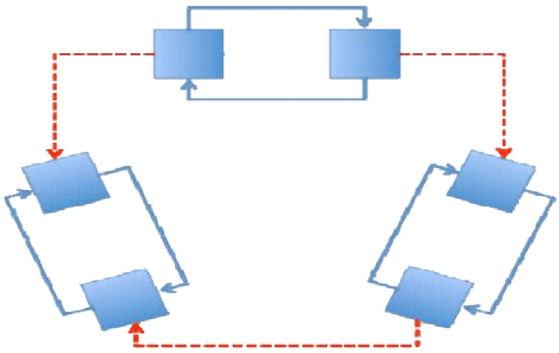


Рисунок 2 - Переплетення 3 циклів, пунктиром показані «фиктивні» ребра.

Недоліком такої трансформації є те, що вона ефективно заплутує тільки код великих функцій, які мають складні структури, що містять кілька циклів. Крім цього застосовується наступний підхід для заплутування всього графа потоку управління: з безлічі блоків функції вибирається N блоків і між ними випадково додаються ребра. Фіктивні переходи захищаються непрозорими предикатами.

• Переплетення функції. Ідея цього заплутує перетворення в тому, що дві або більше функцій об'єднуються в одну функцію. Списки параметрів вихідних функцій об'єднуються, і до них додається ще один параметр, який дозволяє визначити, яка функція в дійсності виконується. Класичний підхід

до переплетення функцій, має малу стійкість [2,3].

• Збільшення складності аналізу потоку даних в програмі. Ідея перетворення полягає в перенесенні локальних змінних в глобальну область видимості і використати або відредагувати перенесених змінних в різних функціях. В загальному випадку не можна змінювати значення змінних, винесених з інших функцій в довільному місці програми, так як це може привести до неправильного виконання модульна програми. Тому будується граф викликів для всіх функцій в модулі (рисунок 2), аналізуючи який для кожної функції обчислюється безліч змінних модифікація яких, не порушить працездатність програми. Такими змінними будуть змінні, винесені з функцій, розташованих на різних шляхах в дереві викликів. Після формування множин відповідних змінних, здійснюється додавання сміттєвого коду, що використовує для обчислень "безпечних" змінних.

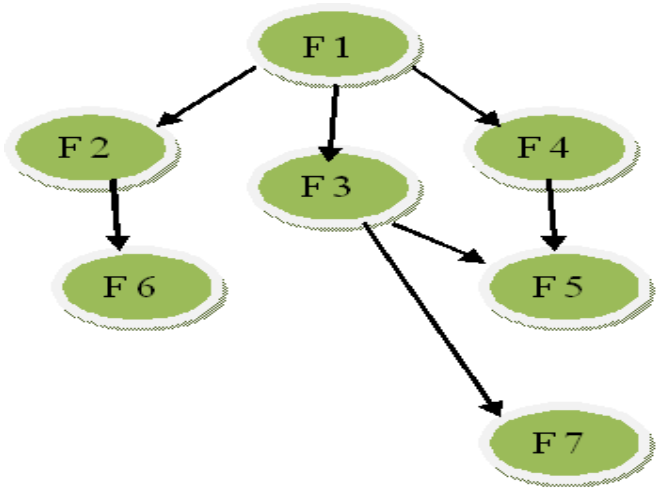


Рисунок 2 - Приклад дерева викликів.

• Розбиття цілочисельних констант. Часто в коді в явному виді зустрічаються константи характерні для певних алгоритмів, наприклад константа 0x67452301 для функції хешування MD5. Пошук констант дозволяє визначити використовуваний алгоритм, що спрощує аналіз програми. Для протидії запропонований алгоритм розбиття констант. Розбиваються тільки константи більше одиниці. Для розбиття випадковим

чином вибирається число менше початкового, яке виступатиме першим доданком, другий доданок виходить.

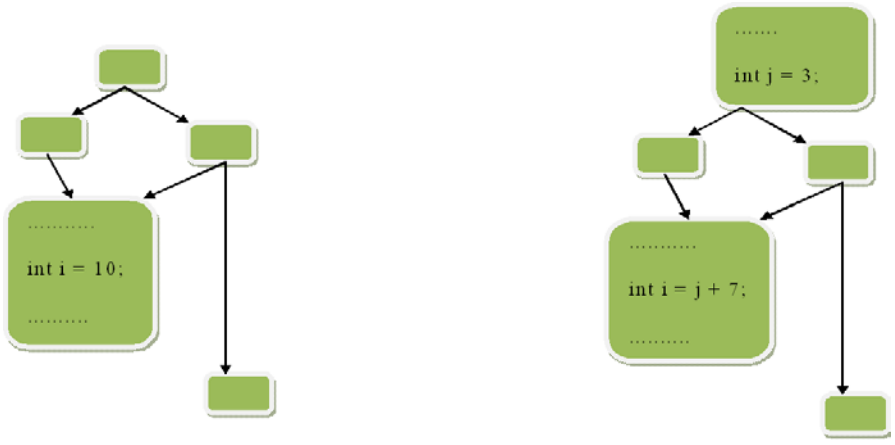


Рисунок 3 - Приклад алгоритму розбиття цілочисельної константи

1.2.4. Непрозорі предикати.

Основною проблемою при проектуванні заплутують перетворень графа потоку управління є те, як зробити їх не тільки дешевими, але і стійкими. Для забезпечення стійкості багато перетворення ґрунтуються на введенні непрозорих змінних і предикатів. Сила таких перетворень залежить від складності аналізу непрозорих предикатів і змінних.

Непроникною змінної називають змінну, що вводиться при обфускації програми, яка задовольняє таким умовам.

1) Значення змінної на стадії обфускації програми може бути визначено в самій програмі і за додатковою інформацією, яка відома на стадії обфускації програми, але відсутня в тексті програми.

2) Завдання визначення значення непроникною змінної по тексту обфускірованної програми є складною.

Непроникний предикат, - це предикат (вираз, що виробляє результат булевого типу), що задовольняє вищевказаним двома ознаками.

Непрозорі предикати можуть бути трьох видів:

PF - предикат, який завжди має значення «брехня»;

PT - предикат, який завжди має значення "істина", і P? - предикат, який може приймати обидва значення, і в момент заплутування поточне значення предиката відомо.

Внесення в програму непроникних змінних і предикатів представляється нам одним з найважливіших методів обфускації програм. Від стійкості непроникних предикатів залежить стійкість багатьох методів обфускації програми, наприклад, внесення недосяжного коду. Основна складність при розробці перетворень потоку управління полягає в тому. щоб зробити їх не тільки дешевими, але і стійкими до автоматичними деобфускаторами. Для досягнення цього багато перетворення покладаються на існування непроникних змінних і предикатів.

Деякі методи отримання непроникних предикатів перераховані нижче:

• Використання різних тотожностей, аналогічно перетворенню внесення тотожностей:

• Побудова складнихбулевих виразів за допомогою еквівалентних перетворень з формули true (або false). У найпростішому випадку можна взяти до довільних нулевих змінних і побудувати з них формулу. Далі за допомогою еквівалентних алгебраїчних перетворень частина дужок (або все) розкриваються, і в результаті виходить шуканий непроникний предикат.

Деякі можливі способи введення непрозорих предикатів і непрозорих виразів коротко перераховані нижче.

• *Використання різних способів доступу до елементів масиву*. Наприклад, в програмі може бути створений масив, який ініціалізується заздалегідь відомими значеннями, далі в програму додаються кілька змінних, в яких зберігаються індекси елементів цього масиву.

• *Використання покажчиків на спеціально створювані динамічні структури.* У цьому підході до програми додаються операції по створенню довідкових структур даних (списків, дерев), і додаються операції над покажчиками на ці структури, підібрані таким чином, щоб зберігалися деякі інваріанти, які і використовуються як непрозорі предикати.

• *Конструювання булевских виразів спеціального виду*. Побудова складних булевских виразів за допомогою еквівалентних перетворень.

• *Внесення недосяжного кода*. Якщо в програму внесені непрозорі предикати видів PF або PT, гілки умови, що відповідають умові "істина" в першому випадку і умові "брехня" в другому випадку, ніколи не будуть виконуватися. Фрагмент програми, який ніколи не виконується, називається недосяжним кодом. Ці гілки можуть бути заповнені довільними обчисленнями, які можуть бути схожі на дійсно виконуючий код, наприклад, зібрані з фрагментів тієї ж самої функції.

• *Внесення мертвого коду.* На відміну від недосяжного коду, мертвий код в програмі виконується, але його виконання ніяк не впливає на результат роботи програми. При внесенні мертвого коду запутиватель повинен бути впевнений, що вставляється фрагмент не може впливати на код, який обчислює значення функції.

• *Внесення надлишкового коду.* Надмірний код, на відміну від мертвого коду виконується, і результат його виконання використовується в подальшому в програмі, але такий код можна спростити або зовсім видалити, так як обчислюється або константне значення, або значення, вже обчислене раніше. Для внесення надлишкового коду можна використовувати алгебраїчні перетворення виразів вихідної програми або введення в програму математичних тотожностей.

1.3 Методи і інструменти для аналізу програм

1.3.1 Методи дослідження програм

1.3.1.1 Дослідження за методом «білого ящика»

При дослідженні за методом «білого ящика» виконується аналіз перш за все вихідного коду. Іноді доступний тільки двійковий код, але можна

провести його декомпіляцію, отримати з двійкового вихідний код і провести його дослідження, що також є аналізом за методом «білого ящика». Цей метод тестування дуже ефективний для виявлення помилок програмування і реалізації в програмному забезпеченні. У деяких випадках дослідження доходить до пошуку відповідностей за заданими шаблонами і може навіть виконуватися автоматично за допомогою статичного аналізатора.

Для методу «білого ящика» характерний один недолік при використанні цього методу часто виявляються нібито потенційні вразливі місця, яких в дійсності не існує. Проте, методи статичного аналізу вихідного коду дозволяють успішно зламувати деякі програми.

Засоби для проведення досліджень, методу «білого ящика» можна розділити на дві категорії: ті, яким потрібен вихідний код, і ті, які автоматично декомпілюються двійковий код і продовжують роботу з цього моменту.

Дослідження за методом «чорної скриньки»

При дослідженні за методом «чорної скриньки» на вхід виконуваної програми подаються різні тестові дані. При такому тестуванні потрібно тільки запуск програми і не проводиться ніякого аналізу вихідного коду. З точки зору безпеки на вхід програми можуть подаватися шкідливі дані з метою викликати збій в роботі програми. Якщо програма дає збій при виконанні якогось тесту, то вважається, що виявлена проблема безпеки.

Аналіз за методом «чорної скриньки» можливий навіть без доступу до двійкового коду. Таким чином, програма може бути проаналізірована по мережі. Все, що потрібно, - це наявність запущеної програми, яка здібна приймати вхідні дані, тобто якщо дослідник здатний відправляти вхідні дані, які приймає програма, і здатний отримати результат про ництва цих даних, значить, можливо тестування за методом «чорної скриньки». Ось чому багато хакерів вибирають для злому саме методи «чорної скриньки».

Аналіз програми за методом «чорної скриньки» не такий ефективний,

як при використанні методу «білого ящика», але цей метод набагато простіше для реалізації і не вимагає високого рівня кваліфікації. В ході тестування за методом «чорного ящика», фахівець, впливаючи на програму, але видаються результатами намагається максимально точно визначити шляхи виконання коду в програмі. При цьому неможливо перевірити дійсне місце введення призначених для користувача даних в коді програми, але тестування за методом чорного ящика більше нагадує реальну атаку в реальному середовищі виконання в порівнянні з використанням методу «білого ящика».

Іноді помилки, виявлені при аналізі по методу «чорного ящика», не можуть бути використані хакерами при реальних атаках на конкретну систему в конкретній мережі. Наприклад, атаку може заблокувати брандмауер.

Дослідження за методом «сірого ящика»

У дослідженнях методу «сірого ящика» об'єднані методи «білого ящика» і способи тестування за допомогою вхідних даних по методу «чорного ящика». Вдалим прикладом простого аналізу але методу «сірого ящика» є запуск програми всередині відладчика і подача на вхід цієї програми різних даних. При цьому йде виконання програми, а відладчик використовується для виявлення помилок і некоректних станів. В цілому, всі методи тестування дозволяють розкрити ризики для програмного забезпечення та потенційні можливості для проведення атак.

1.3.2 Методи аналізу програм

Мета таких методів - виявлення залежностей між компонентами програми, що дає можливість застосувати певні оптимізаційні перетворення, або накладає обмеження на проведені оптимізаційні перетворення.

Методи аналізу програм можуть бути розділені на 4 групи:

1) Синтаксичні. До цієї групи належать методи, засновані тільки на результатах лексичного, синтаксичного і семантичного аналізу програми.

2) Статичні. До цієї групи належать методи аналізу потоків управління і даних і методи, засновані на результатах аналізу потоків управління і даних. Статичні методи аналізу працюють з програмою, не використовуючи інформацію про роботу програми на конкретних початкових даних.

3) Динамічні. Динамічні методи аналізу програм використовують інформацію, отриману в результаті "спостереження" за роботою програми на конкретних вхідних даних. Зауважимо, що самі по собі динамічні методи рідко застосовуються для аналізу програм, оскільки, як правило, необхідна інформація про поведінку програми на різних наборах вхідних даних, яка збирається за допомогою статистичних методів аналізу.

