Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Система динамічної емуляції x64 коду в середовищі Windows»

08-54.БКР.016.00.000 ПЗ

Виконав: студент 4 курсу, групи 1СП-21б
спеціальності 123 — «Комп'ютерна інженерія»
освітня програма – «Системне програмування»
Стоколос Ю. М.

Керівник к.т.н., доц. каф. ОТ
Кожем'яко А. В.

Рецензент к.т.н., доц. каф. ПЗ
Майданюк В. П.

Допущено до захисту
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д.

«12» 06 2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки
Освітньо—кваліфікаційний рівень бакалвр
Галузь знань — 12 Інформаційні технології
Спеціальність — 123 Комп'ютерна інженерія
Освітня програма — Системне програмування

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ проф., д.т.н.
_____Азаров О. Д.
«21» березня 2025 року

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Стоколосу Юрію Миколайовичу

1 Тема роботи: Система динамічної емуляції x64 коду в середовищі Windows, керівник роботи Кожем'яко Андрій Вікторович, к.т.н., доцент кафедри ОТ, затверджено наказом вищого навчального закладу від «20» березня 2025 року №97.

2 Строк подання студентом роботи 13.05.2025

3 Вихідні дані до роботи: технічний опис системи динамічної емуляції машинного коду, технології розробки: мова програмування C++ та асемблер x64, бібліотека Zydis для декодування інструкцій, інструменти Windows API для роботи з захистом пам'яті, технології керування виключеннями в середовищі Windows, цільова платформа системи — Windows x64, середовище розробки — Visual Studio.

4 Зміст розрахунково-пояснювальної записки: вступ; аналіз сучасних методів емуляції машинного коду; теоретичні основи вирішення ключових проблем динамічної емуляції; проєктування та реалізація ядра системи динамічної емуляції; розширення функціональності системи динамічної емуляції; висновки.

5 Перелік графічного матеріалу: блок-схема головного алгоритму обр інструкцій, результат роботи тестової бібліотеки з використанням емуляції та

6. Консультанти розділів роботи приведені в таблиці 1.

Таблиця 1 — Консультанти роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	к.т.н., доц. каф. ОТ Кожем'яко А. В.	21.03.25	13.06.25
Нормоконтроль	ас. каф. ОТ Швець С. І.	14.05.2025	30.05.2025

7. Дата видачі завдання 21.03.2025 р.

8. Календарний план приведений в таблиці 2.

Таблиця 2 — Календарний план

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання	Підпис
1	Постановка задачі розробки системи динамічної емуляції машинного коду	21.03.25	
2	Аналіз існуючих підходів до емуляції коду та вибір архітектурного рішення	21.03.25— 30.03.25	
3	Формалізація теоретичних основ: пам'ять, виключення, рекомпіляція інструкцій	21.03.25— 30.03.25	
4	Реалізація ядра бібліотеки: обробка виключень, рекомпіляція, колбеки	31.03.25— 13.04.25	
5	Розробка тестової бібліотеки та застосунку для перевірки роботи системи	14.04.25— 27.04.25	
6	Оформлення пояснювальної записки та ілюстративного матеріалу	28.04.25— 11.05.25	
7	Перевірка якості виконання бакалаврської кваліфікаційної роботи та усунення недоліків	13.05.25	

Студент _____ Юрій СТОКОЛОС

Керівник роботи _____ к.т.н., доц. Андрій КОЖЕМ'ЯК

АНОТАЦІЯ

УДК 004.23

Стоколос Ю. М. Система динамічної емуляції x64 коду в середовищі Windows. Бакалаврська кваліфікаційна робота зі спеціальності 123 «Комп’ютерна інженерія», освітня програма «Системне програмування».

Вінниця: ВНТУ, 2025. 99 с.

На укр. мові. Бібліогр.: 29 назв; рис.: 8; табл. 1.

Кваліфікаційну роботу на тему "Система динамічної емуляції x64 коду в середовищі Windows" виконано з метою розробки внутрішньопроцесного інструменту, здатного здійснювати детальний контроль за виконанням машинного коду без повної ізоляції середовища. У межах проєкту розроблено систему, яка перехоплює інструкції під час виконання та рекомпілює їх у реальному часі, забезпечуючи можливість логування, модифікації й розширеного моніторингу. Основу реалізації становить архітектура з розділенням відповідальності між обробником винятків, модулем генерації коду та центральним контролером. Ключовою особливістю є використання прав доступу до пам’яті та перехоплення винятків через KiUserExceptionDispatcher, що дає змогу здійснювати динамічну рекомпіляцію x64 інструкцій з точним збереженням логіки оригінальної програми.

Ключові слова: емуляція коду, динамічна емуляція, Windows, аналіз програмного забезпечення, x64.

ABSTRACT

Stokolos Y. M. Dynamic emulation system for x64 code in the Windows environment. Bachelor's qualification work in specialty 123 "Computer Engineering", educational program "System Programming".

Vinnytsia: VNTU, 2025. 99 p.

In Ukrainian. Bibliography: 29 titles; fig.: 8; table 1.

The qualification work on the topic "System of dynamic emulation of x64 code in the Windows environment" was carried out with the aim of developing an intra-process tool capable of detailed control over the execution of machine code without complete isolation of the environment. Within the framework of the project, a system was developed that intercepts instructions during execution and recompiles them in real time, providing the possibility of logging, modification and advanced monitoring. The basis of the implementation is an architecture with separation of responsibility between the exception handler, the code generation module and the central controller. The key feature is the use of memory access rights and exception interception via KiUserExceptionDispatcher, which allows for dynamic recompilation of x64 instructions with accurate preservation of the logic of the original program.

Keywords: code emulation, dynamic emulation, Windows, software analysis, x64.

ЗМІСТ

ВСТУП

Сучасний розвиток системного програмування вимагає ефективних інструментів для аналізу та контролю виконання машинного коду, особливо в контексті забезпечення інформаційної безпеки, зворотної інженерії та розробки засобів динамічного аналізу. З огляду на ускладнення архітектури операційних систем і широке використання 64-розрядних платформ, актуальним постає завдання розробки динамічних інструментів для детального дослідження виконання коду без ізоляції в окремому середовищі. **Актуальність теми** зумовлена потребою у створенні високопродуктивних засобів внутрішньопроцесної динамічної емуляції, які дозволяють контролювати виконання інструкцій у реальному середовищі без значного впливу на продуктивність. Існуючі рішення, такі як Unicorn Engine або QEMU, мають ряд обмежень щодо інтеграції з реальними програмами, що ускладнює їх використання у складних прикладних сценаріях.

Мета роботи полягає в розробці та реалізації ядра системи внутрішньопроцесної динамічної емуляції для архітектури x64, що забезпечує гнучке перехоплення, аналіз та модифікацію інструкцій без повної ізоляції середовища.

Для досягнення мети були поставлені наступні **завдання**:

— проаналізувати існуючі підходи до емуляції машинного коду та виявити їхні переваги й недоліки;

— дослідити архітектуру Windows x64 з акцентом на віртуальну пам’ять, обробку виключень і специфіку виконання інструкцій;

— спроєктувати загальну архітектуру системи динамічної емуляції;

— реалізувати механізм динамічного рекомпілятора та розробити систему обробки виключень;

— реалізувати систему зворотних викликів (callback-механізмів) для взаємодії з кодом під час його виконання;

— перевірити працездатність системи на практичних прикладах.

Об’єктом дослідження є процес динамічного виконання x64 інструкцій у середовищі Windows.

Предметом дослідження є методи реалізації динамічного контролю за виконанням коду.

Методи дослідження. У роботі використано методи системного аналізу, моделювання поведінки програмного коду на низькому рівні, експериментального тестування компонентів системи в середовищі Windows x64. Також були застосовані інструменти дизасемблювання, трасування виконання та аналізу стану пам’яті.

Апробація результатів кваліфікаційної роботи здійснювалась шляхом участі в LIV Всеукраїнській науково-технічній конференції факультету інформаційних технологій та комп’ютерної інженерії (2025).

Публікації результатів. За результатами роботи опубліковані тези доповіді: Богомолов, С., Кожем’яко, А., & Стоколос, Ю. (2025). Система динамічної емуляції x64 коду в середовищі Windows в ВНТКП ВНТУ. Факультет інформаційних технологій та комп'ютерної інженерії. Отримано з https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2025/paper/view/24197.

1 АНАЛІЗ СУЧАСНИХ МЕТОДІВ ЕМУЛЯЦІЇ МАШИННОГО КОДУ

1.1 Поняття емуляції та її види

Емуляція машинного коду [1] — це спосіб імітувати роботу однієї апаратної або програмної платформи за допомогою іншої. Фактично, створюється середовище, яке дозволяє виконувати інструкції іншої архітектури без необхідності фізично мати відповідне обладнання. Такий підхід дає змогу запускати бінарні файли, створені під одну систему, на зовсім іншій, а також ретельно відстежувати їхню роботу — що особливо цінно для цілей аналізу, тестування чи кібербезпеки.

Сфера застосування емуляції досить широка. Вона допомагає забезпечити сумісність між різними пристроями, досліджувати поведінку потенційно небезпечних програм, вивчати роботу низькорівневих систем без ризику пошкодити реальну машину, а також ефективно розробляти та відлагоджувати нове програмне забезпечення. Найчастіше емуляцію використовують у сфері інформаційної безпеки — зокрема для антивірусного аналізу, реверс-інжинірингу, динамічного трекінгу виконання та під час створення відладчиків чи віртуалізованих середовищ.

Методи реалізації емуляції можна поділити на кілька основних категорій. Найпростішим варіантом є повна інтерпретація — коли кожна інструкція виконується по черзі в режимі інтерпретатора, що забезпечує точний контроль, але суттєво знижує швидкість. Динамічна трансляція, навпаки, перетворює блоки коду у власний формат для системи-хоста, зберігає їх у кеші та дозволяє значно прискорити виконання. Апаратна віртуалізація, яка використовує можливості процесора (наприклад, Intel VT-x або AMD-V) [2, 3], дає змогу працювати майже з нативною продуктивністю, але водночас ускладнює перехоплення окремих інструкцій, що може бути критичним для аналізу.

Також можна виділити відмінності за рівнем інтеграції в процес. Ізольовані емулятори — такі як QEMU [4] чи Bochs — виконують код у відокремленому середовищі, що гарантує безпеку, але менш гнучкі в плані взаємодії з виконуваним процесом. А от варіанти, які інтегруються безпосередньо в пам’ять цільового

процесу (так звана in-process емуляція), забезпечують високий рівень контролю, проте вимагають складної реалізації механізмів перехоплення та обробки команд.

1.2 Огляд існуючих рішень для емуляції x64 коду

У галузі системного програмування та кібербезпеки існує чимало інструментів, що дозволяють імітувати виконання машинного коду. Кожен із них створений під певні потреби: одні максимально точно відтворюють поведінку реального процесора, інші ж — фокусуються на швидкому аналізі окремих фрагментів інструкцій. Загалом такі рішення можна згрупувати в три основні класи: емулятори повної архітектури, віртуалізація із залученням апаратної підтримки та вузькоспеціалізовані програмні бібліотеки, що орієнтовані на моніторинг, аналіз або налагодження виконання.

До першого класу належать повноцінні емулятори, як-от QEMU або Bochs. Вони моделюють поведінку процесора на рівні кожної інструкції, часто додатково імітуючи роботу пам’яті, пристроїв введення/виведення та іншого обладнання. QEMU, наприклад, підтримує як користувацький режим, де виконується лише застосунок, так і повноцінну системну емуляцію з повним середовищем. Основна його перевага — гнучкість і здатність запускати програми, зібрані під інші архітектури. Натомість Bochs орієнтований на максимально точну симуляцію всієї платформи, завдяки чому ідеально підходить для навчання або детального відлагодження. Щоправда, за швидкістю він поступається іншим варіантам. На рисунку 1.1 представлено приклад запуску системи Ubuntu Linux у середовищі QEMU.

