Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента	*Візира Владислава Вікторовича*
	(ПІБ)
академічної групи	*121М-20-1*
	(шифр)
спеціальності	*121 Інженерія програмного забезпечення*
	(код і назва спеціальності)
освітньої програми	*«Інженерія програмного забезпечення»*
	(назва освітньої програми)
на тему:	*Методи, алгоритми та програмне забезпечення для реверсивної інженерії під різні архітектури процесорів*

_____*В.В. Візир*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
розділів кваліфікаційної роботи				
спеціальний	*Проф. Алексєєв М.О.*			
економічний	*Проф. Вагонова О.Г.*			
Рецензент	*Проф. Корнієнко В.І.*			
Нормоконтролер	*Доц. Приходченко С.Д.*			

Дніпро
2022

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

_____ І.М. Удовик
(підпис) (прізвище, ініціали)

« » _____ 20 21 року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

студенту *121М-20-1* *Візиру Владиславу Вікторовичу*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Методи, алгоритми та програмне забезпечення для реверсивної інженерії під різні архітектури процесорів*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 10.12.2021 р. № 1036-с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процес реверсивної інженерії файлів різних архітектур.

Предмет досліджень – методи та підходи до аналізу з урахуванням особливостей різних архітектур.

Мета НДР – обрати та обґрунтувати найкращі методи й інструменти реверсивної інженерії з урахуванням внутрішніх алгоритмів роботи та індивідуальних особливостей архітектур процесорів.

Вихідні дані для проведення роботи – теоретичні та експериментальні дослідження внутрішніх особливостей архітектур, алгоритмів роботи інструментів аналізу.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Новизна запропонованих рішень визначається тим, що було досліджено та обрано ефективні методи аналізу виконуваних систем з урахуванням алгоритмів роботи та особливостей архітектур процесорів в залежності від умов та мети дослідження.

Практична цінність результатів полягає у тому, що запропоновані у роботі інструменти й методи дозволяють пришвидшити процес реверсивної інженерії з урахуванням сучасних архітектур вбудованих систем.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати досліджень мають містити опис та обґрунтування алгоритмів аналізу з урахуванням архітектури процесору та кінцевої мети дослідження.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз теми та постановка задачі	12.09.2021-29.09.2021
Дослідження методів, алгоритмів, інструментів аналізу та особливостей архітектур процесорів в контексті реверсивної інженерії	30.09.2021-15.11.2021
Експериментальні дослідження та обґрунтування обраних методик й інструментів	15.11.2021-31.12.2021

6 РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект від реалізації результатів роботи очікується позитивним завдяки підвищенню якості та швидкості процесу реверсингу, що зменшує витрати та підвищує цінність роботи.

7 ДОДАТКОВІ ВИМОГИ

Завдання видав _____ *Алексєєв М.О.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Візир В.В.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *12.09.2021 р.*

Термін подання кваліфікаційної роботи до ЕК *20.01.2022*

РЕФЕРАТ

Пояснювальна записка: 83 стор., 39 рис., 3 додатки, 37 джерел.

Об'єкт дослідження: процес реверсивної інженерії файлів різних архітектур.

Предмет дослідження: методи та підходи до аналізу з урахуванням особливостей різних архітектур.

Мета роботи: обрати та обґрунтувати найкращі методи й інструменти реверсивної інженерії з урахуванням внутрішніх алгоритмів роботи та індивідуальних особливостей архітектур процесорів.

Методи дослідження. Для вирішення поставлених задач використані наступні методи: аналізу даних, алгоритмів та структур даних, принципи об’єктно-орієнтованого програмування.

Новизна отриманих результатів визначається тим, що було досліджено та обрано ефективні методи аналізу виконуваних систем з урахуванням алгоритмів роботи та особливостей архітектур процесорів в залежності від умов та мети дослідження.

Практична цінність результатів полягає у тому, що запропоновані у роботі інструменти й методи дозволяють пришвидшити процес реверсивної інженерії з урахуванням сучасних архітектур вбудованих систем.

Область застосування. Запропоновані рішення й методи можуть буди застосовані компаніями або індивідуальними фахівцями, що надають послуги реверсивної інженерії.

Значення роботи та висновки. Запропоновані інструменти та методи досліджень програм і систем дозволяють прискорити процес реверсивної інженерії та аудити інформаційних систем для пошуку вразливостей та прискорення розробки.

Прогнози щодо розвитку досліджень. Необхідно провести глибоке дослідження окремих архітектур і створити набір готових програм або плагінів, що прискорюють аналіз певної архітектури без участі інженера.

У економічному розділі були проведені розрахунки трудомісткості розробки програмного забезпечення, витрат на створення програмного забезпечення і тривалості його розробки, а також проведені маркетингові дослідження та аналіз перспективності даної галузі.

Список ключових слів: реверсивна інженерія, архітектура процесора, виконуваний файл, дизасемблер, декомпілятор, обфускація, оптимізація, динамічний аналіз, статичний аналіз.

ABSTRACT

Explanatory note: 83 pages, 39 figures, 3 applications, 37 sources.

Object of research is the process of reverse engineering for different CPU architecture.

Subject of research is methods of binary analysis according to unique CPU architecture features.

Purpose of Master's thesis is to choose and explain the most convenient and suitable methods for binary analysis taking into account CPU architecture characteristics and internal algorithms of existing tools.

Research methods. To solve existing problems used such methods as data analysis, principles of object-oriented programming, algorithms and data structure.

Originality of research is in chosen methods and instruments which make process of binary analysis faster and provide higher quality.

The practical value of the results is that the proposed in the paper algorithms, techniques and tools help to increase binary analysis speed and quality in age of embedded devices.

Scope of application. The proposed approaches could be used by companies and individuals which provide reverse engineering services.

The value of the work and conclusions. The proposed methods, algorithms and instruments for binary analysis help to increase quality of reverse engineering as part of development lifecycle or malware analysis sphere.

Research forecast and development. It is necessary to research more unique features of CPU architectures and to create special tools for binary analysis without manual research.

In the Economics section we calculated complexity of software development, the cost of creating software, duration of its development and did marketing research with analysis of sphere perspectives.

List of key words: reverse engineering, processor architecture, executable file, disassembler, decompiler, obfuscation, optimization, dynamic analysis, static analysis.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface;

BE – Big Endian, порядок байт в файлів від меншого до більшого;

BP – Base Pointer, регістр процесора;

BSOD – Blue Screen of Death, назва стану помилки ОС Windows;

CPU – Central Processing Unit;

DSP – Digital Signal Processor;

ELF – Executable and Linkable Format, формат файлів Unix систем;

GPE – General Purpose Register;

IOT – Internet of Things;

IR – Intermediate Representation;

ISA – Base Pointer, регістр процесора;

JVM – Java Virtual Machine;

LE – Little Endian, порядок байт в файлів від більшого до меншого;

LP – Link Pointer, регістр процесора;

PC – Program Counter, регістр процесора;

PE – Portable Executable, формат файлів Windows;

RISC – Reduced Instruction Set Computer;

RTTI – Run-time Type Identification;

SP – Stack Pointer, регістр процесора;

SPE – Special Purpose Register;

SRE – Software Reverse Engineering;

ЗМІСТ

ВСТУП

З розвитком технологій все більшу популярність на ринку мають вбудовані пристрої. Першим елементом цієї сфери ж безліч IoT приладів. Дані продукти в більшості випадків працюють як автономні пристрої, також умови роботи можуть бути екстремальні: високі перепади вологи, температури повітря тощо. Усі ці фактори призводять до використання специфічних типів процесорів при розробці. Їх основними перевагами є невелика ціна, розмір та енергоефективність.

Останнім часом через розвиток чіпів такого типу навіть персональні комп'ютери переходять на подібні архітектури. Такі зміни впливають на індустрію аналізу даних продуктів та програмного забезпечення, що їх використовую. Так як дані пристрої мають великий вплив на життя людей, то й до безпеки їх використання варто приділяти багато уваги.

Метою роботи є дослідження алгоритмів, методів та інструментів реверсивної інженерії з урахування особливостей архітектур процесорів, що використовуються в сучасних пристроях. До таких приладів можна віднести системи розумного дому, прилади зі сфери автомобільної промисловості та інструменти індустріальної роботи для моніторингу та перевірки станів роботи чи середовища.

Через різноманіття архітектур для їх ефективного аналізу потрібно враховувати унікальні особливості кожної з них разом із сучасними інструментами та методами аналізу.

Новизна отриманих результатів визначається тим, що було досліджено та обрано ефективні методи аналізу виконуваних систем з урахуванням алгоритмів роботи та особливостей архітектур процесорів в залежності від умов та мети дослідження.

Запропоновані інструменти та методи досліджень програм і систем дозволяють прискорити процес реверсивної інженерії та аудити інформаційних систем для пошуку вразливостей та прискорення розробки. рішення й методи

можуть буди застосовані компаніями або індивідуальними фахівцями, що надають послуги реверсивної інженерії.

Кваліфікаційна робота складається з чотирьох розділів, висновку та трьох додатків, а також містить 83 сторінок тексту, 39 рисунків та список використаних джерел.

У першому розділі досліджується й описується поняття реверсивної інженерії, після цього досліджується мета реверсингу в сучасності та сфери використання такого аналізу, що напряму впливає на вибір інструментів та методик досліджень

У другому розділі описуються підходити до аналізу виконуваних файлів та досліджуються інструменти, що використовуються під час реверсивної інженерії. Разом з тим досліджуються їх внутрішні алгоритми та методи роботи, які варто взяти до уваги під час аналізу.

У третьому розділі досліджуються внутрішні індивідуальні особливості архітектур процесорів в контексті реверсивної інженерії програмного забезпечення. Після цього були сформовані етапи аналізу файлів, можливі перешкоди й методи їх вирішень. В практичній частині роботи були створені тестові файли для порівняння результатів аналізу різними інструментами для формування висновків.

У четвертому розділі були проведені розрахунки трудомісткості розробки програмного забезпечення, витрат на створення програмного забезпечення і тривалості його розробки, а також проведені маркетингові дослідження та аналіз перспективності даної галузі.

РОЗДІЛ 1
СУТНІСТЬ ТА МЕТА РЕВЕРСИВНОЇ ІНЖЕНЕРІЇ

1.1. Визначення реверсивної інженерії

Інженерія програмного забезпечення полягає в тому, що програміст або інженер змушує виконувати комп'ютер якийсь набір дій. Таке абстрактне формулювання потрібно для того, що показати всю складність та велику кількість рівнів абстракції про які навіть не думають із сучасними інструментам розробки. Для того, щоб зрозуміти в чому суть реверсивної інженерії, потрібно йти від зворотного. Пайплайн розробки програмного забезпечення в першу чергу полягає в написанні коду. Код по суті є звичайним текстовим файлом, з яким зручно працювати саме людям. Незалежно від мови програмування кожна програма виконується саме на процесорі, а отже або через процес компіляції, або за допомогою інших рівнів абстракції, як віртуальна машина java або інтерпретатор python, програмні тексти перетворюються в ассемблерні інструкції. Якщо код в більшості випадків є платформо незалежним, то коди мовою асемблера використовують жорсткий набір певних інструкції та специфічних для CPU операторів. Програма мовою асемблера перетворюється у виконуваний машинний код за допомогою спеціальної програми. Процес перетворення називають асемблюванням або збіркою. Як результат ми отримуємо двійковий файл. Він зберігає в собі усю необхідну інформацію для запуску. До неї відноситься машинний код, ресурси, данні.

Реверсивна інженерія – це процес або практика аналізу програмної системи або її частини для відтворення внутрішньої логіки, використаних залежностей та вихідного коду маючи лише бінарні файли [1]. В якості вхідних даних інженер отримує виконуваний файл. За допомогою набору програмного забезпечення інженер намагається відтворити той набір інструкцій, з яких цей файл був зібраний.

1.2. Призначення реверсивної інженерії

Реверсивна інженерія програмного забезпечення має на меті відтворення вихідного коду з бінарного файлу, а отже може використовуватись для копіювання відомих програм задля створення аналогів, що є предметом дискурсу з точки зору моралі. В такому контексті процес реверсингу насправді є інструментом хакінгу. Хакерів розділяють на так званих “black hat” та “white hat”. Мета black hat хакерів є фінансова вигода від своєї діяльності. Вони використовують в тому числі реверсивну інженерію програмного забезпечення для викрадення інтелектуальної власності ПЗ, створення вірусів, пошуку вразливостей для злому системи. Для боротьби з такими діями й захисту однією із задач в процесі збірки программного забезпечення є набір інструментів, які направлені на зміну вихідного файлу таким чином, щоб ускладнити процес реверсивної інженерії. До таких інструментів відносяться обфускація коду, шифрування даних, заборона процесу відладки тощо.

Так як реверсивна інженерія є відтворенням та аналізом асемблерних інструкцій, то саме цей процес намагається ускладнити програма обфускатор. Обфускація - приведення початкового коду або виконуваного програмного коду до вигляду, який зберігає його функціональність, але ускладнює аналіз, розуміння алгоритму роботи і модифікації при декомпіляції [2]. Заплутування коду може здійснюватися на рівні алгоритму, початкового коду або асемблерного коду. Для створення заплутаного асемблерного коду можуть використовуватися спеціалізовані компілятори, які використовують неочевидні або недокументовані можливості середовища виконання програми. Існують також спеціальні програми, що дозволяють заплутати код на етапі збірки, які називаються обфускаторами. Але цей процес ніколи не зможе повністю захистити власників ПО від реверсингу, а лише робить його складнішим та довшим.

Шифрування даних програми також направлене на ускладнення процесу реверсингу. Незашифровані дані як імена змінних чи строки в середині коду

можуть бути серйозною підказкою щодо контексту чи мети даного коду або функції. Найпростішим прикладом буде змінна password, що показує де саме відбувається процес аутентифікації. Або ж специфічні тексти, які вказують на те, який саме алгоритм шифрування був використаний.

White hat хакери працюють з тими ж проблемами й перешкодами під час реверсингу, але мають іншу мету. Фінансова вигода полягає в виплаті винагороди власниками програмного забезпечення. Їх основною діяльністю є аналіз й пошук вразливостей. Це може бути як замовлення працівника для аудиту програмного комплексу, так і так звані bug bounty програми, які зобов'язуються виплачувати кошти будь кому, хто зможе знайти й повідомити про вразливість системи.

Втім реверсивна інженерія програмного забезпечення не обмежується хакінгом. Метою можуть бути аналіз вразливого програмного забезпечення або ж створення прикладу вразливості на основі потенційної загрози. Цей процес полягає в тому, щоб маючи інформації про вразливість або помилку в системі довести те, що за допомогою цієї вразливості дійсно можна нанести шкоду системі й отримати несанкціонований доступ. Адже інколи існують вразливості, які насправді не несуть великої загрози й витрачати час та кошти на виправлення не має сенсу. Існує навіть спеціальні правила аналізу ризиків, які вираховують ймовірність та ціну нанесеної шкоди і порівнюють її з витратами на захист та виправлення.

Наступною задачею реверсингу є підтримка та робота з так званим legacy програмним забезпеченням. Ринок IoT з кожним роком стає все більше. Також розвивається індустрія Automotive, яка працює з програмним забезпеченням в автомобілях. Ці індустрії пов'язані використанням окремих вбудованих пристроїв, які найчастіше виробляються незалежними постачальниками разом із програмним забезпеченням. Вендори можуть не надавати доступ до вихідних кодів своїх продуктів, що робить неможливим їх перевірку на необхідний рівень безпеки без реверсивної інженерії. Також сюди можна віднести ефективну взаємодію та процес відладки в системі зі сторонніми продуктами.

Реверсивна інженерія може використовуватись як один з етапів в пайплайні прямої інженерії ПЗ. Окрім використання реверсингу як аудиту рівня захисту програми від взлому його можна використовувати на етапі розробки й написання коду. Найчастіше це вивчення недокументованих можливостей ПЗ або навіть цілої операційної системи, яскравим прикладом якої є MacOS, вивчення внутрішньої структури комунікаційних протоколів в мережі, алгоритмів шифрування файлів або ж їх збірки й архівування.

1.3. Висновки до першого розділу

Отже реверсивна інженерія є дуже важливим методом аналізу програмного забезпечення, що використовується в якості аудиту рівня захисту ПЗ або аналізу на вразливості. Процес є зворотнім до розробки та створення програмного забезпечення й направлений на виявлення внутрішньої логіки й архітектурних особливостей системи. Основним об'єктом аналізу є виконувані файли й існуюча документація. Окрім цього реверсинг широко використовується під час розробки системного програмного забезпечення. Основними напрямками є дослідження алгоритмів залежностей, з якими на надається вихідний код, недокументованих можливостей програмного забезпечення або операційних систем, внутрішньої структури зашифрованих файлів, архівів, мережевих пакетів.

РОЗДІЛ 2
АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РЕВЕРСИВНОЇ ІНЖЕНЕРІЇ

2.1. Дебагер (Debugger)

Дебагер – основний вид ПЗ для реверсивної інженерії. Він дозволяє перевірити, що виконує програма в даний момент часу. Програма дозволяє запускати файли в спеціальному середовищі й контролювати процес виконання, робити зупинки, переглядати вміст пам’яті процесу. Користувач даної програми для реверсингу програмного забезпечення отримує ті самі переваги від дебагера, що й програміст під час відладки програми з вихідним кодом. До основного функціоналу дебагера відноситься виконання коду по інструкціям, точки зупинки, аналіз пам’яті процесу в даний момент часу.

Один із найпопулярніших відладчиків в сфері реверсингу під ОС Windows є OllyDbg [3]. Це 32-бітний відладчик для аналізу та модифікації зібраних виконуваних файлів та бібліотек, що працюють в середовищі користувача. Для відладки програм в середовищі ядра з більшими привілеями використовуються інші інструменти тому що структура ПЗ там значно відрізняється. OllyDbg містить в собі багато сторонніх інструментів для аналізу програм. Відладчик має графічний інтерфейс користувача й може працювати без процесу інсталяції, що полегшує його використання на окремих незалежних пристроях (IoT, automotive). До основних можливостей OllyDbg відносять:

- Інструменти аналізу файлу
    - Трейси регістрів процесора
    - Аналіз інструкцій
    - Аналіз циклів та складних структур даних
- Інструменти роботи з динамічними бібліотеками
- Сканування об’єктних файлів
- Підключення до програм в процесі роботи

- Аналіз багатопоточного ПЗ
- Аналіз API функцій стандартної бібліотеки
- Зміна пам’яті та патчінг програми на льоту

OllyDbg прекрасно забезпечує потреби своєї авдиторії й може обходити захист від відладки. Однією з найкращих особливостей OllyDbg є її відкрита архітектура. Це дозволило інженерам з усього світу створювати додаткові плагіни протягом років.