4) Статистичні. Статистичні методи використовують інформацію, зібрану в результаті значної кількості запусків програми на великій кількості наборів вхідних даних.

1.3.2.1 Методи статичного аналізу

Статичний аналіз [6,9] необхідний в мовах, в яких кілька імен можуть бути використані для доступу до однієї і тієї ж області пам'яті. Наприклад, певний елемент масиву a може бути адресований в програмі як a [0] (канонічне ім'я елемента масиву), як a [j] або взагалі як b [-4]. В результаті аналізу кожному оператору, що виконує непряму запис в пам'ять або непряме читання з пам'яті, ставиться у відповідність безліч імен змінних, які можуть справити вплив даної операцією.

Статичне усунення мертвого коду має на меті виявити в програмі код, який виконується, але не впливає на результат роботи програми.

Статична мінімізація кількості змінних має на меті зменшити кількість використовуваних у функції локальних змінних за рахунок об'єднання

змінних, часи життя значень в яких не перетинаються, в одну зміну. Стандартна техніка, яка використовується для мінімізації кількості змінних, полягає в побудові графа перекриття змінних за допомогою ітераційного рішення рівняння потоку даних і подальшої розфарбованих вершин цього графа в мінімальне або близьке до мінімальної кількісті квітів.

Статичне просування констант і копій полягає в просуванні константних виразів якнайдалі по тексту функції. Якщо вираз використовує тільки значення змінних, які в даній точці програми свідомо містять одне відоме при аналізі програми значення, такий вислів може бути обчислено на етапі аналізу програми. Якщо у виразі використовується змінна, яка в даній точці програми свідомо є копією якоїсь іншої змінної, в вираз може бути підставлена вихідна змінна.

Статичний аналіз доменів є розширенням алгоритму просування констант. Він дозволяє визначити безліч значень, які може приймати ця змінна в даній точці програми, якщо це безліч не велика.

Статичний слайсинг - це побудова "скороченої" програми, з якої відстані весь код, що не впливає на обчислення заданої змінної в заданій точці (зворотний слайс), але при цьому програма залишається синтаксично і семантично коректної і може бути виконана. Крім зворотного слайсинга розроблені алгоритми прямого слайсинга. Прямий слайсінг залишає в програмі тільки ті оператори, які залежать від значення змінної, обчисленого в даній точці програми. Методи слайсинга можуть бути корисні при поділі "переплетених" обчислень, коли одночасно обчислюються дві незалежні одна від одної величини. Наприклад, в одному циклі може обчислюватися скалярний добуток двох векторів, а також мінімальний і максимальний елемент кожного вектора, і такі цикли можуть бути розщеплені за допомогою побудови слайсів.

Статистичний аналіз покриття базових блоків програми дозволяє встановити, виконувався чи коли-небудь при виконанні програми на заданій множині наборів вхідних даних заданий базовий блок.

Статистичне порівняння трас дозволяє виявити, чи однакові траси програми, отримані при різних запусках на одному і тому ж наборі вхідних даних. Статистична побудова графа потоку управління будує граф потоку керування на підставі інформації про порядок проходження базових блоків на одному наборі або на безлічі наборів вхідних даних.

Динамічне просування копій уздовж трас необхідно для точного межпроцедурного аналізу залежностей за даними на основі траси виконання програми. Оскільки траса виконання програми, по суті, є одним великим базовим блоком, просування копій - нескладне завдання.

Динамічне виділення мертвого коду дозволяє виявити інструкції програми, які виконувалися при цьому запуску програми, але не справили жодного впливу на результат роботи програми. Якщо аналізується сукупність запусків програми на безлічі наборів вхідних даних, можна говорити про статистичному виділення мертвого коду.

Динамічний слайсінг залишає в трасі програми тільки ті інструкції, які вплинули на обчислення даного значення в даній точці програми (прямий динамічний слайсінг), або тільки ті інструкції, на які вплинуло присвоювання значення даної змінної в даній точці програми. Описані вище динамічні методи не можуть застосовуватися в автоматичному інструменті аналізу програм. Роль цих методів в тому, щоб привернути увагу користувача інструменту аналізу програм до особливостей роботи програми. Надалі користувач може вивчити «підозрілий» фрагмент коду більш детально із застосуванням інших інструментів, щоб підтвердити або спростувати висунуту гіпотезу. Якщо непрозорі предикати і недосяжний код усуваються тільки на підставі статистичного аналізу, завжди залишається можливість, що предикат був істотним (як в прикладі вище). Щоб все ж спростити програму, можна, наприклад, винести імовірно недосяжний код із загального графа потоку керування функції в обробник спеціального виключення, яке порушується кожен раз, коли предикат прийме значення, відмінне від звичайного. З одного боку, граф потоку

керування і потоку даних основної програми в результаті спроститься, а з іншого боку, програма збереже свою функціональність. Перерахування деяких методів заплутування програм з точки зору методів їх можливого розплутування дано в таблиці 1.

Таблиця 1.1 - Методи заплутування програм і методи для їх аналізу.

Метод заплутування	**Метод розплутування**
спотворення імен змінних	перейменування змінних
використання специфічних мовних конструкцій	спрощення специфічних мовних конструкцій
розгортка циклу	візуалізація графа потоку керування функції для виділення потенційних кандидатів на згортку в цикл; виявлення індуктивної змінної, що вимагає (інтерактивного) порівняння базових блоків і (інтерактивних) еквівалентних перетворень виразів, причому при таких перетвореннях вираз може навіть (трохи) ускладнюватися; згортка циклу
використання унікальних змінних в базових блоках	просування копій (статичну і статистичне), мінімізація кількості використовуваних змінних
введення детермінованого диспетчера	статистичне відновлення графа потоку управління
введення недетермінірованного диспетчера	порівняння трас, отриманих на одному і тому ж наборі вхідних даних
переплетення коду декількох базових блоків в один заплутаний базовий блок	статистичне усунення мертвого коду

Продовження таблиці 1.1

введення непрозорих предикатів	статистичний аналіз покриття для виявлення потенційних непрозорих предикатів, пошук за зразками відомих непрозорих предикатів, алгебраїчне спрощення, доведення теорем згортка констант, просування констант, просування копій, статичне усунення мертвого коду - можуть виконуватися після кожного кроку розплутують перетворень
всі методи	

У таблиці 1.2 наведено класифікацію методів заплутування по відношенню до необхідних методів аналізу програм.

Таблиця 1.2 - Класифікація заплутують перетворень

Перетворення	**Складність розпутування** (необхідний тип аналізу)	**Автоматизація** (тип розплутувача)
Видалення коментарів	Одностороннє перетворення	hh
Переформатування програми	Синтаксичний	автомат.
Видалення налагоджувальної інформації	Одностороннє перетворення	hh
Зміна імен ідентифікаторів	Синтаксичний	напівавтомат.
Мовно-специфічні перетворення	Синтаксичний	автомат.
Відкрита вставка функцій	Одностороннє перетворення	підтрим.
Винос групи операторів	Синтаксичний	автомат.
Непрозорі предикати і вирази	Синтаксичний - статистичний (залежить від виду предиката)	автомат. - підтрим.
Внесення недосяжного коду	Залежить від стійкості непрозорих предикатів	автомат., напівавтомат.

Продовження таблиці 1.2

Внесення мертвого коду	Синтаксичний – статистичний	автомат., напівавтомат.
Введення диспетчера	Статичний – статистичний	автомат., напівавтомат.
Локалізація змінних в базовому блоці	Статичний – статистичний	автомат., напівавтомат.
Розширення області дії змінних	Статичний – статистичний	автомат., напівавтомат.
Внесення надлишкового коду	Синтаксичний– статистичний	автомат., підтрим.
Внесення незвідність в граф	Статичний, але залежить від стійкості непрозорих предикатів	автомат., напівавтомат
Усунення бібліотечних викликів	Одностороннє перетворення	підтрим.
Переплетення функцій	Статичний – статистичний	автомат. - підтрим.
Клонування функцій або базових блоків	Статичний	автомат. - підтрим.
Розгортка циклів	Одностороннє перетворення	підтрим.
Розкладання циклів	Статичний	автомат. - підтрим.
Введення диспетчера	Статичний – статистичний	автомат., напівавтомат.
Локалізація змінних в базовому блоці	Статичний – статистичний	автомат., напівавтомат.
Розширення області дії змінних	Статичний – статистичний	автомат., напівавтомат.

У третьому стовпці таблиці вказана ступінь, в якій можливе автоматичне розплутування.

Ступінь автоматизму оцінюється за такою шкалою: автоматичний - пошук в програмі заплутаних фрагментів і їх розплутування повністю автоматично;

напівавтоматичний - пошук в програмі підозрілих фрагментів і їх розплутування окремо виконуються автоматично, але користувач повинен підтвердити застосування розплутувати перетворення;

підтримуваний - пошук в програмі заплутаних фрагментів і застосування розплутують перетворень вимагають істотної участі людини, але процес може бути підтриманий спеціальними інструментальними засобами;

неавтоматичний - автоматизація виконання розплутувати перетворення принципово утруднена.

1.3.3 Програмні інструменти для дослідження і аналізу програм

Всі кошти для дослідження програм можна розбити на 2 класа: статичні і динамічні.

Статичні оперують вихідним кодом програми як даними і будують її алгоритм без виконання. Динамічні вивчають програму, інтерпретуючи її в реальному або віртуальної обчислювальної середовищі. Статичні засоби аналізу є більш універсальними в тому сенсі, що теоретично можуть отримати алгоритм всієї програми, в тому числі і тих блоків, які ніколи не отримають управління. Динамічні засоби можуть будувати алгоритм програми тільки на підставі конкретної її траси, отриманої при певних вхідних даних. Тому завдання отримання повного алгоритму програми в цьому випадку еквівалентна побудови вичерпного набору текстів для підтвердження правильності програми, що практично неможливо, і взагалі при динамічному дослідженні можна говорити тільки про побудову деякої частини алгоритму.

Засоби для дослідження програм також можна класифікувати по меті їх застосування:

• аналізу даних програми;

• аналізу алгоритмів програм;

• для подолання систем захисту;

• обходу систем захисту програм;

На рисунку 1.3 показані основні групи засобів дослідження програм по цілі їх застосування [6]. Два найбільш відомих типу програм, призначених для дослідження ПЗ, як раз і відносяться до різних класів: це відладчик (динамічний засіб) і дизассемблер (засіб статистичного дослідження).

Отладчиком називається програма, яка виконує в собі іншу програму і дозволяє здійснювати контроль за цією виконуваною програмою. За допомогою відладчика можна перевіряти програму покроково, відстежувати маршрут виконання коду, встановлювати точки зупинки і контролювати значення змінних і пам'яті перевіряється (або атакується) програми. Отладчик просто незамінний для визначення логічної структури програми. Існує дві категорії отладчиків: отладчики призначених для користувача програм і отладчики ядра. Відладчики призначених для користувача програм запускаються як звичайна програма під управлінням операційної системи і підкоряються тим же правилам, що і звичайні програми. Таким чином, отладчики цієї категорії здатні виконувати налагодження тільки інших процесів, що виконуються на рівні користувача.

Отладчик ядра є частиною операційної системи, і з його допомогою можна виконувати налагодження драйверів пристроїв і навіть самої операційної системи. Дизассемблер призначений виключно для побудови алгоритму і формує на виході асемблерний текст алгоритму. Дизассемблер дозволяє конвертувати машинний код в код на мові асемблера. Код на мові асемблера є читабельною формою машинного коду (по крайней мере, більш читабельною, ніж рядок бітів). За допомогою дизассемблера можна дізнатися, які машинні інструкції використовуються в машинному коді.