До окремого класу рішень належать апаратні технології віртуалізації — зокрема Intel VT-x і AMD-V. Вони не розбирають кожну інструкцію на льоту, як це роблять емулятори, а дозволяють коду виконуватись безпосередньо на фізичному процесорі, водночас ізольовано від основної системи. Такий підхід відкриває шлях до створення високопродуктивних гіпервізорів на кшталт VMware чи Hyper-V. Хоч основне призначення цих технологій — безпечне розгортання віртуальних машин,

вони також знайшли своє місце в дослідницьких практиках, зокрема для відлагодження драйверів і системних компонентів у контрольованому середовищі.

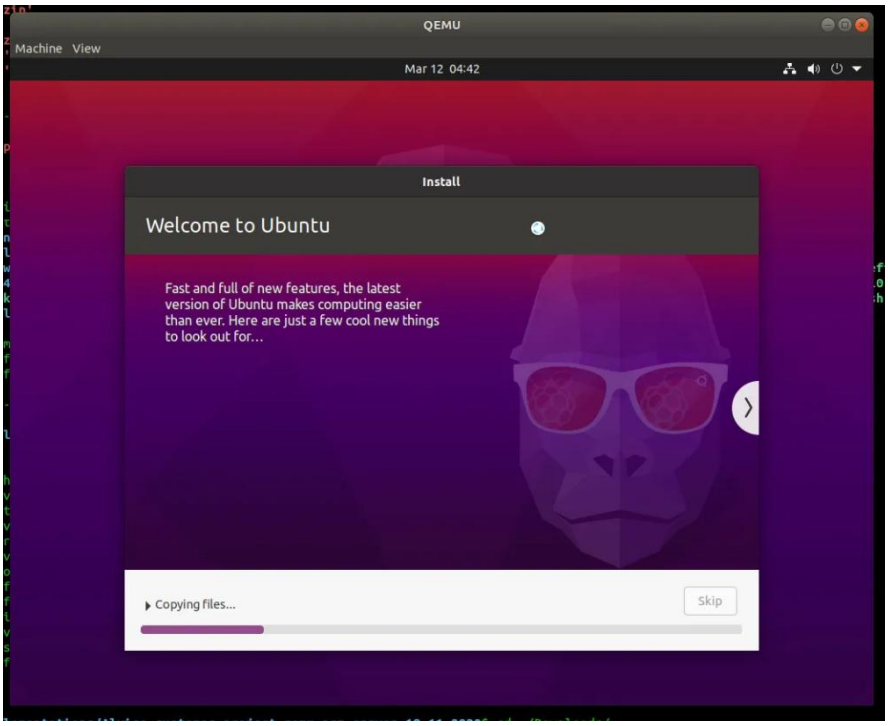

Рисунок 1.1 — Емуляція Ubuntu Linux в QEMU

Серед інструментів, які заслуговують на окрему увагу, можна виділити Unicorn Engine [5] — універсальну бібліотеку, що базується на QEMU, але при цьому легка й зручна для інтеграції. Вона дозволяє емітувати виконання окремих фрагментів коду з можливістю детального відстеження інструкцій, роботи з регістрами та пам’яттю. Цей інструмент активно застосовується у сфері реверс-інжинірингу, для побудови автоматизованих систем аналізу шкідливого ПЗ, а також у різних дослідницьких задачах, де важливо мати контроль над кодом без фізичного виконання.

Те, який інструмент використовувати — Unicorn, гіпервізор або інше рішення — залежить від завдань. Для когось важлива точність та контроль на рівні кожної інструкції, а для інших — продуктивність або зручність інтеграції в уже існуючий C++ застосунок. У будь-якому випадку, жоден із варіантів не позбавлений компромісів: хтось поступається гнучкістю, інший — продуктивністю, а третій — рівнем ізоляції. Саме ці недоліки й штовхають розробників на пошуки нових рішень і методик.

1.3 Ізольоване середовище виконання коду: підходи та обмеження

Ізольоване середовище для виконання коду давно стало фундаментальним інструментом у сфері безпечної емуляції інструкцій процесора. Його активно використовують під час аналізу шкідливого ПЗ, налагодження, а також при запуску потенційно небезпечного або невідомого коду. Суть такого підходу полягає у створенні відокремленої платформи, яка ізолюється від основної операційної системи або процесу. Це дозволяє уникнути небажаних наслідків у реальному середовищі, забезпечуючи повний контроль над тим, як саме виконується досліджуваний код. Найчастіше для досягнення такої ізоляції використовують повноцінні віртуальні машини або спеціалізовані емулятори, що імітують архітектуру на рівні команд та регістрів.

Такі емулятори створюють детальну модель усіх компонентів системи: центрального процесора, оперативної пам’яті, а іноді навіть пристроїв введення-виведення. Весь процес взаємодії з кодом відбувається виключно в межах цієї симульованої системи — жодна інструкція не досягає справжнього апаратного забезпечення. Зазвичай реалізується точне відстеження виконання кожної інструкції, моніторинг доступу до пам’яті, обробка системних викликів тощо. До найвідоміших рішень такого типу можна віднести Bochs або QEMU у режимі повної емуляції, які дозволяють відтворювати роботу коду з високою точністю. Подібний підхід особливо доречний, коли потрібно дослідити внутрішню логіку роботи шкідливого ПЗ або гарантувати, що код не має залежностей від конкретного обладнання.

Однак ізольоване виконання не позбавлене суттєвих мінусів. Передусім — це значне падіння продуктивності, адже повна програмна інтерпретація інструкцій відбувається набагато повільніше, ніж їхнє апаратне виконання. Ще один виклик — це складність точного повторення поведінки реального процесора, особливо коли мова йде про недокументовані або специфічні апаратні функції. У випадках, коли досліджуване ПЗ використовує нестандартні прийоми або намагається виявити підміну середовища, це може призвести до помилкових результатів аналізу.

Окрему проблему становить обмежена інтеграція таких емуляторів з реальною системною інфраструктурою — наприклад, із драйверами, бібліотеками або зовнішніми пристроями. Частково цю проблему намагаються вирішити шляхом використання гібридних підходів, де ізоляція поєднується з вибірковим доступом до системних ресурсів або реалізується на рівні ядра. Проте такі механізми є складними у реалізації та вимагають глибокого розуміння архітектури ОС.

1.4 Концепція внутрішньопроцесної динамічної емуляції коду

Сучасні підходи до емуляції машинного коду дедалі частіше зосереджуються на використанні спеціалізованих бібліотек, які дають змогу відслідковувати, аналізувати або навіть змінювати виконання інструкцій на низькому рівні. Однією з найвідоміших у цій сфері є Unicorn Engine — легка та потужна бібліотека, яка підтримує повну емуляцію інструкцій для низки архітектур. Хоча Unicorn широко застосовується в дослідницьких цілях, зокрема у зворотній інженерії та безпеці, її модель повної ізоляції коду має низку істотних обмежень, особливо при роботі з реальними програмами.

Головний недолік полягає в тому, що Unicorn виконує код у повністю штучному середовищі, вимагаючи створення віртуальної пам’яті, емуляції API, системних викликів та інших елементів ОС. Це суттєво ускладнює використання бібліотеки для складніших задач — розробнику доводиться вручну готувати пам’ять, емулювати оточення, підтримувати внутрішні залежності програми. У результаті така модель часто виявляється непридатною, коли йдеться про складне ПЗ, що активно взаємодіє з ОС і зовнішніми компонентами.

Багато прикладних сценаріїв не вимагають повної емуляції середовища — достатньо лише отримати контроль над окремими ділянками виконання, наприклад, системними викликами чи доступами до пам’яті. У таких випадках створення повної симуляції операційної системи є надмірним і неефективним. Натомість значно доречніше дозволити програмі виконуватись у звичайному середовищі, перехоплюючи лише ті моменти, які підлягають аналізу або модифікації. Це дає змогу зменшити накладні витрати та суттєво спростити процес дослідження.

Втім, реалізація подібного підходу стикається з серйозною проблемою — неможливістю повного контролю над кожною інструкцією в рамках стандартних механізмів ОС. Сучасні системи не надають прямих інтерфейсів для моніторингу інструкцій у реальному часі, тож виникає потреба у використанні непрямих методів. Одним із таких є позбавлення прав на виконання для певного регіону пам’яті — у разі спроби виконати інструкцію з цього регіону процесор згенерує виняток, який можна обробити, перехопити та використати для впровадження необхідної логіки.

Цей виняток стає точкою входу для спеціального механізму — динамічного рекомпілятора, який зчитує інструкції, що мали бути виконані, й створює для них альтернативну версію з додатковим моніторингом або змінами. Такий механізм дозволяє гнучко контролювати хід виконання, перехоплювати системні виклики, логувати доступи до пам’яті та інші дії без потреби повної ізоляції коду. Ключова ідея базується на використанні системної підтримки атрибутів пам’яті: якщо регіон не має права на виконання, то будь-яка спроба його запуску перетворюється на керовану подію.

Упровадження такого механізму відкриває шлях до створення внутрішньопроцесної, динамічної системи контролю, яка діє у реальному середовищі й при цьому не потребує важкої емуляції. Однак така концепція ставить перед розробником ряд технічних викликів. Найбільш критичні з них — це динамічне управління атрибутами пам’яті та перехоплення винятків. Першу задачу можна вважати вирішеною: Windows, як і більшість сучасних ОС, дозволяє змінювати права доступу до пам’яті [6, 7] під час виконання. Зі складнощами ж виникає при роботі з винятками — ОС самостійно керує цим механізмом, і щоб мати змогу втручатись, потрібно перехопити початкову функцію обробки винятків та впровадити власний обробник. Такий обробник має перехоплювати лише ті ситуації, що стосуються цільового регіону пам’яті, передаючи інші назад у штатну систему обробки.

2 ТЕОРЕТИЧНІ ОСНОВИ ВИРІШЕННЯ КЛЮЧОВИХ ПРОБЛЕМ ДИНАМІЧНОЇ ЕМУЛЯЦІЇ

2.1 Організація пам'яті в Windows x64 та структура виконуваних файлів

Операційні системи з 64-розрядною архітектурою, такі як Windows, впроваджують модель віртуальної пам’яті [7], яка дозволяє ефективно ізолювати процеси один від одного, зручно керувати адресним простором і точно налаштовувати права доступу до окремих ділянок пам’яті. Завдяки цій абстракції кожен процес працює у своєму логічному середовищі, хоча фізично всі ці області можуть розміщуватись у спільній оперативній пам’яті. У 64-бітній версії Windows кожен процес має доступ до величезного віртуального простору — до 128 ТБ, де половина зарезервована для користувацького режиму, а інша — для потреб ядра системи. Вся ця структура ґрунтується на поділі пам’яті на сторінки фіксованого розміру, зазвичай 4 КБ. Це дає змогу індивідуально призначати для кожної сторінки окремі параметри доступу. Ключову роль у забезпеченні захисту відіграють таблиці сторінок, які обслуговуються апаратно самим процесором. Завдяки ним операційна система здатна задавати, які ділянки можна лише читати, які — виконувати, а які — змінювати. Такий контроль є основою для безпечного функціонування програм та захисту критичних даних. Приклад відображення віртуальної пам’яті одного з процесів наведено на рисунку 2.1.

