МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

НКЦ заочної освіти

Кафедра інформаційних технологій

Бакалаврська кваліфікаційна робота

на тему: Захист програмного забезпечення методом обфускації

Виконав студент 5 курсу групи КН-5
спеціальність 122 комп'ютерні науки
та інформаційні технології
Гулієв Тогрул Нізамі оглу

Керівник д.т.н., професор
Казакова Надія Феліксівна

Рецензент регіональний координатор
Програми EGAP
Копиченко Іван Юрійович

Одеса 2020

ЗМІСТ

Перелік скорочень 5
Вступ 6
1 Огляд методів захисту програмного забезпечення 9
1.1 Класифікація методів захисту програмного забезпечення 9
1.2 Програмно-апаратні методи захисту 10
1.3 Програмні методи захисту 14
2 Обфускація програмного коду 21
2.1 Загальні положення та основні визначення 21
2.2 Загальний алгоритм обфускації 22
2.3 Види обфускації 24
2.4 Основні методи дослідження обфускованого коду 34
3 Підвищення стійкості методу 39
3.1 Алгоритм комплексного методу обфускації 39
3.2 Обґрунтування стійкості комплексного методу 54
3.3 Приклад застосування 56
4 Планування робіт щодо аналізу та дослідження захисту програмного забезпечення методом обфускації 60
4.1 Зміст етапів планування 60
4.2 Виявлення і опис подій і робіт 61
4.3 Визначення часу виконання робіт 62
4.4 Побудова сіткового графіка 64
4.5 Розрахунок параметрів сіткового графіка 64
4.6 Аналіз сіткового графіка і його оптимізація 66
Висновки 67
Перелік джерел посилання 68
Додаток А Результат маскування функції queens 70

ПЕРЕЛІК СКОРОЧЕНЬ

АС – автоматизована система;

ЕОМ – електронно-обчислювальна машина;

ЗІ − захист інформації

ІС − інформаційна система

ІБ – інформаційна безпека

ІТ − інформаційні технології

ІзОД – інформація з обмеженим доступом;

КЗЗ – комплекс засобів захисту;

КСЗІ – комплексна система захисту інформації;

НД ТЗІ – нормативний документ системи технічного захисту інформації;

НСД – несанкціонований доступ;

ОС – обчислювальна система;

ПЕОМ – персональна електронно-обчислювальна машина;

ПЗ – програмне забезпечення;

СУІБ − система управління інформаційною безпекою

СЗІ − система захисту інформації

ТЗІ – технічний захист інформації

ВСТУП

За останні десятиліття використання інформаційних технологій у світі зростає стрімкими темпами. Їх основу становить програмне забезпечення – продукт інтелектуальної діяльності, який належить одночасно до сфер інформаційних технологій та авторського права.

Комп'ютерні системи і програмне забезпечення для їх функціонування – це основа становлення сучасних виробничих, фінансових та бізнесових технологій у світовій економіці, що потребує значних обсягів нового програмного забезпечення, ринок якого у світі постійно збільшується. Використання неліцензійного програмного забезпечення становить перешкоду на шляху розвитку ринку інформаційних технологій, стримує міжнародне співробітництво, сприяє розвитку тіньового сектору та інших негативних наслідків. Втрачаються можливості інвестування у сфери інформаційних технологій міжнародними фінансовими та виробничими структурами. Саме тому, все більшої актуальності набуває питання комплексного розв'язання проблем правомірного використання об'єктів інтелектуальної власності тощо.

Одним з основних видів правопорушень щодо програмного забезпечення є контрафакція, різновидами якої є відтворення, розповсюдження та використання програмного забезпечення без дозволу власника авторських прав на ці твори (комп'ютерне піратство).

Комп'ютерне піратство – це нелегальне копіювання та розповсюдження програмних продуктів на дисках та через комп'ютерні мережі. Полягає у подоланні різноманітних систем захисту. Для цього існує спеціальний клас програмного забезпечення – так звані “cracks” (укр. краки) – тобто спеціальні програми, готові серійні номери або їх генератори для продукту, котрі знімають з нього обмеження, пов'язані з вбудованим захистом від нелегального використання.

Слід зазначити, що піратські продукти далеко не завжди є точними копіями ліцензійних і можуть містити помилки на деяких стадіях

функціонування, таким чином підриваючи авторитет компаній розробників програми. Також існують випадки умисного змінення коду програми зловмисниками для власних цілей.

Незважаючи на стрімкий розвиток систем захисту програмного забезпечення, за останні роки продовжується ріст комп'ютерного піратства. За даними дослідження BSA Global Software Survey, який регулярно випускає Асоціація виробників програмного забезпечення, 80% програмного забезпечення, встановленого на комп'ютерах в Україні, не має необхідних ліцензій.[1][1)] Використання неліцензованого програмного забезпечення, хоч і трохи знижене, але все ще широко поширене. Неліцензійне програмне забезпечення все ще використовується по всьому світу з тривожними темпами, що становить 37 відсотків програмного забезпечення, встановленого на персональних комп'ютерах − лише на 2 відсотки спостерігається зниження використання з 2016 року.

CIO (Chief Information Officer – IT-директори) повідомляють, що неліцензоване програмне забезпечення стає все більш ризикованим і дорогим. Зловмисне програмне забезпечення від неліцензованого програмного забезпечення коштує компаніям у всьому світі майже 359 мільярдів доларів на рік. Уникнення взломів даних та інших загроз безпеці від зловмисного програмного забезпечення є причиною номер один для забезпечення їх повноцінної ліцензії.

Удосконалення відповідності програмному забезпеченню тепер є економічним фактором на додаток до необхідних заходів безпеки. Коли компанії вживають прагматичні кроки для покращення управління програмним забезпеченням, вони можуть збільшити прибуток на цілих 11 відсотків. У дослідженні також наводяться дані за обсягом і вартості неліцензійного ПЗ, встановленого на персональних комп'ютерах в більш ніж 110 країнах світу. Так в Україні, за даними BSA, вартість всього ПЗ, встановленого без ліцензій за 2017 рік, становила 108 млн. доларів.

[1)] [1] Software Management: Security Imperative, Business Opportunity, https://gss.bsa.org/

У грошовому еквіваленті Україна на третьому місці після Росії (1,2 млрд. доларів) і Чехії (149 млн. доларів) по регіону Центральна та Східної Європи. А за процентним співвідношенням − на шостому.

В цілому проникнення піратського ПЗ в регіоні СЕЕ становить 57%. Приблизно такий же рівень (50% і вище) в Азії, Латинській Америці, Африці і Середньому Сході. Найнижчий рівень піратства, очікувано, спостерігається в країнах Північної Америки (16%) і Західної Європи (26%), що впливає на загальносвітову статистику.

В середньому доля неліцензійного програмного забезпечення в глобальному масштабі сягає 40%, тобто кожні чотири з десяти копій програм виявляються в певному значенні вкраденими у виробника, що зменшує його прибутки. По розрахункам BSA (Business Software Alliance) у 2002 році збитки програмної галузі від піратства склали приблизно 13 мільярдів доларів США.

Країни пострадянського простору мають одні з найвищих показників піратства. Для західних компаній це хоч і приносить відчутні збитки, проте не є критичним для їх бізнесу, однак для українських компаній дана тенденція може зруйнувати усі інвестиційні плани.

Є кілька способів боротьби з піратством. Основними напрямками боротьби є нормативно-правовий та економічний. Вони звертаються до третього методу – захисту програмного забезпечення від зламу і нелегального копіювання програмним та програмно-апаратним шляхом. Захист становить перешкоду на шляху зловмисників і, в кінцевому результаті, дозволяє розробникам програмних продуктів збільшити прибутки.

Виходячи зі сказаного, завданнями дипломної роботи є:

- Проаналізувати методи захисту програмного забезпечення
- Дослідити стійкість методів обфускації програмного коду

1 ОГЛЯД МЕТОДІВ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Класифікація методів захисту програмного забезпечення

Поняття "захист інформації" має широке трактування, оскільки має на увазі практично всі аспекти інформаційної безпеки. Невід'ємною складовою захисту інформації є захист програмного забезпечення.

Захист програмного забезпечення (ПЗ) – це комплекс заходів, спрямованих на захист ПЗ від несанкціонованого придбання, використання, поширення, модифікації, вивчення та відтворення аналогів [2][1)].

Системи захисту програмного забезпечення широко розповсюджені і постійно розвиваються, завдяки розширенню ринку програмного забезпечення та телекомунікаційних технологій. Також існує ряд ознак за якими їх можна класифікувати, серед яких доцільно виділити методи захисту програмного забезпечення.

Класифікація методів захисту ПЗ зображена на рис. 1.1. До програмних відносяться методи, що реалізуються програмним шляхом. Вони не торкаються жодних фізичних характеристик носіїв інформації чи інформаційного середовища, у якому функціонують, а реалізуються лише шляхом виконання певних алгоритмічних перетворень. До програмно-апаратних відносяться методи, що використовують спеціальні пристрої або їх фізичні особливості, для того, щоб ідентифікувати оригінальну версію програми та захистити продукт від нелегального використання.

---

1) [2] Бобало Ю.Я. Інформаційна безпека: навчальний посібник / Ю.Я. Бобало, І.В. Горбатий, М.Д. Кіселичник, А.П. Бондарєв, С.С. Войтусік, А.Я. Горпенюк, О.А. Нємкова, І.М. Журавель, Б.М. Березюк, Є.І. Яковенко, В.І. Отенко, І.Я. Тишик. Львів: Видавництво Львівської політехніки, 2019. – 580 с.

1.2 Програмно-апаратні методи захисту

До найпоширеніших апаратних методів захисту належать системи що використовують:

- Електронні ключі.
- Особливості носіїв інформації.
- Особливості інформаційного середовища.

Електронні ключі за останній час набувають дедалі більшої популярності серед виробників програмного забезпечення. Електронними ключами, в даному випадку, є засоби, основані на використанні так званих "апаратних ключів". Електронний ключ – це апаратна частина системи захисту, що представляє собою плату з мікросхемами пам'яті і, в деяких випадках, мікропроцесором, розміщену в корпусі й призначену для встановлення в один із стандартних портів персонального комп'ютера (COMM, LPT, PCMCIA, USB ...) або слот розширення материнської плати. Також в якості такого пристрою можуть використовуватися СМАРТ-карти. По результатам проведених аналізів, такі методи захисту на даний час є одними з найстійкіших систем захисту ПЗ.

Електронні ключі за архітектурою можна поділити на ключі з пам'яттю (без мікропроцесора) і ключі з мікропроцесором (і пам'яттю).

Менш стійкими є системи з апаратною частиною першого типу. В таких системах критична інформація (ключ дешифрації, таблиця переходів) зберігається в пам'яті електронного ключа. Для дезактивації такого захисту в більшості випадків необхідна наявність у зловмисника апаратної частини системи захисту (основна методика: перехоплення діалогу між програмною та апаратною частинами для доступу до критичної інформації).

Рис. 1.1 − Класифікація методів захисту програмного забезпечення

Більш стійкими є системи з апаратною частиною другого типу. Такі комплекси містять в апаратній частині не лише ключ дешифрації, а й блоки шифрації/дешифрації даних, таким чином при роботі захисту в електронний ключ передаються блоки зашифрованої інформації, а приймаються розшифровані дані. В системах такого типу досить важко перехопити ключ дешифрації, оскільки всі процедури виконуються апаратною частиною, але залишається можливість примусового збереження захищеної програми у відкритому вигляді після обробки системи захисту.

Позитивні сторони даної системи захисту:

- Ускладнення нелегального розповсюдження та використання ПЗ.
- Автоматизація процесу захисту ПЗ.
- Широкий вибір даних систем на ринку.

Негативні сторони:

- Ускладнення розробки та відлагодження ПЗ у зв'язку з обмеженнями зі сторони системи захисту.
- Додаткова вартість апаратної частини системи захисту.
- Підвищення системних вимог (сумісність, драйвери).
- Можлива несумісність систем захисту і системного чи прикладного ПЗ користувача.
- Загроза втрати електронного ключа.
- Сповільнення продажу, оскільки необхідна фізична передача апаратної частини.

Особливості носіїв інформації. Є системи захисту, що використовують "прив'язку" ПЗ до дистрибутивного носія. Даний тип захисту оснований на детальному вивченні роботи контролерів накопичувачів, їх фізичних показників, нестандартних режимів форматування, читання/запису і т. п. При тому на фізичному рівні створюється дистрибутивний носій, що має (ймовірно) неповторні властивості (зазвичай це досягається за допомогою нестандартної розмітки носія інформації і/або запису на нього додаткової інформації – паролю чи

мітки), а на програмному – створюється модуль, налаштований на ідентифікацію і автентифікації носія за його унікальними властивостями.

На даний час використовують два основні принципи створення унікального носія. Перший реалізується суто програмним шляхом за допомогою звичайного обладнання і полягає в записі певного "коду" на відповідну область носія або нестандартній його розмітці. Другий принцип передбачає створення за допомогою спеціального обладнання фізичних відмінностей (в тому числі пошкоджень певних областей) носія. Останній принцип є надійнішим з точки захисту від копіювання, оскільки його майже неможливо відтворити без спеціального обладнання, проте його застосування впливає на ціну кінцевого продукту ПЗ.

Позитивні сторони даної системи захисту:

- Ускладнення нелегального копіювання і розповсюдження ПЗ.
- Захист прав користувача на придбане ПЗ.

Негативні сторони:

- Трудомісткість реалізації системи захисту.
- Сповільнення продажу, оскільки необхідна фізична передача носія.
- Зниження стійкості до відмов ПЗ.
- На час роботи ПЗ зайнятий накопичувач.
- Загроза втрати носія.

Особливості інформаційного середовища. Системи, що використовують особливості інформаційного середовища, при встановленні ПЗ на персональний комп'ютер користувача здійснюють пошук унікальних ознак комп'ютерної системи або вони встановлюються самою системою захисту. Після цього модуль захисту в ПЗ налаштовується на пошук та ідентифікацію даних ознак, за якими надалі визначається авторизоване чи ні використання ПЗ. При цьому можливе застосування методів оцінки швидкісних або інших показників процесора, материнської плати, додаткових пристроїв, операційної системи, читання/запис в мікросхеми енергозалежної пам'яті, запис скритих файлів та ін. Щоправда реалізація оцінки таких характеристик є не простим завданням,

оскільки ці фізичні властивості в більшості випадків не є сталими величинами і мають певну похибку.

Основною проблемою таких систем є зміни конфігурації складових персонального комп'ютера, що може дати хибні спрацювання системи захисту.

Позитивні сторони даної системи захисту:

- Не потрібно додаткових апаратних пристроїв для системи захисту.
- Ускладнений несанкціонований доступ до скопійованого ПЗ.
- Простота у використанні.
- Система захисту є "невидимою" для користувача.

Негативні сторони:

- Хибні спрацювання системи захисту при зміні конфігурації ПК.
- Можливі конфлікти із системним програмним забезпеченням.

1.3 Програмні методи захисту

Програмні методи захисту ПЗ можна поділити на два види:

- Захист від несанкціонованого запуску;
- Захист від дослідження.

1.3.1 Захист від несанкціонованого запуску

Полягає в попередній чи періодичній автентифікації користувача ПЗ або його комп'ютерної системи шляхом запиту додаткової інформації. До цього виду захисту можна віднести:

- Системи парольного захисту.
- Авторизація через мережу Інтернет.