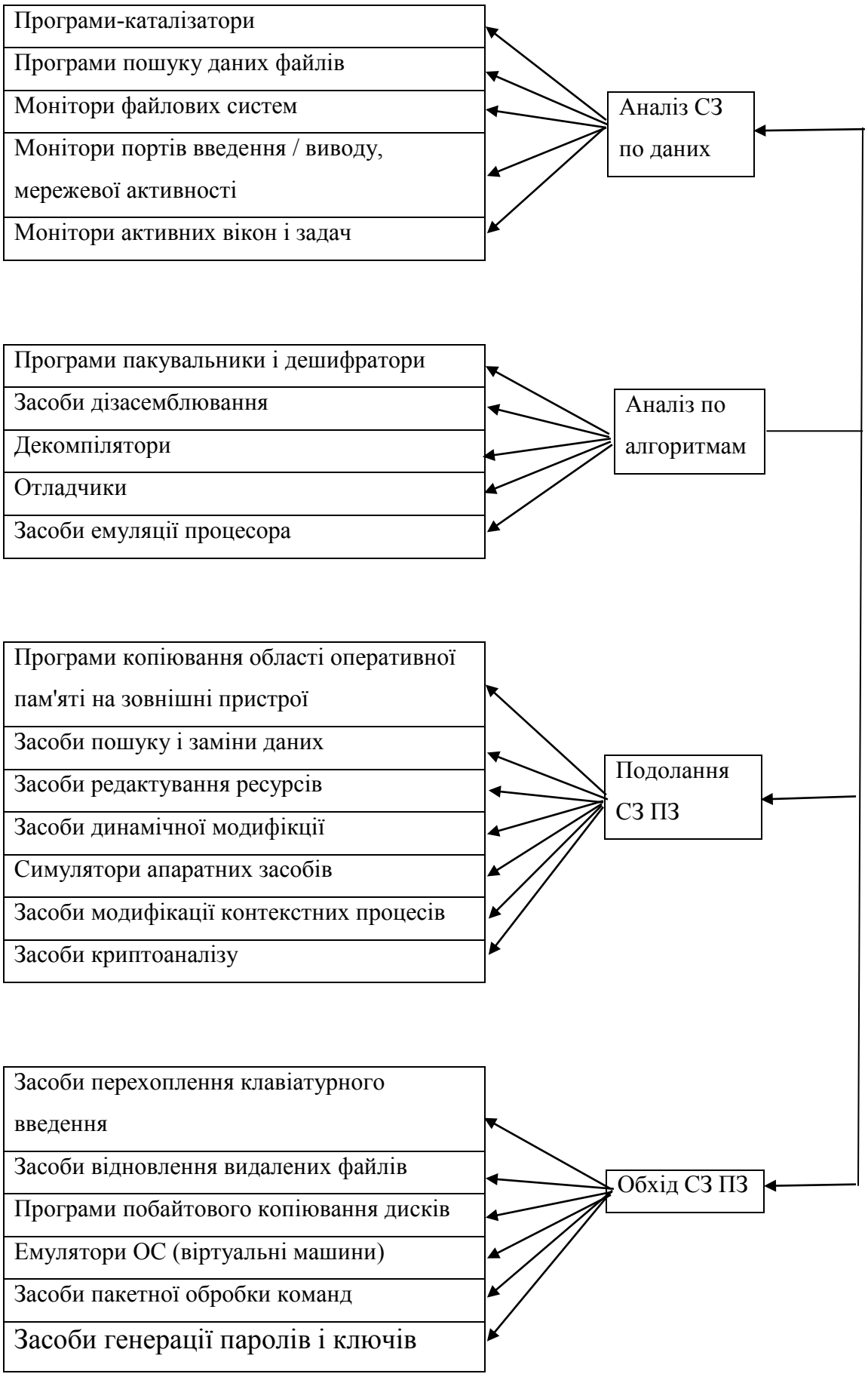
Машинний код є специфічним для конкретної апаратної архітектури. Тому і дизасемблери пишуться спеціально для конкретної апаратної архітектури.

Дизасемблери бувають двох видів - пакетні і інтерактивні.

У *пакетних* дизассемблера аналіз досліджуваної програми проводиться автоматично на основі вибраних налаштувань.

В *інтерактивних* дизассемблера можна контролювати весь процес дизассемблирования. Пакетні дизасемблери набагато простіше в управлінні. Однак вони мають дуже великий недолік - вони практично неспромжні проти простих антіотладочних прийомів і простих захистів. В силу своєї простоти вони свого часу і набули такого поширення, але сьогодні вони практично марні. Вони можуть стати в нагоді для простого дизассемблірування своєї програми, але їх вік вже закінчено [14]. На зміну їм прийшли інтерактивні дизасемблери.

Нижче описані три найпопулярніших програмних засобів для аналізу програм.



Рисинок 1.3 - Засоби дослідження програм

IDA Pro - є одним з найпопулярніших засобів для відновлення вихідного коду і структури програм. Вона є одночасно дизассемблером і отладчиком. На сьогоднішній день IDA є найпотужнішим і найрозвиненішим інтерактивним дизассемблером і дозволяє перетворити бінарний код програми в асемблерний текст, який може бути застосований для аналізу роботи програми. Програма IDA Pro дозволяє хакеру виявити таємно виправлені помилки, пов'язані з безпекою програм, які виконують постачальники програмного забезпечення.

Дослідження коду в IDA Pro визначається трьома можливостями програми:

• потужним засобом аналізу виконуваного коду, вбудованого в дизассемблер. IDA Pro ніколи не робить занадто "самовпевнених" припущень. Привілей на евристичний аналіз надається людині;

• людині надається можливість брати участь в цьому аналізі, уточнювати параметри тих чи інших об'єктів програми, робити виправлення. Таким чином користувач даного інструменту стає актвним учасником процесу дизасемблірования вбудовану мову програмування, вельми близький за своєю структурою до класичного мови С, дозволяє значно нарощувати функціональність даного продукту. Підтримується величезна кількість форматів: Win32 PE ExE, DLL, OCX, DOS EXE, UNIX EXE, MacOS, Java, ARM, dotNET, Xbox, Sony PlayStation, BeOS, OS / 2 і інші.

OllyDbg - безкоштовний 32 бітний відладчик користувацького рівня для операційних систем Windows, призначений для аналізу та модифікації відкомпільованих виконуваних файлів і бібліотек, які працюють в режимі користувача. OllyDbg найпопулярніший відладчик останнім часом, саме їм найчастіше користуються як початківці, так і досвідчені крекери в наші дні. Останнім часом також написано багато статей, що описують його використання [23]. Особливо зручний тим, що дізассемблірует програму "на льоту" і дозволяє вести аналіз програми, коли її вихідні коди

недоступні. Отладчик відображає значення регістрів, може показувати вміст пам'яті, розпізнає процедури, API-функції, переходи, строкові та цифрові константи, є можливість перейменовувати змінні і робити коментарі в дизасемблювати коді, зроблені ним патчі коду відладчик вміє записувати прямо в виконуваний файл.

SoftICE - всім відомий відладчик для Windows, що працює дна рівні ядра. На відміну від прикладного відладчика OllyDbg, SoftICE здатний призупинити всі операції в Windows. Назва відладчика SoftlCE - це вказівка на те, що відладчик в будь-який момент може «заморозити» (від англ. *to ice* - заморожувати) систему і дати повну інформацію і по системі, і по всім працюючим в ній додатків.

З його допомогою можна налагоджувати будь-які програми, що виконуються в операційній системі, в тому числі сервіси і драйвери, що працюють в нульовому кільці захисту. Унаслідок тісної взаємодії відладчика з операційною системою з його допомогою можна отримати багато системної інформації про функціонування операційних систем. Тому SoftlCE просто незамінний для тих, хто вивчає внутрішні механізми функціонування операційної системи Windows. У середовищі дослідників коду SoftlCE вважається кращим отладчиком. Сам відладчик доповнюється також утилітами, головна з яких - це Symbol Loader, яка завантажує виконуваний модуль в пам'ять і здійснює виклик вікна відладчика SoftlCE, тобто встановлює точку зупинки на точку входу програми. При наявності в виконуваному модулі відкладної інформації, яка розпізнається загрузчиком, він також завантажує її в відладчик. Усунення несправностей дозволяє налагоджувати виконуваний код не тільки на автономному комп'ютері, але і виробляти налагодження з віддаленого комп'ютера, з'єднаного з першим за допомогою COM-порту.

Крім цих двох основних типів інструментів дослідження широко використовуються:

декомпілятори - кошти, яке дозволяє перетворити код на мові асемблера або

машинний код в вихідний код на високорівневої мовою, наприклад на C. Також існують декомпілятори для перетворення коду на проміжних мовами на зразок байт коду Java і коду на мові MSIL в вихідний код на зразок Java. Гарна пара дизассемблер / декомпілятор може використовуватися для компіляції програми, що генерують з виконуваного коду програму на мові високого рівня; шістнадцятиричні редактори - програми для перегляду, вивчення, аналізу та редагування внутрішнього устрою виконуваних файлів. Одним з представників таких програм є програма PE Explorer призначена для використання в різних IT-областях: у розробці та тестуванні програмного забезпечення, у зворотній розробці (реверс-інжинірингу) з метою відновлення алгоритмів, в антивірусних лабораторіях і криміналістиці, для виявлення вразливостей і проведення аудиту безпеки для файлів, отриманих з неперевірених джерел;

трасувальники - програми, спочатку запам'ятовують кожну інструкцію, що проходить через процесор, а потім переводять набір інструкцій у форму, зручну для статичного дослідження, автоматично виділяючи цикли, підпрограми і тому подібне;

спостерігаючі системи - програми, що запам'ятовують і аналізують трасу вже не інструкції, а інших характеристик, наприклад викликаних програмою переривання. розпаковують - програми розкривають упаковані (виконувані стислі) виконуються файли;

модифікатори - програми для внесення виправлень в виконуваний файл або бібліотеки DLL;

дампер - програми для аналізу вмісту ділянок оперативної пам'яті.

Висновки

У цьому розділі описуються науково-практичні основи процесу обфускації, виконано короткий опис заплутують перетворень вихідного

коду програм, описані методи і програмні інструменти для дослідження і аналізу програм.

РОЗДІЛ 2. МЕТОДИКА ОБФУСКАЦІЇ КОДУ З ШИФРУВАННЯМ АДРЕС

2.1 Методи обфускації покажчиків на дані і функції

В цьому розділі дипломної роботи описується метод обфускації критично важливого програмного коду заснованого на шифруванні адресу даних використаних в цій ділянці програми.

В основу цього методу покладено прийом обфускації покажчиків на дані та функції, запропонований в [13] і методика заплутування коду програм пропонована в [1].

В роботі [l3] Крис Касперски описав два прийоми для обфускації покажчиків на дані та на функції. Перший з них (для обфускації покажчиків на дані) здійснюється на етапі написання вихідного коду.

Детально розглянемо механізм цього методу.

Процесори підтримують операнди трьох типів: регістр, безпосереднє значення, безпосередній покажчик (рисунок 2.1.). Тип операнда явно задається в спеціальному полі машинної інструкції і тому ніяких проблем в ідентифікації типів операндів не виникає. Крім того, процесори підтримують два види адресації пам'яті: безпосередню і непряму. Тип адресації визначається типом покажчика. Якщо операнд - безпосередній покажчик, то і адресація безпосередня. Якщо ж операнд-покажчик - регістр, - така адресація називається непрямою. Фактично будь-яка змінна в вихідному коді програми після компіляції стає дороговказівником на ту ділянку оперативної пам'яті де зберігається її значення.

Існує два типи вказівників - покажчики на дані і покажчики на функцію. Покажчики на дані використовуються для вилучення значення осередку пам'яті і зустрічаються в арифметичних командах і командах пересилання. Покажчики на функцію використовуються в командах непрямого виклику і, рідше, в командах непрямого переходу.

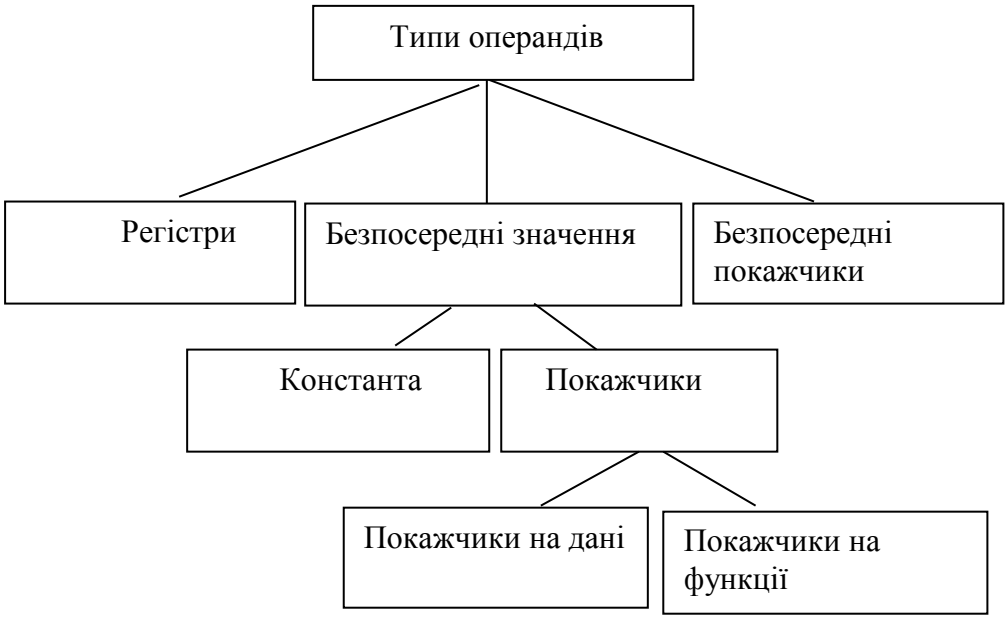


Рисунок 2.1 -Типи операндів

C/C++ і деякі інші мови програмування допускають виконання над покажчиками різних арифметичних операцій, ніж серйозно ускладнюють ідентифікацію типів безпосередніх операндів.

Над покажчиками виконується лише обмежена кількість математичних операцій. Так, абсолютно безглуздо додавання двох покажчиків, а вже тим більше множення або ділення їх один на одного. Віднімання — справа інша. Використовуючи той факт, що компілятор має функції у пам'яті відповідно до порядку їх оголошення в програмі, можна обчислити розмір функції, віднімаючи її покажчик від вказівника на наступну функцію . Такий трюк зустрічається в пакувальників (розпакувальників) виконуваних файлів, захистах з самомодифікуючим кодом, але в прикладних програмах використовується рідко. Застосовувати такі операції можна не тільки до випадків «покажчик» + «покажчик» - покажчик може поєднуватися і з константою. Причому таке поєднання настільки популярне, що мікропроцесори серії 80x86 навіть підтримують для цього спеціальну адресацію — базову. При складанні покажчика з константою більшість компіляторів на перше місце поміщають покажчик, а на друге — константу, яким би не було їх розташування у вихідній програмі.