У Windows виконувані файли мають формат PE (Portable Executable) [8], який є основним стандартом для збереження та запуску машинного коду в цій операційній системі. Цей формат застосовується як до файлів типу EXE, так і до бібліотек DLL, оскільки обидва мають спільну структуру: вона включає DOS-заголовок, PE-заголовок, таблицю секцій та самі секції, які містять код, дані й службову інформацію [8]. Серед найтиповіших секцій варто відзначити: .text — для коду, .data — для ініціалізованих змінних, .rdata — для констант, .reloc — для релокацій, і .idata — для імпортів. Утім, PE-формат є досить гнучким: структура та назви секцій можуть змінюватися в залежності від компілятора, лінкера чи інструментів, які обробляють файл.

Base address	Type	Size	Protection	Use
0x7ffe0000	Private	4 kB	R	USER_SHARED_DATA
0x7ffe3000	Private	4 kB	R	HYPERVISOR_SHARED_DATA
0xacaa280000	Private	1 MB	RW	Stack (thread 36908)
0xacaa400000	Private	2 MB	RW	
0xacaa600000	Private	1 MB	RW	Stack (thread 20028)
0xacaa700000	Private	1 MB	RW	Stack (thread 22712)
0xacaa800000	Private	1 MB	RW	Stack (thread 13420)
0x20c00c00000	Private	64 kB	RW	
0x20c00c10000	Mapped	8 kB	RW	
0x20c00c20000	Mapped	128 kB	R	ApiSetMap
0x20c00c40000	Mapped	16 kB	R	System activation context data
0x20c00c50000	Mapped	24 kB	R	Process activation context data
0x20c00c60000	Private	8 kB	RW	Shim data
0x20c00c70000	Mapped	4 kB	R	Activation context data
0x20c00c80000	Mapped	68 kB	R	C:\Windows\System32\C_1252.NLS
0x20c00ca0000	Mapped	68 kB	R	C:\Windows\System32\C_437.NLS
0x20c00cc0000	Mapped	12 kB	R	C:\Windows\System32\l_intl.nls
0x20c00cd0000	Mapped	24 kB	R	
0x20c00ce0000	Mapped	24 kB	R	
0x20c00cf0000	Mapped	4 kB	R	
0x20c00d00000	Mapped	64 kB	RW	
0x20c00d10000	Mapped	12 kB	R	C:\Windows\System32\l_intl.nls
0x20c00d20000	Mapped	844 kB	R	C:\Windows\System32\locale.nls
0x20c00e00000	Mapped	68 kB	R	C:\Windows\System32\C_1252.NLS
0x20c00e20000	Mapped	68 kB	R	C:\Windows\System32\C_437.NLS
0x20c01000000	Private	16 MB	RW	Process Heap (ID 1)
0x7df4b2e00000	Mapped	1 MB	R	CSR shared memory
0x7df4b2f00000	Private	4 GB	RW	
0x7df5b2f20000	Private	32 MB	RW	
0x7df5b4f30000	Mapped	4 kB	R	
0x7df5b4f40000	Mapped	2 TB	NA	
0x7ff610a90000	Image	676 kB	WCX	C:\Program Files\WindowsApps\Microsoft.WindowsTerminal_1.22.11141.0_x64__8wekyb3d8bbwe\WindowsTerminal.exe
0x7ff610a90000	Image: Commit	4 kB	R	C:\Program Files\WindowsApps\Microsoft.WindowsTerminal_1.22.11141.0_x64__8wekyb3d8bbwe\WindowsTerminal.exe
0x7ff610a91000	Image: Commit	300 kB	RX	C:\Program Files\WindowsApps\Microsoft.WindowsTerminal_1.22.11141.0_x64__8wekyb3d8bbwe\WindowsTerminal.exe
0x7ff610adc000	Image: Commit	160 kB	R	C:\Program Files\WindowsApps\Microsoft.WindowsTerminal_1.22.11141.0_x64__8wekyb3d8bbwe\WindowsTerminal.exe
0x7ff610b04000	Image: Commit	4 kB	RW	C:\Program Files\WindowsApps\Microsoft.WindowsTerminal_1.22.11141.0_x64__8wekyb3d8bbwe\WindowsTerminal.exe
0x7ff610b05000	Image: Commit	16 kB	WC	C:\Program Files\WindowsApps\Microsoft.WindowsTerminal_1.22.11141.0_x64__8wekyb3d8bbwe\WindowsTerminal.exe
0x7ff610b09000	Image: Commit	4 kB	RW	C:\Program Files\WindowsApps\Microsoft.WindowsTerminal_1.22.11141.0_x64__8wekyb3d8bbwe\WindowsTerminal.exe
0x7ff610b0a000	Image: Commit	4 kB	WC	C:\Program Files\WindowsApps\Microsoft.WindowsTerminal_1.22.11141.0_x64__8wekyb3d8bbwe\WindowsTerminal.exe
0x7ff610b0b000	Image: Commit	184 kB	R	C:\Program Files\WindowsApps\Microsoft.WindowsTerminal_1.22.11141.0_x64__8wekyb3d8bbwe\WindowsTerminal.exe
0x7ffaa3530000	Image	60 kB	WCX	C:\Program Files\WindowsApps\Microsoft.WindowsTerminal_1.22.11141.0_x64__8wekyb3d8bbwe\TerminalThemeHelpers.dll
0x7ffacbb20000	Image	708 kB	WCX	C:\Windows\System32\uxtheme.dll
0x7ffacbe90000	Image	228 kB	WCX	C:\Windows\System32\dwmapi.dll
0x7ffaced40000	Image	664 kB	WCX	C:\Windows\System32\msvcp_win.dll
0x7ffacedf0000	Image	3.8 MB	WCX	C:\Windows\System32\KernelBase.dll
0x7ffacf1c0000	Image	1.3 MB	WCX	C:\Windows\System32\ucrtbase.dll
0x7ffacf5f0000	Image	1.2 MB	WCX	C:\Windows\System32\gdi32full.dll
0x7ffacf730000	Image	1.42 MB	WCX	C:\Windows\System32\WinTypes.dll
0x7ffacf8a0000	Image	168 kB	WCX	C:\Windows\System32\win32u.dll
0x7ffacf970000	Image	7.17 MB	WCX	C:\Windows\System32\shell32.dll
0x7ffad00f0000	Image	868 kB	WCX	C:\Windows\System32\oleaut32.dll
0x7ffad03a0000	Image	1.81 MB	WCX	C:\Windows\System32\user32.dll
0x7ffad0be0000	Image	180 kB	WCX	C:\Windows\System32\gdi32.dll
0x7ffad0d60000	Image	808 kB	WCX	C:\Windows\System32\kernel32.dll
0x7ffad0ff0000	Image	1.1 MB	WCX	C:\Windows\System32\rpcrt4.dll
0x7ffad1590000	Image	960 kB	WCX	C:\Windows\System32\SHCore.dll
0x7ffad1790000	Image	3.52 MB	WCX	C:\Windows\System32\combase.dll
0x7ffad1b60000	Image	2.39 MB	WCX	C:\Windows\System32\ntdll.dll

Рисунок 2.1 — Приклад карти пам’яті процесу Windows

Наприклад, хоча код зазвичай розміщується в секції `.text`, це не є суворим правилом — при використанні обфускаторів [9] або під час ручного редагування файлів назви секцій можуть бути змінені, а їхній зміст — нестандартним. Тому орієнтуватися лише на назву секції недоцільно. Надійнішим методом виявлення виконуваного коду є перевірка атрибутів доступу кожної секції, які вказані у відповідних заголовках: саме вони визначають, чи можна певну область пам’яті виконувати, читати чи змінювати.

Для аналізу вмісту PE-файлів зручно користуватися спеціальними утилітами, зокрема CFF Explorer [10] — популярною програмою, яка дозволяє переглядати структуру, заголовки, таблиці секцій та інші складові виконуваного файлу. На рисунку 2.2 показано інтерфейс програми CFF Explorer з таблицею секцій типового PE-файлу, відкритого в цій програмі.

svchost.exe

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00006AE8	00001000	00007000	00001000	00000000	00000000	0000	0000	60000020
fothk	00001000	00008000	00001000	00008000	00000000	00000000	0000	0000	60000020
.rdata	00004454	00009000	00005000	00009000	00000000	00000000	0000	0000	40000040
.data	000009A0	0000E000	00001000	0000E000	00000000	00000000	0000	0000	C0000040
.pdata	000006D8	0000F000	00001000	0000F000	00000000	00000000	0000	0000	40000040
.didat	000000C0	00010000	00001000	00010000	00000000	00000000	0000	0000	C0000040
.rsrc	00000820	00011000	00001000	00011000	00000000	00000000	0000	0000	40000040
.reloc	00000100	00012000	00001000	00012000	00000000	00000000	0000	0000	42000040

Рисунок 2.2 — Інтерфейс програми CFF Explorer з таблицею секцій довільного PE файлу

Завантаження виконуваних файлів у Windows здійснюється спеціалізованим компонентом системи — PE Loader. Його завдання полягає в тому, щоб розібрати структуру PE-файлу, визначити обсяг пам’яті, необхідний для його розміщення, і ініціалізувати цей простір через системний менеджер пам’яті. Далі відбувається копіювання вмісту секцій у відповідні ділянки пам’яті, обробка таблиці релокацій, завантаження зовнішніх бібліотек та прив’язка імпортованих функцій. Після завершення всіх підготовчих дій управління передається точці входу — для EXE-файлів це початок виконання програми, а для DLL — виклик функції `DllMain`.

Ключовим елементом у цьому процесі є гнучка система атрибутів доступу до пам’яті. Кожному регіону призначаються права читання (R), запису (W) і виконання (X), що дозволяє точно контролювати поведінку коду. Наприклад, секція з машинним кодом отримує дозволи лише на читання та виконання, без можливості модифікації. Цей принцип лежить в основі таких захисних технологій, як DEP (Data Execution Prevention) [11], які запобігають виконанню довільного коду в ділянках, призначених лише для даних.

У випадку динамічної емуляції така система прав відіграє важливу роль. Щоб запобігти прямому виконанню оригінального коду, достатньо забрати атрибут виконання для відповідного регіону пам’яті. У разі, якщо процес спробує виконати інструкцію з цієї області, процесор спровокує виключення, яке можна перехопити на рівні системи. Це дозволяє замість прямого запуску активувати власний емулятор, що і забезпечує повний контроль над виконанням інструкцій. Такий

підхід не потребує повної ізоляції або створення окремого середовища, оскільки все відбувається в межах поточного процесу.

2.2 Механізми обробки виключень у Windows

У Windows реалізовано спеціальні механізми, що дозволяють системі правильно реагувати на виключення, які можуть виникати під час виконання програм. Причини таких виключень бувають різними — це можуть бути помилки самого додатку, наприклад, спроба поділу на нуль чи звернення до недопустимої області пам’яті, або ж події, пов’язані з внутрішніми процесами на рівні ядра. У будь-якому випадку, коли трапляється виключення, процесор генерує відповідне переривання, обробка якого покладається на ядро операційної системи.

Подальші дії залежать від того, в якому режимі сталося виключення — користувацькому чи режимі ядра [12]. У випадку, якщо проблема виникла в ядрі, система зазвичай виконує внутрішню обробку, що може включати виклики драйверів або, якщо помилка критична, завершитися появою BSOD. Якщо ж інцидент трапився у просторові користувача, ядро формує дві важливі структури — EXCEPTION_RECORD [13] та CONTEXT. Перша з них містить відомості про тип та код виключення, а також адресу, де воно виникло. Друга ж дозволяє зберегти повний знімок регістрів процесора в момент помилки — це необхідно для точного відновлення стану виконання. Після формування цих структур керування повертається у користувацький режим, де виконується спеціальна функція KiUserExceptionDispatcher [14], що розміщена у бібліотеці ntdll.dll. Саме вона першою реагує на виключення у користувача і є ключовим елементом для реалізації користувацьких механізмів обробки помилок.