В першому випадку "ключову" інформацію вводить користувач, в другому – авторизація здійснюється після з'єднання з Інтернет сервером. Щоправда, принцип роботи цих двох систем відрізняється в основному лише способом авторизації, а алгоритм захисту може бути однаковий.

Ці системи захисту ПЗ на сьогоднішній час є найбільш розповсюдженими. Основний принцип роботи даних систем полягає в ідентифікації та автентифікації користувача шляхом запиту додаткових даних, це може бути назва фірми і/чи ім'я та прізвище користувача та його пароль або лише пароль/реєстраційний код. Запит цієї інформації може відбуватися в різноманітних ситуаціях, наприклад, при старті програми, після завершення строку безоплатного користування ПЗ, після виклику процедури реєстрації або безпосередньо в процесі встановлення ПЗ на персональний комп'ютер. Процедури такого захисту є простими в реалізації і тому досить часто використовуються розробниками ПЗ. Дана система захисту використовує логічні механізми, що зводяться до перевірки правильності ключа і запуску або не запуску ПЗ, в залежності від результатів перевірки. Існують також системи, які шифрують захищене ПЗ і використовують пароль або похідну від нього величину як ключ дешифрації, більшість таких систем використовують слабкі або прості алгоритми шифрування, нестійкі до атак. Це відбувається внаслідок складної коректної реалізації стійких криптоалгоритмів і недоцільності їх використання для захисту недорогих умовно-безплатних програмних продуктів, які становлять більшість ПЗ, що використовують такі системи захисту.[3][1)]

Слабкою ланкою парольного захисту та захисту шляхом авторизації через мережу Інтернет є блок перевірки правильності ключа. Для такої перевірки можна порівняти отриманий ключ з записаним в коді ПЗ правильним або з правильно згенерованим з отриманих додаткових даних ключем. Можливе також порівняння похідних величин від отриманого та правильного ключів, наприклад їх ХЕШ-функції, в такому випадку в коді можна зберегти лише похідну величину, що підвищує стійкість захисту. Шляхом аналізу процедур перевірки можна знайти реальний ключ, записаний в коді ПЗ, знайти правильно згенерований ключ з введених даних, або створити програму перебору ключів

[1)] [3] Програмні технології захисту інформації: конспект лекцій для студентів за напрямом підготовки 6.050103 «Програмна інженерія» факультету інформаційних технологій УжНУ / Розробник: к.т.н. Поліщук В.В. Ужгород: 2018. 80 с.

для визначення ключа з необхідною ХЕШ-сумою. Крім того, якщо системи захисту ПЗ не використовують шифрування, достатньо лише примусово змінити логіку перевірки і отримати безперешкодний доступ до ПЗ. Основна перевага авторизації через інтернет над звичайним парольним доступом полягає в тому, що процедура звичайного парольного доступу у більшості випадків здійснює перевірку авторизованого доступу лише один раз, в той час, як авторизація через інтернет може проводитися періодично.

Позитивні сторони даної системи захисту:

- Надійний захист від зловмисника-непрофесіонала.
- Мінімальні незручності для користувача.
- Відсутність конфліктів із системним та прикладним ПЗ.
- Простота в реалізації та застосуванні.
- Низька вартість.

Негативні сторони:

- Низька стійкість захисту у порівнянні з іншими описаними методами.
- Можливість перехоплення ключів.

1.3.2 Захист програмного забезпечення від дослідження

Захист програмного забезпечення від дослідження полягає в тому, щоб унеможливити або ускладнити декомпіляцію ПЗ. Цей захист використовують для захисту ПЗ від розкриття алгоритму його роботи або несанкціонованих змін коду ПЗ. Найпоширенішими системами цього виду є:

- Криптографічні алгоритми.
- Емуляція процесорів та операційних систем.
- Алгоритми заплутування програмного коду (обфускація).

Криптографічні алгоритми здійснюють функціональне перетворення інформації і використовуються з метою приховати зміст інформації [4], [5][1)]. Криптографічний захист програмного коду починався з програм "пакувальників". Першочергово, основним завданням таких програм було зменшити об'єм виконуючого модуля програми на диску без пошкодження функціональності програми. С цією метою створювалися різноманітні алгоритми, що забезпечували шифрування тексту програми. Але пізніше на першому плані було питання захисту ПЗ від аналізу його алгоритмів та несанкціонованої модифікації. Для досягнення цього використовуються алгоритми компресування даних, прийоми, пов'язані з використанням недокументованих особливостей операційних систем та процесорів, шифрування даних.

На практиці застосовуються наступні механізми захисту програм, основані на шифруванні: шифрування коду програми (у відкритому вигляді код програми знаходиться лише під час виконання програми), шифрування фрагменту програми (частіше обирають критичну частину програми) та шифрування даних (надійною реалізацією спеціалістами признається шифрування даних безпосередньо у вихідному тексті програми). Шифрування може бути статичним та динамічним. При статичному шифруванні весь код (фрагмент коду) один раз шифрується/розшифровується. У зашифрованому вигляді код постійно зберігається на зовнішньому носії, у відкритому вигляді присутній в оперативній пам'яті. При динамічному шифруванні послідовно шифруються/розшифровуються окремі фрагменти або критичні фрагменти програми.

Криптографічний захист реалізується на практиці за допомогою програмних або програмно-апаратних засобів. При використанні програмно-апаратних засобів криптографічні операції виконуються за допомогою спеціального обчислювального пристрою. Апаратна реалізація відрізняється істотною

[1)] [4] Технології захисту інформації [Електронний ресурс] : підручник для студ. спеціальності 122 «Комп’ютерні науки», спеціалізацій «Інформаційні технології моніторингу довкілля», «Геометричне моделювання в інформаційних системах» / Ю. А. Тарнавський; КПІ ім. Ігоря Сікорського. Київ : КПІ ім. Ігоря Сікорського, 2018. 162 с.
[5] Остапов С. Е. Технології захисту інформації : навчальний посібник / С. Е. Остапов, С. П. Євсеєв, О. Г. Король. Х. : Вид. ХНЕУ, 2013. 476 с. (Укр. мов.)

вартістю, однак, збільшує продуктивність і надійність криптосистеми. Програмна реалізація є універсальною, гнучкою і простою у використанні та оновленні, проте вона є низькошвидкісною. Спеціалісти достатньо вивчили питання ненадійності криптографічних систем захисту. Серед основних причин дослідники називають обмеження по застосуванню стійких криптоалгоритмів, неправильну реалізацію криптоалгоритмів, а також людський фактор.

Позитивні сторони даної системи захисту:

- Забезпечує захист ПЗ від аналізу його алгоритмів.
- Методи шифрування збільшують стійкість систем захисту інших типів.

Негативні сторони:

- Шифрування сповільнює виконання коду ПЗ.
- Шифрування коду викликає ускладнення при оновленні (update) та виправленні помилок (bugfix, servicepack) ПЗ.
- Підвищення апаратно-програмних вимог ПЗ.
- Шифрування виконуючого коду вступає у конфлікт з сучасними операційними системами, що забороняють самомодифікацію коду.

Емуляція процесорів та операційних систем полягає у застосуванні спеціальної програми або драйвера, що створює віртуальне середовище яке обробляє дані за унікальним алгоритмом. Таким чином, створюється віртуальний процесор або операційна система (не обов'язково реально існуючі) і програма-перекладач із звичайної системи команд у систему команд створеного процесора чи операційної системи. Після такого перекладу ПЗ може виконуватися лише за допомогою емулятора.

Суть методу полягає в тому, що деякі функції, модулі або повністю програма компілюється під певний віртуальний процесор або операційну систему (ОС), з невідомою потенційному зловмиснику системою команд і архітектурою. Виконання забезпечує вмонтований в результуючий код симулятор. Тобто, завдання дослідження захищених фрагментів зводиться до вивчення архітектури симулятора, створення дизасемблера, і, нарешті, аналізу коду. Це

завдання не легке навіть для спеціаліста, що має знання та досвід в роботі з архітектурою цільової машини. Зловмисник в цьому випадку не має доступу ні до опису архітектури віртуального процесора чи ОС, ні до інформації про організацію використаного симулятора. Вартість зламу істотно зростає.

Одначе, не зважаючи на високу теоретичну ефективність, даний метод досі немає широкого розповсюдження по двом основним причинам. По-перше, метод має властивості, що звужують області його потенційного використання, а саме, в деяких випадках можливі конфлікти з операційними системами або несумісність з обладнанням ЕОМ. По-друге, і, можливо, це більш серйозна причина, складність (а відповідно і ціна) реалізації системи достатньо висока. Якщо врахувати принципову можливість витоку інформації про щойно створену систему, яка моментально призведе до її неефективності та знецінення, стає зрозумілим, чому фірми-виробники захисного ПЗ не квапляться реалізовувати цей метод.

Позитивні сторони даної системи захисту:

- Захист від дослідження алгоритмів ПЗ.
- Негативні сторони:
- Можливі конфлікти з іншим програмним забезпечення.
- Складність створення даної системи захисту і, відповідно, висока ціна.

Алгоритми заплутування програмного коду (обфускація) – це один з перспективних методів захисту програмного коду, що дозволяє ускладнити процес реверсивної інженерії (дослідження) коду захищеного ПЗ.

Позитивні сторони даної системи захисту:

- Захист від дослідження коду ПЗ.
- Не вимагає додаткового ПЗ чи обладнання для функціонування.

Негативні сторони:

- Збільшує об’єм захищеного ПЗ.
- При використанні певних методів обфускації може сповільнюватися виконання програми.
- Існуючі методи не завжди забезпечують достатній рівень.

Таким чином, у першому розділі проведена класифікація методів захисту ПЗ. Аналіз показав, що обфускація є перспективним методом захисту ПЗ, проте він вимагає вдосконалення в напрямку підвищення стійкості до дослідження.

2 ОБФУСКАЦІЯ ПРОГРАМНОГО КОДУ

2.1 Загальні положення та основні визначення

Для обходу захисту ПЗ, необхідно проаналізувати принцип роботи його коду, і те, як він взаємодіє із захищеною програмою, цей процес аналізу називається процесом реверсивної (зворотної) інженерії. Цей процес залежить від властивостей людської психіки, тому використання цих властивостей дозволяє знизити ефективність процесу реверсивної інженерії.

Обфускація ("obfuscation" – заплутування) – це один з методів захисту програмного коду, який дозволяє ускладнити процес реверсивної інженерії коду захищеного програмного продукту [6][1]. Обфускація може застосовуватися не лише для захисту ПЗ, вона має більш широке застосування, наприклад, може бути використана авторами вірусів, для захисту своїх продуктів.

Заплутаною (obfuscated) називається програма, яка на всіх допустимих для вихідної програми вхідних даних видає той самий результат, що й оригінальна програма, але більш складна для аналізу, розуміння й модифікації [7][2]. Заплутана (обфускована) програма виходить в результаті застосування до вихідної, не заплутаної програми заплутуючи перетворень (obfuscating transformations).

Суть обфускації полягає в тому, щоб заплутати програмний код і усунути більшість логічних зв'язків в ньому, тобто трансформувати його так, щоб він був достатньо складний для дослідження та модифікації сторонніми особами (зловмисниками або програмістами, які намагаються дізнатися унікальний алгоритм роботи захищеної програми).

---

[1] [6] Чернов А.В. Анализ запутывающих преобразований программ. В сб. "Труды Института системного программирования", под. ред. В. П. Иванникова. М.: ИСП РАН, 2002.

[2] [7] Chow S., Gu Y., Johnson H., Zakharov V. An approach to the obfuscation of control-flow of sequential computer programs. LNCS 2200, pp. 144--155, 2001.

Таким чином, обфускація сама по собі не призначена для забезпечення найбільш повного та ефективного захисту програмних продуктів, оскільки вона не дає можливості запобігти нелегальне використання програмного продукту. Тому обфускацію зазвичай використовують разом з одним серед існуючих методів захисту, це дозволяє значно підвищити рівень захисту ПЗ в цілому.

Обфускацію, як метод захисту, можна вважати порівняно новою та перспективною. Також вона відповідає принципу економічної доцільності, оскільки її використання не суттєво збільшує ціну програмного продукту і дозволяє при цьому знизати втрати від піратства, знизити можливість плагіату в результаті викрадення унікального алгоритму роботи захищеного програмного продукту.

2.2 Загальний алгоритм обфускації

Існує кілька визначень процесу обфускації. Розглядаючи даний процес з точки зору захисту ПЗ, і трансформації коду програми без можливості потім повернутися до його первинного вигляду (трансформація "в одну сторону"), можна дати таке визначення:

Визначення. Нехай "ПТ" – це процес трансформації, "П1" та "П2" – програма до і після трансформації відповідно. Тоді при "П1+ПТ=П2" програма "П2" буде трансформованим кодом програми "П1". Процес трансформації "ПТ" буде вважатися процесом обфускації, якщо будуть задоволені такі вимоги:

- код програми "П2" в результаті трансформації буде суттєво відрізнятися від коду програми "П1", але при цьому він буде виконувати ті ж самі функції, що й код програми "П1", а також буде працездатним;
- дослідження принципу роботи, тобто процес реверсивної інженерії, програми "П2" буде більш складнішим, трудомістким, і буде займати більше часу, ніж програми "П1";

- при кожному процесі трансформації одного й того ж самого коду програми "П1", коди програми "П2" будуть відрізнятися.
- створення програми, що детрансформує програму "П2" в її найбільш схожий первинний вигляд, буде неефективно.

Оскільки код, одержаний після здійснення обфускації, над однією й тою самою ж програмою різний, то процес обфускації можна використовувати для швидкої локалізації порушників авторських прав (тобто тих покупців, які будуть займатися нелегальним розповсюдженням куплених копій програм). Для цього визначають контрольну суму кожної копії програми, що пройшла обфускацію, і записують її разом із інформацією про покупця у відповідну базу даних. Після цього для визначення порушника достатньо буде, визначивши контрольну суму нелегальної копії програми, спів ставити її з інформацією, що зберігається в базі даних.

Програмний код може бути представлено у двійковому вигляді (послідовність байтів, що є машинним кодом, який виходить після компіляції вихідного коду програми) або вихідному вигляді (текст, що містить послідовність інструкцій певної мови програмування, і цей текст надалі буде підлягати компіляції або інтерпретації на комп'ютері користувача).

Процес обфускації може бути здійснено над будь яким з вище перерахованих видів представлення програмного коду, тому прийнято виділяти наступні рівні процесу обфускації:

- низький рівень, коли процес обфускації здійснюється над асемблерним кодом програми, або навіть безпосередньо над двійковим файлом програми, що зберігає машинний код;
- високий рівень, коли процес обфускації здійснюється над вихідним кодом програми написаному на мові програмування високого рівня.

Здійснення обфускації на низькому рівні вважається менш комплексним процесом, але при цьому важче реалізується по багатьом причинам. Одна з цих причин полягає в тому, що повинні бути враховані особливості роботи

більшості процесорів, оскільки спосіб обфускації можливий на одній архітектурі, може виявитися нездійсненним на іншій.

Також, на сьогоднішній час процес низькорівневої обфускації досліджений мало (мабуть тому, що не набув широкої популярності).

Більшість існуючих алгоритмів та методів обфускації (включаючи ті, що будуть розглянуті в далі) можуть застосовуватися для процесу обфускації як на низькому, так і на високому рівнях.

Інколи може бути не ефективно піддавати обфускації весь код програми (наприклад, тому, що в результаті може значно збільшитися час виконання програми), в таких випадках доцільно здійснювати обфускацію лише найбільш важливих частин коду.