Ідея цього прийому обфускації полягає в тому, що на початку всі дані представляються у вигляді вказівників, а потім виконується шифрування/розшифрування цих покажчиків. Ці операції здійснюються оптимізуючим компілятором, який підтримує математичні операції (типу додавання і віднімання) над покажчиками. При цьому шифрування проводиться на стадії компіляції, а розшифровка цих покажчиків проводиться в манері, не підтримуваної ні IDA-Pro, ні популярними отладчиками покажчиків.

В даний час для розробки програм застосовуються тільки оптимізуючі компілятори. Оптимізуючий компілятор — це компілятор, в якому використовуються різні методи отримання більш оптимального програмного коду при збереженні його функціональних можливостей. Найбільш поширені цілі оптимізації: скорочення часу виконання програми підвищення продуктивності, компактифікація програмного коду, економія пам'яті, мінімізація енерговитрат, зменшення кількості операцій введення-виведення. Оптимізація, як правило, реалізується за допомогою послідовності оптимізуючих перетворень і алгоритмів, які приймають програму і змінюють її для отримання семантично еквівалентного варіанту, який більш ефективний з точки зору якого-небудь набору цілей оптимізації. При застосуванні неоптимизирующих компіляторів аналіз коду відносно простий і цілком доступним для розуміння навіть новачкам у програмуванні, а оптимізують компілятори генеруючи дуже хитрий, заплутаний і витіюватий код.

Процеси шифрування/розшифровки створюються застосуванням наступних процедур при програмуванні вихідного коду:

• спочатку адреса, що вказує на розташування даних, що змінюється операцією додавання з деякою константою. Тим самим адреса розташування самих даних стає невідомим.

• при доступі до цих вказівників адреса змінюється операцією віднімання, але вже не з константою, а з глобальною або статичною

змінною.

Таким чином "ключі шифрування" змінних фактично знаходяться в статичних змінних, а "ключі розшифрування"- у глобальних змінних. На рисунку 2.2 показана схема шифрування покажчиків на дані.

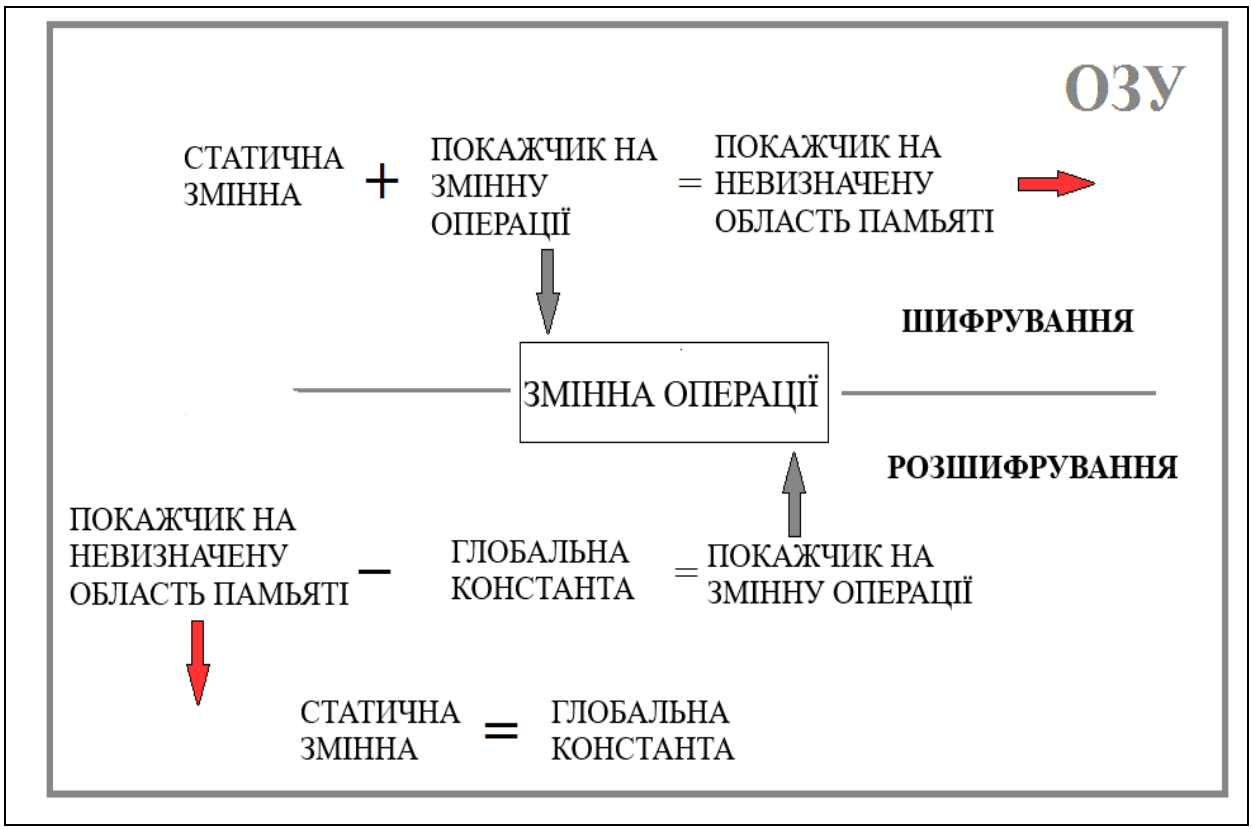


Рисунок 2.2 - Схема шифрування і розшифровки покажчиків

Так як компілятори не оптимізують статичні та глобальні змінні, то й отримати істинні значення змінених адрес за допомогою відладчика неможливо. Розшифрувати значення покажчика цілком можливо (достатньо проаналізувати дизассемблерний код), але на це йде час і крім того, всі засоби для побудови графа потоку керування не спрацьовують. Запропонована схема захисту даних заснована на властивості статичних змінних і констант. Розглянемо їх більш докладно .

Статичні змінні — це різновид глобальних змінних, але з обмеженою областю відомостей - вони доступні тільки з тій функції, в якій були

оголошені. У всьому іншому статичні та глобальні змінні повністю збігаються.

Якщо глобальну або статичну змінну передати функції по посиланню як параметр цієї функції ,то вона буде адресуватися опосередковано — через вказівник. А це означає, що всіх звернень до цим змінним простим контекстним пошуком знайти неможливо. У цьому випадку глобальні змінні не знайде їх і IDA Pro - для цього потрібен був би повноцінний емулятор процесора або хоча б його основних команд [14].

Константи - це такі ж безпосередні значення як і покажчики (рисунок 2.1). Фактично константа це покажчик на область пам'яті де зберігається її значення. Якщо наприклад в програмі записано число 0x123AFD00 то неможливо визначить що визначає це число - константи або покажчик. При цьому однакові вирази можуть інтерпретуватися по різному,наприклад:

• 0x0045AB + 0x123AFD00 – додавання двох цілих чисел. Результат операції нове ціле число.

• 0x0045AB + 0x123AFD00 – складання покажчика з цілим числом (зміщення покажчика). Результат операції новий покажчик.

Це і породжує фундаментальну проблему асемблювання — синтаксичну нерозрізненість констант і змішань (ближніх покажчиків). Таким чином основна ідея запропонованого методу шифрування даних полягає в тому, що після оптимізазії компіляції вихідного коду програм, написаних на мовах високого рівня, в виконуваний код програми (машинний код) синтаксично невиразні константи і покажчики .

Зашифрувати покажчики на функції набагато складніше, оскільки, оптимізатори не підтримують математичних перетворень над ними. Кріс Касперски пропонує шифрування покажчиків на функцію проводити шляхом доопрацювання откомпилірованного коду вже після трансляції, зашифрувавши покажчики безпосередньо в двійковому файлі, виконуючи розшифровку вже в самій програмі. Вся проблема в тому, як знайти дороговказівники в откомпилированном коді? Найпростіше - загнати

покажчики в структуру, попереджену спеціальним маркером — текстовим рядком або константою з унікальним вмістом, після чого залишається тільки знайти цей маркер в програмі і зашифрувати наступні за ним покажчики, що можна зробити як вручну , так і автоматично з допомогою нескладної програми.

Використання цих двох прийомів обфускації не гарантує повного затемнення кода - при прогоні програми під відладчиком хакер дізнається значення регістрів, що визначають яка функція викликається. Але наочність дизассемблерного лістингу буде необоротно втрачено, а використання механізмів реконструкції потоку управління збільшується час аналізу програми на порядок – другий [14].

2.2 Методика обфускації коду з використанням непроникних функцій

В роботі [1] за аналогією з визначенням непроникної змінної і непроникного предиката (див. розділ 1.3.4) визначено поняття непроникної функції і запропонований метод обфускації з її використанням. Приклад вихідного коду програми, обусцірованного запропонованим методом, показаний на рисунку 2.3.

Непроникною називають функцію, результат якої відомий на етапі обфускації, але його складно визначити при дослідженні обфусцірованної програми. У цьому методі кожної змінної (константи), використаної в програмі, ставиться у відповідніть деякий об'єкт, який буде зберігати її значення (рисунок 2.3-а). Далі створюється глобальний масив всіх об'єктів, використовуваних в програмі і таким чином, кожної змінної або константі, використовують в програмі, буде присвоєно унікальний індекс в цьому глобальному масиві.

Далі визначається в програмі функція f яка є непроникною і її входом є ціле число - унікальний ідентифікатор деякої точки в програмі, в якій відбувається звернення до змінної, виходом - індекс в масиві об'єктів. У

вихідній програмі замінюються всі звернення до змінних/констант на звертання до відповідних об'єктів таким чином, що кожен об'єкт адресується як певний елемент глобального масиву об'єктів і номер об'єкта в масиві визначається динамічно за допомогою виклику непроникною функції f. Перетворений після першого етапу код програми показаний на рисунку 2.4.

Таким чином, після першого етапу всі прямі звернення до змінних в коді програми замінюються на непрямі звернення, що ускладнює зворотний аналіз програми. При достатній непроникності функції f завдання визначення, який саме об'єкт є параметром при виклику тієї чи іншої функції, є практично нерозв'язною.

На другому етапі створюється глобальний масив func всіх функцій, що викликаються в програмі. Кожній функції у відповідність втупився унікальний індекс в цьому масиві. Аналогічно функції f вводиться непроникна функціюя g, яка, приймаючи деякий унікальний цілочисельний аргумент, буде повертати номер функції в масиві функцій. Аналогічно тому, як це робилося на першому етапі, замінюються всі прямі виклики функцій на непрямі (рисунок 2.3).

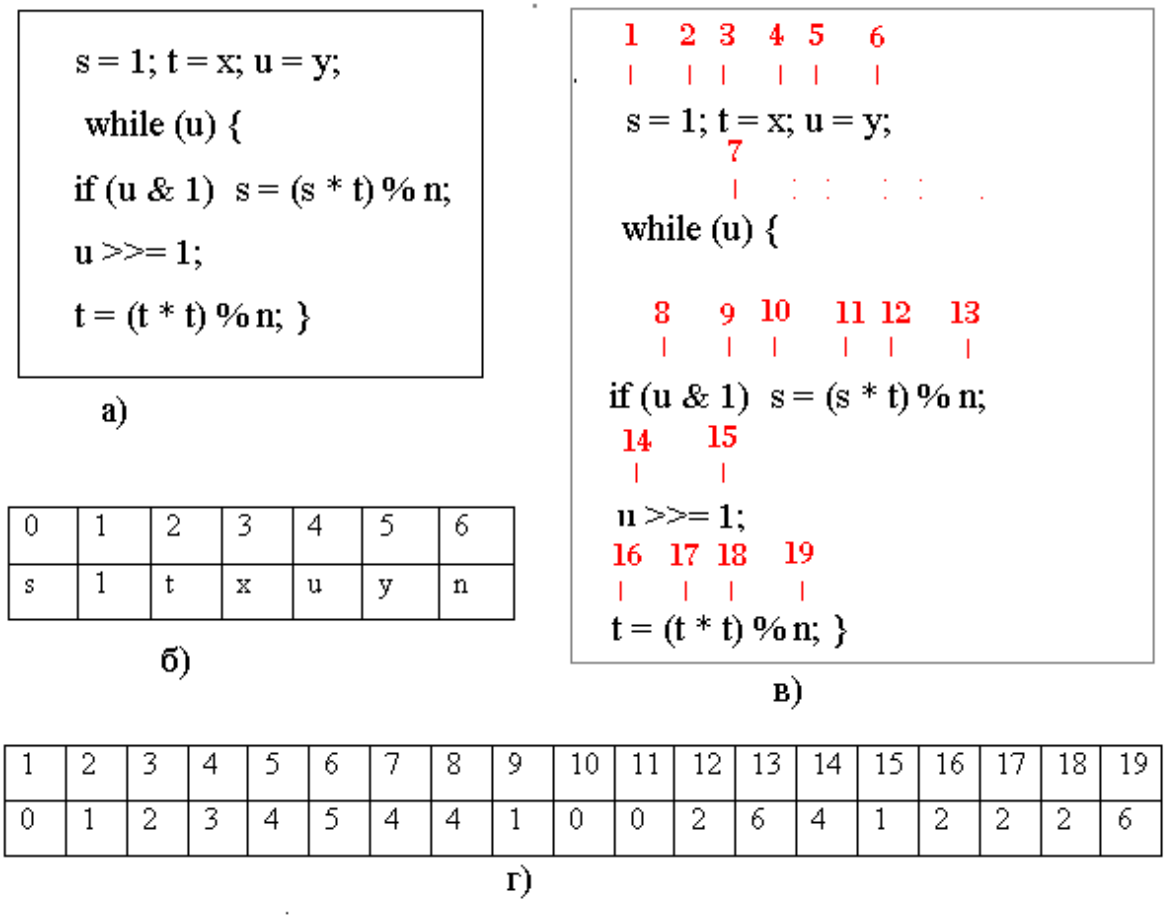


Рисунок 2.3 - Перший етап обфускації

```
obj[f(1)] = obj[f(2)];
obj[f(3)] = obj[f(4)];
obj[f(5)] = obj[f(6)];
while (obj[f(7)]) {
  if (obj[f(8)] & obj[f(9)])
    obj[f(10)] = (obj[f(11)]*obj[f(12)])%obj[f(13)];
  obj[f(14)] >>= obj[f(15)];
  obj[f(16)] = (obj[f(17)]*obj[f(18)])%obj[f(19)];
}
```

Рисунок 2.4 - Результат першого етапу обфускації

Заміна всіх прямих викликів функцій на непрямі, які визначаються на етапі виконання, робить недоступною для статичного аналізу інформацію про послідовність викликів функцій. Фактично єдина інформація, доступна атакуючому після другого етапу - це послідовність виконання умовних операторів і операторів циклу (if, while, switch і т.п.), проте цієї інформації

недостатньо для відновлення логіки роботи програми, так як аргументи цих операторів захищені непроникненою функцією. Як вказують автори цього методу, його стійкість на пряму залежить від стійкості використовуваних непроникних функцій. При цьому наголошується, що проблема побудови таких функцій виходить за рамки даної роботи, дають посилання на приклади їх побудови і вказується що описаний метод дозволяє досягти більш високого рівня приховування даних за рахунок непрямих викликів непроникних функцій.