При завантаженні бібліотеки ntdll.dll до дизасемблера, наприклад, IDA Pro [15], та підключенні відповідних символів, можна розглянути детальний алгоритм роботи функції KiUserExceptionDispatcher. В самому її початку перевіряється глобальний вказівник Wow64PrepareForException. Якщо він заданий, функція викликає адресу, на яку цей вказівник посилається, передаючи туди два аргументи — покажчики на структури EXCEPTION_RECORD і CONTEXT. Це відкриває

можливість підміни цього вказівника на власну функцію для перехоплення обробки виключень. Інтерфейс програми IDA Pro з дизассембльованим початоком функції KiUserExceptionDispatcher наведено на рисунку 2.3.

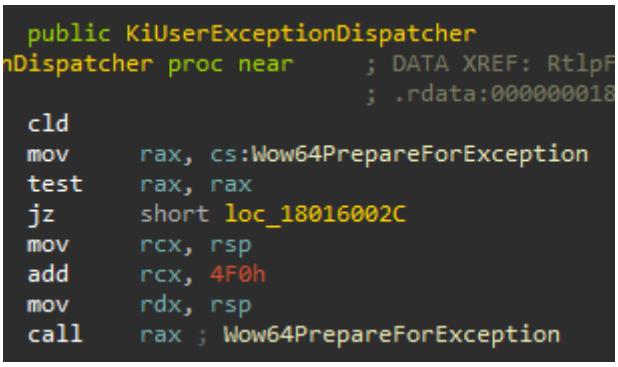

Рисунок 2.3 — Інтерфейс програми IDA Pro з дизассембльованим початоком обробника виключень

Саме це перехоплення і відкриває шлях для повного контролю над процесом обробки виключень. Наша власна функція, яка замінила оригінальний вказівник, може отримати повний доступ до структур, що описують виключення та контекст процесора. Завдяки цьому можна не лише аналізувати, що саме призвело до виключення, але й змінювати значення регістрів процесора у структурі CONTEXT. В результаті, модифікуючи контекст, можна перенаправити виконання коду на будь-яку іншу адресу, наприклад, на рекомпільований варіант початкового коду. Це дозволяє реалізувати гнучкий механізм перенаправлення виконання без прямої зміни початкового коду програми, що є фундаментальним для розробки внутрішньопроцесної динамічної емуляції. Перехоплення функції обробки виключень KiUserExceptionDispatcher через зміну вказівника Wow64PrepareForException дозволяє вирішити одну з проблем динамічної емуляції — перенаправлення виконання на рекомпільований код за допомогою модифікації контексту процесора у випадку виключних ситуацій.

2.3 Thread Information Block та сегментація в x64

Сучасні процесори архітектури x64 мають специфічний підхід до використання сегментації пам'яті [16]. На відміну від старих архітектур, де сегменти активно використовувались для обмеження доступу до різних областей

пам'яті, в x64 сегментація є значною мірою прозорою. Це означає, що сегментні регістри cs, ss, ds та інші налаштовані таким чином, що вказують на один спільний, лінійний адресний простір, і програми не потребують врахування особливостей сегментів при звичайній роботі з пам’яттю. Єдиним винятком є сегментний регістр gs, який має особливу роль в архітектурі x64. Сегмент gs налаштований операційною системою таким чином, що надає кожному потоку доступ до унікального для нього блоку інформації — так званого Thread Information Block (TIB) [17]. TIB являє собою структуру, яка містить різноманітні дані, важливі для роботи конкретного потоку, зокрема адреси меж стеку, покажчики на інформацію про потік, ідентифікатори потоку та процесу, а також спеціальні резервні поля для додаткових потреб системи. В операційних системах Windows x64 доступ до TIB здійснюється саме через сегмент gs.

У процесі проєктування системи динамічної емуляції виникає потреба в надійному сховищі даних, яке було б унікальним для кожного потоку виконання. Це сховище потрібно для того, щоб рекомпільований код під час свого виконання міг звертатися до певних службових даних, не конфліктуючи із виконанням оригінального коду. Зокрема, емулятору потрібно мати змогу зберігати дані, пов'язані з інформацію про контекст виконання, стан потоків та інші потоково-залежні дані. Очевидно, що використання регістрів процесора або стеку потоку для цих цілей не підходить, оскільки це призведе до порушення логіки роботи оригінальної програми. Тому найкращим рішенням є використання саме TIB.

Thread Information Block є ідеальним місцем для зберігання потоково-залежної інформації завдяки його унікальності для кожного потоку і прямому доступу через сегментний регістр gs. Крім того, в структурі TIB передбачено кілька зарезервованих полів, які операційна система не використовує активно. Одним із таких зарезервованих буферів є поле за зміщенням 0x110 [17], яке не використовується Windows у звичайному режимі роботи та є доступним для зберігання службових даних. Це поле і було обрано як перспективне місце для зберігання потоково-залежних даних емулятора.

Ідея полягає в тому, щоб рекомпільований код, який буде створюватись динамічно, міг напряму звертатися до свого службового сховища даних через інструкції з використанням сегменту gs. Такий підхід забезпечить повну позиційну незалежність згенерованого коду, що значно спростить процес динамічної емуляції. Рекомпільований код зможе безпечно оперувати різними потоково-залежними даними та іншою службовою інформацією, при цьому не торкаючись оригінальних регістрів чи стеку потоку. Саме ця концепція лежить в основі реалізації динамічної системи емуляції.

2.4 Модель динамічної рекомпіляції коду

2.4.1 Загальний принцип роботи динамічного рекомпілятора

Концептуальною основою динамічної емуляції машинного коду є процес рекомпіляції, який дозволяє перетворювати окремі інструкції або цілі блоки машинного коду у новий, модифікований варіант, що виконує таку саму роботу, але при цьому забезпечує додаткову функціональність, знаходячись в іншій області пам'яті. До прикладів такої додаткової функціональності можна віднести логування виконуваних операцій, виклики специфічних зворотніх функцій чи здійснення додаткових перевірок на етапі виконання інструкції.

Загальний принцип роботи рекомпілятора досить простий: він отримує на вхід адресу інструкції, з якої потрібно почати рекомпіляцію, а результатом його роботи є адреса нового блоку машинного коду, що був згенерований під час рекомпіляції. Щоб здійснити рекомпіляцію, спочатку необхідно дизасемблювати початковий код, тобто розібрати його на окремі інструкції та проаналізувати їх. Для цього процесу може бути використана спеціалізована бібліотека дизасемблювання; оптимальним вибором є Zydis [18], оскільки вона має дуже високу швидкість роботи, підтримує повний набір інструкцій архітектури x86-64, а також проста у використанні та інтеграції. Рекомпілятор працює з інструкціями у межах окремих блоків. Кожен блок починається з інструкції, яка була подана на вхід рекомпілятора, і завершується першою інструкцією, яка впливає на регістр інструкційного вказівника RIP [19]. Така межа обрана невипадково, адже після виконання RIP-

залежної інструкції [20], наступний блок може бути недосяжним. Важливо враховувати, що за RIP-залежною інструкцією може знаходитись дані, а не машинний код, або ж наступні інструкції можуть бути взагалі недосяжні через постійні переходи чи виклики, які здійснюються з цієї точки. Тому підхід, коли рекомпіляція завершується саме на такій інструкції, дозволяє гарантувати обробку лише тих інструкцій, виконання яких є достовірним.

Для організації процесу рекомпіляції інструкції можна умовно розподілити на три класи, кожен з яких потребує окремої стратегії рекомпіляції. До першого класу належать інструкції, що прямо впливають на RIP (наприклад, безумовні та умовні переходи, виклики та повернення). До другого класу відносяться інструкції з відносною (REL) адресацією [20], які використовують зміщення відносно поточної адреси інструкції, але не впливають прямо на регістр RIP. Третя група — це всі інші інструкції, які не потребують особливої логіки обробки і можуть бути перенесені в згенерований код без додаткових змін.

Важливою концепцією запропонованої моделі рекомпіляції є використання кешування згенерованих блоків коду. Коли певний блок інструкцій вже був рекомпільований одного разу, його результат зберігається в кеші. Всі наступні спроби рекомпіляції цього ж самого блоку не призводять до повторного аналізу і генерації коду, натомість одразу повертають вже готову адресу кешованих даних. Також потрібно враховувати ситуацію, коли рекомпілятор під час обробки інструкцій доходить до фрагменту коду, який вже був рекомпільований та знаходиться у кеші, ще до того як зустрілась перша RIP-залежна інструкція. В такому випадку рекомпіляція достроково завершується на останній необробленій інструкції, а замість подальшого генерування нового коду додається перехід до відповідного кешованого блоку. Такий підхід дозволяє уникнути повторної генерації одного й того самого коду та підвищити загальну ефективність роботи рекомпілятора.

2.4.2 Особливості рекомпіляції RIP-залежних інструкцій

Одним із ключових завдань рекомпілятора є коректна обробка інструкцій, що впливають на значення регістра RIP. Особливість таких інструкцій полягає у їхньому безпосередньому впливі на послідовність виконання коду, через що підхід до їхньої рекомпіляції потребує спеціальних методів та алгоритмів.

З метою оптимізації процесу рекомпіляції передбачається використання спеціальної структури даних, умовно названої глобальною таблицею адрес. Ця таблиця являє собою лінійний масив, який зберігає відповідності між оригінальними адресами інструкцій та адресами їхніх рекомпільованих варіантів. Завдяки цій таблиці рекомпілятор може швидко дізнаватися, чи був вже рекомпільований код за певною адресою, що дозволяє уникнути повторних обчислень та зайвих виключень. Більш детальний опис того, як саме працює ця оптимізація за допомогою глобальної таблиці, буде наведено при описі умовних переходів.

Особливий підхід застосовується до інструкції повернення (ret). Під час генерації нового коду, рекомпілятор спочатку вставляє код, що прочитає адресу повернення зі стеку та тимчасово збереже її у потоково-залежному сховищі. Після цього в рекомпільований код додається спеціальна перевірка цієї адреси. Якщо адреса повернення знаходиться у межах області емуляції, і вже існує рекомпільований варіант для неї, адреса у сховищі замінюється на відповідну рекомпільовану адресу. Потім виконується емуляція здвигу покажчика стеку (RSP) на 8 байт, як це вимагає оригінальна семантика інструкції повернення, і завершується перехід за остаточно визначеною адресою. Такий механізм гарантує повну сумісність із початковою логікою програми та мінімальний вплив на її оригінальні регістри та стек.

Для безумовних переходів [21] та викликів алгоритм рекомпіляції залежить від типу операнда: операнд може бути регістром, безпосереднім значенням (immediate) або задаватися через пам’ять (із базовим регістром RIP чи іншим регістром). В залежності від цього рекомпілятор генерує специфічний код, що

визначає адресу, на яку має здійснюватись перехід або виклик. Ця адреса попередньо зберігається у потоковому сховищі, а після цього рекомпілятор генерує код перевірки адреси через згадану раніше глобальну таблицю. Якщо існує рекомпільований код за цією адресою, адреса переходу буде замінена на адресу відповідного рекомпільованого коду, що суттєво підвищує продуктивність емуляції. В кінці генерується безумовний перехід за цією адресою. Додатково для інструкції виклику (call) генерується вставка оригінальної адреси повернення у стек, щоб повністю зберегти оригінальну поведінку програми.