2.3 Види обфускації

Існує кілька видів обфускації в залежності від того, на трансформацію якої з компонент програми вони орієнтовані.

- Перетворення форматування, яке змінює лише зовнішній вигляд програми. До цієї групи належать перетворення, що видаляє коментарі, відступи в тексті програми або перейменовуються ідентифікатори.
- Перетворення структури даних змінює структуру даних, з якими працює програма. До цієї групи належать, наприклад, перетворення, що змінюють ієрархію класів у програмі, або перетворення, що об'єднують скалярні змінні типу в масив.
- Перетворення потоку управління програми, що змінюють структуру її графа потоку управління, такі як розгортка циклів, виділення фрагментів коду в процедури, та ін.
- Превентивні перетворення, застосовуються проти визначених методів декомпіляції програм або такі, що використовують помилки у відповідних інструментальних засобах декомпіляції.

Також, слід зазначити, що аналіз відкритих літературних джерел показав, що на даний час розроблені та існують нові класифікації методів захисту програмного коду, в якій враховані усі сучасні методи обфускації. Так, наприклад у [8][1)] наведений один з варіантів класифікації методів обфускації. Дані методи заплутування програмного коду було розділено за трьома критеріями: пунктуаційні перетворення, перетворення змінних та перетворення структури коду. На рис. 2.1 показана схема класифікації даних методів.

Рисунок – 2.1 Класифікація обфускаційних методів захисту коду програми

У даному розділі докладніше розглянемо перетворення форматування та перетворення потоку управління. Оскільки саме ці види є найефективнішими з точки зору захисту ПЗ.

2.3.1 Перетворення форматування

До перетворення форматування відноситься видалення коментарів, переформатування програми, видалення відлагоджувальної інформації, заміна імен ідентифікаторів.

[1)] [8] Stepanenko I., Kinzeryavyy V., Nagi A., Lozinskyi I. Modern obfuscation methods for secure coding // Ukrainian Scientific Journal of Information Security, 2016, vol. 22, issue 1, p. 32-37, ISSN 2225-5036 (Print), ISSN 2411-071X (Online) http://infosecurity.nau.edu.ua

Видалення коментарів та переформатування програми застосовуються, коли заплутування виконується на рівні вихідного коду програми. Ці перетворення не потребують лише лексичного аналізу програми. Хоча видалення коментарів – одностороннє перетворення, їх відсутність не дуже ускладнює зворотну інженерію програми, оскільки при зворотній інженерії наявність хороших коментарів до коду програми є швидше виключенням, а ніж правилом. При пере форматуванні програми вихідне форматування втрачається безповоротно, проте програма завжди може бути переформатована з використанням будь-якого інструменту для автоматичного форматування програм.

Видалення відлагоджувальної інформації застосовується, коли заплутування виконується на рівні об'єктної програми. Видалення відлагоджувальної інформації призводить до того, що унеможливлюється відновлення імен локальних змінних.

Заміна імен локальних змінних потребує семантичного аналізу (прив'язки імен) в межах однієї функції. Заміна імен усіх змінних і функцій програми окрім повної прив'язки імен до кожної одиниці компіляції потребує аналізу між модульних зв'язків. Імена, визначені у програмі, що не використовуються у зовнішніх бібліотеках, можуть бути змінені довільно, але погоджено із усіма одиницями компіляції, в той час як імена бібліотечних змінних та функцій мінятися не можуть. Дане перетворення може змінювати імена на короткі, з автоматичною генерацією. З іншого боку, імена змінних можуть бути змінені на довгі ідентифікатори без значення (випадкові) з розрахунком на те, що довгі імена важче сприймаються людиною.

2.3.2 Перетворення потоку управління

Перетворення потоку управління змінюють граф потоку управління однієї функції. Вони можуть створювати у програмі нові функції. На практиці застосовують такі перетворення:

- Відкрита вставка функції.

- Винесення групи операторів.
- Непрозорі предикати.
- Внесення недосяжного коду.
- Внесення мертвого коду.
- Внесення надлишкового коду.
- Перетворення графу потоку управління, який зводиться, до такого, що не зводиться.
- Усунення бібліотечних викликів.
- Переплетення функцій.
- Клонування функцій.
- Розгортка циклів.
- Реструктуризація графу потоку управління.
- Локалізація змінних в базовому блоці.
- Розширення області дії змінних.

Відкрита вставка функції (function inlining) полягає в тому, що тіло функції підставляється в точку виклику функції. Дане перетворення є стандартним для оптимізуючи компіляторів. Це перетворення є одностороннім, тобто по перетвореній програмі автоматично відновити вставлені функції практично неможливо.

Винесення групи операторів (function outlining). Дане перетворення є зворотнім до попереднього і непогано його доповнює. Певна група операторів вихідної програми виділяється в окрему функцію. При необхідності створюються формальні параметри. Перетворення може бути легко обернене компілятором, який (як було сказано вище) може підставляти тіла функцій в точки їх виклику. Відзначимо, що виділення операторів в окрему функцію є складним для здійснення перетворенням. Його автору необхідно провести глибокий аналіз графа потоку керування і потоку даних з врахуванням вказівників, щоб бути впевненим, що перетворення не порушить роботу програми.

Непрозорі предикати (opaque predicates). Основною проблемою при проектуванні заплутуючих перетворень графа потоку управління є те, як зробити

їх не лише дешевими, а й стійкими. Для забезпечення стійкості багато перетворень основані на введенні "непрозорих" змінних і предикатів. Сила таких перетворень залежить від складності аналізу непрозорих предикатів та змінних.

Змінна *v* є непрозорою, якщо існує властивість π відносно цієї змінної, яка заздалегідь відома в момент заплутування програми, але її важко встановити після завершення заплутування програми. Аналогічно, предикат *P* називається непрозорим, якщо його значення відоме в момент заплутування програми, проте його важко встановити після того, як заплутування завершене.

Непрозорі предикати можуть бути трьох видів: P^F – предикат, який завжди має значення "*false*", P^T − предикат, який завжди має значення "*true*", та $P^?$ – предикат, що може приймати обидва значення, і в момент заплутування відоме поточне значення предикату.

Деякі можливі способи введення непрозорих предикатів і непрозорих виразів коротко описані далі.

- Використання різних способів доступу до елементів масиву. Наприклад, у програмі може бути створений масив (наприклад, *a*), який ініціалізується заздалегідь відомими значеннями, надалі в програму додаються кілька змінних (наприклад, *i, j*), у яких зберігаються індекси елементів цього масиву. Тепер непрозорі предикати можуть мати вигляд *a[i]=a[j]*. Якщо ж змінні і та j змінюються у програмі, існуючі на даний час методи статичного аналізу дозволяють лише визначити, що *i* та *j* можуть вказувати на будь-який елемент масиву *a*.
- Використання вказівників на спеціально створені динамічні структури. Цей підхід полягає у додаванні в програму операції, що створюють структури посилання даних (списків, дерев), і додаються операції над вказівниками на ці структури, підібрані таким чином, щоб зберігались деякі інваріанти, які й використовуються як непрозорі предикати.
- Конструювання булевих виразів спеціального виду.
- Побудова складних булевих виразів за допомогою еквівалентних

перетворень з формули *true*. В найпростішому випадку ми можемо взяти k довільних булевих змінних $x_1 \ldots x_k$ і побудувати з них рівність $(x_1 \vee \overline{x_1}) \wedge \ldots \wedge (x_k \vee \overline{x_k})$. Надалі за допомогою еквівалентних алгебраїчних перетворень частина дужок (або всі) розкриваються, і в результаті отримаємо шуканий непрозорий предикат.

– Використання комбінаторних рівностей, наприклад $\sum_{i=0}^{n} C_n^i = 2^n$.

Внесення недосяжного коду (adding unreachable code). Якщо в програму внесені непрозорі предикати видів P^F або P^T, гілки умов, що відповідають умові "*true*" в першому випадку і умові "*false*" в другому випадку, ніколи не будуть виконуватися. Фрагмент програми, який ніколи не виконується, називається недосяжним кодом. Ці гілки можуть бути заповнені довільними розрахунками, які можуть бути схожими на код, що дійсно виконується, наприклад, зібрані із фрагментів тої ж самої функції. Оскільки недосяжний код ніколи не виконується, дане перетворення впливає лише на об'єм заплутаної програми, а не на швидкість її виконання. Загальна задача знаходження недосяжного коду, як відомо, не може бути розв'язана алгоритмічно. Це означає, що для виявлення недосяжного коду необхідно застосовувати різні евристичні методи, наприклад, основані на статистичному аналізі програми.

Внесення мертвого коду (adding dead code). На відміну від недосяжного коду, мертвий код в програмі виконується, але його виконання ніяк не впливає на результат програми. При внесенні мертвого коду потрібно бути впевненим, що вставлений фрагмент не може впливати на код, який розраховує значення функції. Це практично означає, що мертвий код не може мати побічного ефекту, навіть у вигляді модифікації глобальних змінних, не може змінювати оточення працюючої програми, не може виконувати ніяких операцій, які можуть викликати будь-які виключення у роботі програми.

Внесення надлишкового коду (adding redundant code). Надлишковий код, на відміну від мертвого коду виконується, і результат його виконання використовується надалі у програмі, але такий код можна спростити або зовсім

видалити, оскільки обчислення або константне значення, або значення, вже обраховане раніше. Для внесення надлишкового коду можна використовувати алгебраїчні перетворення виразів вихідної програми або введення в програму математичних рівностей. Наприклад можна використати комбінаторну рівність $\sum_{i=0}^{8} C_8^i = 2^8 = 256$ і замінити всюди у програмі використання константи 256 на цикл, який обраховує суму біноміальних коефіцієнтів по наведеній формулі.

Подібні алгебраїчні перетворення обмежені цілими значеннями, оскільки при виконанні операцій з плаваючою комою виникає проблема накопичення похибки обчислень. Наприклад, вираз sin^2x+cos^2x при обчисленні на машині практично ніколи не дасть в результаті значення 1. З іншого боку, при операціях з цілими значеннями виникає проблема переповнення. Наприклад, якщо використання 32-бітної цілої змінної x замінити на вираз *x*p/q*, де *p* і *q* гарантовано мають одне й те саме значення, при виконанні множення *x*p* може відбутися переповнення розрядної сітки, і після ділення на q ми не отримаємо бажаного результату. В якості часткового вирішення завдання можна виконувати множення в 64-бітних цілих числах.

Перетворення графу потоку управління, який зводиться, до такого, що не зводиться (transforming reducible to non-reducible flow graph). Коли цільова мова (байт-код або машинна мова) більш виразна, ніж вихідна, можна використовувати перетворення, що суперечать структурі вихідної мови. В результаті таких перетворень отримаємо послідовність інструкцій цільової мови, що не відповідають жодній із конструкцій вихідної мови.

Наприклад, байт-код віртуальної машини Java містить інструкцію *goto*, а у мові Java оператор *goto* відсутній. Графи потоку управління програм на мові Java завжди зводяться, в той час, як в байт-коді можуть бути представлені графи, що не зводяться.

Можна запропонувати заплутуючи перетворення, які трансформують графи, що зводяться, потоку управління функцій в байт-коді, отриманих в

результаті компіляції Java-програм, в графи, що не зводяться. Наприклад, таке перетворення може полягати в трансформації структурного циклу в цикл з множинними заголовками з використанням непрозорих предикатів. З одного боку, декомпілятор може спробувати виконати зворотне перетворення, видаляючи області в графі, що не зводяться, дублюючи вершини або вводячи нові булеві змінні. З іншого боку, можна за допомогою статичних або статистичних методів аналізу визначити значення непрозорих предикатів, використаних при заплутуванні, і видалити переходи, що ніколи не виконуються. Однак, якщо здогадка про значення предиката виявиться невірною, в результаті отримаємо неправильну програму.

Усунення бібліотечних викликів (eliminating library calls). Більшість програм на деяких мовах суттєво використовують стандартні бібліотеки. Оскільки семантика бібліотечних функцій добре відома, такі виклики можуть дати корисну інформацію при оберненій інженерії програм. Проблема посилюється ще й тим, що посилання на класи бібліотек деяких мов завжди є іменами і ці імена не можуть бути спотворені.

В багатьох випадках можна обійти ці обставини, просто використовуючи в програмі власні версії стандартних бібліотек. Таке перетворення суттєво не змінить час виконання програми, проте значно збільшить її розмір.

Щоправда, для програм на традиційних мовах це не є проблемою, оскільки стандартні бібліотеки, як правило, можуть бути компоновані статично разом із самою програмою. В даному випадку програма не містить ніяких імен функцій із стандартної бібліотеки.

Переплетення функцій (function interleaving). Ідея цього заплутуючого перетворення полягає в тому, щоб дві або більше функцій об'єднати в одну функцію. Списки параметрів вихідних функцій об'єднуються, і до них додається ще один параметр, який дозволяє визначити, яка функція виконується в дійсності.

Клонування функцій (function cloning). Під час оберненої інженерії функції в першу чергу досліджується сигнатура функції, а також то, як ця

функція використовується, у 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 якоїсь іншої, кожний раз нової функції. Може бути створено кілька клонів функції, і до кожного з клонів буде застосований різний набір заплутуючих перетворень.

Розгортка циклів (loop unrolling). Розгортка циклів застосовується в оптимізуючи компіляторах для пришвидшення роботи циклів або їх розпаралелення. Розгортка циклів полягає в тому, що тіло циклу розмножується два або більше разів, умова виходу з циклу й оператор приросту лічильника відповідно модифікується. Якщо кількість повторень циклу відома в момент компіляції, цикл може бути розгорнутий повністю.

Розкладання циклів (loop fission). Розкладення циклів полягає в тому, що цикл із складним тілом розбивається на кілька окремих циклів з простими тілами з тим самим простором ітерації.

Реструктуризація графу потоку управління. Структура графа потоку управління, наявність в графі потоку управління характерних шаблонів для циклів, умовних операторів і т. д. дає цінну інформацію при аналізі програми. Наприклад, по конструкціям графа потоку управління, що повторюються, 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. В якості протидії можна використати таке перетворення графу потоку управління, що робить граф однорідним ("плоским"). Оператори передачі керування на наступні за ними базові блоки, що знаходяться на кінцях базових блоків, замінюються на оператори передачі керування на спеціально створений базовий блок диспетчера, який за попереднім базовим блоком і керуючими змінними обчислює наступний блок і передає керування на нього. Технічно це може бути зроблено перенумеруванням усіх базових блоків і введенням нової змінної, яка містить номер поточного базового блоку, що виконується.

Заплутана функція замість операторів if, for і т. д. буде містити оператор switch, що знаходиться всередині нескінченного циклу.

Локалізація змінних в базовому блоці. Це перетворення локалізує використання змінних одним базовим блоком. Для кожного заплутуваного базового блоку функції створюється свій набір змінних. Всі використання локальних та глобальних змінних у вихідному базовому блоці замінюються на використання відповідних нових змінних. Щоб забезпечити правильну роботу програми між базовими блоками вставляються так звані зв'язні (connective) базові блоки, завдання яких скопіювати вихідні змінні попереднього базового блоку у вхідні змінні наступного базового блоку.

Застосування такого перетворення призводить появу в функції значного числа нових змінних, які використовуються лише в одному-двох базових блоках, що заплутує людину, яка аналізує програму.

При реалізації цього перетворення виникає необхідність точного аналізу вказівників і контекстно-залежного між процедурного аналізу. В іншому випадку не можна гарантувати, що запис по будь-якому вказівнику чи виклик функції не модифікують справжню змінну, а не поточну робочу копію.