0	1	2	3	4	5
=	!=0	&	*	%	>>=

а)

```
func[g(1)](obj[f(1)], obj[f(2)]);
func[g(2)](obj[f(3)], obj[f(4)]);
func[g(3)](obj[f(5)], obj[f(6)]);
while (func[g(4)](obj[f(7)])) {
  if (func[g(5)](obj[f(8)], obj[f(9)]))
    func[g(8)](obj[f(10)], func[g(7)](func[g(6)](obj[f(11)], obj[f(12)]),
      obj[f(13)]));
  func[g(9)](obj[f(14)], obj[f(15)]);
  func[g(12)](obj[f(16)], func[g(11)](func[g(10)](obj[f(17)], obj[f(18)]),
    obj[f(19)]));
}
```

б)

Рисунок 2.5 - Другий етап обфускації

Якщо проаналізувати суть запропонованого методу, то фактично він зводиться до розробки непроникних функцій, на вхід яких подається індекс масиву, де зберігаються параметри обфускації, а їх виходом є індекс масиву, де зберігаються змінні і покажчики на функцію. При досить великій кількості операторів в коді програми, які підлягають обфускації, ця задача аналітичного подання S - блоку підстановок. У криптографії S-блоком називають підстановку: відображення m-бітових входів на n-бітові виходи.

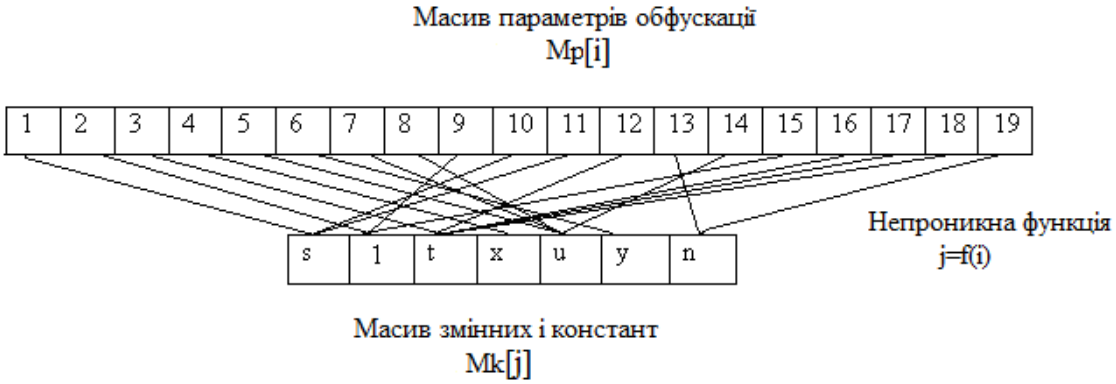


Рисунок 2.6 - Схема заміщення

Дуже важливою властивістю S-блоку є лавинний ефект: скільки вихідних бітів S-блоку змінюється при зміні деякого підмножини вихідних бітів. Вибір хороших S-блоків є складним завданням [8], існує безліч різних ідей, як краще зробити це.

Можна виділити чотири основні підходи:

• Випадково вибрати. Стійкість S-блоків зростає, якщо вони одночасно є і випадковими, і залежними від ключа.

• Вибрати і перевірити. У деяких шифри властивості S-блоків, генерованих випадковим чином, перевіряються.

• Розробити вручну. При цьому математичний апарат використовується вкрай незначно: S-блоки створюються з використанням інтуїтивних прийомів.

• Розробити математично. S-блоки створюються відповідно до математичних законів, тому вони мають гарантовану надійність. Щодо використання S-блоку в якості непроникної функції для довільного коду програм, то по видимому єдиним способом його уявлення, є табличне представлення. При цьому таку таблицю треба буде все одно зберігати в програмі, що і робить даний метод обфусації ненадійним.

2.3 Метод обфускації коду з шифруванням адрес

Метод захисту, запропонований в цій магістерській роботі, передбачає посилення обфускації за рахунок:

• представлення даних у вигляді покажчиків;

• використання шифрування покажчиків (описано в розділі 2.1);

• заплутування коду програми шляхом об'єднання всіх типозованих покажчиків в один масив невизначених (описано в розділі 2.3.)

• зворотного перетворення невизначених покажчиків в типозовані.

Тим самим реалізується підхід, що полягає в застосуванні великої кількості простих прийомів заплутування, кожен з яких окремо легко розплутується, але в сукупності вони роблять завдання аналізу коду (особливо статичного аналізу) досить складною.

Реалізація запропонованого методу здійснюється в послідовному виконанні п'яти етапів:

• Приведення всіх змінних критичного коду програми до покажчиків на дані.

• Опис всіх алгебраїчних або логічних виразів у вигляді функцій.

• Перетворення всіх цих явних покажчиків в невизначені покажчики і об'єднання їх в один масив.

• Шифрування елементів масиву невизначених покажчиків за допомогою заздалегідь визначених глобальних констант.

• Використання шифрованих елементів масиву невизначених покажчиків і заздалегідь визначених статичних змінних глобальних констант для опису всіх алгебраїчних або логічних виразів критичного коду.

Розглянемо докладно всі ці етапи.

1) Приведення всіх змінних критичного коду програми до покажчиків на дані не представляє ніяких труднощів, так як в мові високого рівня C/C ++ існує операція взяття адреси цієї змінної, яка може бути поміщена в

відповідний збірний покажчик.

2) Будь-яке алгебраїчне або логічне вираження оформлюється у вигляді функції.

В загальному випадку можна створити покажчик на ці функції, використовувати його також як і покажчики на дані, але не шифрувати його.

3) Перетворення всіх цих явних покажчиків в невизначені покажчики і об'єднання їх в один масив може здійснюватися в такий спосіб. Створюється масив невизначених покажчиків, розмір якого дорівнює кількості раніше створених раніше покажчиків на змінні. Так як невизначеним вказівникам можуть присвоюватися будь-які адреси незалежно від типу покажчика, то в цей масив заносяться значення адрес отриманих при виконанні етапів 1 і 2.

4) Процес шифрування полягає в зміні значень адрес, які вказують на дійсне розташування значень змінної в пам'яті, за допомогою додавання їх з глобальними константами програми. Довільний вибір значень цих констант, які фактично є «ключами шифрування», може привести до того що «зміщені» значення «зашифрованих» покажчиків будуть посилатися на неприпустимі області пам'яті, що виключає динамічне дослідження коду програми.

5) Процес розшифрування покажчиків проводиться безпосередньо при використанні цих покажчиків в якості параметрів функцій реалізують обфусціруємий оператор критичного коду програми. У цьому випадку проводиться віднімання з покажчика значення статичної змінної, значення якої дорівнює значенню глобальної константи при його шифруванні.

На рисунку 2.7 показана загальна схема описаного методу обфускації стосовно до одного оператора критичного коду програми.

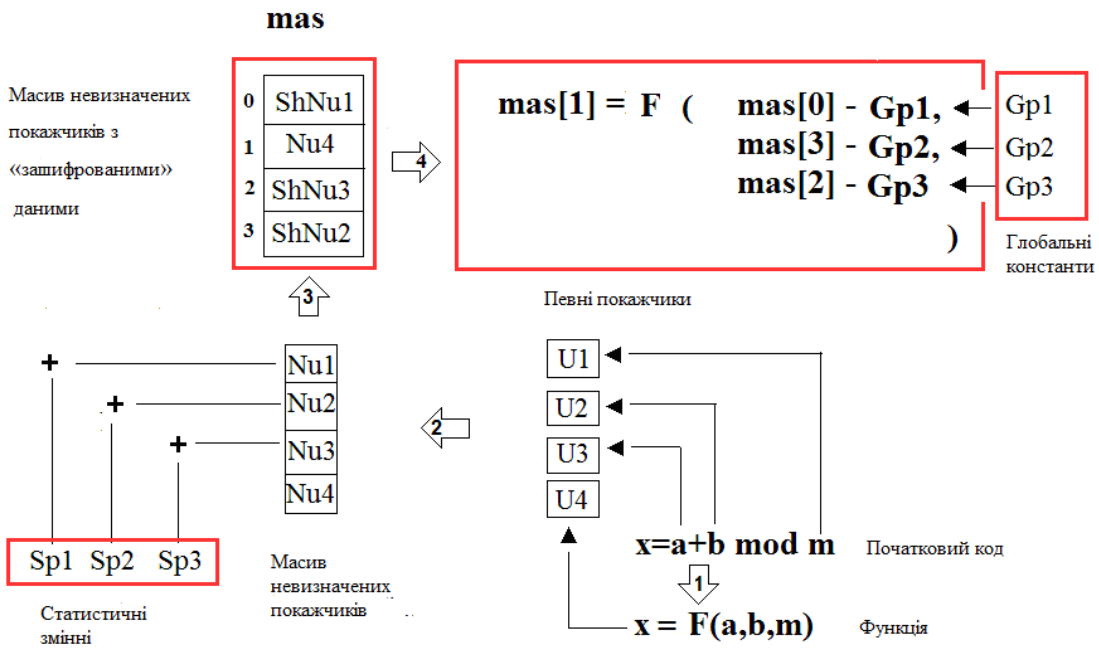


Рисунок 2.7 - Схема обфускації для однієї операції

Таким чином цей метод посилює заплутування ефекту за рахунок спільного застосування раніше відомих і окремо використовуваних способів - шифрування покажчиків, об'єднання їх в один масив і розшифровки покажчиків в момент виклику функцій, що визначають значення обфусціруємих виразів коду програми.

2.4 Тестування методу

З метою перевірки запропонованого методу обфускації розроблена програма для обчислення виразу x = a + b mod m, де a, b і m цілі числа.

```
int main()
{   int a=12;int b=27;int m=5;
// обчислення виразу
     cout << a+b %m << endl;
   return 0;}
```

Ця програма була обфусцірована за допомогою запропонованого методу. Скріншон обфусцірованної програми показаний на рис 2.8

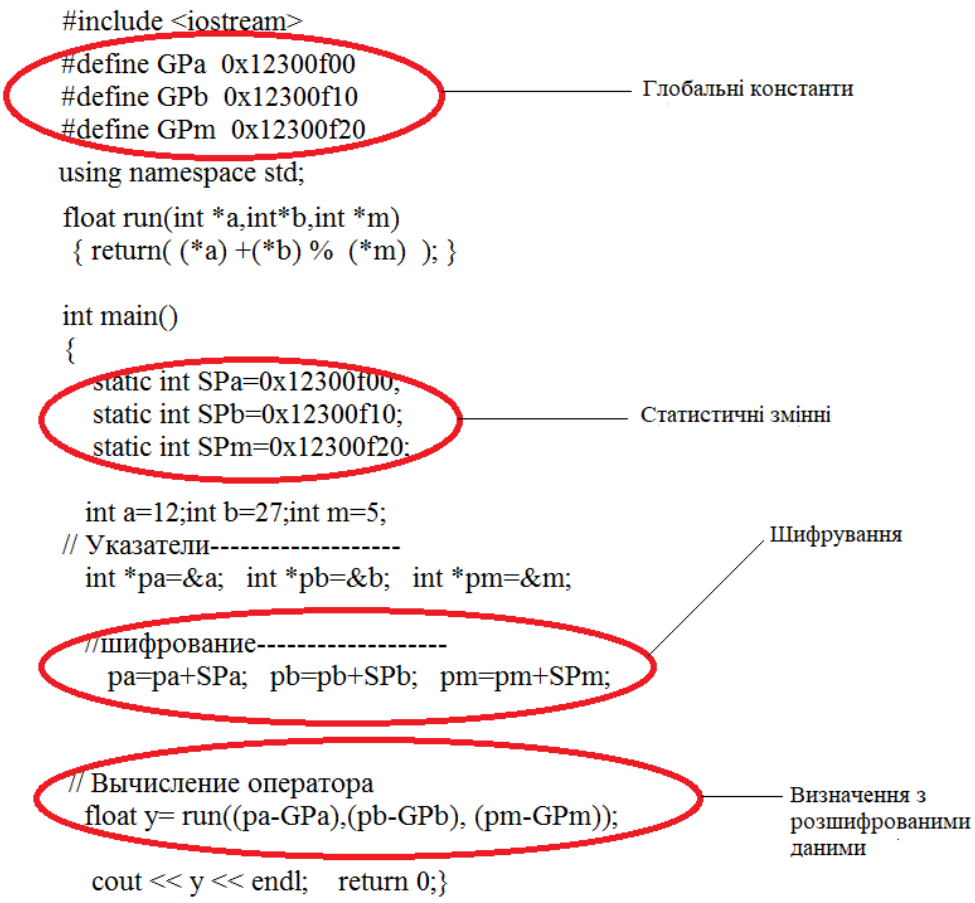


Рисунок 2.8 - Обфусціровна тестова програма

На рисунку 2.9 показані дезассемблірування коди цих програм. Попередній аналіз цих кодів показує, що їх основна відмінність полягає в присутності в обфусцірованному варіанті команд з непрямої адресацій. Це означає що простежити дійсні значення операндів цих команд набагато складніше, ніж посилання на регістри процесора в подібних командах необфусцірованного варіанту. Також слід зазначити природне збільшення виконуваного коду в обфусцірованному варіанті програми.