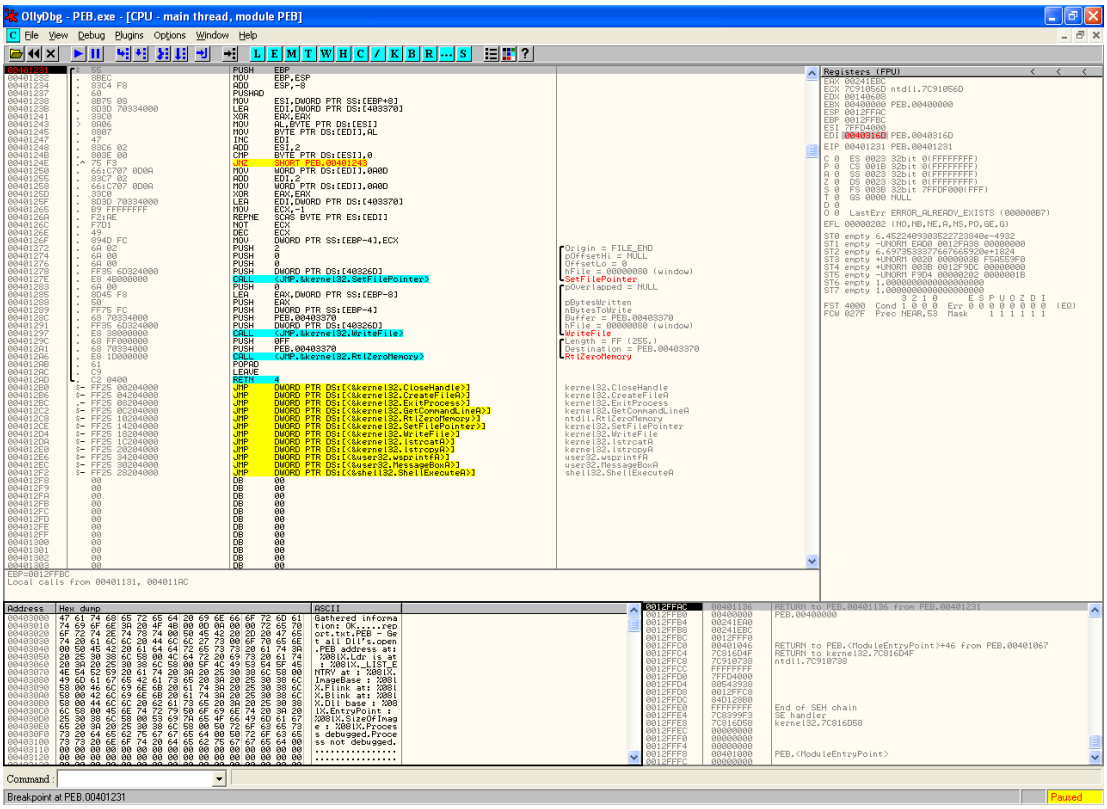

Рис. 2.1. Інтерфейс OllyDbg

Windbg – продукт компанії Microsoft для відладки процесів в операційній системі Windows. Незважаючи на менш привабливий інтерфейс ця програма є незамінною саме завдяки відладці процесів в середовищі ядра [4]. Він також надає можливості для аналізу дампів пам’яті ядра після виникнення bsod у

системі.

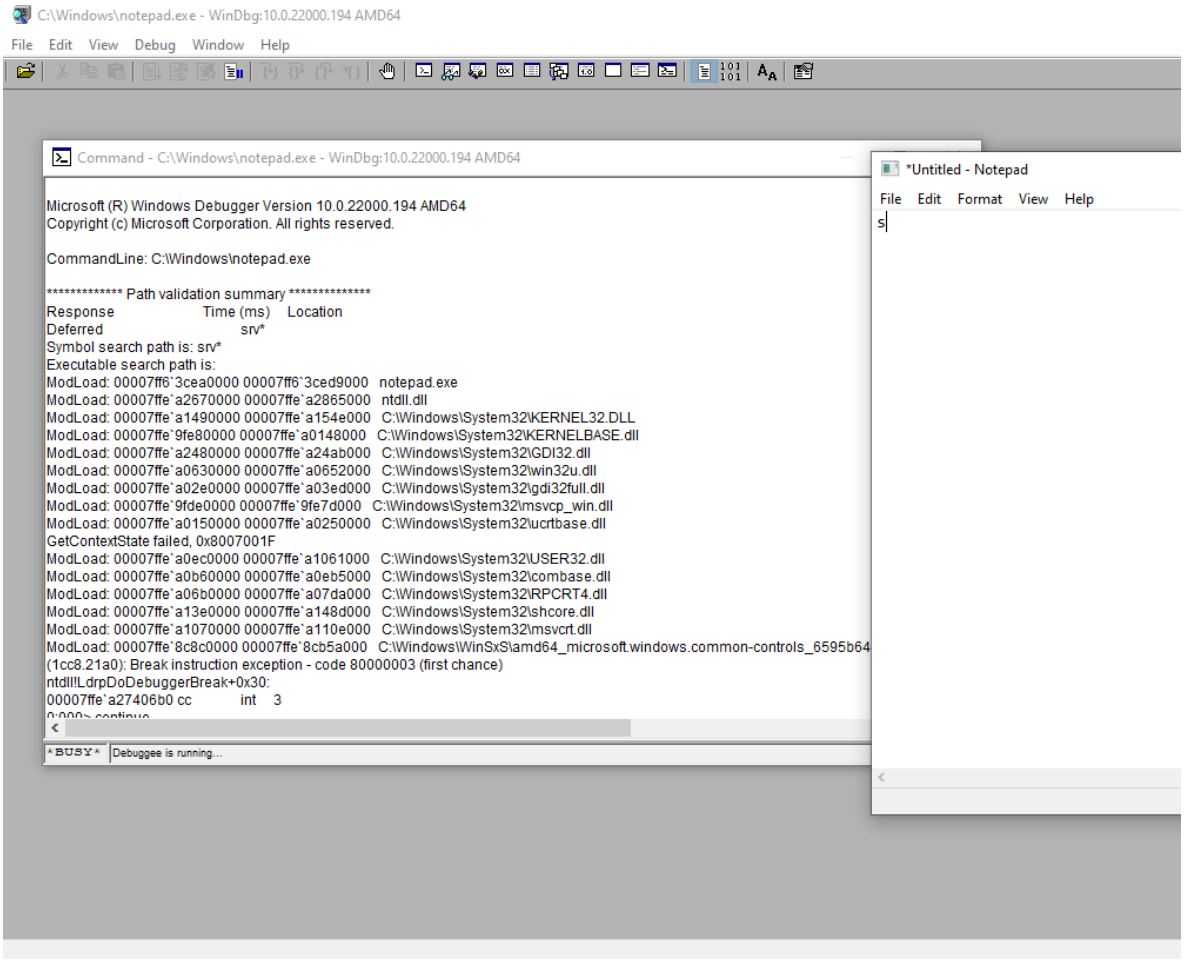

Рис. 2.2. Інтерфейс Windbg

До категорії дебагерів також можна віднести утиліту strace. По суті дана програма не є дебагером, але в певній мірі дозволяє отримати інформацію про програму під час запуску. Strace показує які системні виклики робить програма під час виконання. Системний виклик – рівень абстракції, що відгороджує середовище користувача від середовища ядра. Програми в середовищі користувача являються ізольованими операційною системою. Вони не можуть напряму взаємодіяти з реальними пристроями, такими як диск, монітор, мережева карта тощо. За усі перелічені дії відповідає ядро й програми в його середовищі. Системні виклики є інтерфейсами до цього середовища. Таким чином програма strace показує як саме програма взаємодія з ядром ОС. Якщо

скомпілювати програму, що виводить на екран “Hello World” та запустити її з утилітою strace – то можна побачити цю взаємодію:

```
      // Відкриття й читання файлу стандартної бібліотеки
      openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libstdc++.so.6",
O_RDONLY|O_CLOEXEC) = 3
      read(3, "\177ELF\2\1…

      // Запис тексту на екран
      write(1, "Hello, World: 5\n", 16Hello, World: 5)= 16
      // Завершення процесу
      exit_group(0)
```

У випадку більш складних програм інженер за допомогою strace може аналізувати роботу з мережею, створенням дочірніх процесів, відкриття файлів тощо.

2.2. Шістнадцятирічні редактори

Раніше шістнадцятирічні редактори були окремими програмами з досить обмеженим функціоналом. Попри це, вони справлялись зі своєю основною задачею – відображення даних у бінарному файлі. З часом такі редактори обросли багатьма додатковими функціями, таким як вбудовані дизасемблери, калькулятори, пошук регулярних виразів й навіть шифрування чи дешифрування фрагментів тексту. Також шістнадцятирічні редактори стали однією з функцій більш потужних сервісів, які надають безліч інструментів для аналізу.

Основною задачею редактора є показ вмісту бінарного файлу в шістнадцятирічному вигляді. Певні послідовності байт вже будуть знайомі для інженера й можуть показати йому початок певної секції коду, специфічний заголовок або ж данні в відкритому вигляді. За допомогою редактора можна також змінювати файл для в процесі відладки для того щоб зайти в певну гілку коду або ж обійти специфічну перевірку. Або ж взагалі змінити або додати власні інструкції для більш ефективного аналізу.

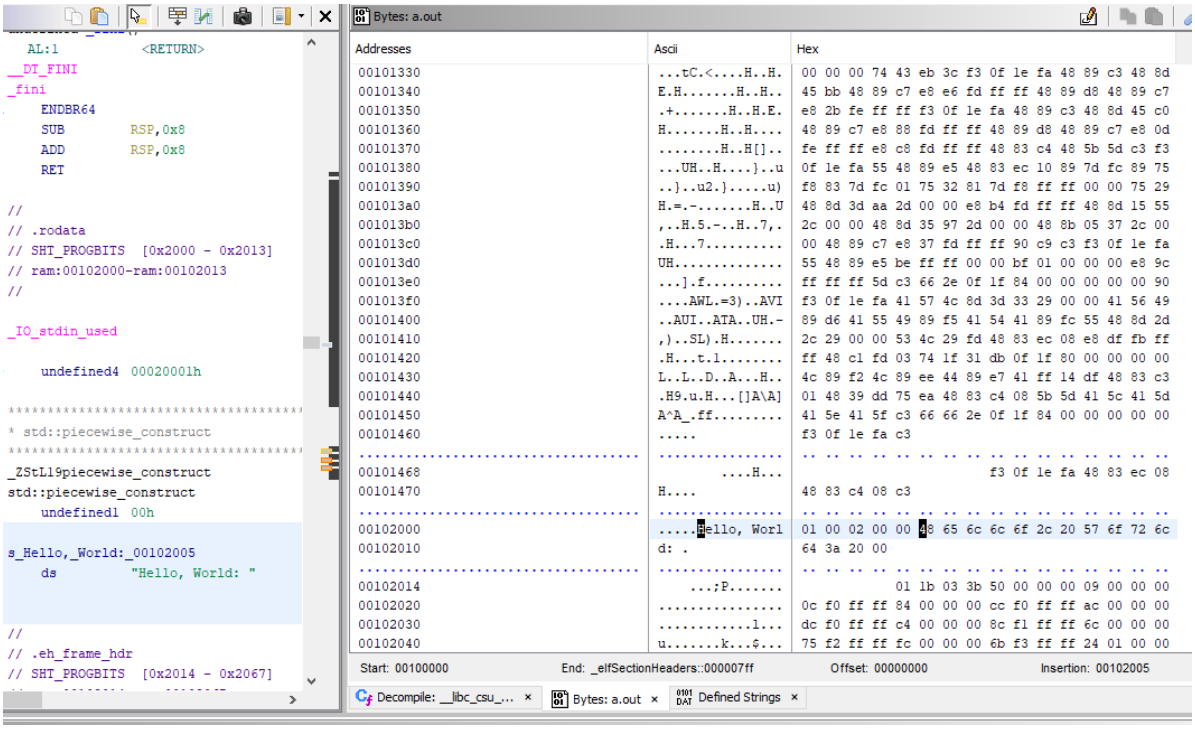

Рис. 2.3. Інтерфейс редактору як функції в Ghidra

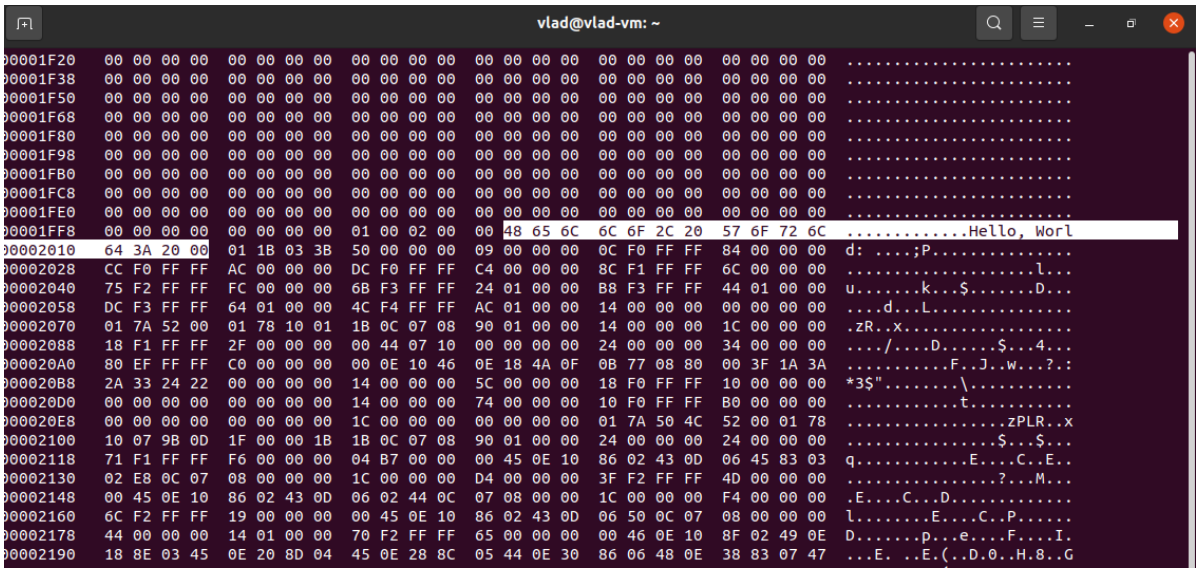

Рис. 2.4. Текстові данні виконуваного файлу hello_world.out

Отже зараз редактори стали частиною програмного забезпечення й використовуються як одна з опцій. Втім редактори як незалежне програмне забезпечення все же користується попитом, особливо якщо воно розвертає весь інтерфейс користувача в терміналі (рис. 2.4). Такими редакторами зручно

користуватись тоді, коли немає необхідності або можливості запускати інше програмне забезпечення з купою залежностей й виконувати повний аналіз файлу, що може займати купу часу, особливо коли файл має великий розмір. В такому випадку зручною є функція перегляду файлу на віддаленій машині або ж на мікроконтролері, доступ до якою є тільки через вікно терміналу.

2.3. Фаззинг

Fuzzing – техніка тестування програмного забезпечення на вразливості. Її ідея полягає в тому, що б передати на вхід у програму помилкові дані й знайти той набір вхідних даних, який може привести до помилки. Програми фаззери налаштовуються таким чином, що передають програмному забезпеченню безліч даних протягом певного проміжку часу. В процесі тестування ці данні можуть бути змінені, цей процес називають мутацією й за його допомогою фаззери шукають послідовності вхідних даних, що призводять до неочікуваних помилок в системі.

Фаззинг як техніка використовується не тільки тестувальниками програмного забезпечення, а й реверсивними інженерами під час пошуку вразливостей. Для них це є етапом реверсингу, який може вказати на потенційно вразливий код в програмі. Наступним етапом є використання цих даних для перевірки й доведення, що така помилка може нанести шкоду системі й дає несанкціонований доступ.

Як приклад можна розглянути програму, яка отримує на вхід число й виводить його на екран. Типовим фаззером буде перебір усіх можливих чисел як позитивні, негативні, нуль, числа, що виходять за розмір типових типів даних. Також можна протестувати передачу чисел з плаваючою точкою й навіть звичайного тексту. Якщо усі варіанти не були враховані розробником, то процес фаззингу може показати помилки, які ведуть до вразливостей, наприклад переповнення буферу. Фаззинг є однією із технік, що знаходить безліч вразливостей в відомих проектах.

Існують безліч фаззерів для різних векторів тестування в залежності від типу програмного забезпечення, що тестується. Це можуть бути або фаззери для програм із графічним інтерфейсом, програм з вводом через аргументи командного рядка, інтерактивним вводом, або навіть для веб орієнтованого ПЗ. Часто тестують не данні, що поступають на вхід до програми, а певні окремі компоненти як бібліотеки чи функції.

Недоліками фаззерів є доволі високий поріг входу й необхідність додаткового дослідження. До їх запуску як автоматичного програмного забезпечення інженер повинен провести велике дослідження, дізнатись особливості роботи певних модулів й знайти спосіб підстановки неправильних даних. Після цього виконується сам процес фаззингу. Але на цьому він не закінчується. Отримані результати необхідно дослідити й довести, що знайдені помилки мають цінність в контексті реверсивної інженерії й експлуатації вразливостей. Інколи досліджуване програмне забезпечення є дуже залежним від його оточення. До такого ПЗ можна віднести код, що виконується на мікроконтролерах. Фаззинг таких програм є більше складним, адже для цього потрібно симулювати поведінку цільової системи, що впливає на отримані результати.

До найбільш популярних фаззерів можна віднести такі проєкти, як AFL та WinAFL. Для тестування процесів в середовищі ядра можна використовувати difuze або IOCTLbf. Для фаззингу в браузерах можна використовувати BFuzz [6, 10]. Усі проєкти мають відритий код, що дає змогу долучитись до розробки, знайти відповіді на питання чи проблеми або ж модифікувати програми виходячи з власних потреб.

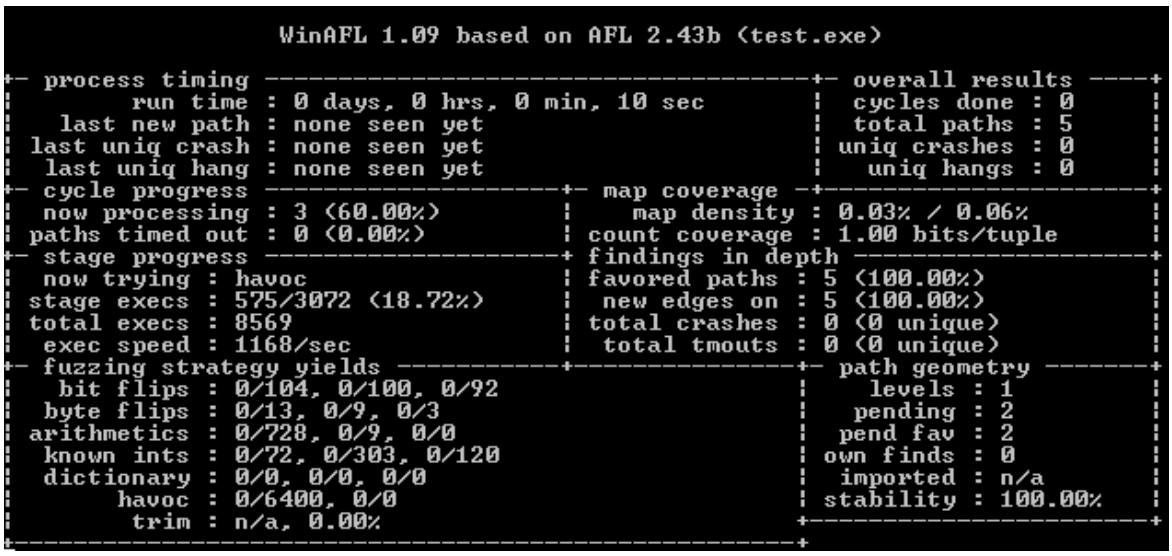

Рис. 2.5. Вивід програми фаззеру WinAFL

2.4. Дизасемблер

Дизасемблер – програма, що транслює машинний код виконуваних файлів в набір асемблерних інструкцій [11]. Це є основною й найважливішою програмою під в процесі реверсивної інженерії. Деякі дизасемблери надають можливості створення власних констант на коментарів, що значно полегшує роботу з файлом. Отже результати роботи дизасемблера є основним робочим середовищем під час реверсингу. Окрім транслювання машинного коду в мову асемблера програма також може відтворити відладочну інформацію за її наявності, наприклад з формату файлів ELF. Окрім цього дизасемблер вміє знаходити виклики API функцій стандартних бібліотек, які завантажуються динамічно.

Перше, на що потрібно звернути уваги під час вибору дизасемблеру для аналізу це те, які саме архітектури процесора він підтримує. Процес роботи дизасемблера на прикладі однієї інструкції є наступним. На вході він отримує бінарний файл, читає його й отримує певний порядок байт. Для спрощення будемо вважати, що нам вже відома архітектура й адреса, з якої починається код. Дизасемблер отримує наступний набір байт з файлу. Після цього він повинен транслювати ці байти в асемблерну інструкцію. Розробник архітектури

процесора, для якого було створено бінарний файл, надає документацію до своїх інструкцій. Розберемо її на прикладі інструкції xor архітектури PowerPC, яка виконує логічну операцію виняткової диз'юнкції.