Розширення області дії змінних. Дане перетворення за суттю є оберненим до попереднього. Це перетворення намагається збільшити час життя змінних на стільки, наскільки можливо. Наприклад, виносячи блочну змінну на рівень функції або виносячи локальну змінну на статичний рівень, розширюється область дії змінної й ускладнюється аналіз програми. Тут застосовується те, що глобальні методи аналізу (тобто, методи, що працюють над однією функцією в цілому) добре обробляють локальні змінні, проте для роботи із статичними змінними необхідні більш складні методи між процедурного аналізу.

Надалі, для заплутування програми можна об'єднати кілька таких статичних змінних в одну змінну, якщо достеменно відомо, що змінні не будуть використовуватися одночасно. Очевидно, що перетворення можна застосовувати лише до функцій, які ніколи не викликають одна одну безпосередньо або через ланцюжок інших викликів.

2.4 Основні методи дослідження обфускованого коду

В даному розділі описані методи, що застосовуються при аналізі обфускованого (заплутаного) коду програми. Ціль таких методів – виявлення залежностей між компонентами програми, що дає можливість застосувати визначені оптимізаційні перетворення, або обмежує певні оптимізаційні перетворення, що проводяться.

Методи аналізу програм можна розділити на чотири групи [9][1]:

Синтаксичні. До цієї групи відносяться методи, основані лише на результатах лексичного, синтаксичного і семантичного аналізу програми.

Статичні. До цієї групи відносяться методи аналізу потоків керування і даних та методи, основані на результатах аналізу потоків управління і даних. Статичні методи аналізу працюють з програмою, не використовуючи інформацію про роботу програми з конкретними початковими даними.

Динамічні. Динамічні методи аналізу програм використовують інформацію, отриману в результаті "спостереження" за роботою програми з конкретними вхідними даними. Відомо, що самі по собі динамічні методи рідко застосовуються для аналізу програми, оскільки, як правило, необхідна інформація про виконання програми з різними наборами вхідних даних, яка збирається за допомогою статистичних методів аналізу.

Статистичні. Статистичні методи використовують інформацію, зібрану в результаті значної кількості запусків програми з великою кількістю наборів вхідних даних.

На даному етапі розвитку методів обфускації синтаксичний аналіз обфускованого коду не є ефективним. Коротка характеристика інших методів аналізу обфускованих програм наведена нижче.

[1)] [9] Обфускация алгоритмических структур / И. В. Чумаченко, Е. Е. Малафеев, Е. Л. Шевцов // Системи обробки інформації. 2006. Вип. 8. С. 87-89.

2.4.1 Статичний аналіз

До методів статичного аналізу належать [9][1)]:

- Статичний аналіз аліасів.
- Статичне усунення мертвого коду.
- Статична мінімізація кількості змінних.
- Статичне просування констант і копій.
- Статичний аналіз доменів.
- Статичний слайсінг.

Статичний аналіз аліасів (alias analysis) необхідний в мовах, в яких кілька імен можуть бути використані для доступу до однієї й тої ж самої області пам'яті. Наприклад, деякий елемент масиву "a" може бути адресований в програмі як a[0] (канонічне ім'я елемента масиву), як a[j] або як a[-4]. В результаті аналізу аліасів кожному оператору, що виконує непрямий запис в пам'ять або непряме зчитування з пам'яті, ставиться у відповідність множина імен змінних, котрих дана операція може торкатися.

Якщо мова допускає аліаси, проведення аналізу вказівників необхідне для коректного аналізу потоків даних і для перетворення програм. У випадку доступу до елементів масивів і полям структур в найпростішому випадку ми можемо вважати, що зчитується чи модифікується одразу весь масив або ціла структура. Для вказівників або посилань в найпростішому випадку ("консервативний" аналіз) можна вважати, що непряме зчитування з пам'яті торкається усіх локальних та глобальних змінних, а непрямий запис в пам'ять може їх всіх модифікувати. Така схема блокує глибоку трансформацію практично будь-якої програми.

Аналіз вказівників не може бути використаний безпосередньо для заплутування чи розплутування програми, проте він є ключовим для точного аналізу властивостей програми.

[1)] [9] Обфускация алгоритмических структур / И. В. Чумаченко, Е. Е. Малафеев, Е. Л. Шевцов // Системи обробки інформації. 2006. Вип. 8. С. 87-89.

Статичне усунення мертвого коду (dead-code elimination) призначене для виявлення у програмі коду, який виконується, але не впливає на результат роботи програми.

Статична мінімізація кількості змінних (variable minimization) призначений для зменшення кількості використаних в функції локальних змінних за рахунок об'єднання змінних, часи життя значень в яких не пересікаються, в одну змінну. Стандартний спосіб, що застосовується для мінімізації кількості змінних, полягає в побудову графу перекриття змінних за допомогою ітераційного вирішення рівняння потоку даних.

Статичне просування констант і копій (constant and copy propagation) полягає в просуванні константних виразів якомога далі по тексту функції. Якщо вираз використовує лише значення змінних, котрі в даній точці програми містять одне заздалегідь відоме при аналізі програми значення, такий вираз може бути обрахований на етапі аналізу програми. Якщо у виразі використовується змінна, яка в даній точці програми є заздалегідь відомою копією якоїсь другої змінної, у виразі може бути представлена початкова змінна.

Статичний аналіз доменів (domain analysis) є розширенням алгоритму просування констант. Він дозволяє визначити множину значень, які може приймати дана змінна в даній точці програми, якщо ця множина не велика.

Статичний слайсінг (slicing) це побудова "скороченої" програми, з якої видалений весь код, що не впливає на обчислення заданої змінної в заданій точці (зворотний слайс), але при цьому програма залишається синтаксично і семантично коректною і може бути виконана. Окрім описаного вище оберненого слайсінгу розроблені алгоритми прямого слайсінгу. Прямий слайсинг залишає в програмі тільки ті оператори, які залежать від значення змінної, обчисленої в даній точці програми. Методи слайсінгу можуть бути корисні при розділенні "переплетених" обчислень, коли одночасно обчислюються дві незалежні один від одного величини. Наприклад, в одному циклі може обчислюватися скалярна величина двох векторів, а також мінімальний і максимальний

елемент кожного вектора, і такі цикли можуть бути розщеплені за допомогою побудови слайсів.

2.4.2 Динамічний і статистичний аналіз

До таких методів належать [9][1)]:

- Статистичний аналіз покриття.
- Статистичне порівняння трас.
- Статистична побудова графу потоку управління.
- Динамічне просування копій уздовж трас.
- Динамічне виділення мертвого коду.
- Динамічний слайсінг.

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

Статистичне порівняння трас дозволяє виявити, чи однаковими є траси програми, одержані при різних запусках на одному і тому ж самому наборі вхідних даних.

Статистична побудова графу потоку управління будує граф потоку управління на підставі інформації про порядок проходження базових блоків на одному наборі або на множині наборів вхідних даних.

Динамічне просування копій уздовж трас необхідне для точного міжпроцедурного 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 даному запуску програми, але ніяк не вплинули на результат роботи програми. Якщо аналізується сукупність запусків

[1)] [9] Обфускация алгоритмических структур / И. В. Чумаченко, Е. Е. Малафеев, Е. Л. Шевцов // Системи обробки інформації. 2006. Вип. 8. С. 87-89.

програми на множині наборів вхідних даних, можна говорити про статистичне 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 лише ті інструкції, на які вплинуло присвоєння значення даної змінної в даній точці програми. Зазначимо, що про точність динамічних методів аналізу можна говорити, лише якщо відоме повне тестове покриття програми (побудова повного тестового покриття – задача, що не має алгоритмічного вирішення). В протилежному випадку статистичне виявлення властивостей програми не дозволяє нам стверджувати, що дана властивість справедлива на всіх допустимих наборах вхідних даних. Наприклад, умова if (leap_year(current_data)) завжди буде рівна значенню "true", якщо поточний рік високосний, і значенню "false" в іншому випадку, проте усунення цього оператора з програми спричинить її неправильну роботу.

Тому описані вище динамічні методи не можуть застосовуватися в автоматичному інструменті аналізу програм. Роль цих методів полягає в тому, щоб привернути увагу користувача інструменту аналізу програм до особливостей роботи програми. Надалі користувач може вивчити "підозрілий" фрагмент коду детальніше із застосуванням інших інструментів, щоб підтвердити або спростувати висунуту гіпотезу.

Якщо непрозорі предикати і недосяжний код усуваються лише на підставі статистичного аналізу, завжди залишається можливість, що предикат був істотним. Щоб все ж таки спростити програму, можна, наприклад, винести ймовірно недосяжний код із загального графу потоку управління функції в програму, що опрацьовує спеціальні виключення, який активується кожного разу, коли предикат прийме значення, відмінне від звичного. З одного боку, граф потоку управління і потоку даних основної програми в результаті спроститься, а з іншого боку, програма збереже свою функціональність.

3 ПІДВИЩЕННЯ СТІЙКОСТІ МЕТОДУ

3.1 Алгоритм комплексного методу обфускації

В даному розділі розглядається запропонований комплексний метод обфускації підвищеної стійкості, надалі, метод маскування програм (ММ). Даний метод призначений для маскування текстів програм, що складаються із сотень функцій по кілька сотень стрічок кожна. Замасковані програми повинні відповідати обмеженням обчислювальної системи, що має безпосередній вплив на вибір методів маскування. Крім цього, великий розмір вихідних програм означає, що застосування ручного аналізу програми при її демаскуванні ускладнене часовими і вартісними обмеженнями. Для демаскування таких програм застосовуються інструментальні засоби аналізу програм і зворотної інженерії, що підтримують повний спектр існуючих статичних та динамічних методів аналізу.

Універсального методу маскування, який можна було б застосувати до всіх програм і який був би стійким до всіх можливих методів аналізу програм, немає [6][1)]. ММ розроблений ґрунтуючись на результатах аналізу опублікованих в [9][2)] і є стійким до статичних методів аналізу та найкраще задовольняє наведеним вище вимогам.

Розглянемо задачу маскування програми реалізованої на мові Сі.

ММ використовує деякі маскуючі перетворення, розглянуті в роботі [9]. Проте, він виконує їх в такій комбінації з новими перетвореннями, що застосування методів аналізу, описаних в [9], не дає результату.

Цей метод застосовується окремо до кожної функцій програми, при цьому структура програми в цілому не змінюється. Для зміни структури

---

1) [6] Чернов А.В. Анализ запутывающих преобразований программ. В сб. "Труды Института системного программирования", под. ред. В. П. Иванникова. М.: ИСП РАН, 2002.

2) [9] Обфускация алгоритмических структур / И. В. Чумаченко, Е. Е. Малафеев, Е. Л. Шевцов // Системи обробки інформації. 2006. Вип. 8. С. 87-89.

маскованої програми можуть застосовуватися стандартні методи відкритої вставки і винесення функції, розглянуті в [9][1)], які не є частиною досліджуваного методу маскування.

При маскуванні кожної функції ММ використовує, разом з локальними неістотними змінними, глобальні неістотні змінні, які формують глобальний неістотний контекст. До маскованої програми вносяться неістотні залежності по даним між істотним і неістотним контекстом функції. Наявність глобального неістотного контексту, спільно використовуваного всіма замаскованими функціями, приводить до появи в замаскованій програмі залежностей по даним між всіма функціями і глобальними змінними.

Метод ММ складається головним чином з перетворень графа потоку управління. В результаті граф потоку управління замаскованої програми значно відрізняється від графа потоку управління вхідної програми. Метод не змінює структуру даних вхідної програми, але вносить до замаскованої програми велику кількість неістотних залежностей по даним. В результаті, замаскована програма складніша ніж вхідна як по управлінню, так і по даним.

Загальна ідея методу може бути охарактеризована таким чином:

- По-перше, збільшити складність графа потоку управління, але так, щоб всі дуги графа потоку управління, внесені при маскуванні, проходилися при виконанні програми. Це дозволяє подолати основну слабкість "непрозорих" предикатів.
- По-друге, збільшити складність потоків даних маскованої функції, "вклавши" в неї програму, яка не впливає на оточення маскованої функції і, як наслідок, не змінює роботи програми. "Холоста" функція будується як з фрагментів маскованої функції, семантичні властивості яких заздалегідь відомі, так і з фрагментів, взятих з бібліотеки маскуючого транслятора. Щоб ускладнити завдання виявлення холостої частини замаскованої функції використовуються мовні конструкції, що

[1)] [9] Обфускация алгоритмических структур / И. В. Чумаченко, Е. Е. Малафеев, Е. Л. Шевцов // Системи обробки інформації. 2006. Вип. 8. С. 87-89.

важко піддаються аналізу (вказівники) і математичні тотожності.

Маскування можна поділити на кілька етапів:

Збільшення розміру графа потоку управління функції. На цьому етапі виконуються перетворення, які змінюють структуру циклів в тілі функції, а також клонування базових блоків.

Руйнування структури графа потоку управління функції. На цьому етапі в граф потоку управління вноситься значна кількість нових дуг. При цьому існуючі базові блоки можуть виявитися розбитими на декілька менших базових блоків. У графі потоку управління можуть з'явитися нові поки що порожні базові блоки. Мета цього етапу − підготувати місце, на яке надалі буде внесений неістотний код.

Генерація неістотного коду. На цьому етапі порожні базові блоки графа потоку управління заповнюються інструкціями, що не впливають на результат виконання програми. Неістотна, "холоста" частина поки що ніяк не стикається з основною, функціональною частиною програми.

"Перемішування" холостої і основної програми. Для цього використовуються як властивості програм, що важко аналізуються (наприклад, вказівники), так і математичні тотожності і нерівності.

3.1.1 Збільшення розміру графа потоку управління

Перетворення перебудови циклів полягає в тому, що в маскованій функції вибираються відповідні гнізда циклів та одиночні цикли і над ними виконуються перетворення множини індексів. До таких перетворень відносяться:

- Пониження розмірності області ітерації.
- Підвищення розмірності області ітерації.
- Зміна порядку обходу простору ітерації.
- Афінні перетворення простору ітерації.
- Часткова або повна розгортка циклу.

На етапі перебудови циклів переглядаються всі цикли в тілі функції. У кожному циклі перевіряються достатні умови для застосування кожного перетворення, визначаючи таким чином множину доступних перетворень. Потім для кожного циклу програми визначається яке перетворення буде до нього застосоване, і виконується перетворення циклів.

На рисунку 3.1 наведено приклад застосування перетворення пониження розмірності індексної множини до функції перемножування двох матриць. Перетворення дозволило перейти від потрійного вкладеного циклу до єдиного циклу.

```
enum {N=128}
typedef double
matr_t[N][N];
void mm(matr_t m1, matr_t
m2, matr_t r)
{
  int i, j, k;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      r[i][j]=0;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      for (k=0; k<N; k++)

r[i][j]+=m1[i][k]*m2[k][j]
;
}
```

(a) функція до перетворення

```
enum {N=128}
typedef double matr_t[N][N];
void mm(matr_t m1, matr_t m2,
matr_t r)
{
  int i, j, k;
  for (i=0; i<N*N; i++)
    r[i/N][i%N]=0;
  for (i=0; i<N*N*N; i++)
    r[i/(N*N)][i%(N*N)/N]+=
m1[i/(N*N)][i%N]*m2[i%N][i%(N*N)/N]
;
}
```

(b) функція після перетворення

Рис. 3.1 − Приклад перетворення пониження розмірності

Перетворення клонування базових блоків полягає в заміні послідовного виконання двох базових блоків (наприклад, *B[i]* і *B[j]*) на оператор розгалуження з виконанням копії базового блоку *B[j]* на кожній з гілок. Для цього базовий блок *B[j]* повинен бути скопійований необхідну кількість разів. На рис. 3.2 наведена схема перетворення.