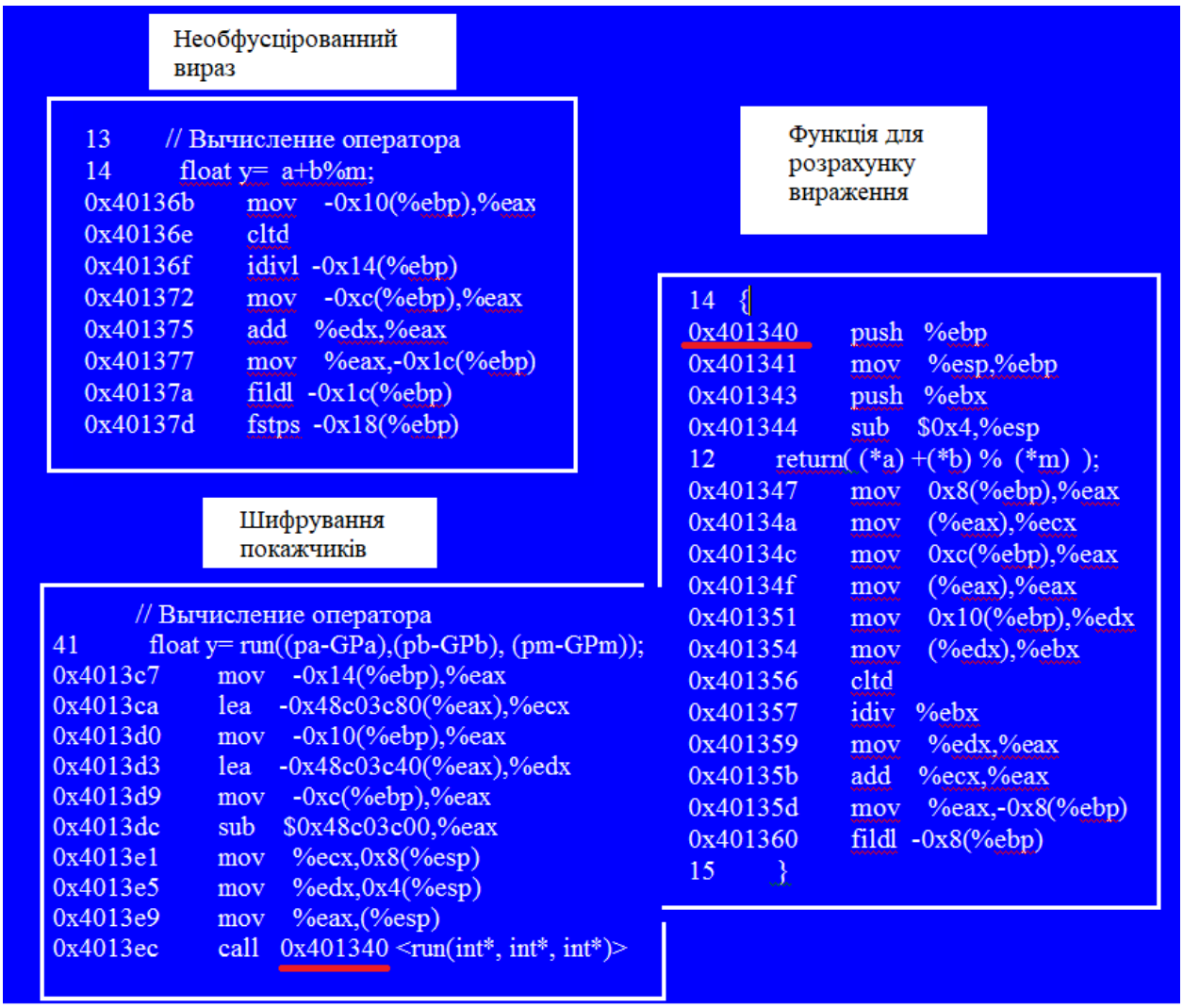


Рисунок 2.9 - Дезассемблірування кодів тестових програм

Надалі слід провести більш детальний аналіз виконуваного коду за допомогою спеціалізованого відладчика і трасувальника.

2.5 Особливості програмної реалізації

Об'єднання в один масив даних і функцій, запропонований в даному методі обфускації, може бути здійснений в такий спосіб:

1) Кожна константа або змінна описується в певному класі в якості даних класу.

2) Створюються об'єкти цих класів.

3) Адреси цих об'єктів поміщаються в масив невизначених покажчиків. Цей масив і буде об'єднаним масивом даних і функцій.

4) Кожна операція в критичному коді також описується методом у відповідному класі. При цьому параметрами цих методів є адреси об'єктів, що містять константи і змінні.

5) Створюються об'єкти класів які містять методи операцій і адреси цих класів також додаються в масив невизначених покажчиків. Таким чином, в цьому масиві невизначених покажчиків поміщаються дані і методи.

6) При реалізації обфусцірованного коду викликаються відповідні методи об'єктів з обов'язковим явним перетворенням відповідних типів методів і їх параметрів.

У наступному прикладі викликається метод Func4 класу FUN4 знаходиться в 12-му елементі масиву невизначених покажчиків mas Параметрами цих методів є об'єкти класів INT_t і INT_n знаходяться в 6-му і 8-му елементах масиву mas.

int t = (((FUN4 *) mas [12]) -> Func4 ((INT_t *) mas [6], (INT_n *) mas [8]));

де (FUN4 *), (INT_t *), (INT_n *) - явні перетворення типів.

В додатку ___ наведено вихідний і його дизасемблірованний код програми, а в додатку ППР наведено обфусцірованний вихідний код і його дизасемблірований вид.

Висновки

1) Розглянуто механізм обфускації покажчиків на дані, що заснована шифруванні адрес даних використовуваних в цій ділянці програми;

2) Розглянуто метод обфускації з використанням непрозорих предикатів, в якому використовується прийом об'єднання покажчиків на дані в окремі масиви покажчиків.

3) Запропоновано метод обфускації коду з шифруванням адрес масиву

невизначених покажчиків з їх подальшою розшифровкою при обчисленні окремих виразів критичного коду програми. Даний метод посилює заплутування ефекту за рахунок спільного застосування раніше відомих і окремо використовуваних способів обфускації.

4) Виконано програмне тестування запропонованого методу стосовно до одного обфусціруємого прикладу і запропонований алгоритм для програмної реалізації всього методу.

3. ЕКОНОМІЧНИЙ РОЗДІЛ

3.1 Вступ

Метою дослідження є аналіз витрат на захист програм методом обфускації.

Справжній дипломний проект присвячений розробці методу захисту програм від вивчення принципу їх роботи (процесу реверсивної або зворотної інженерії). Одним з методів такого захисту програм, який дозволяє ускладнити процес реверсивної інженерії називається обфускація (заплутуванням) коду захисного програмного продукту. Суть процесу обфускації полягає в тому, щоб заплутати програмний код і усунути більшість логічних зв'язків в ньому, тобто трансформувати його так, щоб він був дуже важкий для вивчення і модифікації сторонніми особами (будь то зломщики, або програмісти які збираються дізнатися унікальний алгоритм роботи захисної програми). Одна обфускація не забезпечує повний захист програмних продуктів, так як вона не надає можливості запобігання нелегального використання програмного продукту. Тому обфускацію зазвичай використовують разом з одним з існуючих методів захисту (шифрування програмного коду і т.д.), це дозволяє значно підвищити рівень захисту ПП в цілому.

Існує багато методів визначення ефективності застосування того чи іншого процесу обфускації, до конкретного програмного коду. Ці методи прийнято розділяти на дві групи: аналітичні та емпіричні. Аналітичні методи грунтуються на трьох величинах характеризують наскільки ефективний той чи інший процес обфускації:

1) Стійкість - вказує на ступінь складності здійснення реверсивної інженерії над кодом який пройшов процес обфускації.

2) Еластичність - вказує на те наскільки добре даний процес обфускаціїї захистить програмний код від застосування деобфускаторів.

3) Вартість перетворення - дозволяє оцінити, наскільки більше потрібно

системних ресурсів для виконання коду який пройшов процес обфускації, ніж для виконання оригінального коду програми.

Їх найбільш ефективно застосовувати при порівнянні різних алгоритмів обфускації, але при цьому вони не можуть дати абсолютної відповіді на питання наскільки ефективним є застосування того чи іншого алгоритму, саме до даного програмного коду. Емпіричні ж методи можуть дати сприятливу відповідь на таке питання, тому що вони ґрунтуються на статистичних даних отриманих в результаті досліджень. Для проведення одного з таких досліджень потрібна група людей (якнайкраще знайомих, з реверсивної інженерії), фрагмент коду захисної програми, і набір різних алгоритмів обфускації. Результати такого дослідження будуть включати в себе мінімальну кількість часу, який буде потрібний групі людей, для того щоб вивчити кожен фрагмент коду пройшовшого один з алгоритмів обфускації.

3.2 Маркетингові дослідження

На сьогоднішній день створено досить багато діючих розвинених систем обфускації програм, в яких використовуються різні комбінації описаних в розділі 1 методів обфускації. Індивідуальні особливості кожної з цих систем визначаються своєю стратегією застосування обфускціованих перетворень. Ці системи призначені для:

- безкоштовного використання розробниками ПЗ;
- платного застосування;
- виконання обфускації програм замовника спеціалізованими фірмами.

Застосування таких сторонніх систем пов'язано з одним важливим недоліком, а саме тим, що розробники ПО використовують такі системи в яких достеменно не відома надійність їх застосування. Крім цього в разі

виконання обфускації програм зовнішніми спеціалізованими фірмами існує ймовірність втрати конфіденційності представленим їм вихідного коду програмного забезпечення. Слід зазначити що безкоштовні обфускатори дуже слабкі і придатні тільки для простого перейменування.

У таблиці [3.1] наведені відомі системи обфускації і їх рейтинг в 10-бальною оцінкою.

Таблиця 3.1 Рейтинг відомих обфускаційних програм

Назва	**Рейтинг**	**Ціна**
CodeVeil	5	$900/25200грн
NET Reactor	6	$180/5040грн
SmartAssembly	7	$795/22260грн
CilSecure	8	<$1000/28000грн
DesaWare	8	$1500/32000грн
Dotfuscator	8	$1900/53200грн
Obfuscator.NET	4	$200/5600грн
Goliath.NET	5	$115/3220грн
Phoenix Protector	4	Free/безкоштовно
BitHelmet	4	Free/безкоштовно
Aspose.Obfuscator	3	Free/безкоштовно
C# Source Code Obfuscator	3	Free/безкоштовно

3.3 Визначення трудомісткості обфускації

Запропонований метод обфускації передбачає модифікацію окремих операторів в коді програми. В даному випадку під оператором розуміється виконання окремої операції (привласнення, арифметичної, логічної, бітової і ін.) Над деякими операндами цієї операції. Наприклад, в операції y = K << b; використовуються дві операції (привласнення і зсунення бітів вліво) і три операнда (y, K і b).

Для обфускації однієї операції в запропонованому методі, пропонується створення окремих класів в яких описуються змінні і константи, які є операндами операції, а також класи містять методи реалізуючі кожну окрему операцію в операції.

Таким чином, трудомісткість обфускації деякого коду програми залежить від кількості різних операндів і операцій в коді. При цьому трудомісткість таких перетворень для операндів і операцій відрізняється один від одного незначно і може бути прийнята однаковою.

Розрахуємо трудомісткість обфускації для одного оператора або операції в коді програми:

$$T = tmз + tв + ta + tпр + tопр + tд = 19+74+74+162+74+74=477 \text{люд/годин} \quad (3.1)$$

де, $tmз$ = 19 люд/годин - витрати праці на підготовку і опис поставленого завдання;

ta = 74 люд/годин - витрати праці на дослідження алгоритму рішення завдання;

$tпр$ = 162 люд/годин - витрати праці на обробку блок-схеми алгоритму;

$tв$ =74 - витрати праці на вивчення ТЗ, літературних джерел за темою, тощо;

$tопр$ = 74 – витрати праці на опрацювання програми;

$tд$ - витрати праці на підготовку документації по завданню.