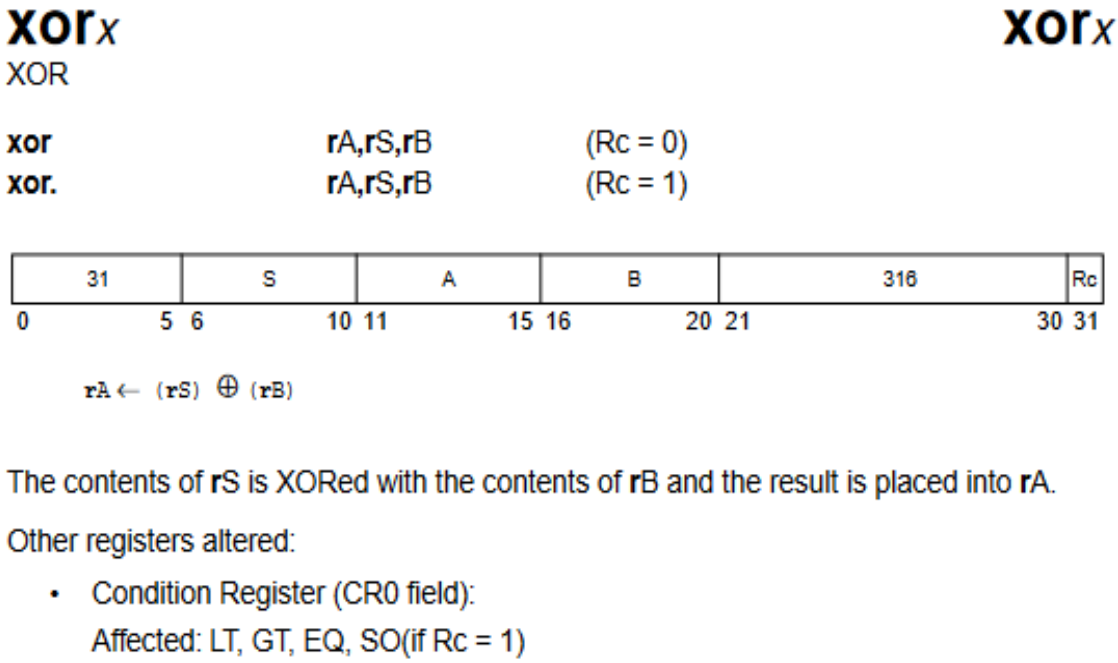

Рис. 2.6. Опис інструкції xor архітектури PowerPC

На схемі (рис. 2.6) можна побачити, що операція xor має розмір 32 біти [12]. Біти з індексами 0-5 та 21-31 є зарезервованими та мають статичне значення. Саме по цих бітах ми можемо зрозуміти те, що працюємо з інструкцією xor. Такі зарезервовані значення унікальні для кожної інструкції. Біти з індексами 6-10, 11-15, 16-20 є так званими аргументами інструкції. Розмір кожного з них - 4 біти й вони відповідають за номер регістру, який приймає участь в роботі. Останній біт з індексом 31 вказує на те, чи повинна інструкція змінювати регістр умовних операцій. Якщо він має значення 1, то в залежності від результату інструкції умовний регістр буде мати число, що відповідає певній умові.

До таких умов відносяться:

- Менше ніж
- Більше ніж
- Дорівнює
- Переповнення

Такі умовні інструкції використовують в парі з інструкціями, які в свою чергу перевіряють значення в умовному регістрі та виконують якусь дію в залежності від результату.

Отримуючи набір байт можемо розкласти його на елементи за кольорами:

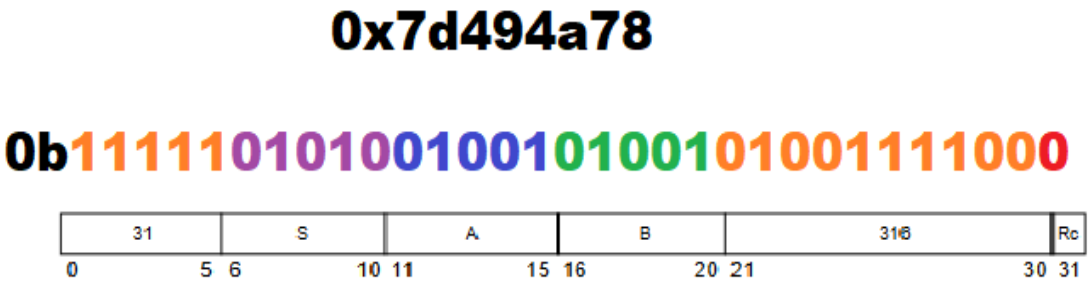

Рис. 2.7. Побітовий парсинг машинного коду в інструкцію асемблера

Червоний колір – біт умовних регістрів. Помаранчевий – зарезервовані значення. Програма дизасемблер знає весь набір можливих інструкцій й розуміє те, що це інструкція XOR. Наступний етап в такому випадку є парсинг аргументів. Біти над полем S вказуються на те, що буде використаний регістр r10, A – r9, B – r9. Таким чином цей набір байт можна транслювати в інструкцію - xor r9, r10, r9 (виконати операцію « r10 XOR r9 » , результат записати в регістр r9).

Отриманий результат можна перевірити написавши власну функції й скомпілювавши код під архітектуру PowerPC. Такі приклади зручно перевіряти

на онлайн компіляторі. Одним з них є CompilerExporer - онлайн компілятор, що являє собою веб додаток й підтримує велику кількість різних компіляторів різних версій [13].

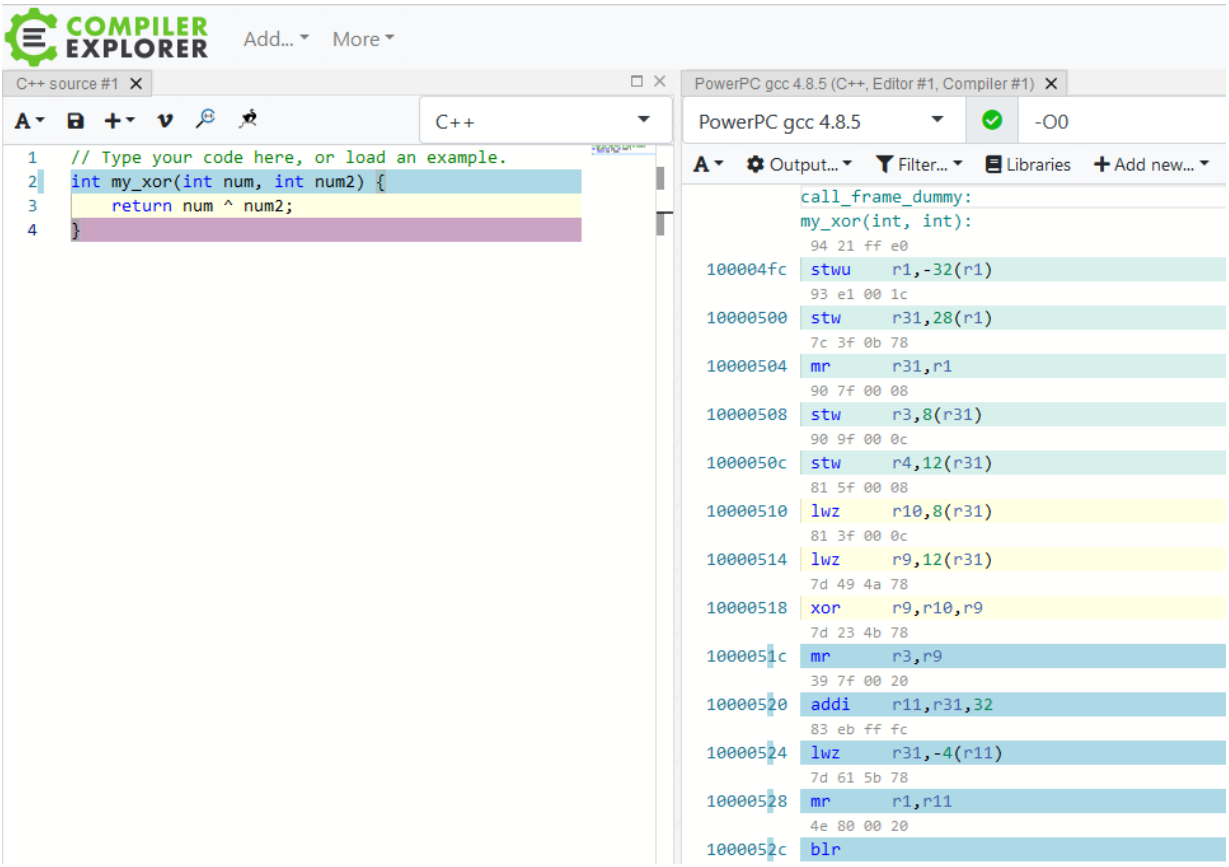

Рис. 2.8. Результат роботи Compile Explorer

Жовтим кольором позначена операція xor, за яку в вихідному файлі відповідають дві інструкції завантаження аргументів та операція xor. Над самою інструкцією за адресою 0x10000518 можна побачити байти, які будуть в двійковому файлі, вони збігаються з прикладом.

Окрім перекладу байтів в набір асемблерних інструкцій до задач дизасемблера відносяться визначення адреси, з якої починається код та визначення API функцій. Різні компілятори під час компіляції та збірки ПЗ можуть використовувати різні базові адреси, з яких починається код у файлі. Такі данні є вводом, що використовує реверсер, під час аналізу файлу. Використання різної адреси призводить до різних результатів дизасемблювання й кількості

знайдених інструкцій. Зміна базової адреси чи її приховування й відновлення під час запуску програми є однією з технік обфускації, що робить аналіз файлу складнішим. Чим точніше компілятор може аналізувати базову адресу автоматично, тип краще він вважається. Це дозволяє впроваджувати процеси автоматизації в аналіз файлів в порівнянні з ручним або напівручним аналізом, що потребує витрат часу в від інженера.

2.5. Декомпілятор

Декомпілятор – програма, задачею якої є трансляція бінарного файлу в вихідний код, що є оберненим процесом до компіляції. Відновлений за допомогою декомпілятора код може відрізнятись від оригінального, що залежить від налаштувань компілятора. До таких налаштувань можна віднести оптимізацію, за якої певні частини коду можуть бути реорганізовані або взагалі видалені. Не зважаючи на це, результат роботи програми-декомпілятора є успішним в тому випадку, коли вихідний код компілюється без помилок й виконує ту саму логіку, що й оригінальний файл.

Як і у випадку з іншими засобами для реверсивної інженерії, обфускація створює перешкоди для процесу декомпіляції, адже це є транслюванням результату роботи дизасемблера в людино-орієнтований вихідний текст програми. Найбільш вдало процесу декомпіляції піддаються файли, в яких збережена метадата та інформації для відладки програмного забезпечення. Такі символи дозволяють повністю відновити оригінальні імена змінних, функцій, структур даних та навіть номери рядків.

```c
#include <stdio.h>

int main()
{
    char* my_str = "Hello, World!\n";
    printf("%s", my_str);
    return 0;
}
```

Рис. 2.9. Вихідний код програми hello_world.c

На наступному зображенні (рис 2.10) можна побачити результат декомпіляції програми hello_world.c (рис. 2.9). На ньому видно яким чином компілятор оптимізував змінну my_str і замінив її на константний текст, адже вона не змінюється в процесі роботи програми.

Decompile: main - (hello_gcc.out)

```
undefined8 main(void)

{
  printf("%s","Hello, World!\n");
  return 0;
}
```

Рис. 2.10. Декомпіляція програми hello_gcc.out

Перші декомпілятори працювали таким чином, що намагалися поєднати певні паттерни асемблерних інструкцій з відомими конструкціями високорівневої мови програмування. Таким чином вони просто запам’ятовували як виглядають умовні конструкції, цикли тощо. Це дозволяло генерувати асемблерний код в стилі мови програмування С. Але це й означало те, що інженери повинні були додавати інформацію про різні відомі паттерни виклику функцій, циклів, розгалужень та інших структур для кожної архітектури процесора. Більш того, різні компілятори могли інтерпретувати такі операції на свій розсуд, що також додавало більше роботи й часу для підтримки декомпіляції всіх можливих варіантів.

Наступним етапом в розвитку декомпіляторів було впровадження алгоритму з використанням IR (Intermediate Representation). IR – абстрактна мова, що використовується компіляторами, віртуальними машина, програмами для аналізу коду для подальшої обробки й відображає вихідний код незалежно від архітектури процесора [14].

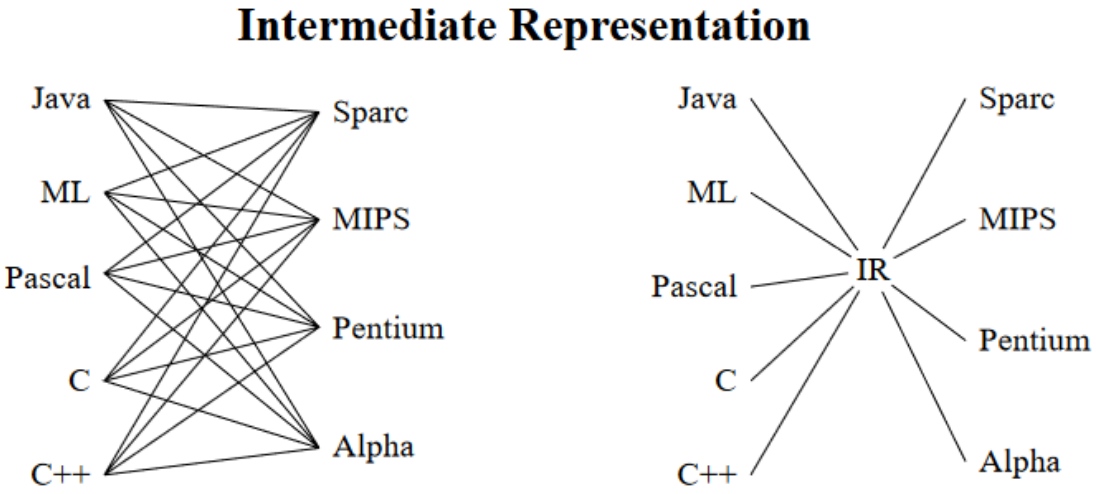

Рис. 2.11. Intermediate Representation scheme

Таким чином маючи програму, що вміє працювати з IR, підтримка усіх можливих архітектур процесора стає значно простішою. Від інженерів потрібно реалізувати лише транслятор із специфічного платформо-залежного асемблеру в

платформо-незалежне внутрішнє представлення IR. Однією з важливих вимог до такого внутрішнього представлення є консистентність зворотнього перетворення. Набір інструкцій, що представлений у формі IR, повинен мати можливість бути перетвореним в будь-яку платформо-специфічну мову в зворотньому порядку й не втратити можливостей оригінального коду.

Кожна інструкція IR представляє собою одну фундаментальну найпростішу операцію, що в свою чергу можуть бути об’єднані в блоки:

```
------ IMark(0x24F275, 7, 0) ------
t3 = GET:I32(0) # get %eax, a 32-bit integer
t2 = GET:I32(12) # get %ebx, a 32-bit integer
t1 = Add32(t3,t2) # addl
PUT(0) = t1 # put %eax
```

Таким чином працюють сучасні декомпілятори Ghidra та IDA, які підтримують велику кількість архітектур. Модуль дизасемблера працює виключно з IR кодом й постійно вдосконалює внутрішні алгоритми. А за підтримку нової архітектури відповідальний той модуль, що транслює інструкції цієї архітектури в IR.

2.6. Високорівневі комплекси ПЗ для аналізу

Описані в минулих розділах програми для аналізу й реверсингу виконуваних файлів є досить потужними інструментами, що використовувались довгий час в комплексі. Володіння лише одним окремим інструментом є недостатнім для повноцінного аналізу файлу або системи файлів. Реверсивний інженер завжди поєднує їх для досягнення результату й використовує кожен з них на певному етапі аналізу. Також вихідні результати однієї програми часто можуть бути вхідними даними для іншого програмного забезпечення. Таким чином було створено ПЗ, що поєднує в собі безліч елементів. Такі продукти

можна порівняти з програмною - інтегрованим середовищем розробки під час написання коду, яке можна замінити на окремі компоненти:

- Текстовий редактор
- Компілятор
    - Лінкувальник
    - Препроцесор
- Інтерактивний відладчик
- Статичний аналізатор коду
- Систему контроля зовнішніх залежностей

Найпопулярнішими інструментами для реверсингу є radare, IDA Pro, angr, Ghidra. Кожен з них має набір переваг й недоліків, різні інтерфейси взаємодії, список підтримуємих архітектур та інструментів. Їх можна також порівняти за ступенем відкритості коду, доступністю та величиною користувачів. Кожен з цих інструментів потребує великої кількості часу для повного опанування. Маючи представлення про можливості кожного з них інженер зможе зробити правильних вибір під час роботи з певною системою.

2.6.1 radare2

Sergi Alvarez будучи реверсивним інженером в сфері криміналістики не міг використовувати програмне забезпечення компанії для власних потреб. Через це він вирішив створити власний шістнадцятирічний редактор з наступним функціоналом:

- Легка переносимість
    - Робота в unix системах
    - Інтерфейс через командний рядок
    - Мова програмування C
    - Малий розмір виконуваного файлу
- Робота з файлами-пристроями диску

- Пошук рядків або hex-пар
- Перевірка та збереження результатів на диск

Ця програма була створена для відновлення видалених файлів з диску. Після цього він доповнив її функціями для відладки, додав підтримку декількох архітектур та можливостями аналізу коду. З часом програма перетворилась в фреймворк під назвою radare [17]. Збільшення функціоналу не вплинуло на базові концепти, які були сформульовані на початку:

- Все є файл
- Набір незалежних програм, що викликаються з терміналу та працюють з потоками вводу та виводу

Пізніше проєкту був потрібен значний архітектурний рефакторинг, результатом якого було створення продукту radare2. Він являє собою набір невеликих утиліт, які можна використовувати незалежно або разом. Основним виконуваним файлом є radare2 – головна утиліта всього фреймворку, яка використовує інші програми за потреби. Вона використовує ядро шістнадцятирічного редактора й відладчик, дозволяє відкривати вхідні дані так, наче вони є простими файлами. До вхідних даних можна віднести звичайні файли, файли дисків, драйвери, процеси в режимі відладки, мережеві підключення. Програма дозволяє переміщуватися файлом, аналізувати дані, дизасемблювати виконуванні файли, змінювати частини коду, шукати, порівнювати й візуалізувати дані.

Rabin2 – програма, що відкриває виконуванні файли. Вона вміє працювати з такими форматами як ELF, PE, Java CLASS, Mach-O. Ця утиліта відкриває файл й аналізує його тип, набір символів для експорту та залежностей.

Rasm2 – програма дизасемблер. Підтримує великий набір архітектур процесору серед яких x86-64, MIPS, Arm, PowerPC, Intel X86 тощо.

Rahash2 – утиліта для визначення хешу від файлу, що підтримує велику кількість алгоритмів.

Radiff2 – утиліта для порівняння бінарних файлів та їх частин.

Rafind2 – програма для пошуку патернів в файлах.

Rarun2 – утиліта для запуску програм з різним середовищем. Вона дозволяє налаштовувати аргументи, права доступу, директорії та змінювати дескриптори файлів. Ця утиліта використовується для фаззинга та тестування.