Умовні переходи [21] рекомпілюються за спеціальним алгоритмом, що дозволяє гнучко реагувати на зміни стану рекомпіляції. Генерується конструкція коду, яка в залежності від результату перевірки умови визначає шлях виконання. Згенерований код читає адреси для обох можливих шляхів (умова виконана або не виконана) безпосередньо із глобальної таблиці. Завдяки цьому механізму, коли за будь-якою адресою в таблиці з'явиться рекомпільований варіант коду, конструкція умовного переходу автоматично почне використовувати саме рекомпільовану адресу замість оригінальної.

Загалом описані вище методи роботи з RIP-залежними інструкціями дозволяють не лише забезпечити правильність емуляції поведінки програми, а й створюють передумови для реалізації в майбутньому таких можливостей емулятора, як детальний моніторинг виконуваного коду, реалізація гнучких систем зворотних викликів та інших функціональних розширень. Крім цього, запропонований підхід є фундаментом для високоефективної оптимізації роботи емулятора і дає змогу створювати позиційно незалежний альтернативний варіант оригінального коду, що суттєво спрощує управління його виконанням.

2.4.3 Особливості рекомпіляції інструкцій з відносною адресацією

Інструкції з відносною адресацією (REL-інструкції) є окремим класом команд, які вимагають особливого підходу в рамках процесу рекомпіляції. Такі інструкції не змінюють напряму регістр RIP, але використовують його значення для обчислення адрес пам'яті через відносні зміщення. До таких інструкцій належать,

наприклад, команди, які використовують RIP-відносну адресацію для доступу до даних або для формування адреси у регістрі, такі як lea.

Основна проблема, яку потрібно вирішити при рекомпіляції REL-інструкцій, полягає в тому, що після перенесення в рекомпільований код оригінальні відносні зміщення втрачають свою коректність. Це пов'язано з тим, що рекомпільований код знаходиться в зовсім іншій області пам’яті, ніж оригінальний, і тому обчислені за початковими зміщеннями адреси будуть вказувати на некоректні дані. Тому рекомпілятор повинен змінити підхід до обчислення цих адрес таким чином, щоб остаточні значення були коректними незалежно від позиції, в якій розташований рекомпільований код. Загальна ідея рекомпіляції REL-інструкцій полягає в тому, щоб замінити початковий механізм RIP-відносної адресації на використання абсолютних адрес. Для цього рекомпілятор спочатку аналізує початкову інструкцію за допомогою дизасемблювання (для цієї мети може бути використана бібліотека Zydis, яка має всі необхідні функції для роботи з інструкціями x86-64). Після дизасемблювання інструкції рекомпілятор визначає абсолютну адресу пам'яті, на яку реально вказувала початкова інструкція, базуючись на її початковому розташуванні в пам'яті. Якщо рекомпілятор зустрічає інструкцію типу lea з RIP-відносною адресацією, він перетворює її на еквівалентну інструкцію завантаження регістра безпосереднім (абсолютним) значенням адреси, до якої вона повинна була вказувати спочатку. Таким чином, рекомпільована інструкція не залежатиме від свого поточного розташування і буде працювати коректно незалежно від місця виконання.

Для інших інструкцій, які використовують RIP-відносну адресацію, але не є інструкцією lea, підхід складніший. Рекомпілятор повинен тимчасово задіяти один з регістрів процесора, який не використовується оригінальною інструкцією. Цей регістр використовується для того, щоб спочатку завантажити в нього абсолютну адресу, отриману на етапі аналізу, після чого згенерувати модифіковану інструкцію, яка використовуватиме цей регістр замість регістра RIP для доступу до потрібної області пам’яті.

3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ЯДРА СИСТЕМИ ДИНАМІЧНОЇ ЕМУЛЯЦІЇ

3.1 Загальна архітектура системи емуляції

Архітектура системи динамічної емуляції побудована таким чином, щоб забезпечити чітке розділення обов’язків між її основними компонентами, при цьому зберігаючи простоту взаємодії між ними та можливість масштабування. Система складається з чотирьох ключових модулів: менеджера виключень, конструктора шелл-кодів, емулятора коду та центрального контролера виконання, який координує роботу всієї системи та забезпечує її цілісність.

Менеджер виключень (exception manager) відповідає виключно за перехоплення виключень, які виникають у користувацькому режимі виконання процесу. Його задача — отримати інформацію про виняток, зберігши вказівники на структури EXCEPTION_RECORD та CONTEXT, і передати ці дані для подальшої обробки вищому рівню системи — центральному контролеру виконання. Сам менеджер не приймає жодних рішень щодо характеру виключення чи того, як з ним поводитись — він лише транспортує інформацію. Такий підхід дозволяє ізолювати низькорівневе перехоплення від високорівневої логіки, спрощуючи і тестування, і розширення.

Конструктор шелл-кодів (shellcode builder) — це модуль, що реалізує побудову машинного коду в динаміці. Він дозволяє формувати послідовності інструкцій відповідно до потреб системи: вставки переходів, викликів, звернення до потоково-залежного сховища через GS-сегмент, оперування регістрами та інші дії, необхідні при трансформації інструкцій. Він є критичним для забезпечення гнучкості та позиційної незалежності згенерованого коду, оскільки дозволяє формувати shellcode-інструкції без прямої прив’язки до абсолютних адрес.

Емулятор коду (code emulator) реалізує механізм динамічної рекомпіляції. Він виконує дизасемблювання оригінального коду, аналіз інструкцій, трансформацію інструкцій у функціонально еквівалентний код з додатковими можливостями (логування, моніторинг, виклики зворотніх функцій) та кешування згенерованих

блоків коду Він не має прямого доступу до системи виключень, але на запит центрального контролера рекомпілює відповідний код.

Центральний контролер (emulation controller) — це головний клас, що об’єднує деякі вищезгадані компоненти в єдину систему. Він наслідує функціональність як менеджера виключень, так і емулятора коду, але при цьому реалізує вищий рівень логіки. Саме цей компонент відповідає за початкове налаштування: зміну прав доступу до регіону пам’яті, який підлягає емуляції, активацію механізмів перехоплення, запуск підсистеми рекомпіляції, та прийняття рішення при кожному виключенні. Після того, як менеджер виключень передає йому інформацію про виняток, контролер аналізує, чи поточна адреса відповідає контрольованій області. Якщо так — він викликає рекомпілятор, отримує адресу згенерованого блоку коду і змінює поле RIP у контексті потоку, перенаправляючи виконання на новий код. Усі інші виключення він перенаправляє назад у систему або обробляє за власною логікою.

Основні класи системи, разом з призначенням та основною функцією наведені в таблиці 3.1.

Таблиця 3.1 — Основні класи системи

Назва класу	Призначення	Основні функції
c_x64_exceptions_manager	Перехоплення виключень у користувацькому режимі	Передача обробки виключень у головний контролер
c_x64_shellcode_builder	Генерація машинного коду	Побудова інструкцій, вставка логіки взаємодії з GS-сегментом
c_x64_code_emulator	Дизасемблювання та трансформація інструкцій	Рекомпіляція блоків, кешування, генерація обгорнутого коду
c_x64_emulation_controller	Головний керуючий клас, який координує усю систему емуляції	Ініціалізація компонентів, зміна прав доступу, обробка виключень та рекомпіляція

Життєвий цикл роботи системи виглядає наступним чином. Після ініціалізації, центральний контролер позбавляє обрану область пам’яті права на виконання. Після цього, при першій спробі виконати інструкцію з цієї області, виникає виняток. Менеджер виключень його перехоплює і передає до контролера. Контролер перевіряє, чи входить адреса у зону емульованого регіону, і, якщо так — викликає рекомпілятор, отримує нову адресу, змінює RIP і відновлює виконання вже з рекомпільованого коду.

3.2 Ефективна система зберігання даних

Однією із важливих задач у розробці динамічної емуляції є створення ефективної системи зберігання та швидкого доступу до даних, що асоціюються з конкретними адресами пам’яті. Для цієї мети була розроблена спеціалізована структура даних під назвою c_page_table_storage. Вона забезпечує високоефективний пошук, зберігання та оновлення інформації, використовуючи підхід, аналогічний до ієрархічного механізму трансляції віртуальних адрес у фізичні в архітектурі x64 процесорів. Основна ідея реалізованої структури полягає в імітації чотирирівневої таблиці сторінок (Page Map Level 4, PML4) [22], яка використовується в апаратній реалізації трансляції віртуальних адрес процесором. Структура представлення віртуальних адрес організована у вигляді спеціального об’єднання virtual_address_t, який розбиває адресу на складові елементи відповідно до стандартного формату адреси в архітектурі x64: PML4-індекс, PDPT-індекс, PD-індекс, PT-індекс та offset. Об’єднання зображено в лістингу 3.1.

Лістинг 3.1 — Об’єднання virtual_address_t

```
union virtual_address_t {
        uintptr_t value;
        struct {
                uintptr_t offset : 12;
                uintptr_t pt_index : 9;
                uintptr_t pd_index : 9;
                uintptr_t pdpt_index : 9;
                uintptr_t pml4_index : 9;
                uintptr_t reserved : 16;};};
```

Кожен рівень таблиці сторінок реалізовано як окремий клас або структуру, а саму таблицю представляє клас c_page_table, що забезпечує навігацію по цих рівнях. Пошук конкретного елемента здійснюється послідовно, починаючи з верхнього рівня (PML4) і рухаючись до нижнього рівня (PT), де вже безпосередньо розташовані значення, які відповідають адресам. При необхідності вставки нового елемента, відсутні вузли створюються динамічно, що дозволяє ефективно використовувати пам’ять. Приклади основних методів, таких як insert_page_value та get_page_value_pointer, наведено в лістингу 3.2.

Лістинг 3.2 — Методи insert_page_value та get_page_value_pointer класу c_page_table

```
inline T* get_page_value_pointer (uintptr_t page_address) {
	virtual_address_t va {};
	va.value = page_address;
	auto pml4 = m_level4.m_entries[va.pml4_index];
	if (!pml4) {
		return nullptr;
	}
	auto pdpt = pml4->m_entries[va.pdpt_index];
	if (!pdpt) {
		return nullptr;
	}
	T* pd = (T*)pdpt->m_entries[va.pd_index];
	if (!pd) {
		return nullptr;
	}
	return (T*)&pd[va.pt_index];
}
inline T* insert_page_value (uintptr_t page_address, const T& value) {
	const c_simple_lock_guard lock (m_internal_mutex);
	virtual_address_t va {};
	va.value = page_address;
	auto pml4 = m_level4.m_entries[va.pml4_index];
	if (!pml4) {
		pml4 = new page_table_t ();
		m_level4.m_entries[va.pml4_index] = pml4;
	}
	auto pdpt = pml4->m_entries[va.pdpt_index];
	if (!pdpt) {
		pdpt = new page_table_t ();
		pml4->m_entries[va.pdpt_index] = pdpt;
	}
	T* pd = (T*)pdpt->m_entries[va.pd_index];
	if (!pd) {
		pd = new T[NUMBER_OF_PAGE_TABLE_ENTRIES] {};
```

```
                pdpt->m_entries[va.pd_index] = (page_table_t*)pd;}
        T* page_value_pointer = (T*)&pd[va.pt_index];
        *page_value_pointer = value;
        return page_value_pointer; }
```

На основі цієї ієрархічної таблиці сторінок побудовано клас вищого рівня — c_page_table_storage. Він призначений для безпосереднього зберігання даних, асоційованих із кожною адресою, та дозволяє ефективно працювати з блоками даних розміром у сторінку пам'яті (4 КБ). Це дозволяє значно скоротити кількість динамічних алокацій пам’яті та збільшити локальність даних. Вставка та отримання елементів у цьому класі виконуються методами insert_value, insert_values та get_entry. Ці методи наведлені в лістингу 3.3.