3.4 Витрати на створення методу обфускації

Витрати на розробку програмного забезпечення Кпо включають витрати на заробітну плату виконавця програми Ззп та вартість машинного часу, необхідного для налагодження програми на ЕОМ [8]:

Кпо = Ззп + Змч =15502,5 + 954 = 16456.5 *грн.*, (3.2)

Заробітна плата виконавця визначається за формулою:

Ззп = t * Зпр = 477 * 32.5 = 15502,5 *грн.,* (3.3)

де *t* - загальна трудомісткість побудови моделі загроз; Зпр =28,60 - мінімальна заробітна плата програміста з нарахуваннями, грн/годину.

Виходячи з середньої місячної заробітної плати програміста 6000 грн (в тому числі ЄСВ) при 22 робочих днів у місяці отримаємо

ЄСВ становить 22% від ЗП і дорівнює 273.

Спр = 5727 / (22* 8) = 32.5 *грн./год.*

Вартість машинного часу, необхідного для налагодження програми:

Змч = t * Смч = 477 * 2 = 954 *грн.,* (3.4)

Де t- трудомісткість налагодження програми на ЕОМ, що визначається за формулою (3.1), людино-годин;

$C_{МЧ}$ - вартість 1-го машино-години, грн. / год.

$$C_{МЧ} = W_{Ч} \cdot Ц_{Э} + \frac{Ф_{ОСТ} \cdot Н_{А.ЭВМ}}{F_{г}} + \frac{Ф_{пр.ПО} \cdot Н_{А.ПО}}{F_{г}} \quad (3.5)$$

При розробці ПО на комп'ютері наступної конфігурації: Duron 2100 \ 512Mb DDR \ 40Gb IDE \ Video Int \ 50x Asus \ Sound Int \ Monitor Samsung 15 '\ Колонки \ Mouse \ Pad \ FDD 3,5 ", який був придбаний в січні 2015 р . за вартістю 12380 гривень. На 01.01.2018 залишкова вартість даного комп'ютера становить:

$$Ф_{ОСТ} = Ф_{П} - Ф_{П} \cdot Н_{а\ ЭВМ}, \quad (3.6)$$

де $Ф_п$ - первісна вартість комп'ютера, грн .;

$Н_{а.ЭВМ}$ - річна норма амортизації, яка становить 50%.

Підставивши початкову вартість і річну норму амортизації ЕОМ, отримуємо:
$Ф_{ОСТ}$ = 12380 - 12380 * 0,5 = 6190 грн.

У вартість ПО включається операційна система "Windows 2010 Professional" вартістю 1 798 грн.Так як операційна системи Windows була придбана в січні 2016 р то річна норма амортизації на ПО $Н_{а.ПО} = 60\%$.

$Ф_{пр.\ ПО}$ =1798 – 1798 * 0,5 = 899 грн.

$F_Г$- річний фонд робочого часу (при 40-а годинному робочому тижні

$F_Г$-= 2008 годин);

W - потужність комп'ютера, кВт · год;

$Ц_э$ - тариф на електроенергію, $Ц_э$ = 0,9 грн. / кВт · год.

Виходячи з того, що комп'ютер наведеної конфігурації споживає

0,30 кВт · год:

$С_{МЧ}$ = 0.3 * 0.9 + 0.5 * (899 + 6190) / 2008 = 2 грн. / год.

Висновки

В економічному розділі проаналізовано витрати на захист програм методом обфускації при використанні існуючих систем обфускації. Витрати на розробку програмного забезпечення складають близько 16456.5 грн.

ВИСНОВКИ

Основні результати магістерської роботи можна сформулювати наступним чином:

1. Проведено аналіз науково-практичних основ процесу обфускації, виконаний короткий огляд заплутуючих перетворень вихідного коду програм, описані методи і програмні інструменти для дослідження і аналізу програм.

2. Розглянуто механізм обфускації покажчиків на дані, що заснований на шифруванні адрес даних використаних в цій ділянці програми;

3. Розглянуто метод обфускації з використанням непрозорих предикатів, в якому використовується прийом об'єднання покажчиків на дані в окремі масиви покажчиків.

4. Запропоновано метод обфускації коду з шифруванням адрес масиву невизначених покажчиків з їх подальшою розшифровкою при обчисленні окремих виразів критичного коду програми. Даний метод посилює заплутуючий ефект за рахунок спільного застосування раніше відомих і окремо використовуваних способів обфускації.

5. Виконано програмне тестування запропонованого методу стосовно до одного обфусціруємого виразу і запропонований алгоритм для програмної реалізації всього методу.

6. Виконано оцінку трудомісткості обфускації за пропонованим методом і витрати на її реалізацію.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Анисимов А. В., Иванов И. Ю. Подходы к защите программного обеспечения от атак злонамеренного хоста. Интернет- ресурс: http://mognovse.ru/upj-informaciyi-a-v-anisimov-i-yu-ivanov-podhodi-k-zashite-pro.html.

2. Варновский Н.М.,.ЗахаровВ.Л.,. Кузюрин И.И.,. Шакуров А.В. Современное состояние исследований в области обфускации программ:определения стойкости обфускации.Труды ИСИ РАН. том 26. вып. 3, 2014.

3. Варновский Н.М.,.ЗахаровВ.Л.,. Кузюрин И.И.,. Шакуров А.В. Современные методы обфускации программ: сравнительный анализ и классификация. Известия ЮФУ. Технические науки 2013.

4. Н.П. Варновский. В.А. Захаров. Н.Н. Кузюрин. А.В. Чернов.А.В. Шокуров. Об особенностях применения методов обфускации программ для информационной защиты микроэлектронных схем Труды ИСИ РАН. том 28. вып. 2, 2015 .

5. Айрапетян Р. А. Отладчик SoftlCE. Подробный справочник. — М.: СОЛОН-Пресс, 2003.

6. Чернов А. В., Анализ запутывающих преобразований программ - Труды Института Системного программирования РАН-2009

7. Казарин О.В.Теория и практика защиты программ. -М.: Издательский дом "Вильямс» –2004

8. Ховард М, Лебланк Д.Защищенный код: Пер. с англ, — 2-е изд., М-: Издательско-торговый дом «Русская Редакция», 2004

9.Лифшиц Ю.М. Запутывание (обфускация) программ Обзор 2004. Интернет-ресурс http://logic.pdmi.ras.ru/~yura/of.html

10.Обфускация и защита программных продуктов. Интернет-ресурс http://citforum.ck.ua/security/articles/obfus/#liter

11. Анатолий Ализар. Математическая обфускация: криптографическая защита программного кода . Журнал “Хакер” Авг 15, 2014 Интернет-ресурс https://xakep.ru/2014/08/15/crypto-obfuscation/

12. Хогланд, I per, Мак-Гроу, Гари. Взлом программного обеспечения: анализ и использование кода.: Пер. с англ. -М.: Издательский дом "Вильямс", 2005.

13. Касперски Крис. Техника хакерских атак.Фундаментальные основы хакерства. — М.СОЛОН_Пресс,2004.

14. Касперски Крис Образ мышления – дизассемблер IDA Pro — М.СОЛОН_Р,2010.

15. К. Касперски К., Рокко Е. Искусство дизассемблирования— СПб.: БХВ-Петербург, 2008.

16. Касперски Крис, Техника и философия хакерских атак. Изд.2. - М.: “Солон-Пресс”, 2005.

17. Панов А. С. Реверсинг и защита программ от взлома. — СПб.: БХВ-Петербург,2006.

18. Абашев А. А., Жуков И. Ю., Иванов М. А. и др. Ассемблер в задачах защиты информации Учебно-справочное издание - М: КУДИЦ-ОБРАЗ, 2004. - 544 с.

19. Пирогов В. Ю. Ассемблер и дизассемблирование. — СПб.: БХВ-Петербург,2006.

20. Расторгуев СП., Дмитриевский Н.Н. Искусство защиты и «раздевания» программ. - М.: «Софтмаркет», 1991

21. Рикардо Нарваха. Введение в крэкинг с нуля, используя OllyDbg. Интернет — ресурс: http://www.allasm.ru/crack_01.php

22. С.С. Гайсарян, А.В. Чернов, А.А. Белеванцев и др. О некоторых задачах анализа и трансформации программ Труды Института системного программирования РАН -2011

23. Умяров Н.Х., Губенко Н.Е.Анализ и выбор методов защиты программного продукта от копирования с использованием обфускации. Материалы конференции “Информатика и компьютерные технологии-2011” -Киев 2011

24. Методичні рекомендації до підготовки та захисту дипломної роботи (проекту) для студентів галузі знань 1701 «Інформаційна безпека» та спеціальності 125 «Кібербезпека» / Т.В. Бабенко, М.В. Корнєєв, О.В. Кручінін, Д.С. Тимофєєв ; Нац. гірн. ун-т. – Д. : НГУ, 2016. – 44 с.

ДОДАТОК А. Перелік матеріалів дипломної роботи

1. Титульна сторінка.doc
2. Завдання.doc
3. Реферат.doc
4. Зміст.doc
5. Вступ.doc
6. Розділ 1.doc
7. Розділ 2.doc
8. Розділ 3.doc
9. Висновки.doc
10. Список використаної літератури.doc
11. Додаток А.doc
12. Додаток Б.doc
13. Додаток В.doc
14. Додаток Г.doc

Презентація.pptx

Додаток Б

Тестові програми

1. Вихідний код необфусцірованної тестової програми

```
#include <iostream>
using namespace std;
int main()
{
   int a=12;int b=27;int m=5;
// Вычисление оператора
  float y=  a+b%m;
    cout << y << endl;
    return 0;
}
```

2. Фрагмент вихідного коду необфусцірованної тестової програми

```
11      {
0x401340    lea    0x4(%esp),%ecx
0x401344    and    $0xfffffff0,%esp
0x401347    pushl  -0x4(%ecx)
0x40134a    push   %ebp
```

```
0x40134b    mov    %esp,%ebp
0x40134d    push   %ecx
0x40134e    sub    $0x34,%esp
0x401351    call   0x41c300 <__main>
12          int a=12;int b=27;int m=5;
0x401356    movl   $0xc,-0xc(%ebp)
0x40135d    movl   $0x1b,-0x10(%ebp)
0x401364    movl   $0x5,-0x14(%ebp)
13      // Обчислення оператора
14          float y=  a+b%m;
0x40136b    mov    -0x10(%ebp),%eax
0x40136e    cltd
0x40136f    idivl  -0x14(%ebp)
0x401372    mov    -0xc(%ebp),%eax
0x401375    add    %edx,%eax
0x401377    mov    %eax,-0x1c(%ebp)
0x40137a    fildl  -0x1c(%ebp)
0x40137d    fstps  -0x18(%ebp)
16          cout << y << endl;
0x401380    mov    -0x18(%ebp),%eax
0x401383    mov    %eax,(%esp)
0x401386    mov    $0x489940,%ecx
0x40138b    call   0x459630 <std::ostream::operator<<(float)>
```

```
0x401390    sub    $0x4,%esp

0x401393    movl   $0x478b80,(%esp)
```

3. Вихідний код обфусцірованної тестової програми

```
#include <iostream>

using namespace std;


int GPa=0x12300f00;

int GPb=0x12300f10;

int GPm=0x12300f20;

float run(int *a,int*b,int *m)

 {

 return( (*a) +(*b) %  (*m)  );

 }

int main()

{

    static int SPa=0x12300f00;

    static int SPb=0x12300f10;

    static int SPm=0x12300f20;


   int a=12;int b=27;int m=5;

// Покажчики ----------
```

```
   int *pa=&a;
   int *pb=&b;
   int *pm=&m;

   // шифрування ------------------
   pa=pa+SPa;
   pb=pb+SPb;
   pm=pm+SPm;

// Обчислення оператора
  float y= run((pa-GPa),(pb-GPb), (pm-GPm));
    cout << y << endl;
    return 0;
}
```