Проєкт є повністю відкритим й доступним, що дає можливість розширювати функціонал плагінами. Для підтримки автоматизації та скриптінгу під час реверсингу файлів redare2 має вбудований пайп r2pipe для швидкої передачі результатів однієї команди на вхід в іншу. Також він доступний не лише як unix-подібний виконуваний файл, а й як бібліотека. До списку підтримуємих мов програмування можна віднести:

- Python
- Javascript
- Go
- Ruby
- Perl
- Rust

Бібліотеки є рівнем абстракції над викликом бінарних файлів й не відходять від концепції фреймворку. Вони оперують лише функціями, які приймають на вхід команду виклику бінарного файлу й повертають результат. Це знижує поріг входу й опанувавши ручний аналіз radare2 автоматизація не займе багато часу. Приклад програми аналізу в мові python:

```
import r2pipe

r2 = r2pipe.open('/bin/true')
r2.cmd('aaaa')
r2.cmd('pdf @@f > out')
r2.quit()
```

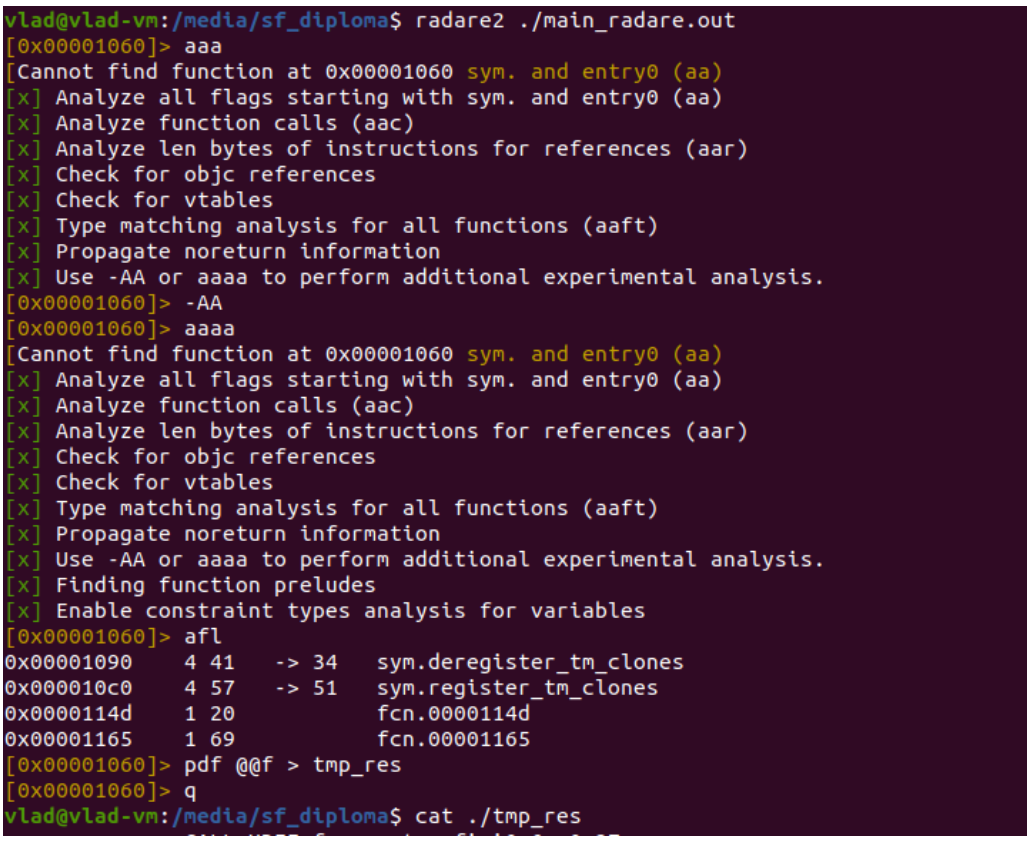

Рис. 2.12. Використання radare2 для аналізу списку функцій

На рис. 2.12 можна пробачати інтерфейс програми radare2. Для прикладу був створений файл з функцією my_sum, яка додає два числа. При базовому аналізі ця функція була знайдена за адресою 0x0000114d, але її ім’я ще не було визначено. Під час інтерактивної сесії можна власноруч змінювати імена функцій і змінних для полегшення відтворення логіки програми. Наступною командою був записаний результат дизасемблеру знайдених функцій у файл. Так як файл містить відладочну інформацію і зберіг імена змінних й функцій, то користувачу доступне її ім’я – my_sum.

За адресою 0x0000116d програма записує числа 5 та 7 в пам’ять стеку з відповідним зміщенням. Потім ці дані переміщуються зі стеку в регістри, які використовуються для передачі аргументів в функцію. Можна побачити набір зайвих операцій копіювання, спочатку копіювання в пам’ять стеку, потім в регістри edx, eax, потім в регістри esi, edi. Якщо скомпілювати файл з вищим рівнем оптимізації, то ці інструкції зникають, й функція my_sum навіть не викликається, замість цього на екран виводиться пораховане під час компіляції

число тому, що змінні статичні й не змінюються під час роботи. Якщо ж функція бути приймати числа від користувача програма, то такої оптимізації не буде.

```
20: fcn.0000114d ();
          ; var int64_t var_8h @ rbp-0x8
          ; var int64_t var_4h @ rbp-0x4
          0x0000114d      55              push rbp
          0x0000114e      4889e5          mov rbp, rsp
          0x00001151      897dfc          mov dword [var_4h], edi
          0x00001154      8975f8          mov dword [var_8h], esi
          0x00001157      8b55fc          mov edx, dword [var_4h]
          0x0000115a      8b45f8          mov eax, dword [var_8h]
          0x0000115d      01d0            add eax, edx
          0x0000115f      5d              pop rbp
          0x00001160      c3              ret
69: fcn.00001165 ();
          ; var int64_t var_ch @ rbp-0xc
          ; var int64_t var_8h @ rbp-0x8
          ; var int64_t var_4h @ rbp-0x4
          0x00001165      55              push rbp
          0x00001166      4889e5          mov rbp, rsp
          0x00001169      4883ec10        sub rsp, 0x10
          0x0000116d      c745f4050000.   mov dword [var_ch], 5
          0x00001174      c745f8070000.   mov dword [var_8h], 7
          0x0000117b      8b55f8          mov edx, dword [var_8h]
          0x0000117e      8b45f4          mov eax, dword [var_ch]
          0x00001181      89d6            mov esi, edx
          0x00001183      89c7            mov edi, eax
          0x00001185      e8bfffffff      call sym.my_sum
          0x0000118a      8945fc          mov dword [var_4h], eax
          0x0000118d      8b45fc          mov eax, dword [var_4h]
          0x00001190      89c6            mov esi, eax
          0x00001192      488d3d6b0e00.   lea rdi, qword [0x00002004] ; "%d"
          0x00001199      b800000000      mov eax, 0
          0x0000119e      e8adfeffff      call sym.imp.printf
          0x000011a3      b800000000      mov eax, 0
          0x000011a8      c9              leave
          0x000011a9      c3              ret
```

Рис. 2.13. Дизасемблинг файлу main_radare.out

```
[0x00001060]> pd @ 0x1155
          0x00001155      4883ec08        sub rsp, 8
          0x00001159      ba0c000000      mov edx, 0xc
          0x0000115e      488d359f0e00.   lea rsi, qword [0x00002004] ; "%d"
          0x00001165      bf01000000      mov edi, 1
          0x0000116a      b800000000      mov eax, 0
          0x0000116f      e8dcfeffff      call sym.imp.__printf_chk
          0x00001174      b800000000      mov eax, 0
          0x00001179      4883c408        add rsp, 8
          0x0000117d      c3              ret
          0x0000117e      6690            nop
```

Рис. 2.14. Функція main файлу main_radare_optimizied_O1.out

2.6.2 Angr

angr — це багато-архітектурний інструмент аналізу виконуваних файлів з можливостями динамічного посимвольного виконання та статичного аналізу [19]. Фреймворк має відкритий код і написаний мовою python, архітектура гнучка до створення розширень і написання власних плагінів. Весь проєкт складається з деяких підпроєктів:

- Завантажувач виконуваних файлів і бібліотек - CLE
- Бібліотека з інформацією про підтримуємі архітектури – archinfo
- Бібліотека мови python, що надає інтерфейс до VEX IR (PyVEX)
- Код аналізу файлів – angr

Сам фреймворк має наступний функціонал в контексті аналізу бінарних файлів та систем:

- Відновлення графу виконання програми
- Посимвольне виконання коду програми
- Патчінг виконуваних файлів
- Статичний аналіз файлів

Angr працює з VEX Intermediate Representation й підтримує наступні архітектури:

- aarch64
- amd64
- avr
- mips
- ppc
- s390x
- soot
- x86

Втім за потреби можна додавати підтримку інших архітектур додавши код, який транслюватиме інструкції специфічної архітектури в IR. В цьому випадку angr зможе аналізувати файли даної архітектури так само, як і файли архітектур з основного списку.

Пайплайн роботи фреймворку починається із створення об'єкту проєкта та завантаження файлу за допомогою ArchInfo та завантажувача CLE. Їх основною задаче є отримати інформацію про файл, під яку архітектуру процесору він був побудований, який в ньому порядок байт (big-endian або little-endian), яка в нього базова адреса. Перехід від бінарного файлу до його представлення в віртуальному адресному просторі є досить складною задачею. Саме цим займається модуль CLE. Також він показує список зовнішніх бібліотек, від яких залежить файл, базову інформацію про завантажений адресний простір.

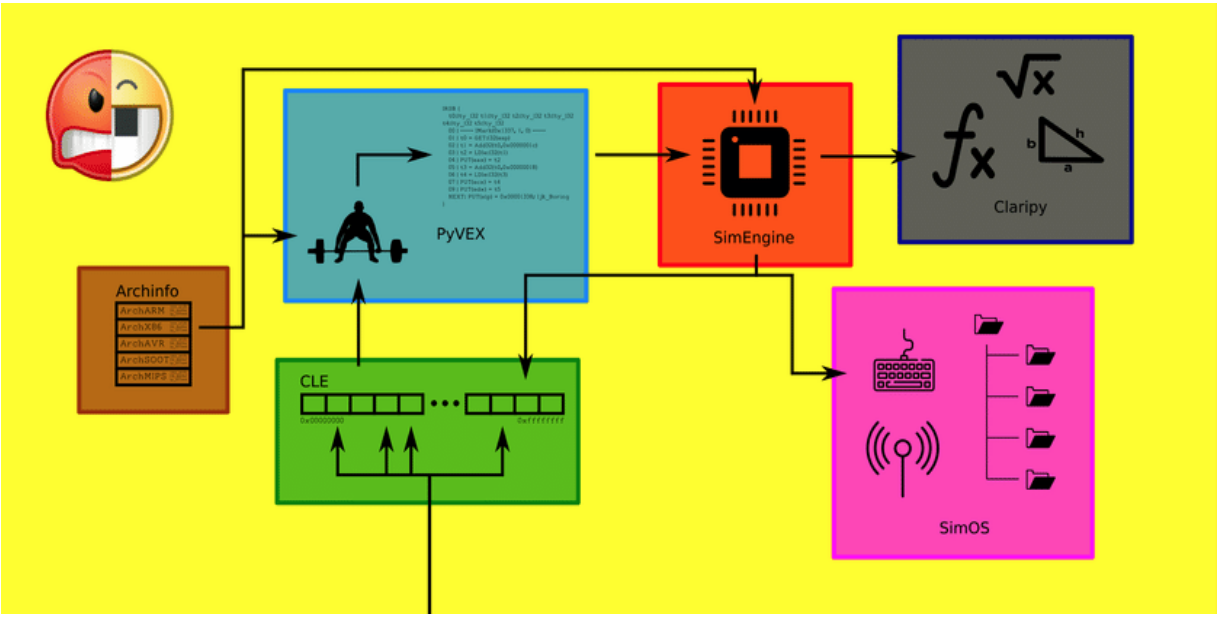

Рис. 2.15. Схема роботи фреймворку angr

На рис 2.16 можна побачити, як angr завантажив і розпакував викуваний файл, отримав інформацію про віртуальні адреси, залежності. За допомогою текстових символів ми можем побачити наявність функції my_sum. Також була визначена архітектура файлу.

```
>>> proj.loader.main_object
<ELF Object main_angr.out, maps [0x400000:0x404017]>
>>> proj.loader.shared_objects
OrderedDict([('main_angr.out', <ELF Object main_angr.out, maps [0x400000:0x404017]>), ('libc.so.6', <ELF Object
libc-2.31.so, maps [0x500000:0x6f14d7]>), ('ld-linux-x86-64.so.2', <ELF Object ld-2.31.so, maps [0x700000:0x72f1
8f]>), ('extern-address space', <ExternObject Object cle##externs, maps [0x800000:0x87ffff]>), ('cle##tls', <ELF
TLSObjectV2 Object cle##tls, maps [0x900000:0x91500f]>)])
>>> hex(proj.loader.main_object.entry)
'0x401060'
>>> hex(proj.loader.main_object.min_addr)
'0x400000'
>>> hex(proj.loader.main_object.max_addr)
'0x404017'
>>> proj.loader.main_object.segments
<Regions: [<ELFSegment flags=0x4, relro=0x0, vaddr=0x400000, memsize=0x600, filesize=0x600, offset=0x0>, <ELFSeg
ment flags=0x5, relro=0x0, vaddr=0x401000, memsize=0x235, filesize=0x235, offset=0x1000>, <ELFSegment flags=0x4,
 relro=0x0, vaddr=0x402000, memsize=0x180, filesize=0x180, offset=0x2000>, <ELFSegment flags=0x4, relro=0x1, vad
dr=0x403db8, memsize=0x248, filesize=0x248, offset=0x2db8>, <ELFSegment flags=0x6, relro=0x0, vaddr=0x404000, me
msize=0x18, filesize=0x10, offset=0x3000>]>
>>> proj.loader.find_symbol('my_sum')
<Symbol "my_sum" in main_angr.out at 0x401149>
>>> proj.arch
<Arch AMD64 (LE)>
>>> proj.arch.max_inst_bytes
15
>>> proj.arch.sp_offset
48
>>>
```

Рис. 2.16. Приклад завантаження файлу main_radare.c в angr

Наступним етапом аналізу є трансляція специфічних до архітектури інструкцій в IR. На рис. 2.17 можна побачити результат роботи дизасемблера з функцією my_sum. Результат збігається з іншими дизасемблерами. Рис 2.17 показує частину цієї функції в вигляді intermediate representation. Можна орієнтуватись порівнюючи інструкції і блоки за однією віртуальною адресою. Взявши найпростішу інструкцію 0x40115d видно те, як angr враховує величину змінних і виконує перетворення з 64-бітного числа до 32-бітного Результат навпаки переводиться в 64-бітне число.

```
>>> proj.factory.block(0x401149).pp()
        my_sum:
401149  endbr64
40114d  push    rbp
40114e  mov     rbp, rsp
401151  mov     dword ptr [rbp-0x4], edi
401154  mov     dword ptr [rbp-0x8], esi
401157  mov     edx, dword ptr [rbp-0x4]
40115a  mov     eax, dword ptr [rbp-0x8]
40115d  add     eax, edx
40115f  pop     rbp
401160  ret
>>>
```

Рис. 2.17. Результат дизасемблеру функції my_sum в angr

Після цього angr має певний набір аналізаторів, які можна використовувати в залежності від цілей. Це може бути аналіз графу викликів функцій, або ж посимвольне виконання програми. В ньому angr оперує станами та блоками інструкцій VEX. Код розбивається на логічні частини й виконання йде блок за блоком. Під час такого виконання можна переглядати та змінювати пам’ять процесу та вміст регістрів. Даний аналіз можна використовувати для перевірки файлу на вразливості або ж аналізу поведінки програми в цілому адже дуже легко імітувати зовнішні умови та змінювати їх в процесі виконання.

```
11 | ------ IMark(0x401151, 3, 0) ------
12 | t17 = Add64(t14,0xfffffffffffffffc)
13 | t20 = GET:I64(rdi)
14 | t19 = 64to32(t20)
15 | STle(t17) = t19
16 | PUT(rip) = 0x0000000000401154
17 | ------ IMark(0x401154, 3, 0) ------
18 | t21 = Add64(t14,0xfffffffffffffff8)
19 | t24 = GET:I64(rsi)
20 | t23 = 64to32(t24)
21 | STle(t21) = t23
22 | PUT(rip) = 0x0000000000401157
23 | ------ IMark(0x401157, 3, 0) ------
24 | t25 = Add64(t14,0xfffffffffffffffc)
25 | t28 = LDle:I32(t25)
26 | t27 = 32Uto64(t28)
27 | PUT(rdx) = t27
28 | PUT(rip) = 0x000000000040115a
29 | ------ IMark(0x40115a, 3, 0) ------
30 | t29 = Add64(t14,0xfffffffffffffff8)
31 | t32 = LDle:I32(t29)
32 | t31 = 32Uto64(t32)
33 | ------ IMark(0x40115d, 2, 0) ------
34 | t33 = 64to32(t31)
35 | t35 = 64to32(t27)
36 | t6 = Add32(t33,t35)
37 | PUT(cc_op) = 0x0000000000000003
38 | t37 = 32Uto64(t33)
```

Рис. 2.18. Частина функції my_sum в IR

2.6.3 Ghidra

Ghidra – SRE фреймворк, який був створений Агентством Національної Безпеки США. Код написаний мовою програмування java й являється повністю відкритим [20]. Так як була використана java, то й підтримуються усі системи, що вміють працювати з Java Virtual Machine. Доступність коду дозволяє створювати власні плагіни для роботи й брати участь в розробці продукту.

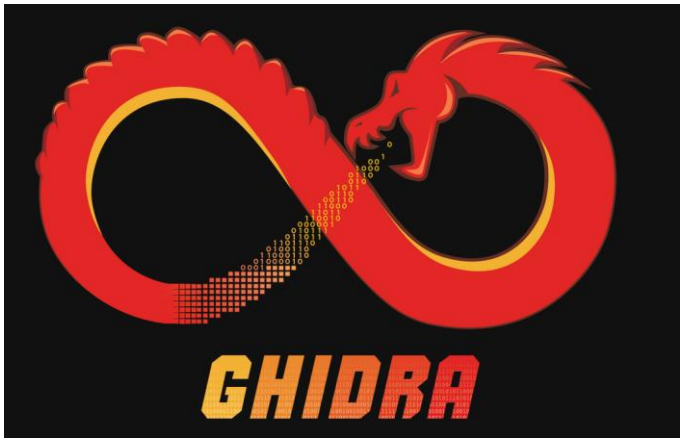

Рис. 2.19. Логотип ghidra

Важко перелічити увесь функціонал цього програмного забезпечення навіть не рахуючи сторонні плагіни, втім до головних можливостей фреймворку можна віднести:

- Дизасемблер
- Декомпілятор
- Побудова графів виклику
- Скриптінг
- Пошук та патчінг файлів

Програма має графічний інтерфейс користувача й працює як IDE для ручного аналізу файлів. Також є можливість для автоматизації використовуючи ghidra API. Це робить цю SRE універсальним програмним продуктом для автоматизації щойно проведеної вручну роботи.

Під час запуску програма пропонує створити прєкт та завантажити файл для аналізу. В наступному вікні можна вибрати набір аналізаторів, що будуть обробляти виконуваний файл. Їх список постійно оновлюється з часом. З найголовніших можна виділити декомпіляцію, аналіз зовнішніх бібліотек, повний аналіз функції і графу викликів, аналіз текстових рядків, аналіз стеку та констант. Після аналізу інженеру доступна інформації про імпорти та експорти файлу, список функцій та структур даних, результати дизасемблєнгу та декомпіляції.

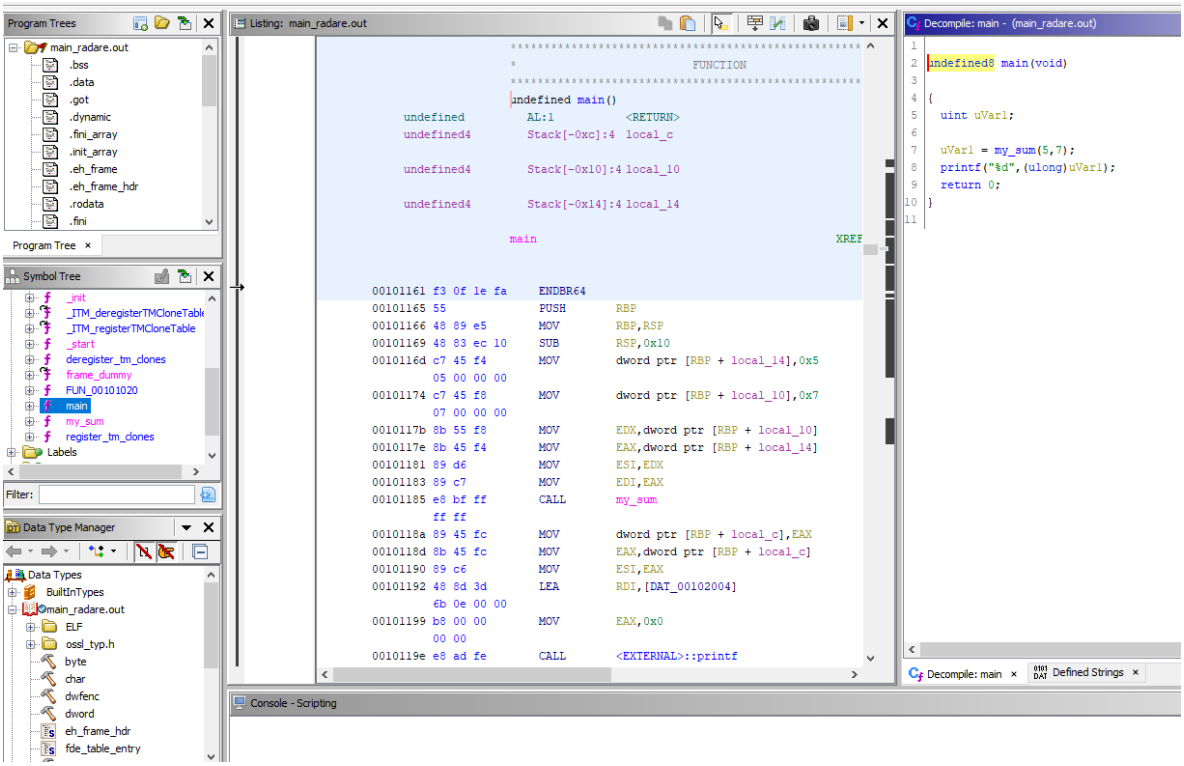

Рис. 2.20. Інтерфейс Ghidra

Робочий простір є інтерактивним й дозволяє на ходу змінювати базову адресу функцій, імена змінних та функцій, додавати коментарі, створювати власні функції за певною адресою. Такий функціонал є корисним під час аналізу файлів, в яких відсутня відладочна інформація й які були обфусковані. Програма також дозволяє змінювати ассемблерні інструкції.