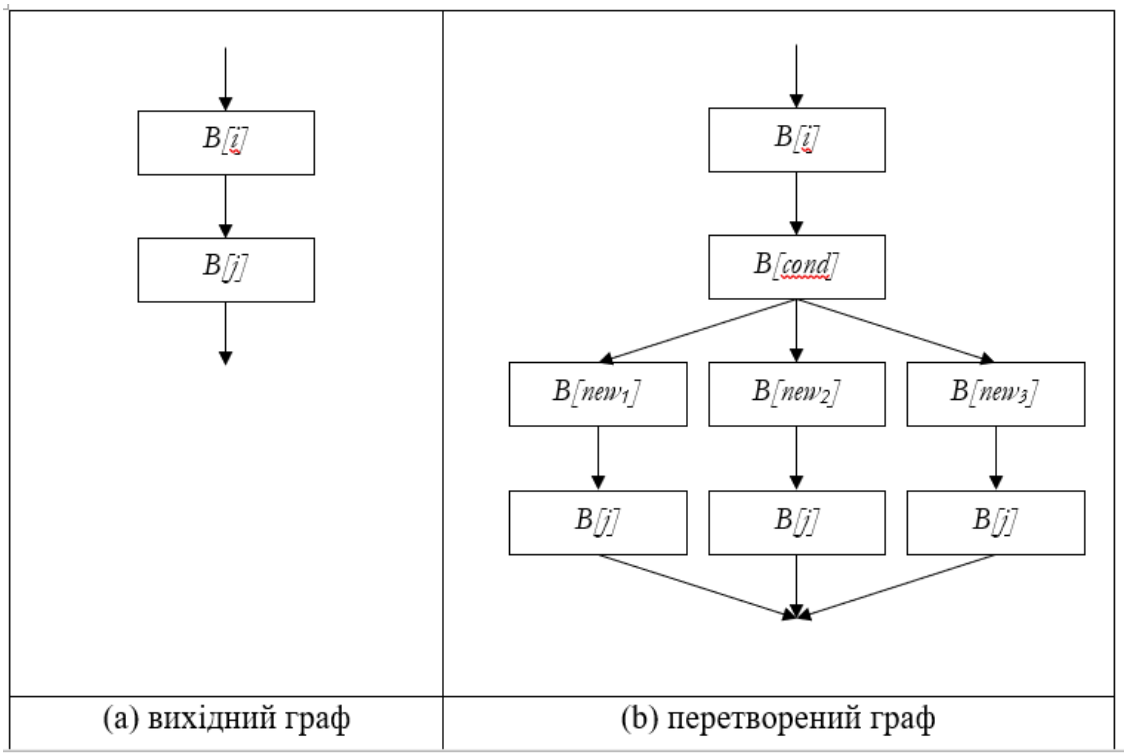


Рис. 3.2 − Схема перетворення клонування базових блоків

На рис. 3.2 базовий блок *B[j]* був розмножений двічі. Базовий блок *B[cond]* міститиме інструкції, необхідні для того, щоб кожна з трьох копій базового блоку *B[j]* виконувалася приблизно з однаковою частотою. Базові блоки $B[new_1]$, $B[new_2]$, $B[new_3]$ також можуть містити інструкції для підтримки рівномірного розподілу потоку виконання по трьом альтернативам.

Вже на етапі клонування базових блоків починається побудова паралельної "холостої" функції, яка надалі буде об'єднана з основною функцією для отримання результуючої замаскованої функції. Спочатку холоста програма будується так, що містить тільки типи даних, змінні і інструкції, необхідні для коректної роботи програми з клонованими базовими блоками. При створенні паралельної функції одночасно будуються всі структури, що містять результати аналізу потоку управління і потоків даних паралельної функції. Використовуються такі методи побудови паралельної функції, при яких її семантичні властивості заздалегідь відомі.

Для клонування вибирається базовий блок *B[j]*, у якого дуга графа потоку управління, що виходить з блоку задовольняє наступним умовам:

- Дуга не виходить з базового блоку “вхід” і не входить в базовий блок “вихід”.
- Дуга має високу відносну частоту проходження.
- Дуга є єдиною дугою, що виходить з блоку *B[j]*.

Лічильники. Базові блоки, отримані в результаті клонування, залишаються повністю еквівалентними один одному. Проте, щоб маскування було ефективнішим, необхідно, щоб всі базові блоки при роботі замаскованої програми виконувалися, причому бажано, з приблизно однаковою частотою. Для цієї мети використовуються так звані недетерміновані лічильники (НЛ). Недетерміновані лічильники є абстрактним типом даних, над яким визначені наступні операції:

```
init: int, env -> counter
get: counter, env -> int
next: counter, env -> counter
```

Тут *env* − це середовище виконання програми, що є джерелом недетермінації лічильника. Операція *init* ініціалізує лічильник. Перший параметр операції задає межу значень *N*, які генеруватиме лічильник. Операція *get* генерує ціле значення в діапазоні від 0 до *N-1*, яке може використовуватися для вибору однієї з дуг для виконання функції. Операція next модифікує поточний стан лічильника так, щоб при наступному виконанні операції get вона видавала інший результат. Операція *next* − це для кожного конкретного лічильника насправді набір операцій $next_1$, $next_2$ і т. д., реалізації яких в замаскованій програмі можуть істотно відрізнятися одна від одної.

Типи даних реалізації лічильників, їх змінні стани і інструкції, відповідні операціям над лічильниками, є частиною паралельної "холостої" функції. Змінна *C*, яка зберігає стан лічильника, може бути створена як на рівні локальних змінних маскованої функції, так і винесена в статичні змінні всієї одиниці компіляції. Останнє дозволяє розділити одну й ту ж саму змінну стану

між різними лічильниками різних функцій, що корисно ще і тим, що створює в замаскованій програмі міжпроцедурні залежності по даним.

Існують такі реалізації найпростіших видів лічильників:

- Лічильник по модулю. Це − простий вид лічильника. Стан лічильника зберігається в змінній цілого типу, який знаходиться на рівні статичних змінних одиниці компіляції. Операція *init* полягає в записі в змінну лічильника довільного числа, наприклад, значення якого-небудь параметра функції або глобальної змінної. Операція *get* полягає в отриманні залишку від ділення на *k* поточного значення змінної лічильника. Операція *next* полягає в додаванні або відніманні довільного числа, не кратного *k,* причому набір операцій *next* отримується різним вибором цього числа.
- Лінійний конгруентний лічильник. Цей вид лічильника реалізує добре відомий лінійний конгруентний метод отримання псевдовипадкових чисел [10][1)]. Операції *init* і *get* не змінюються, а операція *next* визначається як *next(cntr)(C1*cntr+C2) mod C3*, де *C2* і *C3* − взаємно прості числа. Можна також застосовувати поліноміальні конгруентні лічильники, а також будь-який інший алгоритм отримання псевдовипадкових рівномірно розподілених чисел [10].
- Криптографічні хеш-функції. Для реалізації лічильників можуть використовуватися криптографічні хеш-функції, наприклад, Md5 або SHA, або симетричні криптосистеми. Тоді операція *next* може виглядати таким чином *next(cntr) f(cntrx)*. Тут *f* − хеш-функція, а *x* − деякі довільні дані, наприклад, значення якої-небудь локальної змінної або параметра функції. Відзначимо, що алгоритми обчислення криптографічних хеш-функцій мають специфічний вигляд (вони складаються з побітових операцій), і тому можуть бути достатньо легко розпізнані при демаскуванні.

[1)] [10] Керниган Б. В., Ритчи Д.М. Язык программирования Си. СПб.: Невский диалект, 2001.

- Динамічні структури даних. Наприклад, може бути створений кільцевий список, який заповнюється числами від *0* до *k-1* в довільному порядку. Операція *get* полягає в читанні значення поточного елементу списку, а операція *next* - в просуванні покажчика на наступний елемент списку.
- Розрив базових блоків. Це перетворення полягає в тому, що базовий блок достатньої довжини розділяється на два або більше менших базових блоків. Розрив базових блоків ніяк не відбивається на коді функції і полягає в модифікації допоміжних структур, використовуваних для представлення графа потоку управління.

3.1.2 Руйнування структурності графа потоку управління

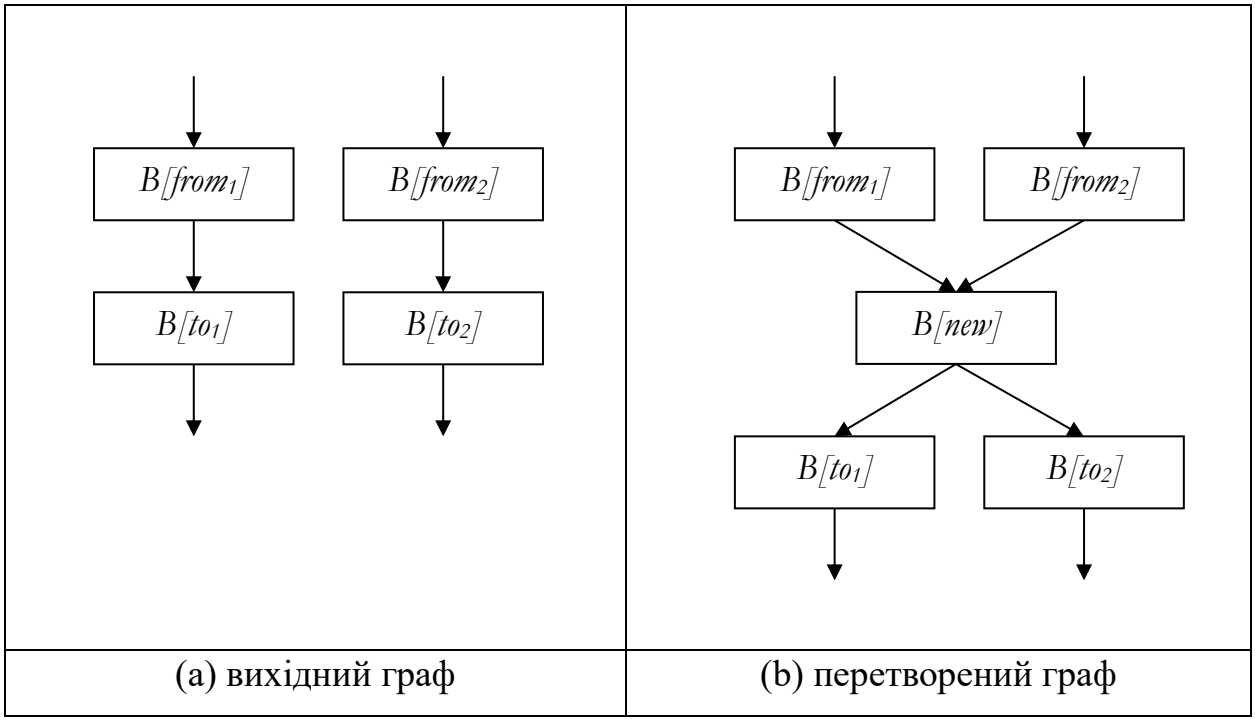
Ця група маскуючих перетворень містить з'єднання дуг і створення псевдоциклів.

Зачеплення дуг. Схема перетворення показана на рис. 3.3. Для перетворення вибираються дві випадкові дуги графа потоку управління функції. При цьому перевага надається "віддаленим" одна від одної дугам, де відстань вимірюється як мінімальна з довжин двох найкоротших шляхів по графові від кінця однієї дуги до початку іншої. Дві вибрані дуги не повинні мати спільного початку або спільного кінця. Ключовим для забезпечення надійності з'єднання дуг є предикат *P*, який в кінці виконання нового базового блоку *B[new]* забезпечує повернення на "правильний" шлях виконання. Цей предикат назвемо *зворотнім*.

До перетворення після виконання базового блоку $B[from_1]$ завжди виконувався базовий блок $B[to_1]$, а після базового блоку $B[from_2]$ завжди виконується базовий блок $B[to_2]$. В результаті виконання цього перетворення створюється новий базовий блок *B[new]*, який виконується і після $B[from_1]$, і після $B[from_2]$. Новий базовий блок завершується обчисленням предиката *P*, залежно від якого управління передається або на базовий блок $B[to_1]$, або на

базовий блок *B[to₂]*. Предикат *P* повинен гарантувати, що управління повернеться на ту гілку, з якого воно прийшло в блок *B[new]*.

Реалізовані деякі прості види *зворотних* предикатів. Простий вид *зворотного* предиката − це звичайна булева змінна. Змінна може бути оголошена як на рівні локальних змінних, так і на рівні глобальних змінних.



(a) вихідний граф | (b) перетворений граф

Рис. 3.3 − Схема перетворення з'єднання дуг.

Зворотні предикати можуть бути побудовані на основі хеш-функції. Нехай хеш-функція *f* перетворює цілочисельний тип в булевий. Введемо змінну *v*, яку використовуватимемо як аргумент *f*. Таким чином, предикат *P* рівний *f(v)*.

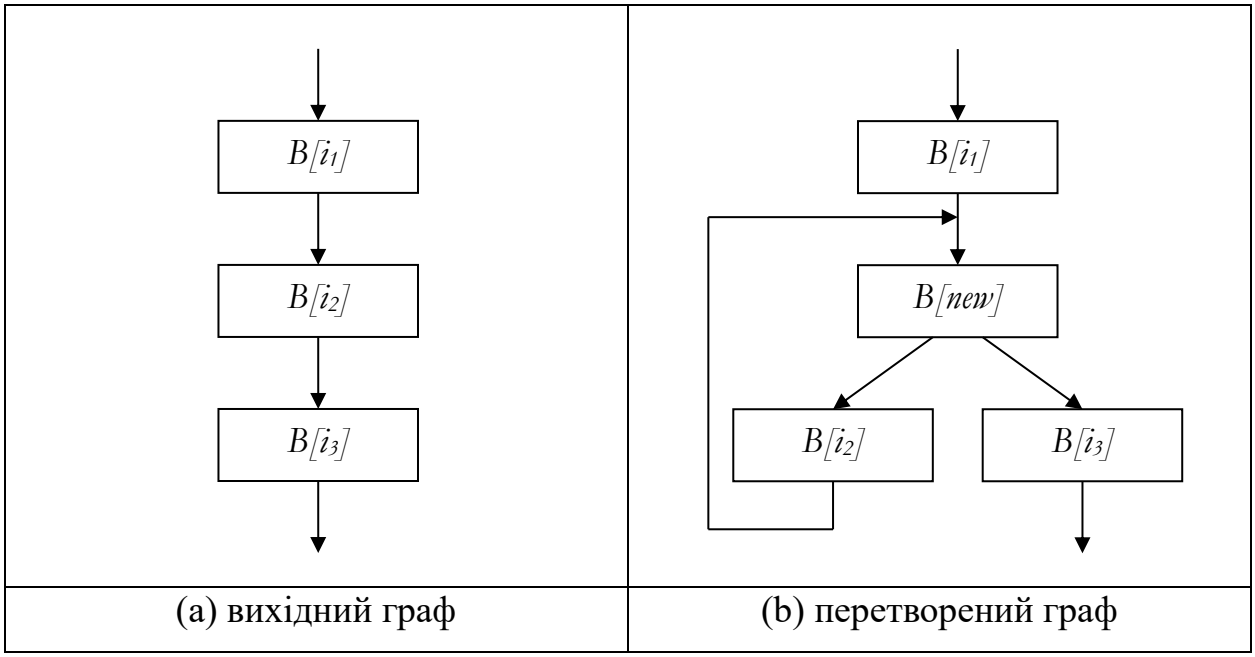
Встановлення значення *P* в *true* еквівалентне присвоєнню змінній *v* будь-якого значення *x*, на якому *f(x)= true*. Такі значення *x* можуть братися з масиву *Ptrue*, який індексується довільним виразом *e*. Тоді встановлення значення предиката *P* в *true* виконується присвоєнням *v < - Ptrue[e]*. Установка значення предиката *P* в *false* виконується аналогічно. Для побудови *зворотних*

предикатів можуть бути використані динамічні структури даних, аналогічно тому, як вони використовувалися для побудови лічильників.