4. Фрагмент вихідного коду обфусцированної тестової програми

```
18      {
0x401369    lea    0x4(%esp),%ecx
0x40136d    and    $0xfffffff0,%esp
0x401370    pushl  -0x4(%ecx)
0x401373    push   %ebp
0x401374    mov    %esp,%ebp
```

```
0x401376    push   %ebx
0x401377    push   %ecx
0x401378    sub    $0x40,%esp
0x40137b    call   0x41c3a0 <__main>
19          static int SPa=0x12300f00;
20          static int SPb=0x12300f10;
21          static int SPm=0x12300f20;
27         int a=12;int b=27;int m=5;
0x401380    movl   $0xc,-0x1c(%ebp)
0x401387    movl   $0x1b,-0x20(%ebp)
0x40138e    movl   $0x5,-0x24(%ebp)
28      // Покажчики ----------
29         int *pa=&a;
0x401395    lea    -0x1c(%ebp),%eax
0x401398    mov    %eax,-0xc(%ebp)
30         int *pb=&b;
0x40139b    lea    -0x20(%ebp),%eax
0x40139e    mov    %eax,-0x10(%ebp)
31         int *pm=&m;
0x4013a1    lea    -0x24(%ebp),%eax
0x4013a4    mov    %eax,-0x14(%ebp)
33         // шифрування ----------
34         pa=pa+SPa;
```

```
0x4013a7    mov    0x47e00c,%eax
0x4013ac    shl    $0x2,%eax
0x4013af    add    %eax,-0xc(%ebp)
35        pb=pb+SPb;
0x4013b2    mov    0x47e010,%eax
0x4013b7    shl    $0x2,%eax
0x4013ba    add    %eax,-0x10(%ebp)
36        pm=pm+SPm;
0x4013bd    mov    0x47e014,%eax
0x4013c2    shl    $0x2,%eax
0x4013c5    add    %eax,-0x14(%ebp)
40      // Обчислення оператора
41        float y= run((pa-GPa),(pb-GPb), (pm-GPm));
0x4013c8    mov    0x47e008,%eax
0x4013cd    shl    $0x2,%eax
0x4013d0    neg    %eax
0x4013d2    mov    %eax,%edx
0x4013d4    mov    -0x14(%ebp),%eax
0x4013d7    lea    (%edx,%eax,1),%ecx
0x4013da    mov    0x47e004,%eax
0x4013df    shl    $0x2,%eax
0x4013e2    neg    %eax
0x4013e4    mov    %eax,%edx
```

```
0x4013e6    mov    -0x10(%ebp),%eax
0x4013e9    add    %eax,%edx
0x4013eb    mov    0x47e000,%eax
0x4013f0    shl    $0x2,%eax
0x4013f3    neg    %eax
0x4013f5    mov    %eax,%ebx
0x4013f7    mov    -0xc(%ebp),%eax
0x4013fa    add    %ebx,%eax
0x4013fc    mov    %ecx,0x8(%esp)
0x401400    mov    %edx,0x4(%esp)
0x401404    mov    %eax,(%esp)
0x401407    call   0x401340 <run(int*, int*, int*)>
0x40140c    fstps  -0x2c(%ebp)
0x40140f    mov    -0x2c(%ebp),%eax
0x401412    mov    %eax,-0x18(%ebp)
43          cout << y << endl;
0x401415    mov    -0x18(%ebp),%eax
0x401418    mov    %eax,(%esp)
0x40141b    mov    $0x489940,%ecx
0x401420    call   0x4596d0 <std::ostream::operator<<(float)>
0x401425    sub    $0x4,%esp
0x401428    movl   $0x478c20,(%esp)
0x40142f    mov    %eax,%ecx
```

```
0x401431    call   0x459440 <std::ostream::operator<<(std::ostream&
(*)(std::ostream&))>
0x401436    sub    $0x4,%esp
44          return 0;
0x401439    mov    $0x0,%eax
45      }
0x40143e    lea    -0x8(%ebp),%esp
0x401441    pop    %ecx
0x401442    pop    %ebx
0x401443    pop    %ebp
0x401444    lea    -0x4(%ecx),%esp
0x401447    ret
```

Додаток В

Приклад обфускації критично важливого програмного коду

1. Початковий код необфусцірованної програми

```
#include <iostream>
using namespace std;
int main()
{    int s,x=2,t,u,y=3,n=4;
   // Критичний код----------
   s=1;
   t=x;
   u=y;
   while(u)
   { if(u&1) s=(s+t)%n;
   u>>=1;
   t=(t*t)%n;
   }
   //------------------------------
   cout << "S=  " <<s<< endl;
   return 0;
}
```

Результат роботи програми

```
 s= 3
```

2. Дизасемблювати код необфусцірованної програми

```
6       {
0x401340    lea    0x4(%esp),%ecx
```

```
0x401344    and    $0xfffffff0,%esp
0x401347    pushl  -0x4(%ecx)
0x40134a    push   %ebp
0x40134b    mov    %esp,%ebp
0x40134d    push   %ecx
0x40134e    sub    $0x34,%esp
0x401351    call   0x41c340 <__main>
7           int s,x=2,t,u,y=3,n=4;
0x401356    movl   $0x2,-0x18(%ebp)
0x40135d    movl   $0x3,-0x1c(%ebp)
0x401364    movl   $0x4,-0x20(%ebp)
9           // Критичний код----------------
10          s=1;
0x40136b    movl   $0x1,-0xc(%ebp)
11          t=x;
0x401372    mov    -0x18(%ebp),%eax
0x401375    mov    %eax,-0x10(%ebp)
12          u=y;
0x401378    mov    -0x1c(%ebp),%eax
0x40137b    mov    %eax,-0x14(%ebp)
13          while(u)
0x40137e    jmp    0x4013aa <main()+106>
0x4013aa    cmpl   $0x0,-0x14(%ebp)
0x4013ae    jne    0x401380 <main()+64>
14          { if(u&1) s=(s+t)%n;
0x401380    mov    -0x14(%ebp),%eax
0x401383    and    $0x1,%eax
0x401386    test   %eax,%eax
0x401388    je     0x401399 <main()+89>
0x40138a    mov    -0xc(%ebp),%edx
```

```
0x40138d    mov    -0x10(%ebp),%eax
0x401390    add    %edx,%eax
0x401392    cltd
0x401393    idivl  -0x20(%ebp)
0x401396    mov    %edx,-0xc(%ebp)
15          u>>=1;
0x401399    sarl   -0x14(%ebp)
16          t=(t*t)%n;
0x40139c    mov    -0x10(%ebp),%eax
0x40139f    imul   -0x10(%ebp),%eax
0x4013a3    cltd
0x4013a4    idivl  -0x20(%ebp)
0x4013a7    mov    %edx,-0x10(%ebp)
17          }
19          //------------------------------
20          cout << "S=  " <<s<< endl;
0x4013b0    movl   $0x47f025,0x4(%esp)
0x4013b8    movl   $0x489940,(%esp)
0x4013bf    call   0x47a280 <std::basic_ostream<char, std::char_traits<char> >&
std::operator<< <std::char_traits<char> >(std::basic_ostream<char,
std::char_traits<char> >&, char const*)>
0x4013c4    mov    %eax,%edx
0x4013c6    mov    -0xc(%ebp),%eax
0x4013c9    mov    %eax,(%esp)
0x4013cc    mov    %edx,%ecx
0x4013ce    call   0x459690 <std::ostream::operator<<(int)>
0x4013d3    sub    $0x4,%esp
0x4013d6    movl   $0x478bc0,(%esp)
0x4013dd    mov    %eax,%ecx
```

```
0x4013df     call   0x4593e0 <std::ostream::operator<<(std::ostream&
(*)(std::ostream&))>
0x4013e4     sub    $0x4,%esp
21          return 0;
0x4013e7     mov    $0x0,%eax
22      }
0x4013ec     mov    -0x4(%ebp),%ecx
0x4013ef     leave
0x4013f0     lea    -0x4(%ecx),%esp
0x4013f3     ret
```

3. Вихідний код обфусцірованної програми

```
#include <iostream>
#include <stdio.h>
using namespace std;
// Класи цілих------- ----------------------
class INT_1  //- 1--------mas[0]-----------
{
  public:
  const int one=1;
  int f (){};
};
class INT_2  //- 2--------mas[1]-----------
{
  public:
  const int two=2;
  int f (){};
};
class INT_3  //- 3---------mas[2]----------
```

```
{
  public:
  const int tree=3;
  int f (){};
};
class INT_s  //- s---------mas[3]----------
{
  public:
  int s;
  int f (){};
};
class INT_u  //-u---------mas[4]----------
{
  public:
  int u;
  int f (){};
};
class INT_x  //-x---------mas[5]----------
{
  public:
  int x;
  int f (){};
};
class INT_t  //- t-------mas[6]------------
{
  public:
  int t;
  int f (){};
};
class INT_y  //-y---------mas[7]----------
```

```
{
  public:
  int y;
  int f (){};
};
class INT_n  //-n--------mas[8]-----------
{
  public:
  int n;
  int f (){};
};
// Класи Операцій ------------------------------
class FUN1 //---------mas[9]---------------------
{
  public:
 int Func1(INT_s *ps,INT_1* pn)
  { int s=ps->s;
    int n=pn->one;
   return (s+n);  }
};
class FUN3 //---------mas[10]--------------------
{
  public:
 int Func3(INT_u *pu)
{
    int u=pu->u;
    if(u==0)  return (0);else return(1);}
};
class FUN4 //---------mas[12]--------------------
{
```

```
 public:
int Func4(INT_t *pt,INT_n* pn)
 { int tn=pt->t;
   int nn=pn->n;
     return (( tn *tn)%nn);  }
};
INT_1 const_1;
   INT_s  per_s;
   INT_t per_t;
   INT_u per_u;
   INT_n  per_n;
   FUN1 pfun1;
   FUN3 pfun3;
   FUN4 pfun4;
int main()
{
per_n.n=4;
per_t.t=7;
per_s.s=1;
// Масив об'єктів ----------------
   void * mas[13];
   mas[0]=&const_1;
   mas[3]=&per_s;
   mas[4]=&per_u;
   mas[6]=&per_t;
   mas[8]=&per_n;
   mas[9]=&pfun1;
   mas[10]=&pfun3;
   mas[12]=&pfun4;
   /*---------------------------------
```

```
   int s,x=2,t,u,y=3,n=4;
```

Критичний код

```
   s=1;    //---S  ----
   t=2;
   u=y;
   while(u) // ---U ----
   { if(u&1) s=(s+t)%n;
   u>>=1;
   t=(t*t)%n; //-----T -----
   }
   ------------------------------*/
   //---- S  -------------
    int s=( ((FUN1*)mas[9])->Func1((INT_s*)mas[3],(INT_1*)mas[0])) ;
 // ----- WHILE(U) --------
 while(((FUN3*)mas[10])->Func3((INT_u*)mas[4]))
 {
  //  --- T --------------
   int t=( ((FUN4*)mas[12])->Func4((INT_t*)mas[6],(INT_n*)mas[8])) ;
 }
     cout << "s=  " <<s  << endl;
   return 0; }
```

4. Дезассемблірованний код обфусцірованної програми

```
0x401340    lea    0x4(%esp),%ecx
0x401344    and    $0xfffffff0,%esp
0x401347    pushl  -0x4(%ecx)
0x40134a    push   %ebp
0x40134b    mov    %esp,%ebp
```

```
0x40134d    push   %ecx
0x40134e    sub    $0x54,%esp
0x401351    call   0x41c3a0 <__main>
0x401356    movl   $0x4,0x489014
0x401360    movl   $0x7,0x48900c
0x40136a    movl   $0x1,0x489008
0x401374    movl   $0x47e000,-0x44(%ebp)
0x40137b    movl   $0x489008,-0x38(%ebp)
0x401382    movl   $0x489010,-0x34(%ebp)
0x401389    movl   $0x48900c,-0x2c(%ebp)
0x401390    movl   $0x489014,-0x24(%ebp)
0x401397    movl   $0x489018,-0x20(%ebp)
0x40139e    movl   $0x489019,-0x1c(%ebp)
0x4013a5    movl   $0x48901a,-0x14(%ebp)
0x4013ac    mov    -0x44(%ebp),%ecx
0x4013af    mov    -0x38(%ebp),%edx
0x4013b2    mov    -0x20(%ebp),%eax
0x4013b5    mov    %ecx,0x4(%esp)
0x4013b9    mov    %edx,(%esp)
0x4013bc    mov    %eax,%ecx
0x4013be    call   0x42d8e8 <FUN1::Func1(INT_s*, INT_1*)>
0x4013c3    sub    $0x8,%esp
0x4013c6    mov    %eax,-0xc(%ebp)
0x4013c9    jmp    0x4013e8 <main()+168>
0x4013cb    mov    -0x24(%ebp),%ecx
0x4013ce    mov    -0x2c(%ebp),%edx
0x4013d1    mov    -0x14(%ebp),%eax
0x4013d4    mov    %ecx,0x4(%esp)
0x4013d8    mov    %edx,(%esp)
0x4013db    mov    %eax,%ecx
```

```
0x4013dd    call   0x42d938 <FUN4::Func4(INT_t*, INT_n*)>
0x4013e2    sub    $0x8,%esp
0x4013e5    mov    %eax,-0x10(%ebp)
0x4013e8    mov    -0x34(%ebp),%edx
0x4013eb    mov    -0x1c(%ebp),%eax
0x4013ee    mov    %edx,(%esp)
0x4013f1    mov    %eax,%ecx
0x4013f3    call   0x42d910 <FUN3::Func3(INT_u*)>
0x4013f8    sub    $0x4,%esp
0x4013fb    test   %eax,%eax
0x4013fd    setne  %al
0x401400    test   %al,%al
0x401402    jne    0x4013cb <main()+139>
0x401404    movl   $0x47f025,0x4(%esp)
0x40140c    movl   $0x489940,(%esp)
0x401413    call   0x47a360 <std::basic_ostream<char, std::char_traits<char> >&
std::operator<< <std::char_traits<char> >(std::basic_ostream<char,
std::char_traits<char> >&, char const*)>
0x401418    mov    %eax,%edx
0x40141a    mov    -0xc(%ebp),%eax
0x40141d    mov    %eax,(%esp)
0x401420    mov    %edx,%ecx
0x401422    call   0x459770 <std::ostream::operator<<(int)>
0x401427    sub    $0x4,%esp
0x40142a    movl   $0x478ca0,(%esp)
0x401431    mov    %eax,%ecx
0x401433    call   0x4594c0 <std::ostream::operator<<(std::ostream&
(*)(std::ostream&))>
0x401438    sub    $0x4,%esp
0x40143b    mov    $0x0,%eax
```

```
0x401440    mov    -0x4(%ebp),%ecx
0x401443    leave
0x401444    lea    -0x4(%ecx),%esp
0x401447    ret
```

ДОДАТОК Г. Відгуки керівників розділів

Г.1 Відгук керівника економічного розділу

Керівник розділу _____ _____

(підпис) (ініціали, прізвище)