2.6.4 IDA

IDA Pro є одним із найпопулярніших SRE фреймворків. Програма є інтерактивним дизассемблером, що означає активну участь інженера в процесі аналізу. IDA не є автоматичним аналізатором програм. Дії користувача зберігаються на диск і за потреби сесію аналізу можна відкрити знову чи передати комусь зі збереженим контекстом.

IDA підтримує багато форматів виконуваних файлів під різні архітектури процесорів та має в собі вбудовану командну мову під назвою IDC для прискорення рутинних дій. Також в IDA існує великий набір плагінів для, які розширюють функціонал. Основною перевагою IDA є те, що вона дозволяє інтерактивно змінювати будь-яку інформацію, що присутня в графічному інтерфейсі. Після завантаження файлу для аналізу можна давати імена функціям, змінним, структурам даних. Програма надає можливість будувати діаграми та графи викликів для спрощення розуміння досліджуємої системи. В IDA також присутній функціонал по автоматичному визначенню використаних стандартних бібліотек. Для роботи з вмістом файлів є аналізатори строкових даних, шістнадцятирічний редактор та можливість патчінгу даних та окремих інструкцій.

Під час першого завантаження файлу в проект IDA визначає тип файлу та пропонує вручну вибрати архітектуру процесора та базову адресу. Втім є можливість вибору автоматичного аналізу. Після цього відкривається головне вікно програми, де можна перевірити список знайдених функцій, переглянути результат роботи дизасемблера, список імпортів на експортів. Як було сказано вище, IDA дозволяє змінити все, що є на екрані. Це особливо важливо під час аналізу файлів, які були обфусковані, або файлі, в яких відсутні символи для відладки. На рис 2.21 можна побачити інтерфейс зміни функції. Є можливість вручну визначити адреси, чи є функція статичною, чи має вона повертати значення, як використовується регістр BP, розмір функції тощо.

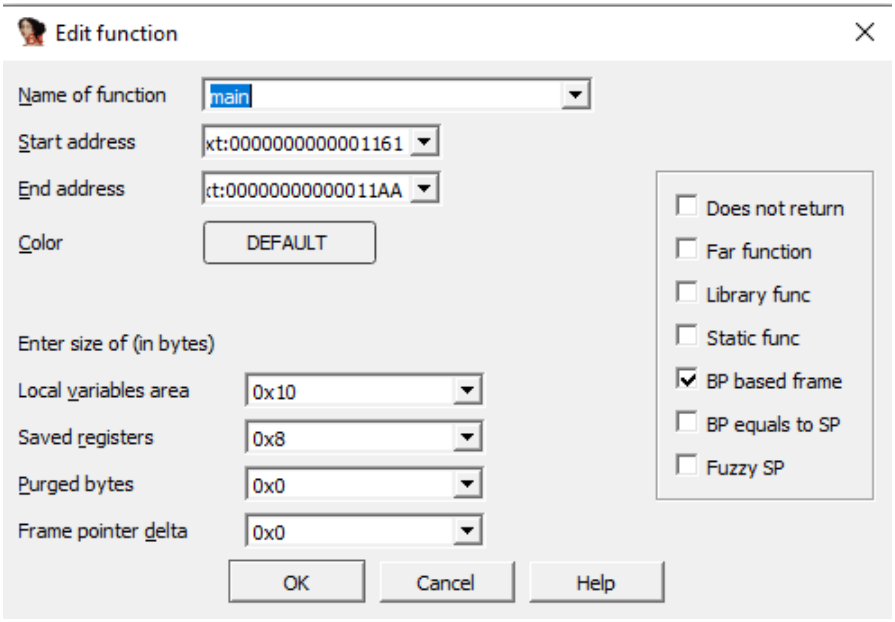

Рис. 2.21. Вікно зміни функції main

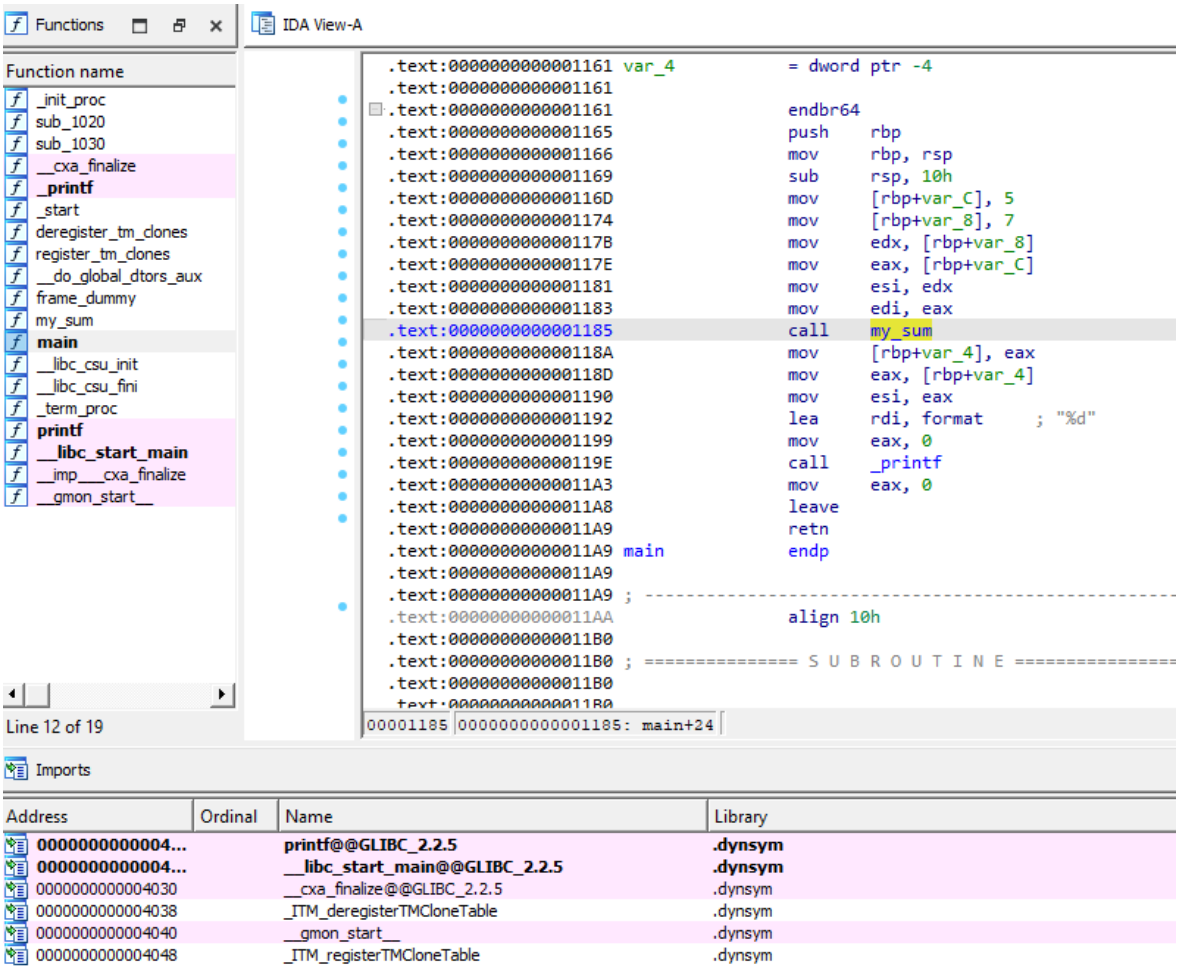

Рис. 2.22. Інтерфейс IDA Pro

Окрім основних можливостей IDA має великий набір плагінів [5]. Декомпілятор Hex-Rays є одним з таких додатків, що дозволяє транслювати набір процесорних інструкцій в людино-орієнтований код мови С. Декомпілятор працює з кодом, який був генерований багатьма C++ компіляторами незалежно від архітектури.

Lighthouse – плагін, що підсвічує інструкції, які виконуються файлом в тому випадку, якщо вони приймають участь в роботі певного алгоритму чи функції. Це допомагає приділяти увагу лише тій частині файлу, яка справді потрібна.

ClassInformer – плагін для аналізу класів та віртуальних функцій з RTTI секції файлів, що були побудовані компілятором Microsoft Visual Studio.

BinDiff – плагін що порівнює двійкові файли на рівні результатів дизасемблеру замість порівнювання сирих байтів.

ida-x86emu – плагін для емуляції виконання асемблерних інструкцій в ізольованому середовищі. Плагін, на відміну від відладчика, може емулювати будь-яку частину коду без ризику пошкодити середовище.

IDA Pro, окрім наведеного вище функціоналу, має вбудований відладчик для архітектур x86-64. Єдиним недоліком програми, що відрізняє її від описаних вище інструментів є те, що вона повністю пропрієтарна. Вихідний код закритий і вартість ліцензії може бути суттєвою перешкодою. Серед планів покупки є безкоштовна версія IDA Freeware з лімітованим функціоналом і обмеженим набором архітектур виключно для некомерційного використання.

2.7. Висновки до другого розділу

Для реверсингу програмного забезпечення існують безліч класів програм , компонентів і методик. Знаючи цілі й особливості кожного з них можна ефективно досліджувати комплексні системи. Для динамічного аналізу поведінки програм використовуються спеціалізовані дебаги, що мають вбудований обхід захисту від відладки, показ регістрів та стану пам’яті,

шістнадцятирічний редактор з функція пошуку паттернів та порівняння файлів. Також для тестування програмного забезпечення на вразливості використовують програми-фаззери, що шукають послідовність вхідних даних, яка призведе до помилки, яку можна використати для створення вразливості. Втім така робота потребує попереднього дослідження системи. Для таких досліджень використовують дизасемблери та декомпілятори, що перетворюють байти й набір асемблерних інструкцій на людино-орієнтований вихідний код. Такий набір програмного забезпечення може мати інтерфейс командного рядку, що допомагає проводити аналіз безпосередньо на віддаленому пристрої задля точного збігу середовища виконання. Для реверсингу на власній системі краще використовувати високорівневі інтерактивні комплекси, що підходять для статичного аналізу. Дослідження файлу за допомогою динамічного аналізу можливе через використання внутрішнього представлення та спеціальних інструментів, що можуть з ним працювати й емулювати виконання всієї програми, або ж її частини.

РОЗДІЛ 3
ОСОБЛИВОСТІ АРХІТЕКТУР ПРОЦЕСОРА ТА ВИЗНАЧЕННЯ КРАЩІХ ЗАСОБІВ ДЛЯ ЇХ АНАЛІЗУ

3.1. ARM

ARM (Advanced RISC Machines) – сімейство архітектур RISC процесорів. RISC (Reduced Instruction Set Computer) – комп'ютер, що спроектований згідно з принципом зменшення складності інструкцій незважаючи на зростання їх кількості для виконання певної задачі.

Процесори на архітектурі ARM використовувались в більшості випадків на SoC та інших мобільних типах девайсів через свою енергоефективність та низьку вартість виробництва. Втім останнім часом розробники все частіше використовують ARM архітектуру для більш потужних систем.

Набір інструкцій ARM значно змінився якщо порівнювати його з першими версіями архітектури. Лише з появою ARMv3 архітектура отримала 32-бітну підтримку. Починаючи з версії ARMv4 був представлений альтернативний набір інструкцій під назвою Thumb. Дані інструкції виконувались в окремому стані процесора з можливістю зміни режимів. Головною відмінністю Thumb був зменшений розмір інструкцій в два рази (16 біт порівняно з 32 біт). Це дозволяло підвищити щільність коду, втім деякі функції потребували більшу кількість Thumb інструкцій. До того ж режим ARM краще підходить для підвищення продуктивності під час роботи з критичним до часу кодом, який в більшості випадків виконується саме на мікроконтролерах.

ARM є load-store архітектурою, в якій робота з пам’яттю виконується через вміст регістрів замість роботи з вмістом пам’яті напряму. Поля в інструкціях мають фіксований розмір, що полегшує їх декодування. Майже всі інструкції можна виконувати як умовні. Усі ці особливості створюють хороший баланс між швидкістю виконання, розміром файлів, малими розмірами процесора та низькими енерговитратами під час роботи.

ARM має 31 регістр загального використання. Втім в кожний момент часу доступні лише шістнадцять з них. Інша половина використовується для пришвидшення обробки виключень. Доступний набір регістрів використовується кодом без привілеїв в просторі користувача. Зміна цього режиму доступна через генерування виключення. Також можуть існувати певні обмеження доступу до пам'яті чи функціям процесора.

Усі стани процесора зберігаються в окремому спеціальному регістрі під назвою CPSR – Current Program Status Register [21]. Він має спеціальну маску і окремі біти цього регістру відповідають за певну інформацію. Це біти для умовних інструкцій, інформації про те, чи відбулося переповнення числа після виконання минулої інструкції, біт для роботи з виключення, біти з інформацією про режим роботи процесора, біти про поточний порядок байт (Big Endian або Little Endian) тощо.

Bits	Name	Function
[31]	N	Negative condition code flag
[30]	Z	Zero condition code flag
[29]	C	Carry condition code flag
[28]	V	Overflow condition code flag
[27]	Q	Cumulative saturation bit
[26:25]	IT[1:0]	If-Then execution state bits for the Thumb IT (If-Then) instruction
[24]	J	Jazelle bit
[19:16]	GE	Greater than or Equal flags
[15:10]	IT[7:2]	If-Then execution state bits for the Thumb IT (If-Then) instruction
[9]	E	Endianness execution state bit: 0 - Little-endian, 1 - Big-endian
[8]	A	Asynchronous abort mask bit
[7]	I	IRQ mask bit
[6]	F	FIRQ mask bit
[5]	T	Thumb execution state bit
[4:0]	M	**Mode field**

Рис. 3.1. Біти регістру CPSR

Три регістри з 16-ти доступних виконують специфічні ролі. Stack pointer (SP) – регістр, що вказує на початкову адресу стеку процесу. В більшості випадків для цього використовують регістр R13.

Link register (LR\LP) – регістр, що використовується як адреса наступної інструкції під час перевірки умови. Також може бути використаний для збереження адреси повернення після обробки виключення. В усіх інших випадках цей регістр (R14) можна використовувати для власних потреб.

Program counter (PC) – регістр R15, в більшості випадків вказує на інструкцію, що знаходить через дві після тієї, що виконується в даний момент часу.

Наступні 13 регістрів не мають спеціальних платформо залежних функцій і використовуються виключно програмним забезпеченням. На рис. 3.2 можна побачити результат компіляції функції, яка повертає одне з двох вхідний чисел. Зеленим кольором виділені інструкції ініціалізації для роботи з функцією, в них використовується спеціальний регістр SP. В наступних блоках можна побачити роботу з пам’яттю через інструкції load, store та інструкцію умовного переходу blt, що також неявно використовує спеціальний регістр, який встановила інструкція cmp після порівняння двох чисел.

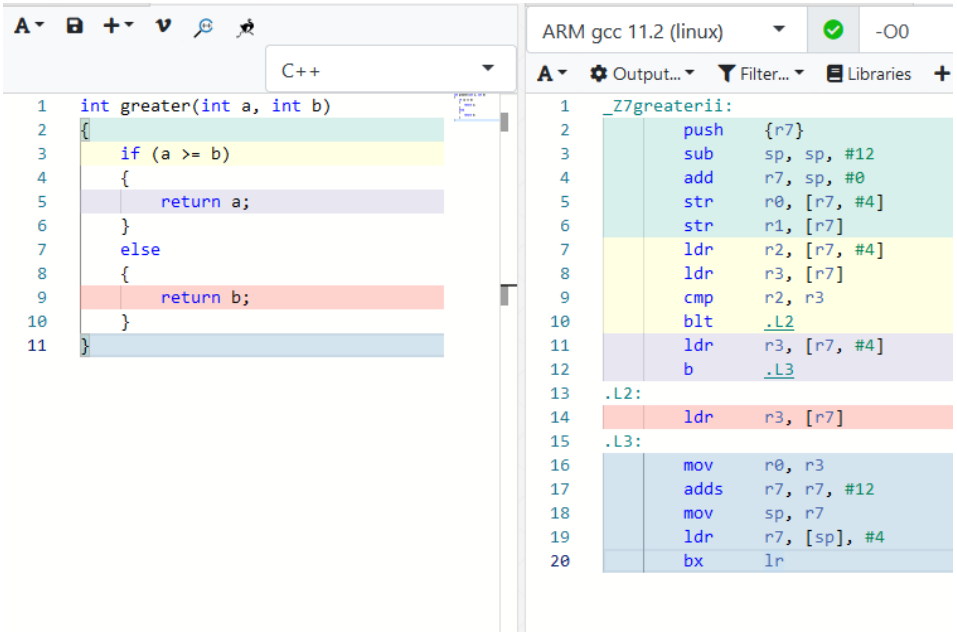

Рис. 3.2. Приклад функції для архітектури ARM

3.2. Tricore

Tricore – 32 бітна архітектура мікроконтролерів від компанії Infineon. Вона поєднує елементи RISC архітектури, MCU та DSP в одному чіпі. DSP (Digital Signal Processor) – спеціалізований процесор, архітектура якого оптимізована для обробки цифрових сигналів в реальному часі. Задачею таких процесорів найчастіше є цифрова фільтрація, перетворення Фур'є, пошук сигналу тощо [22]. В математичному розрізі ці операції зводяться до поелементному множенню елементів багатокомпонентних векторів дійсний чисел. DSP процесори оптимізовані саме для таких задач, більш того вони мають велику енергоефективність, саме тому їх використовують в портативних пристроях. MCU (Microcontroller Unit) – комп'ютер на одному чіпі, включає в себе один або декілька процесорів разом із пам'яттю, пристроями вводу та виводу, таймерами та іншою периферією [23]. Використовується здебільшого у вбудованих системах. Усі ці особливості разом із вбудованими механізмами захисту роблять пристрої Tricore ідеальним рішенням для широко спектру автомобільних та індустріальних пристроїв.

Рис. 3.3. Рекламний пост Infineon про сфери використання продукції

Tricore ISA (Instruction Set Architecture) комбінує в собі можливості для обчислень в режимі реального часу та можливості load-store RISC архітектури в одному модулі. Архітектура підтримує 32-бітний адресний простір та дозволяє інтеграції з різними системними функціями як мультипроцесинг. Tricore підтримує 16-бітні та 32-бітні інструкції. 16-бітні є підмножина 32-бітних інструкцій й використовуються для зменшення розміру коду, пам’яті та витрат енергії. Підтримка систем реального часу досягається через мінімальну затримку переривань шляхом уникнення довгих багатоциклічних інструкцій та покращеної системи переривань з можливостями швидкої зміни контексту. Отже до основних можливостей архітектури можна віднести:

- 4 Гб адресного простору
- 16-бітні та 32-бітні інструкції
- Пріоритет інструкцій без зайвих циклів
- Модуль передбачення переходів в умовних інструкціях

Цікавим є те, що саме передбачення переходів стало однією з найвідоміших вразливостей процесорів Intel під назвою Spectre.