Розміщення *зворотних* предикатів зазвичай супроводжується істотним недоліком, що знижує ступінь його стійкості. Всі операції з *зворотним* предикатом сконцентровані в невеликій області графа потоку управління. Використовується два способи подолання цього недоліку. По-перше, інструкції встановлення значення *зворотного* предиката позначаються як кандидати на просування вгору в графі потоку управління функції. На завершальному кроці перемішування інструкцій ці інструкції будуть переміщені вгору якомога вище. По-друге, інструкції ініціалізації *зворотних* предикатів можуть бути розміщені не в самих базових блоках $B[from_1]$ або $B[from_2]$, а в базових блоках, з яких управління може потрапити лише в потрібний блок.

Створення псевдоциклів**.** Перетворення полягає у внесенні в граф потоку управління функції зворотної дуги. При цьому контролюється, щоб тіло отриманого циклу виконувалося лише один раз. Схема перетворення показана на рис. 3.4. Тут з'єднуються дуги $B[i_1] \rightarrow B[i_2]$ і $B[i_2] \rightarrow B[i_3]$. Предикат P, що знаходиться в кінці базового блоку $B[new]$, повинен забезпечити одноразове виконання базового блоку $B[i_2]$.

Стійкість перетворення створення псевдоциклу до аналізу визначається стійкістю *зворотного* предиката. На відміну від перетворення з'єднання дуг, в якому інструкції встановлення значення *зворотного* предиката P можуть бути розміщені достатньо "далеко" від точки з'єднання дуг, перетворення створення псевдоциклу більш обмежене. Встановлення значення предиката P, щоб керування не поверталося до псевдоциклу, не може бути винесене з блоку $B[i_2]$.



(a) вихідний граф | (b) перетворений граф

Рис. 3.4 − Схема побудови “псевдоциклу”

3.1.3 Генерація неістотного коду

У масковану функцію додається велика кількість неістотного коду. Неістотний код будується таким чином, що його властивості точно відомі компілятору, що маскує програму. Наприклад, у разі використання аліасів в неістотному коді при генерації коду одночасно будується і точна множина абстрактних комірок пам'яті для кожного звернення за вказівником.

Етап генерації неістотного коду складається з наступних підетапів:

- Генерація множини типів.
- Генерація множини змінних.
- Генерація неістотного коду.

Генерація множини типів. Готуються визначення типів, які потім використовуватимуться в холостому коді. Типи для холостого коду будуються одним з наступних способів:

- Безпосередньо використовуються типи, визначені в маскованій програмі.

- Використовуються вбудовані типи.
- З вбудованих типів і типів, визначених в маскованій програмі, шляхом вбудовування в шаблонні типи маскуючого компілятора. Наприклад, з типу t, визначеного в маскованій програмі, можуть бути побудовані типи масиву елементів типу t, списки, дерева з елементами типу t.
- Модифікацією структурних типів, визначених в маскованій програмі.

Для кожного типу, що додається в масковану програму, в маскуючому компіляторі будується *клас-реалізатор*, на який спираються всі функції з генерації інструкцій для маніпуляцій з цим типом.

Генерація множини змінних. Для генерації множини "холостих" змінних використовуються типи, побудовані на попередній стадії. Змінні розміщуються як на рівні локальних змінних функції, так і на рівні глобальних змінних.

Генерація неістотного коду. Для генерації неістотного коду можна використати, наприклад, операції, які були отримані за допомогою методів getbinaryops, getunaryops, getassignops інтерфейсу Typeimplementer інтегрованого середовища Poirot. Інструкції неістотного коду розміщуються в базових блоках упереміш з інструкціями початкової функції і керуючими інструкціями.

3.1.4 «Перемішування» програм

На попередніх етапах граф потоку управління функції був збільшений в розмірах, потім в нього було внесено велику кількість неістотного коду. І предикати, додані в розширений граф потоку управління для моделювання початкового графа потоку управління, і неістотний код використовують множину змінних, що не пересікаються з основними змінними маскованої функції. В результаті функція може бути розшарована на істотну і неістотну частину, хоча це може бути ускладнено необхідністю проведення аналізу вказівників.

Щоб ускладнити відділення неістотної частини замаскованої функції в неї додається велика кількість штучних залежностей по даним між істотною і

неістотною частиною. Для цього на даний час використовуються булева, теоретико-числова і комбінаторна тотожність, а також масиви і динамічні структури даних.

Булеві тотожності. Булеві тотожності застосовується для ускладнення булевих виразів, які визначають умовні переходи. На відміну від інших видів тотожностей, булева тотожність довільної складності легко може бути отримана автоматично. Розглянемо як приклад основний метод отримання булевої тотожності, реалізований в даний час.

Булеві тотожності довільної складності можуть бути отримані з булевих тотожностей відносно простої структури за допомогою набору еквівалентних перетворень. Нехай ми хочемо скласти булеву тотожність змінними v_1, v_2 ..., v_k, серед яких є як змінні вихідної функції, так і неістотні змінні, внесені при маскуванні. Нехай e – вираз, що підлягає ускладненню. Тоді будується вираз $e \cdot o$, де $o = \left(v_1 \vee \overline{v_2}\right) \wedge \ldots \wedge \left(v_k \vee \overline{v_k}\right)$. Далі до виразу, що вийшов, застосовуються m кроків еквівалентних перетворень, які можна знайти, наприклад, в [11][1)]. В результаті отримаємо вираз e', еквівалентний e, який використовується для умовного переходу в замаскованій програмі.

Такий метод отримання булевої тотожності аналогічний методу отримання непрозорих предикатів, розглянутому в попередньому розділі, але в даному випадку використання тотожності не приводить до появи недосяжної дуги графа потоку управління, яка достатньо легко може бути виявлена. Тому використання булевої тотожності в нашому випадку виявляється значно складніше.

Комбінаторні тотожності. Всі тотожності, що розглядаються далі, як комбінаторні, так і теоретико-числові узяті, в основному, з [11]. В якості прикладу розглянемо наступну біноміальну тотожність:

$$2^n - \sum_{k=0}^{n} C_n^k = 0.$$

[1)] [11] Введение в криптографию. Под общей редакцией Ященко В.В. М.: МЦМНО, 1999.

Тотожність може використовуватися таким чином: в якості *n* береться неістотна змінна, що набуває цілих значень на невеликому інтервалі (наприклад, від 0 до 5). Генеруються інструкції, що обчислюють суму біноміальних коефіцієнтів і записують результат в тимчасову змінну, для визначеності @1. Генерується інструкція зсуву, що обчислює 2^n і записує результат в іншу тимчасову змінну, наприклад @2. Далі в початковій функції вибирається адитивний цілий вираз, результат якого зберігається в деякій тимчасовій змінній, наприклад @3, і будується новий вираз @1 + @3 - @2. Цей вираз завжди буде рівний виразу @3, але містить залежність по даним від змінної n.

Деякі інші тотожності, які можуть використовуватися аналогічним чином, наведені нижче.

$$0 = \sum_{k=0}^{n} (-1)^k C_n^k.$$

$$n2^{n-1} = \sum_{k=1}^{n} kC_n^k.$$

$$n(n-1)2^{n-2} = \sum_{k=2}^{n} k(k-1)C_n^k.$$

$$n = \sum_{k=0}^{n} (-1)^k 4^{n-k} C_{2n-k+1}^k - 1.$$

Теоретико-числові тотожності. Як приклад розглянемо відому малу теорему Ферма.

$$a^{p-1} \equiv 1 (\mathrm{mod}\, p).$$

для будь-якого цілого $a \neq 0$, $(\mathrm{mod}\, p)$ і простого *p*. При маскуванні генерується випадкове просте число *p.* Далі генеруються інструкції для обчислення $a^{p-1} \bmod p$, причому піднесення до ступеня обчислюється за допомогою розкладання *p-1* в двійкову систему. Ці інструкції утворюють достатньо довгу лінійну послідовність, яка може бути розподілена по базових блоках маскованої функції. Результат обчислення виразу додається як множник в який-небудь мультиплікативний вираз вихідної програми, або як множник до змінної.

Інші теоретико-числові тотожності, які також можуть використовуватися для внесення залежностей по даним, наведені нижче. Узагальненням малої теореми Ферма є теорема Ейлера:

$$x^{\varphi(n)} \equiv x(\mathrm{mod}\, n).$$

де n та x довільні цілі числа, $\varphi(n)$ – функція Ейлера, що рівна кількості взаємно простих з n цілих чисел, менших n.

$$(n-1)! \equiv -1(\mathrm{mod}\, n)$$ (теорема Вілсона)

тоді і лише тоді, коли n – просте число.

Використання масивів і динамічних структур даних. Динамічні структури даних можуть використовуватися і для створення штучних залежностей між даними. Головний розрахунок тут робиться на те, що на даний час не існує задовільного алгоритму аналізу аліасів, що виникають при використанні вказівників та індексів масивів.

Як простий спосіб можна запропонувати розміщення всіх локальних змінних одного типу в масиві. У тексті функції замість імені змінної тепер використовуватиметься індекс масиву. Навіть випадків, коли індекс завжди є літеральною константою, буде досить, щоб заплутати прості алгоритми аналізу аліасів, які розглядають масив як єдине ціле.

У момент маскування програми відомо, які елементи масиву зайняті істотними, а які – неістотними змінними. Більш того, розподіл неістотних змінних по масиву може вибиратися довільним чином. Це може використовуватися для побудови залежностей по даним. Нехай f – функція, що ставить у відповідність довільному цілому значенню деякий індекс в масиві, за яким знаходиться неістотна змінна. Тоді штучні залежності між даними будуються за допомогою виразів виду $vars[f(e_1)] = e_2$ або $vars[f(e_1)] = vars[f(e_2)]$. Тут $vars$ – це масив змінних, e_1, e_2 – вирази, які містять як істотні, так і неістотні змінні.

Один з простих способів використання динамічних структур даних для внесення залежностей між даними полягає в тому, що всі значення змінних всіх типів зберігаються в списку, що знаходиться в динамічній пам'яті. Для доступу до змінних замість їх імені використовується розіменування

спеціальних вказівників. Крім цього, вказівники на неістотні змінні час від часу змінюють своє положення в списку. В результаті виявиться, що всі звернення до колишніх локальних змінних функції звертаються до об'єктів в області динамічної пам'яті. Для розділення істотних і неістотних змінних потрібен буде аналіз аліасів в динамічній пам'яті, здатний працювати з динамічними структурами даних довільної глибини. В даний час не існує такого методу статичного аналізу аліасів.

3.2 Обґрунтування стійкості комплексного методу

Стійкість замаскованої програми до автоматичного аналізу обґрунтовується наступними спостереженнями:

- Більшість методів статичного аналізу потоків даних не підтримують аналіз масивів з точністю до елементу. Для таких методів аналізу всі звернення до масиву локальних змінних будуть рівноправними, що призведе до виявлення помилкових залежностей між даними.
- Для виявлення залежностей по даним між змінними глобального контексту потрібний міжпроцедурний аналіз програми. За наявності великого числа глобальних змінних (як істотних, так і неістотних) і великого числа функції, глобальний аналіз виявиться або неточним, або складним.
- Те ж саме зауваження справедливе і для аналізу вказівників на область динамічної пам'яті. Існуючі методи аналізу або неточні, або непридатні до програм великого розміру.

Стійкість замаскованої програми до ручного аналізу обґрунтовується наступними міркуваннями:

- Напівстатичний аналіз (трасування) замаскованої програми не дозволяє в ній виявити явних закономірностей, таких як дуги графа потоку управління, що ніколи не виконуються, або блоки, що регулярно виконуються, як диспетчер. Відсутність явних статистичних

закономірностей робить напівстатичний аналіз менш ефективним, ніж у випадках, розглянутих у попередньому розділі.

- Інструкції, що забезпечують стійкість замаскованої функції, розподілені по всіх базових блоках функції, а не сконцентровані на невеликій ділянці, як в схемі диспетчера. Тому демаскування вимагає аналізу всієї функції, а не певної її частини.
- Великий розмір замаскованих функцій навіть для відносно невеликих функцій вихідної програми є перешкодою для ручного аналізу.
- Кожне перетворення, що становить метод маскування, набуває параметрів в широких межах (як правило, випадковим чином). Інформація, отримана в результаті аналізу однієї замаскованої функції, лише частково може бути застосована до аналізу іншої замаскованої функції.
- Граф потоку управління має таку структуру, що його візуалізація може дати незадовільний результат.

Алгоритми візуалізації можуть відобразити граф потоку управління так, що це лише ускладнить розуміння алгоритму програми, або взагалі не зможуть відобразити такий граф.

Наведені тут міркування, звичайно, не замінюють формальних доказів тверджень про складнісить демаскування. Проте слід зазначити, що використовуваний на даний час підхід, який полягає в зведенні якого-небудь обчислювально-важкого завдання до завдання демаскування програми, дозволяє стверджувати про складність демаскування методами статичного аналізу лише у гіршому випадку.

Крім того, на даний час не існує математичного апарату, придатного для оцінки складності напівстатичного і динамічного аналізу програм, який має ефективну евристичну компоненту.

3.3 Приклад застосування

Як приклад застосування досліджуваного методу маскування розглянемо невелику програму, яка вирішує задачу про 8 ферзів. Текст цієї програми наведено далі (на мові Сі).

```
#include <stdio.h>
static int up[15], down[15], rows[8], x[8];
static void print(void)
{
  int k;
  for (k=0; k<8; k++)
    printf("%c",x[k]+'1');
  printf("\n");
}
static void queens(int c)
{
  int r;
  for (r=0; r<8; r++)
    if (rows[r] && up[r-c+7] && down[r+c])
    {
      rows[r]=up[r-c+7]=down[r+c]=0;
      x[c]=r;
      if (c==7)
        print();
      else
        queens(c+1);
      rows[r]=up[r-c+7]=down[r+c]=1;
    }
}

int main(void)
{
  int i;
  for (i=0; i<15; i++)
    up[i]=down[i]=1;
  for (i=0; i<8; i++)
    rows[i]=1;
  queens(0);
  return 0;
}
```

Для маскування обрано основну функцію queens цієї програми. Граф потоку управління функції queens наведено на рис. 3.5.