Лістинг 3.3 — Методи insert_value, insert_values та get_entry класу c_page_table_storage

```
    inline T* get_entry (uintptr_t address) {
        auto page_value_pointer = this->get_page_value_pointer (address);
        if (!page_value_pointer) {
            return nullptr;
        }
        auto page_description = *page_value_pointer;
        if (!page_description) {
            return nullptr;
        }
        virtual_address_t va {};
        va.value = address;
        return &page_description->m_values[va.offset];
    }
    inline T* insert_value (uintptr_t address, const T& entry_to_insert) {
        const c_simple_lock_guard lock (m_mutex);
        auto page_value_pointer = this->get_page_value_pointer (address);
        if (!page_value_pointer || !*page_value_pointer) {
            page_value_pointer = this->insert_page_value (address, new
page_description_t<T> ());}
        auto page_description = *page_value_pointer;
        virtual_address_t va {};
        va.value = address;
        T* final_entry = &page_description->m_values[va.offset];
        *final_entry = entry_to_insert;
        return final_entry;
    }
    inline void insert_values (uintptr_t address, size_t size, const T& value) {
        for (int i = 0; i < size; ++i) {
            insert_value (address + i, value);}}
```

Окремою важливою функцією реалізованої структури є організація зворотного відображення адрес. Це означає, що для будь-якої адреси рекомпільованого коду можна легко знайти відповідну адресу оригінального коду. Така можливість критично важлива при обробці виключень, що можуть виникати під час виконання рекомпільованого коду, і дозволяє оперативно визначати оригінальну адресу, відповідальну за помилку. Для зберігання таких зворотних зв'язків використовується окремий екземпляр цієї ж структури.

З огляду на багатопоточну природу роботи емулятора, була реалізована проста, але ефективна синхронізація доступу до структур даних. Для цього був створений клас м'ютексу c_simple_mutex, який базується на атомарних операціях процесора та гарантує швидке блокування ресурсів у багатопоточному середовищі. Для зручності використання м'ютексу розроблено також допоміжний клас RAII-обгортки — c_simple_lock_guard, що автоматично звільняє м'ютекс після завершення роботи з ним. Реалізацію цих класів синхронізації наведено в лістингу 3.4.

Лістинг 3.4 — Реалізація класів c_simple_mutex та c_simple_lock_guard

```
class c_simple_mutex {
	volatile char m_val {};
public:
	inline void lock () {
		while (_InterlockedExchange8 (&m_val, 1)) {
			_mm_pause (); }}
	inline void unlock () {
		_InterlockedExchange8 (&m_val, 0); } };
template <typename T>
class c_simple_lock_guard {
	T& m_mutex;
public:
	inline c_simple_lock_guard (T& mutex) : m_mutex { mutex } {
		m_mutex.lock (); }
	inline ~c_simple_lock_guard () {
		m_mutex.unlock (); }};
```

Структура c_page_table_storage стала ефективним рішенням, що дозволяє швидко знаходити та оновлювати інформацію, пов'язану з будь-якою адресою пам'яті, а також забезпечує додаткову функціональність у вигляді зворотного відображення адрес рекомпільованого коду.

3.3 Реалізація обробника виключень

Ключовим компонентом системи динамічної емуляції є механізм перехоплення виключень, який дозволяє отримати повний контроль над виконанням програми під час виникнення виняткових ситуацій. У межах реалізації цієї роботи було створено клас c_x64_exceptions_manager, який відповідає за низькорівневе перехоплення функції обробки виключень KiUserExceptionDispatcher операційної системи Windows. Завдяки цьому перехопленню забезпечується можливість подальшої обробки всіх винятків на високому рівні.

Функція KiUserExceptionDispatcher, розташована у системній бібліотеці ntdll.dll, містить у собі виклик глобального покажчику Wow64PrepareForException на початку функції (див. рисунок 2.3). Якщо цей покажчик не є нульовим, система передає керування за адресою, на яку він вказує.

Функція search_for_callback_ptr виконує аналіз тіла системної функції KiUserExceptionDispatcher, яка міститься в бібліотеці ntdll.dll, з метою знаходження адреси глобального покажчика Wow64PrepareForException. Вона спочатку отримує базову адресу ntdll через GetModuleHandleA, а потім — адресу самої функції через GetProcAddress. Далі функція перевіряє, що початкові байти KiUserExceptionDispatcher відповідають очікуваному шаблону. Якщо перевірка проходить успішно, зчитується зміщення, що використовується в інструкції mov, і на його основі обчислюється фактична адреса покажчика, яку функція й повертає. Це дозволяє динамічно виявити точне розташування Wow64PrepareForException у пам’яті, без використання символів чи фіксованих адрес. Код функції search_for_callback_ptr зображений в лістингу 3.5

Лістинг 3.5 — Реалізація функції для пошуку покажчика

```
std::optional<void**> c_x64_exceptions_manager::search_for_callback_ptr () {
	auto ntdll = GetModuleHandleA ("ntdll.dll");
	if (!ntdll) {
		EMU_LOG_ERROR ("Failed to get ntdll base!");
		return std::nullopt;
	}
```

```
		uint8_t* dispatcher_bytes = reinterpret_cast<uint8_t*>(GetProcAddress (ntdll, "KiUserExceptionDispatcher"));
		if (!dispatcher_bytes || *dispatcher_bytes != 0xFC) { // cld
			EMU_LOG_ERROR ("Invalid KiUserExceptionDispatcher ptr!");
			return std::nullopt;
		}
		dispatcher_bytes++;
		if (!(dispatcher_bytes[0] == 0x48 && dispatcher_bytes[1] == 0x8B && dispatcher_bytes[2] == 0x05)) { // mov rax, qword ptr ds:[rip+...]
			EMU_LOG_ERROR ("Invalid KiUserExceptionDispatcher bytes!");
			return std::nullopt;
		}
		return reinterpret_cast<void**>(dispatcher_bytes + *reinterpret_cast<int*>(dispatcher_bytes + 3) + 7); }
```

Для реалізації перехоплення винятків у системі динамічної емуляції було створено так званий проксі хендлер — функція, яка викликається при виникненні виключення та забезпечує передачу керування на вищий рівень обробки. Щоб передати управління до цієї функції, у системі динамічно створюється шеллкод, який підготовлює необхідний контекст і викликає проксі хендлер з правильними аргументами.

Генерація цього шеллкоду здійснюється у методі build_proxy_shellcode_caller. Під час виконання цей метод формує послідовність інструкцій, які виділяються в пам’яті з правами на виконання. У згенерованому коді спочатку створюється shadow space через інструкцію sub rsp, 0x28, що відповідає вимогам угоди виклику функцій у x64. Далі в регістр r8 записується покажчик на поточний екземпляр класу c_x64_exceptions_manager (тобто this), після чого в rax завантажується адреса функції proxy_handler. Потім відбувається виклик цієї функції через call rax, після чого стек відновлюється інструкцією add rsp, 0x28, і завершення виконання здійснюється через ret. Код функції build_proxy_shellcode_caller зображено в лістингу 3.6.

Лістинг 3.6 — Реалізація функції build_proxy_shellcode_caller

```
void* c_x64_exceptions_manager::build_proxy_shellcode_caller () {
	uint8_t local_shellcode[] {
		0x48, 0x83, 0xEC, 0x28,
		// sub rsp, 0x28
		0x49, 0xB8, 0x88, 0x77, 0x66, 0x55, 0x44, 0x33, 0x22, 0x11, // mov r8, imm64
		0x48, 0xB8, 0x88, 0x77, 0x66, 0x55, 0x44, 0x33, 0x22, 0x11, // mov rax, imm64
```

```
                0xFF, 0xD0,
                        // call rax
                0x48, 0x83, 0xC4, 0x28,
                // add rsp, 0x28
                0xC3
                        // ret
        };
        *(void**)(local_shellcode + 0x6) = this;
        *(void**)(local_shellcode + 0x10) = proxy_handler;
        auto shellcode = VirtualAlloc (NULL, 0x1000, MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
        if (!shellcode) {
                EMU_LOG_ERROR ("Failed to allocate memory for shellcode!");
                return nullptr;
        }
        memcpy (shellcode, local_shellcode, sizeof (local_shellcode));
        return shellcode; }
```

Використання саме регістра r8 обумовлене тим, що згідно з угодою про виклик функцій у Windows x64 [23], перші чотири аргументи передаються в регістрах rcx, rdx, r8, і r9. Функція proxy_handler приймає три аргументи — PEXCEPTION_RECORD, PCONTEXT та c_x64_exceptions_manager*. Перші два регістри (rcx і rdx) займаються системою автоматично при виклику обробника винятку, тому третій аргумент (покажчик на менеджер) передається вручну через r8.

Функція proxy_handler є статичною і слугує посередником між шеллкодом і віртуальним методом handle_exception. Вона викликає handle_exception на переданому екземплярі менеджера виключень, дозволяючи реалізувати конкретну логіку обробки вже у класі-спадкоємці, яким є головний контролер емуляції.

Щоб цей шеллкод почав викликатись замість стандартного обробника, необхідно змінити значення покажчика Wow64PrepareForException, що використовується у функції KiUserExceptionDispatcher. Саме цим займається метод setup. Він викликає search_for_callback_ptr, який за допомогою аналізу байткоду KiUserExceptionDispatcher визначає адресу самого покажчика, після чого змінює його значення на адресу згенерованого шеллкоду. Для цього використовується атомарна операція [24] _InterlockedExchange64, яка гарантує коректну зміну значення навіть у багатопотоковому середовищі. Також, для зміни прав доступу до

пам’яті була використана системна функція VirtualProtect. Код функції setup показано в лістингу 3.7.

Лістинг 3.7 — Реалізація функції setup

```
bool c_x64_exceptions_manager::setup () {
	if (m_initialized) {
		return false;
	}
	auto callback_ptr = search_for_callback_ptr ();
	if (!callback_ptr.has_value ()) {
		EMU_LOG_ERROR ("callback_ptr not found!");
		return false;
	}
	m_callback_ptr = callback_ptr.value ();
	DWORD old {};
	if (!VirtualProtect (m_callback_ptr, sizeof (*m_callback_ptr), PAGE_READWRITE, &old)) {
		EMU_LOG_ERROR ("Failed to protect callback pointer!");
		return false;
	}
	m_proxy_shellcode = build_proxy_shellcode_caller ();
	if (!m_proxy_shellcode) {
		EMU_LOG_ERROR ("Failed to build proxy shellcode caller!");
		VirtualProtect (m_callback_ptr, sizeof (*m_callback_ptr), old, &old);
		return false;}
	m_next_exception_handler = reinterpret_cast<decltype(m_next_exception_handler)>(_InterlockedExchange64 (reinterpret_cast<volatile LONG64*>(m_callback_ptr), reinterpret_cast<LONG64>(m_proxy_shellcode)));
	VirtualProtect (m_callback_ptr, sizeof (*m_callback_ptr), old, &old);
	m_initialized = true;
	return true;}
```

Оригінальне значення покажчика зберігається у полі m_next_exception_handler, щоб згодом, після обробки власного виключення, можна було передати керування системному обробнику, якщо це необхідно. Також, при знищенні об’єкта менеджера, це значення відновлюється, а виділена пам’ять під шеллкод звільняється. Код функції proxy_handler показано в лістингу 3.8.