До набору регістрів Tricore відносяться 32 регістри загального призначення GPR, регістр PC, два спеціальних регістра для інформації про стан виконання PCXI та PSW [26].

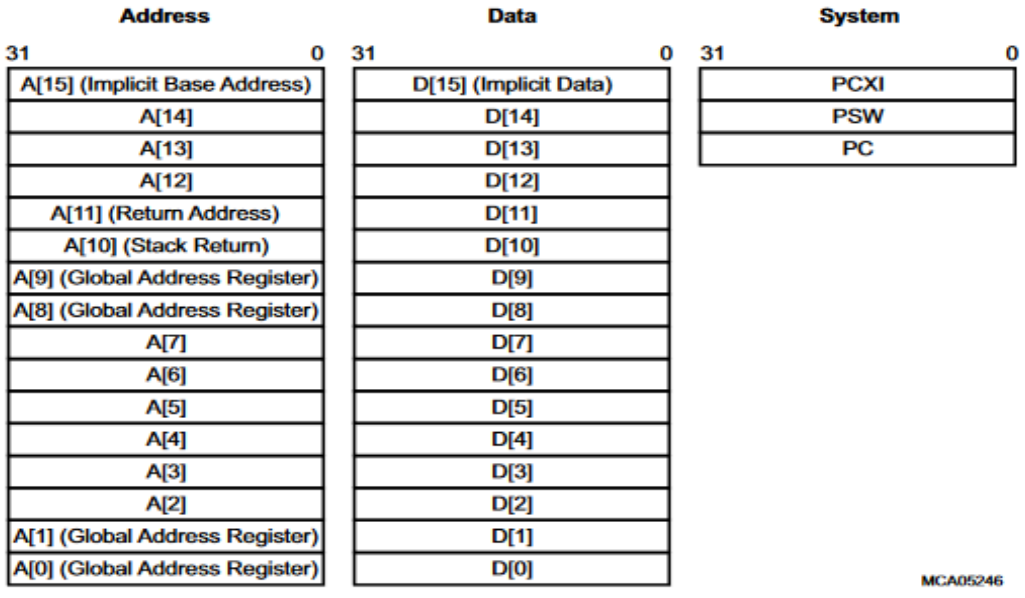

Рис. 3.4. Схема регістрів архітектури Tricore

32 регістри загального призначення поділені на шістнадцять регістрів для збереження адрес та шістнадцять регістрів для збереження даних (рис. 3.4). Це доволі унікальна архітектурна особливість, яку варто взяти до уваги під час аналізу файлів під архітектуру Tricore. Так функція, що повертає посилання на дані, запише результат саме в регістр A. Але якщо повертати тип даних напряму, то результат буде записаний в регістр D. Таким чином не існує одного специфічного регістру для повернення значень з функцій. Чотири регістри виконують спеціальні архітектурно-залежні функції. A11 зберігає адресу повернення після виклику функції або під час умовних інструкцій, A10 (SP) вказує на початок стеку. A15 та D15 – регістри, що використовуються в 16-бітних інструкціях як комірки для збереження даних чи адреси. Це спрощує процес кодування таких інструкцій.

Архітектура також підтримує 64-бітні числа. Для цього існує вісім E регістрів, що накладаються на регістри даних D та вісім P регістрів, що накладаються на регістри адрес A. Таким чином регістр E0 вказує на ту саму пам’ять, що й регістр D0 і займає 64 біти. Наступний регістр E1 вказує на пам'ять регістру D2 так як D1 - це наступні 32 біти регістру E0. Для підтримки чисел з плаваючою точкою окремих регістрів не існує й використовується набір GPR. Дані з плаваючою точкою зберігаються та відновлюються автоматично використовуючи функцію швидкої зміни контексту.

Аргументи в функції передаються або через регістри або через стек. Перші чотири аргументи з даними будуть передані через регістри D4, D5, D6, D7. Аргументи з вказівникам (int*) будуть передані через регістри A4, A5, A6, A7. Аргументи, що виходять за ці межі, будуть передані через стек. Втім існує альтернативна конвенція викликів функцій, що гарантує передачу усіх аргументів через стек. Ця особливість також впливає на аналіз адже функція з сигнатурою «void func(int a)» будет значно відрізнятись від функції з сигнатурою «void func(int* a)».

3.3. PowerPC

PowerPC – RISC архітектура створена компаніями Apple, IBM та Motorola, що пізніше була перейменована в Power ISA. Оригінально архітектура була створена для використання в персональних комп'ютерах Apple до переходу на процесори Intel. Після цього на ринку персональних комп'ютерів її майже не використовували, втім вона набула популярності для вбудованих високопродуктивних процесорів. Однією із задач даної архітектури була сумісність з багатьма типами систем та створення однієї ISA для персональних комп'ютерів, вбудованих контролерів, наукових та графічних обчислювальних станцій та мікропроцесорів.

Унікальними властивостями даної архітектури, що є широка підтримка операцій над числами з плаваючою точкою. Для цього існують окремі регістри та набір інструкцій. Архітектура також підтримує як LE так і BE порядки байт.

Процесори PowerPC можуть працювати в двох режимах:

- Supervisor mode
- User mode

User mode використовується лише програмами користувача, supervisor mode може використовувати лише операційна система. Ці два рівні розділяють доступ до регістрів та дозволяють операційній системі краще контролювати поведінку програм, керувати віртуальною пам'яттю та критичними ресурсами. Інструкції, які контролюють стан процесора та механізм трансляції адрес або спеціальних регістрів, можуть бути виконані лише в режимі підвищених supervisor mode [25].

Регістри PowerPC поділені за категоріями і перша з них це 32 регістри загального призначення (GPRs). Вони використовуються програмним забезпеченням згідно з власними цілями. Наступною категорією є регістри з плаваючою точкою (FPRs). Це 32 64-бітних регістри для окремих інструкцій з плаваючою точкою. Результати операцій і порівняння FPR регістрів записуються

в спеціально статус регістр FPSCR, але в окремих випадках можуть бути записані в звичайний умовний регістр CR [26].

Наступною групою є регістри спеціального призначення (SPR), на відміну від інших архітектур вони не обираються з набору базових регістрів, а є окремою групою й доступ до них відбувається за допомогою спеціальних інструкцій.

CR (Condition Register) – 32 бітний регістр для збереження результатів операцій, що використовуються переважно умовними конструкціями. Дані в регістрі розділені на 8 груп по 4 біти (рис. 3.5).

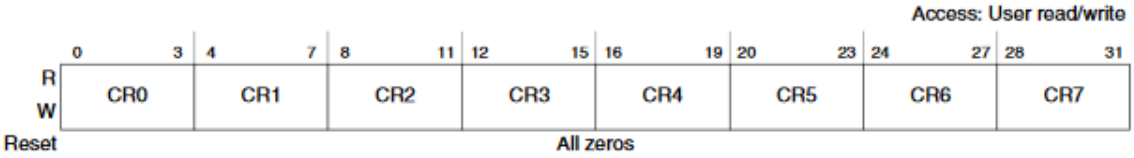

Рис. 3.5. CR регістр архітектури PowerPC

В кожній групі біти також розподілені за спеціальною маскою (рис. 3.6). Кожен біт відповідає за те, чи була виконана специфічна умова. CR0 використовується для всіх операцій з цілими числами. Для роботи з числами с плаваючою точкою використовується група R1.

Bits	Description
0	Negative (LT)—This bit is set when the result is negative.
1	Positive (GT)—This bit is set when the result is positive (and not zero).
2	Zero (EQ)—This bit is set when the result is zero.
3	Summary overflow (SO)—This is a copy of the final state of XER[SO] at the completion of the instruction.

Рис. 3.6. Значення бітів в групі регістра CR

XER регістр використовується для зберігання результату математичних операцій з числами там має наступну структуру (рис. 3.7). Біти в інструкції вказують чи відбулося певне переповнення числа в результаті математичних операцій. До них відносяться overflow, summary overflow та carry.

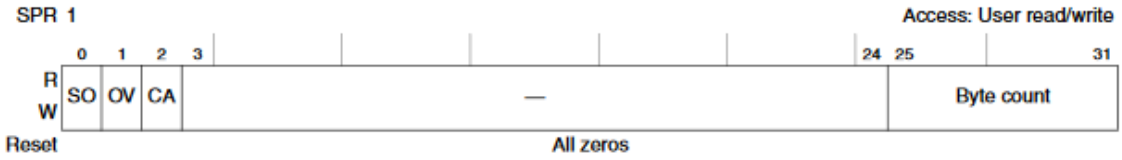

Рис. 3.7. Регістр XER архітектури PowerPC

LR регістр зберігає адресу альтернативного переходу під час виконання умовних інструкцій. CTR регістр є спеціальним лічильником для умовних інструкцій та циклів.

PowerpPC має додатковий набір інструкцій під назвою VLE (Variable Length Encoding). Це набір інструкцій, який є копію оригінального набору, але має розмір 16 або 32 біти. 16-бітні версії інструкцій дозволяють створювати ефективні з точки зору розміру файли для вбудованих систем, де щільність коду впливає на фінальну вартість всього пристрою. Альтернативний варіант кодування працює на рівні сторінок пам'яті і один файл може мати як VLE, так і non-VLE інструкції. Спеціальний біт повідомляє про формат кодування для певної сторінки. Такі інструкції мають 16 бітне вирівнювання та мають лише big endian порядок байт. Деякі регістри та частини регістру CR недоступні в VLE режимі. Втім, незважаючи на це, VLE інструкції виконуються в тому самому оточенні, що й non-VLE з точки зору віртуальної пам'яті, системи переривань тощо.

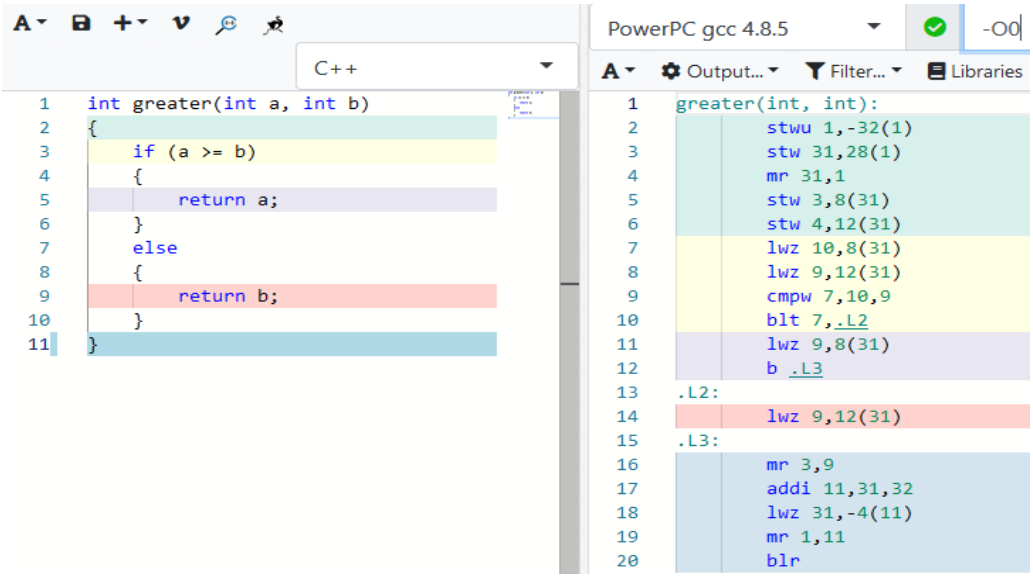

Рис. 3.8. Результат компіляції під архітектуру PowerPC

3.4. Компілятори, оптимізація та обфускація

Для роботи з усіма описаними архітектурами потрібні програми компілятори. В контексті реверсивної інженерії вони потрібні для навчання та створення програм, щоб підтвердити певну теорію стосовно досліджуваного файлу. Специфічні компілятори мають свої особливості побудови, які можуть бути підказкою під час реверсингу. Якщо архітектура є власністю однієї компанії і працює лише на її чіпах, то для даної архітектури використовується специфічне програмне забезпечення від розробника архітектури. Деякі компілятори є цілком доступні та безкоштовні, деякі з них є більш закриті. В такому випадку розробник може дозволити навіть використання в комерційних цілях та вимагає за це лише реєстрацію на сайті. Ну і остання модель розповсюдження, що викликає складності для реверсингу - це повністю платні системи, деякі з них можуть надаватись безкоштовно разом з приладом, під який буте створюватись програмне забезпечення, але це влаштовує тільки тоді, коли для аналізу постачають як файли, так і сам девайс. Найкращим варіантом в такому випадку буде наявність місячної підписки адже специфічне програмне забезпечення може бути потрібне лише для роботи із специфічним файлом і необхідності в повному придбанні немає.

Одним із найпопулярніших компіляторів з підтримкою різних архітектур і навіть кроскомпіляції є gcc. Він повністю безкоштовний і список підтримуємих архітектур дуже великий. Компілятор використовує інтерфейс командного рядку, що є дуже зручним при побудові маленьких нескладних специфічних прикладів. Втім він також використовується для великих проєктів, але в цьому випадку краще використовувати додатковий фронтенд як make або cmake.

Із описаних вище архітектур gcc підтримує усі, але для підтримки Tricore потрібні специфічні компоненти від розробника архітектури, які надаються за ліцензією. Це робить підтримку платною, але в той же час за кошти дозволяє використовувати готовий проєкт під TriCore з мінімальною зміною коду.

Для Tricore розробник пропонує використовувати наступні компілятори:

- Tasking
- Hightec
- Wind River

Під архітектуру ARM можна виділити gcc та keil. Втім специфічні пристрої можуть вимагати своє програмне забезпечення. Для компіляції коду під PowerPC використовують gcc, Green Hills, CodeWarrior, S32 Design Studio.

Наступним елементом, що впливає на вигляд коду після компіляції є налаштування компіляторів та особливо рівень оптимізації та обфускації коду. З точки зору оптимізації достатньо знати базові техніки і прийоми, які використовують компілятори для пошуку потенційних вразливостей. Одним із прикладів вразливості з оптимізацією є CWE-14: Compiler Removal of Code to Clear Buffers [28]. Ця вразливість виникає, коли пам’ять з важливою інформацією очищується базовими функціями, які для цього не призначені. Так очищення пам’яті в кінці блоку через функцію memset (рис. 3.9) може бути видалене компілятором в рамках оптимізації через те, що пам’ять далі ніким не використовується і усі операції з нею можна пропустити. В такому випадку конфіденційна інформація залишається доступною.

Example Language: **C**

```
void GetData(char *MFAddr) {
  char pwd[64];
  if (GetPasswordFromUser(pwd, sizeof(pwd))) {

    if (ConnectToMainframe(MFAddr, pwd)) {

      // Interaction with mainframe
    }
  }
  memset(pwd, 0, sizeof(pwd));
}
```

Рис. 3.9. Приклад коду з вразливістю CWE-14

Обфускація – процес перетворення тексту програми або виконуваного коду ускладнюючи аналіз коду та розуміння внутрішніх алгоритмів, але при цьому зберігаючи оригінальний функціонал. Обфускація може відбуватись на рівні алгоритмів, вихідного тексту програми або на рівні асемблерних інструкцій. Спеціальні компілятори та специфічне програмне забезпечення може робити обфускації в процесі компіляції або після неї.

Аналіз обфускованого коду в більшості випадків зводиться до навичок інженера, що аналізує файл та розумінні алгоритмів обфускації. Втім одним із варіантів є програми, що виконують автоматичну деобфускацію. Однією з них ж є Miasm IR. Вона використовує своє внутрішнє представлення коду для деобфускації програми.

Одним із найпростіших варіантів обфускації коду є перейменування функцій та імен змінних щоб ускладнити загальне розуміння логіки коду. Цей захист вирішується поступовим аналізом з перейменування функцій та змінних в інтерактивному дизасемблері.

Наступним методом обфускації є додавання зайвого коду що не впливає на виклик програми. Для роботи з таким варіантом файлу можна використовувати плагіни, що показують код, який приймає участь в виконанні і видаляє блоки, які не використовуються. Також одним із засобів аналізу заплутаного коду є посимвольне виконання блоків. Емуляція через IR також дозволяє обійти такий метод обфускації, як додавання складних умовних конструкцій які важко проаналізувати статично. Під час виконання ці умови завжди ведуть в одну гілку, бо інша гілка виконання є зайвим кодом для ускладнення аналізу.

Таким чином під час аналізу файлів й для боротьби з оптимізаціями та обфускаторами основним засобом є знання можливих технік обфускації та варіантів оптимізації коду. Знання їх алгоритмів та особливостей дозволяють виконувати їх аналіз вручну або з використання специфічного програмного забезпечення та плагінів.

3.5. Етапи аналізу файлів

Аналіз файлу може бути довготривалим проєктом в залежності від його розміру, складності та мети аналізу. Втім кожен аналіз включає в себе певний набір етапів. Саме ці етапи будуть використані в порівнянні інструментів аналізу під різний набір архітектур.

Кожен аналіз починається із визначення типу файлу та його архітектури. Виконуванні файли в залежності від платформи виконання можуть мати різні типи. В середовищі Linux таким типом є ELF файл. ELF (Executable and Linkable Format) – формат двійкових файлів, що включає в себе заголовок файлу, таблицю заголовків програми, таблицю заголовків секцій, самі секції і сегменти. В операційній системі Windows частіше всього зустрічаються PE файли. PE (Portable Executable) – формат файлу, що містить всю інформації, яка необхідна PE завантажувачу для відображення файлу в пам’яті. Він містить в собі вказівники для динамічних бібліотек, таблиці експорту та імпорту API функцій, дані для управління ресурсами. Аналіз таких типів файлів є найпростішим адже вони містять в собі усю необхідну інформації про розміщення коду, списки залежностей, архітектуру тощо. На практиці у вбудованих системах таких файлів не буде, замість них будуть сирі бінарні файли без заголовків і всю необхідну інформації потрібно буде знаходити вручну.

Отже першим етапом аналізу є визначення архітектури та початкової віртуальної адреси. Є різні методи розпізнавання базової адреси та архітектури. Деякі компілятори мають певний набір байт, що можуть вказувати на те, під яку архітектуру був побудований файл. Так про ARM код можуть підказати повторюванні байти 0x70 0x47, що є інструкцією bx lr для виходу з функції. Одним із методів аналізу є перебір готових архітектур і вибір найкращого варіанту за кількістю отриманих констант та функцій.

Для визначення функцій є декілька підходів. Перший із них - статистичний аналіз інструкцій виклику. В різних архітектурах є свої інструкції виклику функцій. В більшості випадків вини мають наступний вигляд: call (address).

Таким чином проаналізувавши усі інструкції виклику і повернення можна створити готову таблицю. Але цей підхід працює, коли адреса функції відома під час компіляції і є константною. Перешкодою в даному випадку можуть бути або функції, які не викликаються, або функції, адреса який вираховується в процесі роботи. Рішенням може бути посимвольна емуляція певної частини коду й перевірка адреси виклику в процесі роботи.

Наступною технікою розпізнавання функції є визначення прологу і епілогу. Кожна функція перед початком своєї роботи повинна провести певну ініціалізацію. В більшості випадків це бути ініціалізація регістрів стеку. В кінці роботи функція повинна закрити кадр стеку використовуючи для цього також статичний набір інструкцій. Різні компілятори і архітектури використовують різні підходи ініціалізації стеку. Чим довший набір байт при цьому використовується, тип надійніші будуть результати. На рис. 3.10 можна побачити пролог функції my_sum, який виділений зеленим кольором. Епілог функції знаходиться за адресою 0x0010115f і складається з двох інструкцій 5d c4. Такий набір символів іноді може бути недостатнім для визначення і викликати колізії.

```
                        my_sum                                          XREF
00101149 f3 0f 1e fa     ENDBR64
0010114d 55              PUSH       RBP
0010114e 48 89 e5        MOV        RBP,RSP
00101151 89 7d fc        MOV        dword ptr [RBP + local_c],EDI
00101154 89 75 f8        MOV        dword ptr [RBP + local_10],ESI
00101157 8b 55 fc        MOV        EDX,dword ptr [RBP + local_c]
0010115a 8b 45 f8        MOV        EAX,dword ptr [RBP + local_10]
0010115d 01 d0           ADD        EAX,EDX
0010115f 5d              POP        RBP
00101160 c3              RET
```

Рис. 3.10. Пролог функції my_sum архітектури ARM

Втім оптимізуючі компілятори будуть працювати із кадром стеку через глобальні регістрі зменшуючи розмір прологів і епілогів.