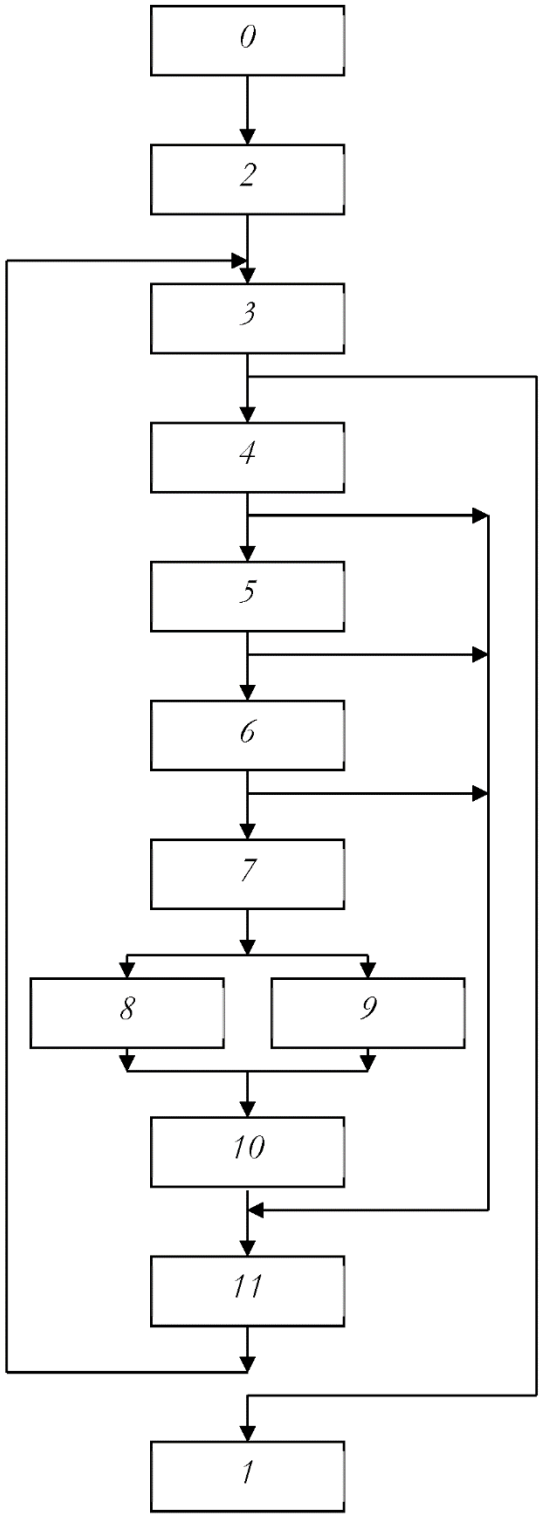


Рис. 3.5 − Граф потоку управління функції queens

Результат маскування функції queens наведено у додатку А.

Граф потоку управління замаскованої функції наведений на рис. 3.6.

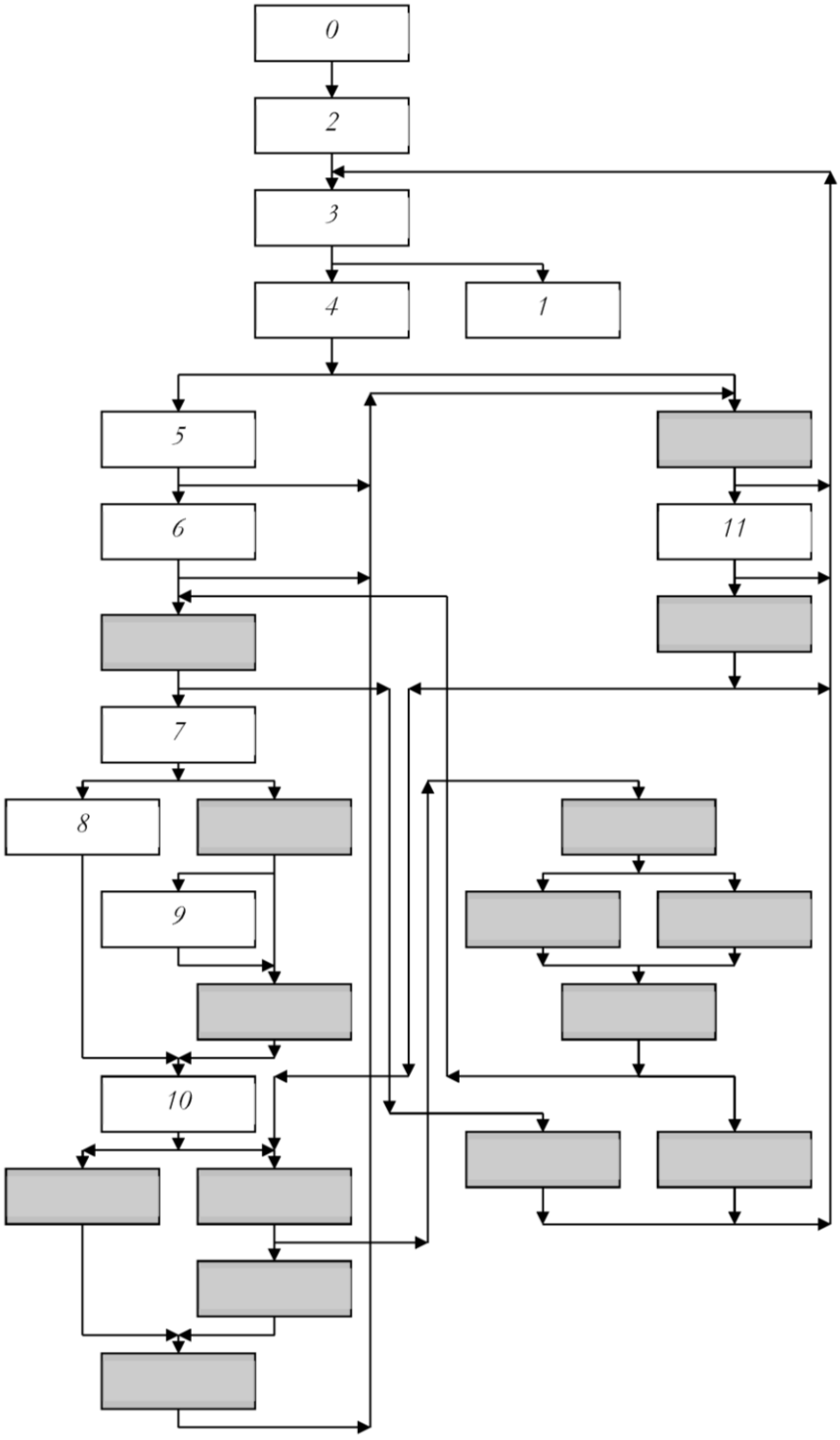


Рис. 3.6 − Граф потоку управління замаскованої функції queens

Принципи розглянутого методу маскування відображається в прикладі таким чином:

– Масив q використовується для зберігання локальних змінних функції queens. В результаті методи аналізу потоку даних, які не розрізняють

окремі елементи масиву, покажуть залежності по даним між всіма операціями з масивом *q*, тобто між всіма локальними змінними функції queens.

- Для протидії алгоритмам аналізу потоку даних, що розрізняють елементи масиву, для доступу до масиву *q* використовується змінна, записана в елемент масиву *q[11]*. Змінна ініціалізувалася значенням 6 в рядках 9-18 функції. Для цього використовується константа 32, що відображалася в циклі як кількість біт в цілому слові, і константа 13 – кількість елементів в масиві локальних змінних, яка зберігається в елементі *q[12]* для подальшого використання. Для аналізу програми необхідно встановити, що елементи *q[11]* і *q[12]* є константами, що в даному випадку можливо для методів, які розрізняють елементи масиву, але обчислення цих констант потребує інтерпретації програми.
- Для протидії алгоритмам просування констант, які могли б розповсюдити константні значення *q[11]* і *q[12]* по функції, значення *q[11]* переобраховується в рядках 82, 129, 145. При цьому значення *q[11]* кожного разу не змінюється.
- У випадку, якщо в результаті аналізу замаскованої функції вдалося виділити кожен елемент масиву в окрему змінну, замаскована функція все ще містить весь неістотний код, внесений при маскуванні. Алгоритм виявлення мертвого коду не дасть результатів, оскільки в ньому використовуються тотожності в рядках 70-79 і 91-96, що вносять хибні залежності по даним між основною і неістотною частиною функції queens.

У розділі розроблено та досліджено комплексний метод обфускації програмного коду. Обґрунтована підвищена стійкість запропонованого методу до дослідження. Теоретичні результати підтверджено прикладом практичної реалізації.

4 ПЛАНУВАННЯ РОБІТ ЩОДО АНАЛІЗУ ТА ДОСЛІДЖЕННЯ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ МЕТОДОМ ОБФУСКАЦІЇ

4.1 Зміст етапів планування

Для проведення раціонального планування дослідження захисту програмного забезпечення методом обфускації програмного коду за методом синхрокільця є скорочення термінів виконання досліджень при мінімальних затратах трудових, матеріальних і фінансових ресурсів. Для забезпечення можливості більш чіткого планування в дипломній роботі використана система сіткового планування і управління (СПУ).

Основа системи СПУ полягає в побудові графічних моделей процесів, забезпеченні можливості оцінки поточного стану і прогнозування подальшого ходу розробки.

Чітке узгодження всіх робіт в часі, виявлення вирішальної (критичної) за термінами послідовності робіт від початку до кінця розробки і зосередження уваги керівників на цих роботах, оптимізація проектування, як в часі (забезпечення мінімальної тривалості циклу), так і по вартості (забезпечення мінімуму витрат), можливість оперативного коректування складних планів за допомогою електронно-обчислювальної техніки – ось далеко не повний перелік переваг методів СПУ.

В пояснювальній записці до дипломної роботи планування робіт щодо дослідження захисту програмного забезпечення методом обфускації програмного коду методом СПУ було здійснено в такій послідовності:

1 Розділення комплексу робіт на окремі етапи;
2 Виявлення і опис всіх подій і робіт, необхідних для виконання поставленої мети;
3 Визначення часу проведення кожної роботи;
4 Побудова сіткового графіка;

5 Розрахунки параметрів сіткового графіка;

6 Аналіз сіткового графіка і його оптимізація.

4.2 Виявлення і опис подій і робіт

Всі події і роботи, що входять в комплекс робіт, зведено в таблицю в порядку їх послідовності, тобто складається бібліотечний список. Цей список включає перелік подій та їх індексацію, а також перелік робіт та їх коди (табл. 4.1.).

Таблиця 4.1 - Перелік подій та робіт в ході аналізу та дослідження захисту програмного забезпечення методом обфускації

Номер події	**Найменування події**	**Код роботи**	**Зміст роботи**
0	Видача технічного завдання науково-дослідної роботи.	0-1	Отримати у керівника завдання науково-дослідної роботи.
1	Опрацювання технічного завдання.	1-2 1-3	Сформулювати мету дослідної роботи та погодити її з керівником дипломної роботи. Скласти попередній перелік програмного забезпечення та обладнання необхідного для виконання дипломної роботи.
2	Пошук та опрацювання інформаційних джерел.	2-3 2-4	Уточнити замовлення на поставки програмного забезпечення та обладнання для виконання дипломної роботи. Робота в бібліотеках зі спеціалізованою літературою, періодичними виданнями та пошук інформації у глобальній мережі Internet.
3	Придбання програмного забезпечення та обладнання	3-5	Сформувати перелік загроз для програмного забезпечення та проаналізувати документацію на програмне забезпечення.
4	Підготовка обладнання для проведення дослідження	4-6 4-5	Встановити програмне забезпечення. Перевірити функціональність програмного забезпечення.
5	Налагодження програмного забезпечення	5-6 5-10	Провести експериментальні дослідження.

Номер події	Найменування події	Код роботи	Зміст роботи
6	Проведення дослідження	6-7	Провести обробку результатів досліджень та їх оформлення.
7	Обробка та оформлення результатів досліджень	7-8	За результатами проведених досліджень описати проведену роботу з аналізу загроз та механізмів захисту, та сформулювати висновки.
8	Формулювання рекомендацій щодо механізмів захисту програмного забезпечення	8-9 8-11	Оформити результати дослідної роботи згідно з вимогами методичних вказівок до дипломного проектування.
9	Підготовка, оформлення та погодження техніко-економічного розділу науково-дослідної роботи з консультантом із економічних питань	9-11	Оформити результати техніко-економічного розділу науково-дослідної роботи згідно з вимогами методичних вказівок до дипломного проектування.
10	Підготовка, оформлення і погодження розділу охорони праці та техніки безпеки науково-дослідної роботи з консультантом із охорони праці	10-11	Оформити результати розділу охорони праці та техніки безпеки науково-дослідної роботи згідно з вимогами методичних вказівок до дипломного проектування.
11	Представлення роботи науковому керівнику та завідувачу кафедри для попереднього захисту результатів	11-12	Внести доповнення та поправки до пояснювальної записки дипломної роботи.
12	Представлення роботи на рецензію	12-13	Підготувати графічну частину до захисту науково-дослідної роботи.
13	Захист результатів дипломної роботи в екзаменаційній комісії		

4.3 Визначення часу виконання робіт

Під час планування тривалості виконання робіт в ході дослідження захисту програмного забезпечення методом обфускації програмного коду

використовувалися узагальнені дані проведення минулих робіт (з врахуванням досвіду їх виконання).

Використовуються дві ймовірні оцінки часу, що даються відповідальними виконавцями кожній роботі: мінімальний час (t_{min}) та максимальний час (t_{max}). Ці оцінки переважно визначаються в днях. Вони є вихідними для розрахунку очікуваного часу виконання роботи, який визначається таким чином:

$$t_0 = \frac{3t_{\min} + 2t_{\max}}{5} \qquad (4.1)$$

Мірою розкиду відхилень очікуваного часу виконання роботи є дисперсія σ_{t0}^2, яка визначається за формулою:

$$\sigma^2 = \left(\frac{t_{\max} - t_{\min}}{5} \right)^2 \qquad (4.2)$$

Розв'язання двох рівнянь (4.1) і (4.2) для кожної роботи дає необхідні значення очікуваного часу і його дисперсію, потрібну для оцінки ймовірності виконання події в заданих термінах.

Результати розрахунку трудомісткості робіт представлено у таблиці 4.2.

Таблиця 4.2 - Трудомісткість виконання робіт та виконавців в ході аналізу та дослідження захисту програмного забезпечення методом обфускації

Код роботи	**t_{min}, дні**	**t_{max}, дні**	**t_0, дні**	**Кількість виконавців**	**Категорія виконавців**
0-1	1	2	1.4	2	Студент, Науковий керівник
1-2	7	9	8	2	Студент, Науковий керівник
1-3	5	11	7.4	2	Студент, Науковий керівник
2-3	5	11	7.4	2	Студент, Науковий керівник
2-4	13	15	13.8	1	Студент
3-5	10	15	12	1	Студент
4-5	10	15	12	2	Студент, Науковий керівник
4-6	10	15	12	2	Студент, Науковий керівник
5-6	11	14	12.2	1	Студент
5-10	5	10	7	2	Студент
6-7	11	15	12.6	1	Студент
7-8	6	9	7.2	2	Студент, Науковий керівник
8-9	5	10	7	2	Студент, Науковий керівник

Код роботи	t_{min}, дні	t_{max}, дні	t_0, дні	Кількість виконавців	Категорія виконавців
8-11	7	11	8.6	1	Студент
9-11	10	12	10.8	2	Студент, Науковий керівник
10-11	5	12	8	2	Студент,
11-12	1	2	1.4	1	Студент
12-13	1	2	1.4	1	Студент

4.4 Побудова сіткового графіка

Згідно з визначенням, сіткою є орієнтований граф з направленням зв'язків від вихідної події до кінцевої.