Лістинг 3.8 — Код функції proxy_handler

```
void c_x64_exceptions_manager::proxy_handler (PEXCEPTION_RECORD record, PCONTEXT context, c_x64_exceptions_manager* manager) {
	manager->handle_exception (record, context);
	if (manager->m_next_exception_handler) {
		manager->m_next_exception_handler (record, context);}}
```

Реалізований механізм дозволяє прозоро вставити свою логіку обробки без втручання в системні викликані механізми, а також інтегрується з загальною архітектурою шляхом делегування обробки до віртуального методу handle_exception.

3.4 Реалізація рекомпілятора інструкцій

3.4.1 Розробка генератора динамічного коду

У рамках системи динамічної емуляції особливу роль відіграє генерація машинного коду безпосередньо під час виконання. Для реалізації цієї задачі був створений клас c_x64_shellcode_builder, який дозволяє зручно формувати 64-бітний x86 код у вигляді послідовності інструкцій, що зберігаються в пам’яті з правами на виконання. Його основне призначення — будування шеллкодів, які можуть бути виконані нативно без проміжної інтерпретації. Під час розробки генератору динамічного коду використовувалась бібліотека Zydis, яка дуже тісно інтегрована в реалізацію багатьох функцій.

Клас базується на концепції послідовного додавання інструкцій у внутрішній буфер m_bytes, при цьому він підтримує два рівні API: базовий низькорівневий рівень і більш високорівневі функції, побудовані на його основі. Низькорівневі методи, такі як current_offset, bytes, qword, dword та word, дозволяють безпосередньо додавати у буфер байти, які відповідають машинним інструкціям. Ці методи є фундаментом для створення будь-якої інструкції. Наприклад, метод mov_reg_qword, який вставляє інструкцію переміщення значення в регістр, реалізовано як комбінацію фіксованих байт для певного регістра і значення, що додається за допомогою qword. Базові низькорівневі методи наведені в лістингу 3.8, реалізація методу mov_reg_qword наведена в лістингу 3.9.

Лістинг 3.8 — Реалізація базових низькорівневих функцій генератора

```
uint64_t c_x64_shellcode_builder::current_offset () {
        return m_bytes.size ();}
c_x64_shellcode_builder& c_x64_shellcode_builder::bytes (const std::vector<uint8_t>& bytes)
{
        push_bytes (bytes.data (), bytes.size ());
        return *this;}
```

```
c_x64_shellcode_builder& c_x64_shellcode_builder::bytes (const void* buffer, size_t size) {
        push_bytes (buffer, size);
        return *this;}
c_x64_shellcode_builder& c_x64_shellcode_builder::qword (uint64_t qword_value) {
        push_bytes (&qword_value, sizeof (qword_value));
        return *this;}
c_x64_shellcode_builder& c_x64_shellcode_builder::dword (uint32_t dword_value) {
        push_bytes (&dword_value, sizeof (dword_value));
        return *this;}
c_x64_shellcode_builder& c_x64_shellcode_builder::word (uint16_t word_value){
        push_bytes (&word_value, sizeof (word_value));
        return *this;}
```

Лістинг 3.9 — Реалізація функції mov_reg_qword

```
c_x64_shellcode_builder& c_x64_shellcode_builder::mov_reg_qword (ZydisRegister reg, uint64_t value) {
        switch (reg) {
                case ZYDIS_REGISTER_RAX: return bytes ({ 0x48, 0xB8 }).qword (value);
                case ZYDIS_REGISTER_RBX: return bytes ({ 0x48, 0xBB }).qword (value);
                case ZYDIS_REGISTER_RCX: return bytes ({ 0x48, 0xB9 }).qword (value);
                case ZYDIS_REGISTER_RDX: return bytes ({ 0x48, 0xBA }).qword (value);
                case ZYDIS_REGISTER_RBP: return bytes ({ 0x48, 0xBD }).qword (value);
                case ZYDIS_REGISTER_RSP: return bytes ({ 0x48, 0xBC }).qword (value);
                case ZYDIS_REGISTER_RSI: return bytes ({ 0x48, 0xBE }).qword (value);
                case ZYDIS_REGISTER_RDI: return bytes ({ 0x48, 0xBF }).qword (value);
                case ZYDIS_REGISTER_R8: return bytes ({ 0x49, 0xB8 }).qword (value);
                case ZYDIS_REGISTER_R9: return bytes ({ 0x49, 0xB9 }).qword (value);
                case ZYDIS_REGISTER_R10: return bytes ({ 0x49, 0xBA }).qword (value);
                case ZYDIS_REGISTER_R11: return bytes ({ 0x49, 0xBB }).qword (value);
                case ZYDIS_REGISTER_R12: return bytes ({ 0x49, 0xBC }).qword (value);
                case ZYDIS_REGISTER_R13: return bytes ({ 0x49, 0xBD }).qword (value);
                case ZYDIS_REGISTER_R14: return bytes ({ 0x49, 0xBE }).qword (value);
                case ZYDIS_REGISTER_R15: return bytes ({ 0x49, 0xBF }).qword (value);
                default:
                {
                        EMU_LOG_ERROR ("Invalid reg value for mov_reg_qword!");
                        utils::abort_execution ();}}return *this;}
```

На основі базових методів реалізовані і більш складні функції, які спрощують формування готових інструкцій. Серед прикладів — jump_ptr_address, який формує послідовність інструкцій для непрямого переходу на адресу, розташовану в пам’яті, а також інструкції з префіксом GS, наприклад store_reg_gs або load_reg_gs, що дозволяють зручно взаємодіяти з потоково-залежним сховищем через GS-сегмент.

Щоб уникнути повторного кодування однакових інструкцій, всі такі інструкції кешуються за допомогою класу c_encoded_instruction_tree. Механізм кешування інструкцій, реалізований у класі c_encoded_instruction_tree, відіграє

важливу роль у підвищенні продуктивності генератора динамічного коду. Його основна ідея полягає в тому, щоб уникати повторного кодування машинних інструкцій, які мають однакові параметри. Кожного разу, коли відбувається спроба сформувати інструкцію, наприклад, для запису значення з регістра у пам’ять або навпаки, система спочатку перевіряє, чи така інструкція вже була колись сформована. Якщо вона вже є в кеші, повторне кодування не відбувається — байти беруться безпосередньо з збереженого результату.

Структурно кеш реалізовано у вигляді дерева, де кожен вузол відповідає певному параметру інструкції — наприклад, регістру, зміщенню або наявності префіксу сегменту. Вузли формують шлях на основі послідовності аргументів, які описують конкретну інструкцію. Наприкінці цього шляху зберігається масив байтів, що представляє собою закодовану інструкцію. Якщо шлях, що відповідає новому запиту, вже існує — результат просто повертається без повторного виклику Zydis.

Приклад використання цього механізму можна побачити у функціях генератора store_reg_gs та load_reg_gs, які були згадані раніше. В них із вхідних параметрів формується ключ, що представляє собою кортеж з регістра, базового регістра адресації, індексу, множника та зміщення. Цей ключ передається до c_encoded_instruction_tree, який перевіряє, чи існує відповідна гілка в дереві. Якщо інструкція була використана раніше, кеш одразу повертає готовий набір байтів. У протилежному випадку викликається внутрішня лямбда-функція, яка формує інструкцію через Zydis, після чого результат зберігається в кеші та стає доступним для майбутніх викликів.

В лістингу 3.10 можна побачити реалізацію функції store_reg_gs, яка використовує c_encoded_instruction_tree для генерації інструкції з допомогою Zydis.

У процесі генерації динамічного коду важливою задачею є підтримка інструкцій із відносною адресацією — таких, які містять зміщення до цільової адреси, обчислене відносно поточного положення інструкції. Це особливо актуально для команд типу jmp, call, lea з використанням rip, а також для умовних

переходів. Проблема полягає в тому, що під час генерації інструкції часто ще невідомо, де саме буде знаходитися ціль переходу або адреса даних.

Лістинг 3.10 — Реалізація функції store_reg_gs

```
c_x64_shellcode_builder& c_x64_shellcode_builder::store_reg_gs (int index, ZydisRegister reg) {
	static c_encoded_instruction_tree cache {};
	auto key = std::make_tuple (index, reg);
	const auto& cached_bytes = cache.get_or_encode (key, [&] () -> std::vector<uint8_t> {
		ZydisEncoderRequest enc_req {};
		memset (&enc_req, 0, sizeof (enc_req));
		enc_req.mnemonic = ZYDIS_MNEMONIC_MOV;
		enc_req.operand_count = 2;
		enc_req.prefixes |= ZYDIS_ATTRIB_HAS_SEGMENT_GS;
		auto& mem_operand = enc_req.operands[0];
		auto& reg_operand = enc_req.operands[1];
		mem_operand.type = ZYDIS_OPERAND_TYPE_MEMORY;
		mem_operand.mem.base = ZYDIS_REGISTER_NONE;
		mem_operand.mem.displacement = GS_STORAGE_VALUE_OFFSET (index);
		mem_operand.mem.index = ZYDIS_REGISTER_NONE;
		mem_operand.mem.scale = 0;
		mem_operand.mem.size = ZydisRegisterGetWidth (ZYDIS_MACHINE_MODE_LONG_64, reg) / 8;
		reg_operand.type = ZYDIS_OPERAND_TYPE_REGISTER;
		reg_operand.reg.value = reg;
		uint8_t new_bytes[ZYDIS_MAX_INSTRUCTION_LENGTH];
		ZyanUSize new_instr_length = sizeof (new_bytes);
		if (!ZYAN_SUCCESS (ZydisEncoderEncodeInstruction (&enc_req, new_bytes, &new_instr_length))) {
			EMU_LOG_ERROR ("Failed to encode instruction!");
			utils::abort_execution ();
		}
		return std::vector<uint8_t> (new_bytes, new_bytes + new_instr_length);
	});
	return bytes (cached_bytes.data (), cached_bytes.size ());}
```

Щоб вирішити цю задачу, в системі реалізовано два допоміжні механізми — структури prebuilt_relative_insn_info_t та relative_insn_t.

Структура prebuilt_relative_insn_info_t зберігає інформацію про те, де саме в байтах інструкції знаходиться поле зі зміщенням і скільки байтів це поле займає. Наприклад, для стандартного 32-бітного відносного переходу значення зміщення розміщується на певному зсуві від початку інструкції, і саме ці два параметри — зміщення та розмір — зберігаються у цій структурі. Це дозволяє у подальшому

легко обчислити правильне значення зміщення, коли стане відома адреса цільового блоку коду.

У свою чергу, структура relative_insn_t є обгорткою навколо вже згенерованої, але ще не "залінкованої" інструкції. Вона зберігає вказівник на об'єкт генератора c_x64_shellcode_builder, позицію інструкції у буфері, її повну довжину, а також екземпляр prebuilt_relative_insn_info_t, який описує розташування зміщення. Коли адреса призначення стає відомою, викликається метод link, який виконує обчислення дійсного зміщення (враховуючи довжину інструкції) і записує його прямо у байти, що зберігаються у внутрішньому буфері генератора.

В лістингу 3.11 зображено реалізацію методу link, який використовується для зв’язування раніше згенерованої інструкції з її фактичною ціллю.

Лістинг 3.11 — Реалізація методу link

```
void c_x64_shellcode_builder::relative_insn_t::link (int64_t destination_offset) {
        switch (m_prebuilt_info.m_value_size) {
                case 4: *(int32_t*)(m_builder->m_bytes.data () + m_insn_offset + m_prebuilt_info.m_value_offset) = int32_t (destination_offset - m_insn_offset - m_insn_size); break;
                case 2: *(int16_t*)(m_builder->m_bytes.data () + m_insn_offset + m_prebuilt_info.m_value_offset) = int16_t (destination_offset - m_insn_offset - m_insn_size); break;
                case 1: *(int8_t*)(m_builder->m_bytes.data () + m_insn_offset + m_prebuilt_info.m_value_offset) = int8_t (destination_offset - m_insn_offset - m_insn_size); break;
                default:
                {
                        EMU_LOG_ERROR ("Invalid relative value size!");
                        utils::abort_execution ();}}}
```

В лістингу 3.12 показано вигляд структур prebuilt_relative_insn_info_t та relative_insn_t.