Найкращим способом пришвидшити виклик функції це взагалі відмовитись від функції і вбудувати її код прямо в контекст виклику. Такі функції називаються вбудованими. Вони не мають прологів та епілогів і не викликаються через спеціальні інструкції. Єдиним способом ідентифікації вбудованих функцій є пошук повторюваних блоків в багатьох місцях коду. Функції вбудовуються за рішенням компілятора й навіть спеціальна директива inline в вихідному коді не гарантує, що функція буде вбудованою. Таким чином в більшості випадків компілятор буде вбудовувати функції з простим набором інструкцій, де збільшення об’єму коду буде виправдане порівняно із зайвим часом на зміну контексту під час виклику функції. На рис. 3.11 можна побачити те, як оптимізатор вбудував функцію my_sum, яка виводила на екран текст hello та повертала суму двох чисел. В даному випадку оптимізація не тільки вбудувала виклик функції, а й оптимізувала рахування суми чисел тому, що виклик функції використовував один зовнішній аргумент argc:

```
int main(int argc, char** argv)
{
    int a_arg = argc;
    int b_arg = argc + 5;
    int res = my_sum(a_arg, b_arg);
    printf("%d", res);
    return 0;
}
```

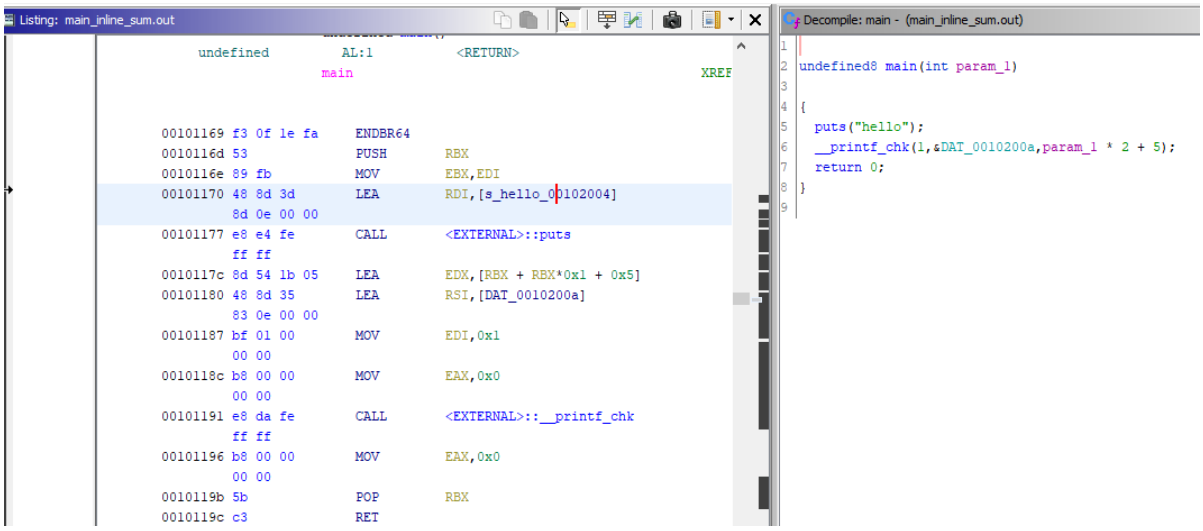

Рис. 3.11. Вбудована функція my_sum

3.6. Порівняння інструментів для аналізу під різні архітектури

Для порівняння програм IDA, Ghidra та Angr потрібно підготувати тестові дані для аналізу. Для оцінки буде використаний тестовий файл з кодом на мові C. Даний файл включає в собі декілька функцій з умовними конструкціями, циклами та структурами даних. Файл буде скомпільований з різними налаштуваннями оптимізації. В якості критеріїв порівняння будуть використані список знайдених функції, якість декомпіляції, граф виклику. Також будуть розглянуті додаткові можливості інструментів. Отже для аналізу використовуємо функції if_func з математичними обчисленнями та умовними конструкціями, while_true_func для перевірки аналізу циклу. Дана функція не повертає значення, ця помилка була додана спеціально. Функція for_func з двома вбудованими циклами та внутрішнім викликом функції if_func для перевірки графу викликів. В останній функції був доданий код, який ніколи не буде виконано.

Для компіляції під архітектуру ARM використано компілятор gcc:

```
$ arm-linux-gnueabi-gcc ./final_test.c -o final_test_arm_base.out -O0
```

В результаті даної кросскомпіляції був створений файл. За допомогою програми readelf перевіряємо те, що файл побудований під архітектуру ARM (рис. 3.12). Таким же чином компілюємо ще два файли з різними параметрами оптимізації:

```
$ arm-linux-gnueabi-gcc ./final_test.c -o final_test_arm_O1.out -O1
$ arm-linux-gnueabi-gcc ./final_test.c -o final_test_arm_O2.out -O2
```

Після цього завантажуємо файл у SRE Ghidra. Вона підтримує архітектуру ARM тому без проблем зчитала із заголовку усю необхідну інформацію і завантажила файл за віртуальною адресою 0x00010000. На рис. 3.13 можна побачити декомпіляцію функції main. В цілому суть функції зрозуміла, ida розпізнала 4 виклики імпортованої функції scanf.

```
vlad@vlad-vm:/media/sf_diploma$ readelf -h ./final_test_arm_base.out
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x1045c
  Start of program headers:          52 (bytes into file)
  Start of section headers:          7580 (bytes into file)
  Flags:                             0x5000200, Version5 EABI, soft-float ABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

Рис. 3.12. Elf заголовок файлу final_test_arm_base.out

Decompile: main - (final_test_arm_base.out)

```
1
2  undefined4 main(void)
3
4  {
5    undefined4 uVar1;
6    undefined4 local_1c;
7    undefined4 local_18;
8    undefined4 local_14;
9    undefined4 local_10;
10   undefined4 *local_c;
11
12   local_c = &__stack_chk_guard;
13   local_1c = 0;
14   local_18 = 0;
15   local_14 = 0;
16   local_10 = 0;
17   __isoc99_scanf(&DAT_00010b48,&local_1c);
18   __isoc99_scanf(&DAT_00010b48,&local_18);
19   __isoc99_scanf(&DAT_00010b48,&local_14);
20   uVar1 = __isoc99_scanf(&DAT_00010b48,&local_10);
21   if (((uint)local_c ^ (uint)&__stack_chk_guard) != 0) {
22                     /* WARNING: Subroutine does not return */
23     __stack_chk_fail(uVar1,(uint)local_c ^ (uint)&__stack_chk_guard,0);
24   }
25   return 0;
26 }
```

Рис. 3.13. Декомпіляція функції main

Функція if_func була декомпільована правильно за виключенням повертаючого типу. В коді це int, в декомпіляторі – unsigned int.

В результаті декомпіляції while_true_func було додано повертюче значення хоча це може бути результат роботи компілятора, адже в такому випадку не існує гарантованого результату. Цикл while було відображено як цикл for. Функція for_func була визначена майже правильно. Перший цикл for був змінений на інструкції if через те, що всередині знаходиться повернення із функції і наступної ітерації зовнішнього циклу ніколи не буде. Втім дизасембльований вид не містить доданої помилки про повернення числа тільки в разі виконання умови. Непрацюючий код компілятор взагалі видалив в рамках базової оптимізації.

Decompile: for_func - (final_test_arm_base.out)