Кожна подія з більшим порядковим номером зображується правіше попередньої. Нумерування здійснюється після побудови сіткового графіка.

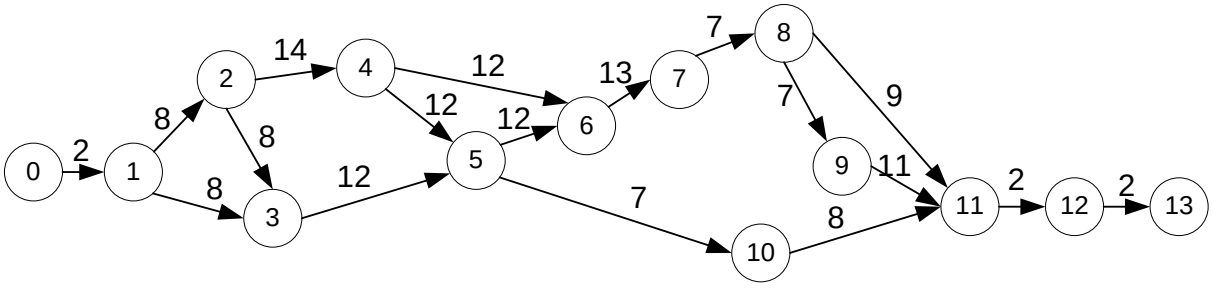


Рис. 4.1 − Побудова сіткового графіка

4.5 Розрахунок параметрів сіткового графіка

До основних параметрів сіткового графіка належать:

t(L) - тривалість шляху;

$t(L_{кр}) = T_{кр}$ - тривалість критичного шляху;

R(L) - резерв часу шляху;

Tp_i - ранній термін здійснення події;

$TП_I$ - пізній термін здійснення події;

$Трп_{ij}$- ранній термін початку роботи;

$Трз_{ij}$ - ранній термін завершення роботи;

$Тпп_{ij}$ - пізній термін початку роботи;

$Тпз_{ij}$ - пізній термін завершення роботи;

$Rп_{ij}$ - повний резерв часу роботи;

$Яв_{ij}$ - вільний резерв часу роботи;

$Кн_{ij}$ - коефіцієнт напруженості роботи.

Ці параметри можна розрахувати різними методами: аналітичним (за формулами), табличним, графічним, матричним і з використанням ЕОМ. Параметри сіткового графіку розраховані аналітично. Тривалість шляху визначається як сума тривалостей робіт, що складають даний шлях. Визначимо довжину всіх шляхів:

Т 1= 1+2+4+6+7+8+11+12+13 = 2+8+14+12+13+7+9+2+2= 69 днів.

Т 2= 1+2+4+5+6+7+8+11+12+13=2+8+14+12+12+13+7+9+2+2=81 день.

Т 3= 1+2+4+5+10+11+12+13=2+8+14+12+7+8+2+2= 55 днів.

Т 4= 1+2+4+6+7+8+9+11+12+13=2+8+14+12+13+7+7+11+2+2= 78 днів.

Т 5= 1+2+4+5+6+7+8+9+11+12+13=2+8+14+12+12+13+7+7+11+2+2= 90 днів.

Т 6= 1+2+3+5+6+7+8+11+12+13=2+8+8+12+12+13+7+9+2+2= 75 днів.

Т 7= 1+2+3+5+6+7+8+9+11+12+13=2+8+8+12+12+13+7+7+11+2+2 = 84 дні.

Т 8= 1+3+5+10+11+12+13=2+8+12+7+8+2+2=55 днів.

Т9 = 1+3+5+6+7+8+11+12+13=2+8+12+12+13+7+9+2+2=67 днів.

Т10 = 1+3+5+6+7+8+9+11+12+13=2+8+12+12+13+7+7+11+2+2=76 днів.

Отже, тривалість критичного шляху складає 90 днів. Резерв шляху виконання робіт наведено в таблиці 4.3.

Таблиця 4.3 – Резерв шляху виконання робіт

Номер шляху	Тривалість часу, дні	Резерв часу, дні
1	69	14
2	81	26
3	55	0

Номер шляху	Тривалість часу, дні	Резерв часу, дні
4	78	23
5	90	35
6	75	20
7	84	29
8	55	0
9	67	12
10	76	21

4.6 Аналіз сіткового графіка і його оптимізація

Одним з перших кроків аналізу побудованого графіка був перегляд топології сітки. При цьому перевірена нумерація подій, встановлена доцільність вибору робіт і структури сітки. Поряд із встановленням зайвих робіт і перевіркою доцільності встановленого рівня їх деталізації розглядались питання про можливість паралельного виконання робіт, виходячи з особливостей запланованого процесу і кількості робітників.

Наступним кроком аналізу сіткового графіка була проведена його оптимізація. При цьому було розв'язано задачі щодо виявлення можливостей скорочення тривалості критичного шляху і кращого розподілу різноманітних видів ресурсів (трудових, матеріально-технічних, фінансових).

Час виконання робіт, що лежать на критичному і підкритичному шляхах, в загальному було скорочено, застосуванням наступних правил:

а) замінено послідовний порядок виконання робіт паралельним там, де це дозволяє технологія;

б) поділено роботи і по можливості суміщено їх у часі;

в) змінено топологію сіткового графіка шляхом перегляду технології виконання робіт.

Після досягнення заданого терміну розробки була проведена оптимізація розподілу ресурсів. Черговість оптимізації за окремими видами ресурсів встановлювалась в залежності від значення кожного з них в даних конкретних умовах або за вказівками консультанта.

ВИСНОВКИ

У роботі проведено класифікацію методів захисту програмного забезпечення.

Встановлено, що метод обфускації є перспективним, проте потребує вдосконалення в напрямку підвищення стійкості обфускованого програмного коду до дослідження.

Розроблено та досліджено комплексний метод обфускації програмного коду.

Обґрунтовано підвищення стійкості до дослідження програмного коду, обфускованого комплексним методом.

Отримані результати підтверджено прикладом практичної реалізації.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 Software Management: Security Imperative, Business Opportunity, https://gss.bsa.org/

2 Бобало Ю.Я. Інформаційна безпека: навчальний посібник / Ю.Я. Бобало, І.В. Горбатий, М.Д. Кіселичник, А.П. Бондарєв, С.С. Войтусік, А.Я. Горпенюк, О.А. Нємкова, І.М. Журавель, Б.М. Березюк, Є.І. Яковенко, В.І. Отенко, І.Я. Тишик. Львів: Видавництво Львівської політехніки, 2019. 580 с.

3 Програмні технології захисту інформації: конспект лекцій для студентів за напрямом підготовки 6.050103 «Програмна інженерія» факультету інформаційних технологій УжНУ / Розробник: к.т.н. Поліщук В.В. Ужгород: 2018. – 80 с.

4 Технології захисту інформації [Електронний ресурс] : підручник для студ. спеціальності 122 «Комп’ютерні науки», спеціалізацій «Інформаційні технології моніторингу довкілля», «Геометричне моделювання в інформаційних системах» / Ю. А. Тарнавський; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 2,04 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2018. – 162 с.

5 Остапов С. Е. Технології захисту інформації : навчальний посібник / С. Е. Остапов, С. П. Євсеєв, О. Г. Король. – Х. : Вид. ХНЕУ, 2013. – 476 с. (Укр. мов.)

6 Чернов А.В. Анализ запутывающих преобразований программ. В сб. "Труды Института системного программирования", под. ред. В. П. Иванникова. М.: ИСП РАН, 2002.

7 Chow S., Gu Y., Johnson H., Zakharov V. An approach to the obfuscation of control-flow of sequential computer programs. LNCS 2200, pp. 144-155, 2001.

8 Stepanenko I., Kinzeryavyy V., Nagi A., Lozinskyi I. Modern obfuscation methods for secure coding // Ukrainian Scientific Journal of Information Security, 2016, vol. 22, issue 1, p. 32-37, ISSN 2225-5036 (Print), ISSN 2411-071X (Online) http://infosecurity.nau.edu.ua

9 Обфускация алгоритмических структур / И. В. Чумаченко, Е. Е.

Малафеев, Е. Л. Шевцов // Системи обробки інформації. 2006. Вип. 8. С. 87-89.

10 Керниган Б. В., Ритчи Д.М. Язык программирования Си. СПб.: Невский диалект, 2001.

11 Введение в криптографию. Под общей редакцией Ященко В.В. М.: МЦМНО, 1999.

12 Barak B., Goldreich O., Impagliazzo R., Rudich S., Sahai A., Vadhan S., Yang K.. On the (Im)possibility of Obfuscating Programs. LNCS 2139, pp. 1-18, 2001.

13 Wroblewski G. General Method of Program Code Obfuscation. PhD Thesis. Wroclaw, 2002.

14 Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979.

15 Дал У., Дейкстра Э., Хоор К. Структурное программирование/ М.: Мир, 1975.

16 Ben-Or M., Canetti R., Goldreich O. Asynchronous secure computation// Proc 25th ACM Symposium on Theory of Computing. 1993. P.52-61.

17 Micali S., Rogaway Ph. Secure computation// Advances in Cryptology - CRYPTO'91, Proceedings, Springer-Verlag LNCS, V.576. 1992. P.392-404.

18 Collberg C., Thomborson C., Low D. A Taxonomy of Obfuscating Transformations. Department of Computer Science, The University of Auckland, 1997.

ДОДАТОК А

Результат маскування функції queens

```
static void queens(int c)
{
  int q[13u];
  q[10] = 0;
  q[8] = 0;
  q[1] = (c + 7);
  q[4] = 0;
  q[7] = 0;
  q[2] = -1;
  q[11] = 0;
 L127:  if (!q[2]) goto L128;
  q[11]++;
  q[2] = (unsigned) q[2] >> 1;
  goto L127;
 L128:  q[12] = q[11]+q[2]-19;
  q[11] = q[11] % q[12];
  L111:
  if (q[q[11]+4] >= 8) goto L45;
  q[3] = q[10];
  q[q[11]+2] = ((q[8] + q[7]) + 1);
  q[9] = (q[10] + 1);
  q[1] = ((q[1] + c) + 2);
  q[q[11]+4] = (q[10] + 2);
  q[7] = ((7 + q[q[11]+2]) - q[1]);
  if ((v3)[q[3]] == 0) goto L94;
  q[4] = ((v8)[q[3]] + (v2)[((q[3] - c) + 7)]);
  if ((v7)[((q[3] - c) + 7)] == 0) goto L94;
  if (q[7] < 0) goto L96;
  if (q[7] <= 7) goto L97;
  L96:
  q[7] = ((q[3] - c) + 7);
  L97:  if ((v4)[(q[3] + c)] == 0) goto L94;
  q[1] = (q[8] + (v2)[q[7]]);
  q[2] = (((c * (c + 1)) * q[3]) % 4);
  goto L99;
  L122:  if (q[5] <= 0) goto L101;
  (v4)[(q[3] + c)] = 0;
  (v2)[((q[3] - c) + 7)] = 0;
  (v7)[((q[3] - c) + 7)] = 0;
  (v3)[q[3]] = 0;
  (v5)[c] = q[3];
  (v8)[c] = q[q[11]+2];
  q[5] = ((q[5] + 1) != 2);
  goto L102;
  L101:  q[7] = (q[7] + 2);
  if (q[7] <= 14) goto L103;
  q[7] = (q[3] + c);
  L103:  q[4] = (v2)[q[7]];
  (v4)[(q[3] + c)] = 0;
  q[1] = 0;
  (v7)[((q[3] - c) + 7)] = 0;
  (v8)[q[3]] = q[1];
  (v3)[q[3]] = 0;
  (v5)[c] = q[3];
  q[5] = ((q[5] + 4) != 5);
```

```
  q[1] = (v8)[c];
  L102:  if (c != 7) goto L105;
  L104:  print();
  q[4] = ((c * 5) + 4);
  q[q[11]+2] = (v8)[c];
  (v8)[c] = q[4];
  goto L106;
  L105:  q[q[11]] = ((c * 5) + 3);
  (v2)[((q[3] - c) + 7)] = ((q[8] + q[1]) - 7);
  L115:
  if ((q[6] % 5) < 2) goto L108;
  q[4] = ((q[q[11]] % 5) + c);
  if ((q[4] % 7) != 0) goto L109;
  q[4] = 1;
  L109:  q[1] = ((q[4] * q[4]) % 7);
  q[1] = ((q[1] * q[1]) * q[1]);
  c = ((c + (q[1] % 7)) - 1);
  queens(((c + 1)));
  q[11] = (q[q[11]+5] + q[12]) % 13;
  L106:  (v4)[(q[3] + c)] = 1;
  (v2)[(q[3] + c)] = q[q[11]+2];
  (v7)[((q[3] - c) + 7)] = 1;
  (v8)[q[3]] = 1;
  (v3)[q[3]] = 1;
  q[4] = (((v2)[(q[3] + c)] + c) + 7);
  L94:  q[1] = (q[4] > 5);
  if ((v3)[q[9]] == 0) goto L111;
  if ((v7)[((q[9] - c) + 7)] != 0) goto L112;
  if (q[1] != 0) goto L111;
  if (q[4] <= 5) goto L111;
  L112:  if ((v4)[(q[9] + c)] == 0) goto L111;
  q[6] = (((q[9] + c) * 5) + 1);
  q[1] = ((q[q[11]] + q[8]) + 1);
  goto L115;
  L108:  (v2)[((q[9] - c) + 7)] = q[5];
  (v4)[(q[9] + c)] = 0;
  (v8)[q[9]] = (v3)[q[9]];
  (v7)[((q[9] - c) + 7)] = 0;
  q[7] = (q[9] + c);
  (v3)[q[9]] = 0;
  q[8] = ((q[5] + q[2]) + 7);
  (v5)[c] = q[9];
  (v8)[c] = q[q[11]+2];
  if (c != 7) goto L117;
  L116: print();
  q[1] = (v8)[c];
  (v8)[c] = q[4];
  goto L118;
  L117:  (v8)[(c + 1)] = q[1];
  queens(((c + 1)));
  L118:  q[8] = ((v2)[(q[9] + c)] + q[1]);
  q[1] = ((q[8] + q[7]) + q[5]);
  if (((v1)[((((q[9] - c) + 7) ^ q[5]) % 4)] % 4) != 1) goto
L120;
  (v2)[((q[9] - c) + 7)] = ((q[7] + q[9]) - c);
  (v4)[(q[9] + c)] = 1;
  q[4] = (q[q[11]+2] + 1);
  (v8)[q[9]] = 1;
  (v7)[((q[9] - c) + 7)] = 1;
  (v3)[q[9]] = 1;
```

```
q[11] = (q[11] + q[q[11]+6]) % 13;
goto L111;
L120:  q[2] = ((((v7)[(q[9] - c)] + 7) | 1211) % 6);
q[4] = (q[8] + ((v7)[(q[9] - c)] | 1211));
q[7] = (q[7] + 1);
L99:  if ((v6)[q[2]] > (v6)[(q[2] + 1)]) goto L122;
(v2)[(q[9] + c)] = ((v6)[q[2]] + c);
q[8] = (((q[1] + q[4]) + q[9]) - 7);
(v4)[(q[9] + c)] = 1;
q[4] = (v8)[q[9]];
(v8)[q[9]] = 1;
(v7)[((q[9] - c) + 7)] = 1;
q[7] = (q[7] - 1);
(v3)[q[9]] = 1;
q[q[11]+5] = (q[11] + q[12]) % 13;
goto L111;
L45: ;
   }
```