```
1
2  int for_func(int param_1,int param_2,undefined4 *param_3,undefined4 *param_4)
3
4  {
5    undefined4 local_28;
6    undefined4 local_24;
7    int local_20;
8    undefined4 local_1c;
9    int local_18;
10   int local_14;
11   undefined4 local_10;
12   undefined4 *local_c;
13
14   local_c = &__stack_chk_guard;
15   local_28 = *param_3;
16   local_24 = *param_4;
17   local_1c = 0;
18   local_18 = param_1;
19   if (param_1 <= param_2) {
20     local_14 = if_func(param_1,param_1 + 1,&local_28,&local_24);
21     param_1 = printf("%d",local_14);
22     for (local_20 = param_2; local_20 < 1; local_20 = local_20 + 1) {
23       local_10 = if_func(local_20,local_20 + 2,&local_24,&local_28);
24       param_1 = printf("%d",local_10);
25     }
26     if (local_14 < 0x2e) {
27       param_2 = 4;
28     }
29     else {
30       param_2 = 5;
31     }
32   }
33   if (((uint)local_c ^ (uint)&__stack_chk_guard) != 0) {
34                     /* WARNING: Subroutine does not return */
35     __stack_chk_fail(param_1,(uint)local_c ^ (uint)&__stack_chk_guard,0);
36   }
37   return param_2;
```

Рис. 3.14. Декомпіляція функції for_func

Результати декомпіляції і дизасемблеру архітектури x86-64 такі самі, як і для arm. IDA Pro так само знайшла всі необхідні функції і побудувала граф викликів (рис.3 15). Отже при базовому аналізі ці інструменти схожі між собою і різниця мінімальна.

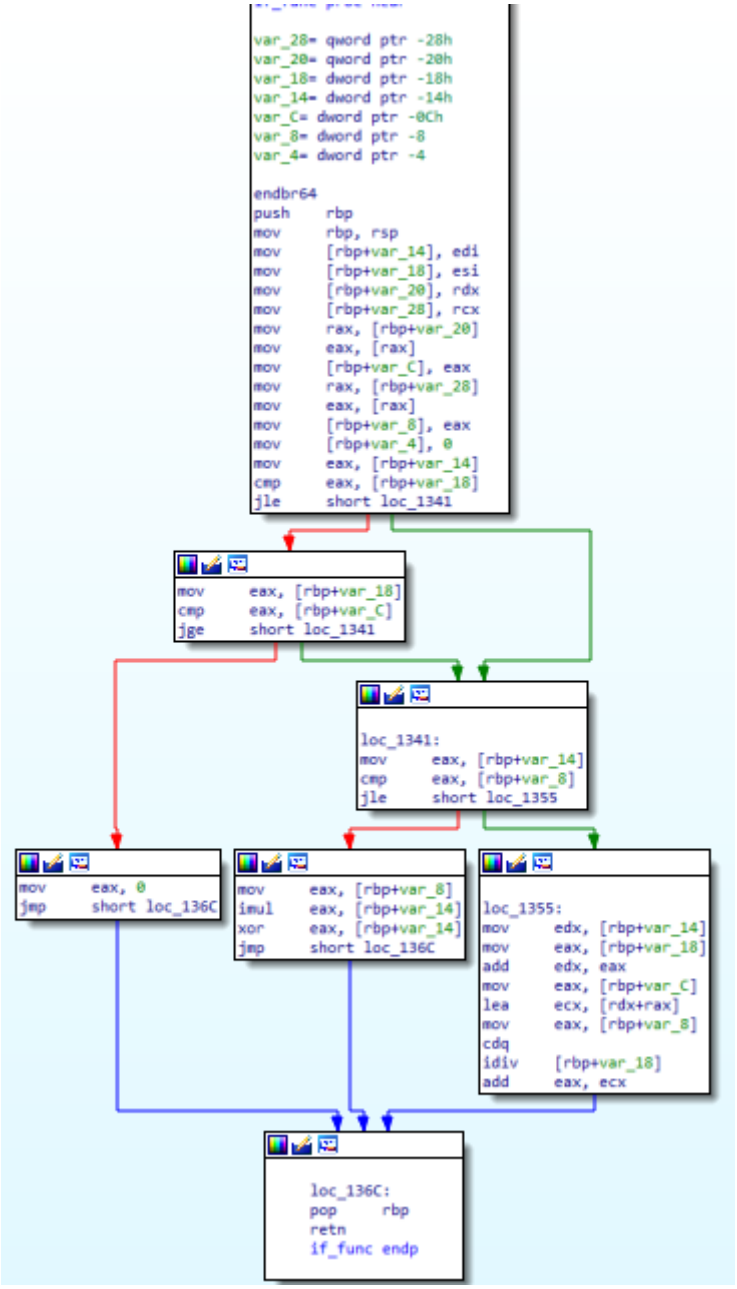

Рис. 3.15. Граф викликів функції if_func в IDA Pro

Останнім етапом дослідження була робота з інструментом Angr через виконання коду мовою python.

Angr є величезним фреймворком для аналізу файлів. Він також автоматично визначив тип архітектури, базову адресу та список функцій програми (рис. 3.16). Для цього був використаний аналізатор CFGFast. Після цього можна аналізувати як окремі блоки коду, так і цілі функції. Angr дозволяє провести емуляцію частини коду надаючи можливість читати та писати в регістри та пам’ять процесу. Під час такої емуляції антидебаг системи вже не працюють адже ми запускаємо не сам файл, а результат його аналізу у вигляді блоків внутрішнього представлення.

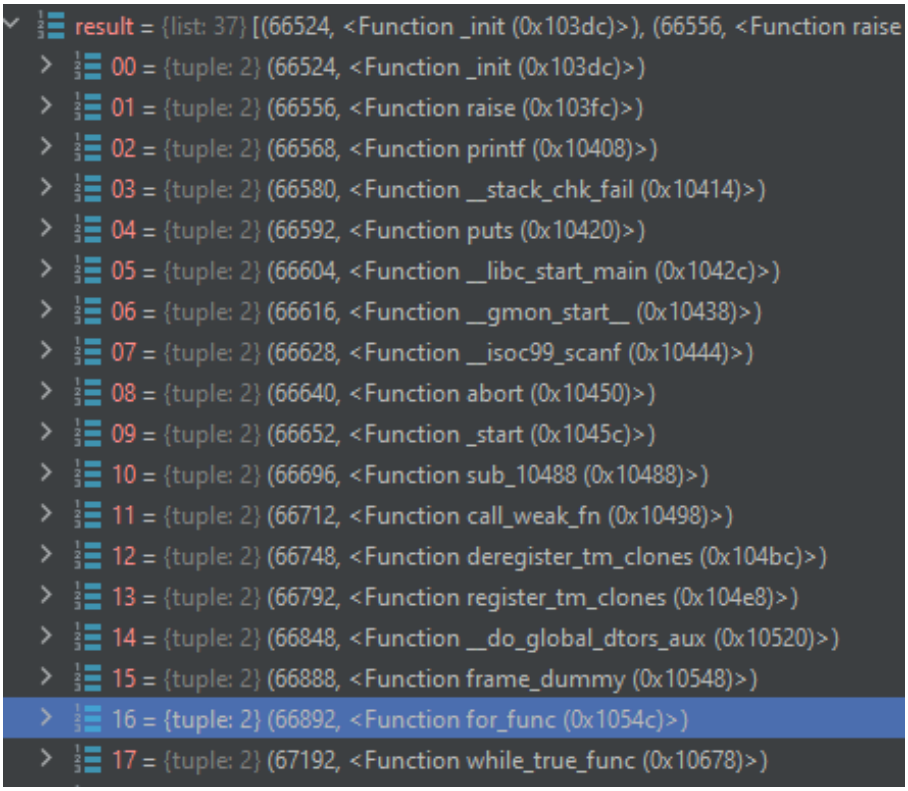

Рис. 3.16. Список функцій файлу в angr

Отже angr має великі можливості для аналізу, втім він має вище поріг входу й менш наглядний інтерфейс. Використання цього фреймворку більше підходить для написання скриптів, що аналізують певні файли.

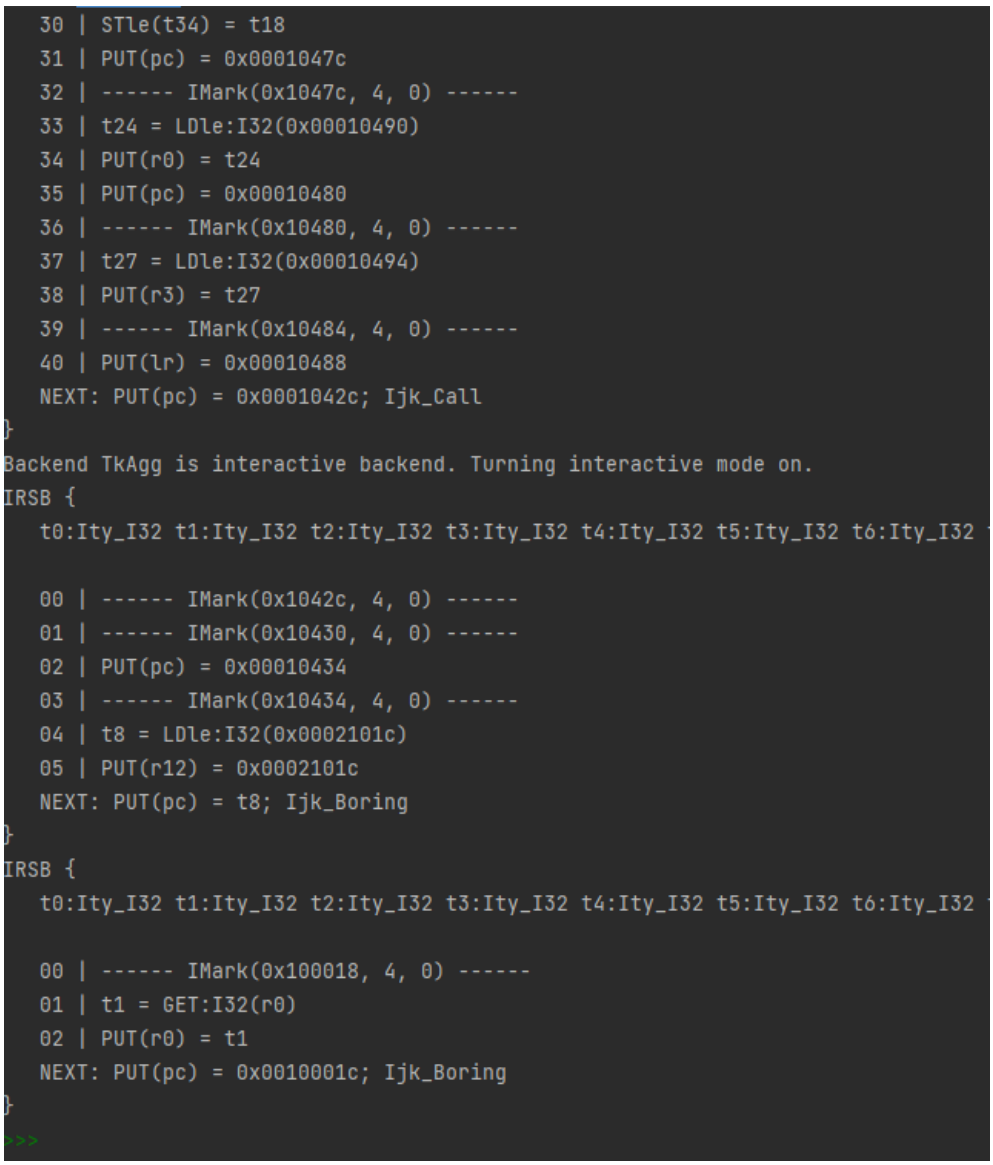

Рис. 3.17. Вивід емуляції стартової функції файлу в angr через VEX IR

Також цей інструмент є чудовим рішенням для підтвердження вразливостей через можливості динамічного аналізу. На рис. 3.17 можна побачити вивід програми, що емулює виконання програми блоками внутрішнього представлення. Якщо ж потрібен статичний аналіз, то тоді краще обрати Ghidra або IDA. Вибір з цих двох інструментів слід починати з того, а чи підтримують вони її. Якщо ж підтримку мають обидва інструмента, то тут вибір краще віддати тому, з чим більше досвіду має інженер.

3.7. Висновки до третього розділу

Архітектура процесора, під який був побудований досліджуємий файл відіграє дуже важливу роль при аналізі. Це в першу чергу впливає на вибір інструменту для даної системи. В сучасному світі з розвитком вбудованих систем, IoT, автомобільної автономної індустрії набувають популярності процесори на RISC архітектурі. До основних переваг даних рішень відносять енергоефективність, швидкість роботи, невеликі розміри адже чіп буде працювати в відокремленому пристрої в складних умовах. До таких архітектур можна віднести ARM, Tricore, PowerPC. Кожна з архітектур має свої особливості, які були додані інженерами для досягнення кращої щільності коду й швидкості обробки. Такі особливості разом із конвенцією викликів, методами роботи з регістрами та пам’яттю, налаштуваннями компіляторів для оптимізації й обфускації роблять створюють велику кількість унікальних варіантів. Приймаючи до уваги всю необхідну інформацію інженер може зробити потрібний вибір. Angr чудово справляється в динамічному аналізі й має відкритий код та потенціал до власного розширення, в свою чергу має обмежену кількість підтримуємих архітектур. IDA Pro та Ghidra краще підходять до статичного аналізу програм та систем. IDA має свої переваги у вигляді найширшої підтримки чіпів. Ghidra, на відміну від конкурента, має відкритий код й краще дозволяє працювати з великою кількість файлів одночасно й з файлами великого розміру. А такі файли найчастіше зустрічаються на ринку вбудованих пристроїв як одна велика прошивка модуля.

РОЗДІЛ 4
ЕКОНОМІКА

При розробці програмного забезпечення важливими етапами є визначення трудомісткості розробки ПЗ і розрахунок витрат на створення програмного продукту.

4.1. Визначення трудомісткості розробки програмного забезпечення

Задані дані:

1. передбачуване число операторів – 250;
2. коефіцієнт складності програми – 1,5;
3. коефіцієнт корекції програми в ході її розробки – 0,4;
4. годинна заробітна плата програміста, грн/год – 230;
5. коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1,1;
6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 1,5;
7. вартість машино-години ЕОМ, грн/год – 14 грн.

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки ПЗ може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

$$t=t_o+t_и+t_a+t_п+t_{отл}+t_д \text{ , людино-годин,} \qquad (4.1)$$

де t_o - витрати праці на підготовку й опис поставленої задачі (приймається 50);

$t_и$ - витрати праці на дослідження алгоритму рішення задачі;

t_a - витрати праці на розробку блок-схеми алгоритму;

$t_п$ - витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ - витрати праці на налагодження програми на ЕОМ;

$t_д$ - витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у ПЗ, яке розробляється.

Умовне число операторів (підпрограм):

$$Q = q * C * (1 + p), \quad (4.2)$$

де q - передбачуване число операторів;

C - коефіцієнт складності програми;

p - коефіцієнт корекції програми в ході її розробки.

$$Q=250*1,5*(1+0,4)= 525 \text{ людино-годин} \quad (4.3)$$

Витрати праці на вивчення опису задачі $t_и$ визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_u=\frac{Q * B}{(75 \dots 85) * k}, \text{ людино-годин,} \quad (4.4)$$

де B - коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

k - коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності.

$$t_u=\frac{525*1,1}{75*1,5}=\frac{577,5}{112,5}=5.1, \text{ людино-годин} \quad (4.5)$$

Витрати праці на розробку алгоритму рішення задачі:

$$t_a = \frac{Q}{(20 \ldots 25) * k}, \text{ людино-годин,} \quad (4.6)$$

$$t_a = \frac{525}{23*1,5} = 16.2, \text{ людино-годин} \quad (4.7)$$

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20\ldots25)*k}, \text{ людино-годин} \quad (4.8)$$

$$t_n = \frac{525}{20 * 1,5} = 17.5, \text{ людино-годин} \quad (4.9)$$

Витрати праці на налагодження програми на ЕОМ:

- за умови автономного налагодження одного завдання:

$$t_{отл} = \frac{Q}{(4\ldots5)*k}, \text{ людино-годин,} \quad (4.10)$$

$$t_{отл} = \frac{525}{4 * 1,5} = 87.5, \text{ людино-годин} \quad (4.11)$$

- за умови комплексного налагодження завдання:

$$t^k_{отл} = 1,5 * t_{отл}, \text{ людино-годин.} \quad (4.12)$$

$$t^k_{отл} = 1,5*87,3 = 131.25, \text{ людино-годин} \quad (4.13)$$

Витрати праці на підготовку документації:

$$t_д = t_{др} + t_{до}, \text{ людино-годин,} \quad (4.14)$$

де $t_{др}$ - трудомісткість підготовки матеріалів і рукопису.

$$t_{др} = \frac{Q}{15..20*k}, \text{ людино-годин.} \quad (4.15)$$

$$t_{др} = \frac{525}{17*1,5} = 20.6, \text{ людино-годин} \quad (4.16)$$

$t_{до}$ - трудомісткість редагування, печатки й оформлення документації

$$t_{до} = 0,75 * t_{др}, \text{ людино-годин} \quad (4.17)$$

$$t_{до} = 0,75*20 = 15.5, \text{ людино-годин} \quad (4.18)$$

$$t_д = 20+15 = 36.1, \text{ людино-годин} \quad (4.19)$$

Тепер розрахуємо трудомісткість ПЗ:

$$t = 50+5.1+16.2+17.5+87.5+36.1 = 213, \text{ людино-годин.} \quad (4.20)$$

4.2. Витрати на створення програмного забезпечення

Витрати на створення ПЗ включають витрати на заробітну плату виконавця програми з/п і витрат машинного часу, необхідного на налагодження програми на ЕОМ.

$$К_{по}=З_{зп}+З_{мв}, грн. \quad (4.21)$$

Заробітна плата виконавців визначається за формулою:

$$З_{зп}=t*С_{пр}, грн, \quad (4.22)$$

де: t - загальна трудомісткість, людино-годин;

$С_{пр}$ - середня годинна заробітна плата програміста, грн/година

$$З_{зп}=213*230=48990, грн. \quad (4.23)$$

Вартість машинного часу, необхідного для налагодження програми на ЕОМ:

$$З_{мв}=t_{отл}*С_{мч}, грн, \quad (4.24)$$

де $t_{отл}$ - трудомісткість налагодження програми на ЕОМ, год.

$С_{мч}$ - вартість машино-години ЕОМ, грн/год.

$$З_{мв}=87.5*15=1313, грн. \quad (4.25)$$

Визначені в такий спосіб витрати на створення програмного забезпечення є частиною одноразових капітальних витрат на створення АСУП.

$$К_{по}=48990+1313=50320, грн. \quad (4.26)$$

Очікуваний період створення ПЗ:

$$T=\frac{t}{B_k*F_p}, \text{міс}, \quad (4.27)$$

де B_k - число виконавців;

F_p - місячний фонд робочого часу (при 40 годинному робочому тижні F_p=176 годин).

$B_k = 1$

$$T=\frac{213}{1*176}=1.2, \quad \text{міс} \quad (4.28)$$

4.3. Маркетингові дослідження

Ринок реверсивної інженерії програмного забезпечення існує в рамках окремих замовлень для реверсу конкретної програми, вбудованого приладу с набором програмного забезпечення тощо. Такі замовлення стають потрібними як допоміжний етап в процесі розробки ПЗ. Широкий набір доступних систем і складність автоматизації організовують ринок таким чином, що в рамках маркетингу між собою змагаються компанії, що надають персональні послуги реверсингу. Кожне окреме замовлення може мати досить специфічну мету, що також призводить до конкуренції серед постачальників даних послуг. До таких завдань можна віднести або реверсинг специфічного протоколу передачі даних, який буде виконано за тиждень, або ж повне дослідження специфічного приладу, що має декілька програмних модулів, які взаємодіють між собою. Таке дослідження може тривати місяці. До послуг реверсивної інженерії можна додати проведення аудити конкретної програми або цілої системи, типовими клієнтами таких аудитів будуть розробники програмних комплексів, які працюють з конфіденційною інформацією.

4.4. Оцінка економічної ефективності

В даній кваліфікаційній роботі були розглянуті інструменти з їх внутрішніми алгоритмами та техніки й архітектурні особливості реверсивної інженерії програмного забезпечення. Більшість з них є засобами мануального аналізу й потребують використання людино-годин. Втім деякі інструменти можуть бути використані з певним відсотком автоматизації таких досліджень. Методи, алгоритми та програмні комплекси, які були розглянуті в даній роботі використовуються в контексті розробки окремого продукту й подаються як послуги для замовника. Таким чином економічна ефективність даних досліджень напряму залежить від економічної ефективності подальшої розробки продукту з використання результатів реверсивної інженерії. Незважаючи на досить розпливчасті умови для аналізу можна зробити висновок, що попит на дані послуги реверсивної інженерії буде збільшуватись й економічна ефективність буде зростати через постійне розширення ринку IoT. Розробка під конкретні пристрої вимагає експертизи з їх аналізу. Окрім цього в умовах віддаленої роботи зростає рівень користування інструментами для доступу до ресурсів й даних, втрата та перехоплення якої може негативно вплинути на економічний стан та репутацію. Такі ризики в свою чергу провокують збільшення замовлень на аудит та перевірку інформаційної та програмної безпеки.

4.5. Висновки до розділу

У розділі були проведені розрахунки трудомісткості розробки програмного забезпечення, витрат на створення програмного забезпечення і тривалості його розробки, а також проведені маркетингові дослідження та аналіз перспективності даної галузі.

ВИСНОВКИ

В даній кваліфікаційній роботи були розглянуті да досліджені методи, алгоритми та програмне забезпечення, що використовується в процесі аналізу виконуваний файлів та систем. Кожен тип програмного забезпечення вирішує конкретну задачу й повинен бути обраним в залежності від мети реверсингу. Для проведення досліджень на власній системі краще використовувати високорівневі інтерактивні комплекси, що підходять для статичного аналізу. Дослідження файлу за допомогою динамічного аналізу можливе через використання внутрішнього представлення та спеціальних інструментів, що можуть з ним працювати й емулювати виконання всієї програми, або ж її частини.

Були розглянуті популярні архітектури процесорів та їх внутрішні алгоритми роботи, які можна використати під час процесу реверсинга. Ці особливості в першу чергу впливають на вибір інструменту для даної системи. Кожна з архітектур має свої особливості, які були додані інженерами для досягнення кращої щільності коду й швидкості обробки. Такі особливості разом із конвенцією викликів, методами роботи з регістрами та пам’яттю, налаштуваннями компіляторів для оптимізації й обфускації роблять створюють велику кількість унікальних варіантів. Angr чудово справляється в динамічному аналізі й має відкритий код та потенціал до власного розширення, в свою чергу має обмежену кількість підтримуємих архітектур. IDA Pro та Ghidra краще підходять до статичного аналізу програм та систем. IDA має свої переваги у вигляді найширшої підтримки чіпів. Ghidra, на відміну від конкурента, має відкритий код й краще дозволяє працювати з великою кількість файлів одночасно й з файлами великого розміру.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Reverse engineering. Wikipedia: website. – URL: https://en.wikipedia.org/wiki/Reverse_engineering
2. Obfuscation (software). Wikipedia: website. – URL: https://en.wikipedia.org/wiki/Obfuscation_(software)
3. OllyDbg. website. – URL: https://www.ollydbg.de/
4. Setting Up Debugging (Kernel-Mode and User-Mode). Microsoft: website. – URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/getting-set-up-for-debugging
5. Software Reverse Engineering Tools. Apriorit: website. – URL: https://www.apriorit.com/dev-blog/366-software-reverse-engineering-tools
6. Finding Vulnerabilities in Closed Source Windows Software by Applying Fuzzing. Apriorit: website. – URL: https://www.apriorit.com/dev-blog/640-qa-fuzzing-for-closed-source-windows-softwares
7. The Evolution of Reverse Engineering. Apriorit: website. – URL: https://www.apriorit.com/dev-blog/732-reverse-engineering-automation-evolution
8. Reverse engineering in a right way. Apriorit: website. – URL: https://www.apriorit.com/dev-blog/364-how-to-reverse-engineer-software-windows-in-a-right-way
9. Fuzzing. Owasp: website. – URL: https://owasp.org/www-community/Fuzzing
10. AFL source code. Google: website. – URL: https://github.com/google/AFL
11. Disassembler. Wikipedia: – URL: https://en.wikipedia.org/wiki/Disassembler
12. Infineon Technologies AG. TriCore 32-bit Unified Processor Core. Volume 2. User Manual / Infineon Technologies AG. – Munich, Germany. – 2012. – URL: https://www.infineon.com/dgdl/tc1_6__architecture_vol2.pdf?fileId=db3a3043372d5cc801374ad9c0653ad9
13. Compiler Exporer. Matt Godbolt: website. – URL: https://godbolt.org/

14. Intermediate Representation. Prof. David Walker: website. – URL: https://www.cs.princeton.edu/courses/archive/spring03/cs320/notes/IR-trans1.pdf
15. Intermediate Representations. Loyola Marymount Universiry: website. – URL: https://cs.lmu.edu/~ray/notes/ir/
16. Intermediate Representation. Wikipedia: website. – URL: https://en.wikipedia.org/wiki/Intermediate_representation
17. Radare2 Framework Overview. Radare: website. – URL: https://book.rada.re/first_steps/overview.html
18. Rasm2 introduction. Radare: website. – URL: https://book.rada.re/tools/rasm2/intro.html
19. Angr Documentation. Angr: website. – URL: https://docs.angr.io/
20. Ghidra Source Code. Natioan Security Agency: website. – URL: https://github.com/NationalSecurityAgency/ghidra
21. Arm CPSR. Keil: website. – URL: https://www.keil.com/pack/doc/CMSIS/Core_A/html/group__CMSIS__CPSR.html
22. Digital Signal Processor. Wikipedia: website. – URL: https://en.wikipedia.org/wiki/Digital_signal_processor
23. Microcontroller. Wikipedia: website. – URL: https://en.wikipedia.org/wiki/Microcontroller
24. Infineon Technologies AG. TriCore 32-bit Unified Processor Core. Volume 1. Core Architecture / Infineon Technologies AG. – Munich, Germany. – 2008. – URL: https://www.infineon.com/dgdl/tc_v131_corearchitecture_v__138.pdf?fileId=db3a304412b407950112b409c4500359
25. Freescale Semiconductor, Inc. MPCxxx Instruction Set / Freescale Semiconductor, Inc. – URL: https://www.nxp.com/docs/en/reference-manual/MPC82XINSET.pdf

26. Special registers in the PowerPC. IBM: website. – URL: https://www.ibm.com/docs/en/aix/7.1?topic=overview-special-registers-in-powerpc
27. Infineon Technologies AG. TriCore 32-bit Unified Processor Core. Embedded Applications Binary Interface (EABI) / Infineon Technologies AG. – Munich, Germany. – 2007. – URL: https://www.infineon.com/dgdl/TriCore_EABI_v2_3.pdf?fileId=db3a304412b407950112b40f8d7a142b
28. CWE-14: Compiler Removal of Code to Clear Buffers. Mitre: website. – URL: https://cwe.mitre.org/data/definitions/14.html
29. Freescale Semiconductor. Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture / Freescale Semiconductor. – 2005. – URL: https://www.nxp.com/files-static/product/doc/MPCFPE32B.pdf?&srch=1
30. International Business Machines Corporation. PowerPC Microprocessor Family: The Programming Environments Manual for 32 and 64-bit Microprocessors / International Business Machines Corporation. – USA. – 2005. – URL: https://wiki.alcf.anl.gov/images/f/fb/PowerPC_-_Assembly_-_IBM_Programming_Environment_2.3.pdf
31. Freescale Semiconductor. Variable-Length Encoding (VLE) Extension Programming Interface Manual / Freescale Semiconductor. – 2006. – URL: https://www.nxp.com/docs/en/supporting-information/VLEPIM.pdf
32. Obfuscation. Techtarget: website. – URL: https://www.techtarget.com/searchsecurity/definition/obfuscation
33. Miasm IR getting higher. Miasm: website. – URL: https://miasm.re/blog/
34. Portable Executable. Wikipedia: website. – URL: https://ru.wikipedia.org/wiki/Portable_Executable
35. Executable and Linkable Format. Wikipedia: website. – URL: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

36. Кріс Касперскі. Искусство Дизассемблирования / Кріс Касперскі., Ева Рокко. – 2008
37. Dennis Yurichev. Reverse Engineeting for Begginers / Dennis Yurichev. – URL: https://beginners.re/paywall/

ДОДАТОК А

КОД ПРОГРАМИ

Приклад файлу для аналізу **hello_gcc.c**:

```
#include <stdio.h>
#include <string.h>

int main()
{
    const char* my_str = "Hello, World!\n";
    printf("%s", my_str);
    if (strlen(my_str) == 0)
    {
        return 1;
    }
    return 0;
}
```

Приклад файлу **main_radare.c** для аналізу інструментом radare2:

```
#include <stdlib.h>
#include <stdio.h>

inline int my_sum(int a, int b)
{
    printf("hello\n");
    return a + b;
}
int main(int argc, char** argv)
{
    int a_arg = argc;
    int b_arg = argc + 5;
    int res = my_sum(a_arg, b_arg);
    printf("%d", res);
    return 0;
}
```

Приклад файлу **final_test.c** для порівняння інструментів під різні архітектури:

```
#include <stdio.h>

const int g_constant = 45;

int for_func(int a, int b, int *c_pointer, int* d_pointer);
int if_func(int a, int b, int *c_pointer, int* d_pointer);
int while_true_func(int a, int b, int *c_pointer, int* d_pointer);

int for_func(int a, int b, int *c_pointer, int* d_pointer)
{
```

```
    int c = *c_pointer;
    int d = *d_pointer;
    int res = 0;

    for (int i = a; i <= b; i++)
    {
        int tmp = if_func(i, i + 1, &c, &d);
        printf("%d", tmp);

        for (int j = b; j <= 0; j++)
        {
            int tmp2 = if_func(j, j + 2, &d, &c);
            printf("%d", tmp2);
        }

        if (tmp > g_constant)
        {
            return 5;
        }

        return 4;

        // dead code
        if (b > d)
        {
            while_true_func(a, b, &b, &a);
            return g_constant;
        }
    }
}

int while_true_func(int a, int b, int *c_pointer, int* d_pointer)
{
    int c = *c_pointer;
    int d = *d_pointer;
    int res = 0;

    while(a + d <= c + d)
    {
        printf("I am string inside strange loop\n");
        a--;
        d--;
        c++;
        d *= 2;
    }
}

int if_func(int a, int b, int *c_pointer, int* d_pointer)
{
    int c = *c_pointer;
    int d = *d_pointer;
    int res = 0;
```

```
    if (a > b && b < c)
    {
        return 0;
    }
    else if (a > d)
    {
        return a ^ d * a;
    }
    else
    {
        return a + b + c + d / b;
    }
}
int main()
{
    int a_arg = 0;
    int b_arg = 0;
    int c_arg = 0;
    int d_arg = 0;

    scanf("%d", &a_arg);
    scanf("%d", &b_arg);
    scanf("%d", &c_arg);
    scanf("%d", &d_arg);
}
```

Приклад програми **angr_emulation.py** на python, що аналізує файл в angr та проводить його емуцяцію в VEX IR

```
import angr
from angrutils import *
import matplotlib.pyplot as plt
import networkx as nx


def draw_graph(graph):
    nx.draw(graph, with_lables=True)
    plt.show()

def main():
    proj = angr.Project("final_test_arm_base.out")
    cfg = proj.analyses.CFGFast()
    func_list = [x for x in proj.kb.functions.items()]
    state = proj.factory.entry_state()
    simgr = proj.factory.simulation_manager(state)

    while True:
        state.block().vex.pp()
        simgr.step()
        state = simgr.active[0]


if __name__ == '__main__':
    main()
```

ДОДАТОК Б

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ДНІПРОВСЬКА ПОЛІТЕХНІКА»

Факультет інформаційних технологій
Кафедра програмного забезпечення комп'ютерних систем

ВІДГУК

Керівника економічної частини Професора Вагонової О.Г.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

На магістерську роботу
Студента ІІ курсу групи 121м-20-1 Візира Владислава Вікторовича
(прізвище, ім'я, по батькові)

На тему: Методи, алгоритми та програмне забезпечення для реверсивної інженерії під різні архітектури процесора

«___»_____2022 р. _____

ДОДАТОК В

ПЕРЕЛІК ДОКУМЕНТІВ НА ДИСКУ

Ім’я файла	**Опис**
Пояснювальні документи	
Diploma_Vizyr.doc	Пояснювальна записка роботи. Документ Word.
Diploma_Vizyr.pdf	Пояснювальна записка роботи в форматі PDF
Вихідні коди та скомпільовані файли для аналізу	
Source_and_binaries.zip	Архів. Містить коди файлів для аналізу, файлу динамічної емуляції та скомпільовані програми для аналізу.
Презентація	
Presentation_Vizyr.ppt	Презентація роботи