Лістинг 3.12 — Реалізація структур prebuilt_relative_insn_info_t та relative_insn_t

```
struct prebuilt_relative_insn_info_t {
   int64_t m_value_offset {};
   int m_value_size {};
};
struct relative_insn_t {
   c_x64_shellcode_builder* m_builder {};
   int64_t m_insn_offset {};
   prebuilt_relative_insn_info_t m_prebuilt_info {};
   int m_insn_size {};
   void link (int64_t destination_offset); };
```

Практичне використання цього механізму можна побачити у таких методах генератора коду, як conditional_jump або lea_rax_rip. Наприклад, метод conditional_jump формує умовний перехід із шаблонним зміщенням (заповненим нулями) і одразу повертає relative_insn_t, який зберігає всю потрібну інформацію для подальшого патчингу. Коли в подальшому стає відомою ціль переходу, користувач викликає link, передаючи туди фінальний зсув у байтах. В лістингу 3.13 зображено реалізацію метода conditional_jump. Видно, що спачатку створюється статична карта всіх можливих умовних переходів, яка буде використовуватись при генерації.

Лістинг 3.13 — Реалізація функції conditional_jump

```
c_x64_shellcode_builder::relative_insn_t c_x64_shellcode_builder::conditional_jump (ZydisMnemonic mnemonic) {
    static std::map<int, std::pair<std::vector<uint8_t>, prebuilt_relative_insn_info_t>> mnemonic_to_prebuilt_insn {
        { ZYDIS_MNEMONIC_JB, { { 0x0F, 0x82, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
        { ZYDIS_MNEMONIC_JBE, { { 0x0F, 0x86, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
        { ZYDIS_MNEMONIC_JL, { { 0x0F, 0x8C, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
        { ZYDIS_MNEMONIC_JLE, { { 0x0F, 0x8E, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
        { ZYDIS_MNEMONIC_JNB, { { 0x0F, 0x83, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
        { ZYDIS_MNEMONIC_JNBE, { { 0x0F, 0x87, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
        { ZYDIS_MNEMONIC_JNL, { { 0x0F, 0x8D, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
        { ZYDIS_MNEMONIC_JNLE, { { 0x0F, 0x8F, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
        { ZYDIS_MNEMONIC_JNO, { { 0x0F, 0x81, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
        { ZYDIS_MNEMONIC_JNP, { { 0x0F, 0x8B, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
        { ZYDIS_MNEMONIC_JNS, { { 0x0F, 0x89, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
        { ZYDIS_MNEMONIC_JNZ, { { 0x0F, 0x85, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
        { ZYDIS_MNEMONIC_JO, { { 0x0F, 0x80, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
        { ZYDIS_MNEMONIC_JP, { { 0x0F, 0x8A, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
        { ZYDIS_MNEMONIC_JS, { { 0x0F, 0x88, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } } },
```

```
                { ZYDIS_MNEMONIC_JZ, { { 0x0F, 0x84, 0x00, 0x00, 0x00, 0x00 }, { 2, 4 } }
},
                { ZYDIS_MNEMONIC_JCXZ, { { 0xE3, 0x00 }, { 1, 1 } } },
                { ZYDIS_MNEMONIC_JECXZ, { { 0x67, 0xE3, 0x00 }, { 2, 1 } } },
                { ZYDIS_MNEMONIC_JRCXZ, { { 0xE3, 0x00 }, { 1, 1 } } },
                { ZYDIS_MNEMONIC_JMP, { { 0xE9, 0x00, 0x00, 0x00, 0x00 }, { 1, 4 } } },
        };
        auto it = mnemonic_to_prebuilt_insn.find (mnemonic);
        if (it == mnemonic_to_prebuilt_insn.end ()) {
                EMU_LOG_ERROR ("Unknown mnemonic for conditional jump: %d",
mnemonic);
                utils::abort_execution ();
        }
        auto current_offset_value = current_offset ();
        bytes (it->second.first);
        return { this, int64_t (current_offset_value), it->second.second, int (it->second.first.size
()) };}
```

3.4.2 Організація потоково-залежного сховища даних

Як вже зазначалося раніше, для реалізації динамічної емуляції критично важливою є наявність окремого сховища службових даних, яке буде унікальним для кожного потоку виконання. Це сховище дозволяє зберігати важливу інформацію, пов’язану з рекомпільованим кодом, не конфліктуючи при цьому з оригінальним кодом чи стеком потоку. Основною особливістю реалізації є те, що доступ до цих даних здійснюється через сегментний регістр gs, а самі дані зберігаються в Thread Information Block — TIB. Завдяки архітектурним властивостям Windows x64, доступ до довільного поля TIB можна здійснити за відомим зсувом, використовуючи команду з префіксом gs. Цим зсувом і є результат множення індексу в сховищі на вісім та додавання до базового значення 0x110.

У системі реалізовано перелік іменованих індексів e_gs_indexes, кожен з яких відповідає за певне поле в сховищі. Усі ці індекси використовуються або для збереження стану під час роботи зворотніх функцій, або для підтримки службової логіки, або для забезпечення тимчасової незалежної пам’яті у межах одного потоку. Список індексів показано в лістингу 3.14.

Лістинг 3.14 — Список всіх можливих індексів потокового сховища

```
enum e_gs_indexes : int {
    gs_indexes_begin,
    gs_index_emu_begin = gs_indexes_begin,
```

```
gs_index_emu_end,
gs_index_global_table,
gs_index_next_rip,
gs_index_called_from,
gs_index_call_destination,
gs_index_callbacks_queue,
gs_index_callbacks_queue_size,
gs_index_callback_cpu_context,
gs_index_callback_stack,
gs_index_memory_pages_table,
gs_index_memory_instruction_info,
gs_index_memory_access_address,
gs_index_memory_access_size,
gs_index_memory_access_flags,
gs_index_backup_internal,
gs_index_backup_rflags,
gs_index_backup_rel,
gs_index_backup_gp_register_1,
gs_index_backup_gp_register_2,
gs_index_backup_gp_register_3,
gs_index_memory_operand_value,
gs_index_temp_rip_for_jump,
gs_indexes_end};
```

Весь перелік розпочинається з умовних маркерів gs_index_emu_begin і gs_index_emu_end, які визначають межі всієї області, що використовується системою динамічної емуляції. Ці значення, хоча і не беруть участі безпосередньо в генерації коду, можуть бути використані для перевірки, чи адреса належить до простору керування шеллкод-білдером.

Індекс gs_index_global_table використовується для зберігання покажчика на глобальну таблицю рекомпільованих адрес, яка виступає як ключовий елемент оптимізації всієї системи. Ця таблиця являє собою лінійний масив, у якому зберігається відповідність між оригінальними адресами інструкцій і адресами вже згенерованих рекомпільованих блоків. Коли система намагається рекомпілювати нову інструкцію, вона спочатку звертається до цієї таблиці і перевіряє, чи існує вже для неї відповідний скомпільований код. Якщо такий запис знайдено — повторна рекомпіляція не виконується, що дозволяє значно зекономити ресурси і уникнути зайвих обчислень та обробки винятків. Уся ця логіка детальніше описується в підрозділах, присвячених умовним переходам та організації кешу рекомпільованих блоків.

Індекс gs_index_next_rip використовується при рекомпіляції RIP-залежних інструкцій, зокрема ret, jmp та call. У таких випадках виникає необхідність зберегти адресу наступної інструкції, яка має виконуватись після завершення обробки поточної. Ця адреса визначається під час рекомпіляції і заноситься у потоково-залежне сховище за допомогою цього індексу. Подальше виконання, в тому числі умовне або безумовне перенаправлення потоку, відбувається з урахуванням цієї збереженої інформації, що дозволяє дотриматись логіки оригінального коду.

Індекси gs_index_called_from та gs_index_call_destination застосовуються у випадках, коли необхідно організувати виклик динамічного шеллкоду безпосередньо з рекомпільованого коду. У таких сценаріях шеллкод, який генерується, зберігає адресу місця виклику (тобто звідки було здійснено перехід) у полі gs_index_called_from, а потім передає адресу самого цільового шеллкоду в поле gs_index_call_destination. Після цього виконується безумовний перехід на нову адресу. Таке розділення дозволяє у подальшому завершити виконання динамічного шеллкоду з поверненням до точки виклику, використовуючи збережену адресу. Детальний опис самого механізму динамічних шеллкодів буде представлений у наступному розділі, присвяченому розширенню функціональності системи емуляції.

Особливу групу індексів становлять gs_index_callbacks_queue та gs_index_callbacks_queue_size, які відповідають за чергу зворотних викликів. Вони дають змогу накопичувати та обробляти зворотні функції перед фактичним виконанням інструкції. При цьому перший індекс зберігає адресу самої черги, а другий — її поточний розмір. За цими полями йде група індексів, яка безпосередньо пов’язана з виконанням колбеків: gs_index_callback_cpu_context зберігає копію контексту процесора для передачі в обробник, а gs_index_callback_stack використовується як тимчасовий стек під час виконання коду зворотного виклику.

Індекс gs_index_memory_pages_table зберігає адресу спеціалізованої таблиці сторінок пам’яті, яка застосовується при роботі з коллбеками доступу до пам’яті. Ця таблиця реалізована як ієрархічна структура, що дозволяє швидко визначити, чи належить певна адреса або сторінка до області, за якою зареєстрований коллбек.

Зокрема, під час рекомпіляції інструкцій, які виконують пам’яттєві операції, система звертається до цієї таблиці, виконує бітові операції над адресою, щоб вирахувати відповідний індекс, і перевіряє наявність ознаки коллбеку. Якщо вона знайдена — генерується додатковий код для збору інформації та виклику відповідного зворотного обробника. Завдяки цьому підхід до обробки пам’яттєвих доступів є масштабованим, що важливо при моніторингу великих областей пам’яті без втрати швидкодії.

Індекси gs_index_memory_instruction_info, gs_index_memory_access_address, gs_index_memory_access_size та gs_index_memory_access_flags зберігають відповідно мета-дані про інструкцію, що виконується, адресу пам’яті, до якої здійснюється доступ, її розмір та права доступу (читання, запис тощо).

У системі передбачено також індекси, які використовуються лише у внутрішніх технічних цілях. Це gs_index_backup_internal, gs_index_backup_rflags, gs_index_backup_rel, а також gs_index_backup_gp_register_1, gs_index_backup_gp_register_2, gs_index_backup_gp_register_3 та gs_index_temp_rip_for_jump. Вони забезпечують тимчасове збереження регістрів, прапорців або інших значень, які потрібно зберегти перед виконанням певних трансформацій коду і відновити пізніше.

3.4.3 Реалізація рекомпіляції блоку інструкцій

Основною функцією системи динамічної емуляції є перекомпіляція коду, що реалізована у функції get_recompile_code. Ця функція є ключовою частиною архітектури системи емуляції, оскільки вона здійснює трансляцію інструкцій із оригінального коду в модифікований еквівалент, здатний виконуватись з додатковим контролем.

Процес рекомпіляції починається з перевірки, чи знаходиться вхідна адреса у допустимому діапазоні емуляції. Далі накладається взаємовиключення на глобальну таблицю рекомпільованих адрес для забезпечення потокобезпечності. Перш ніж розпочинати нову рекомпіляцію, виконується перевірка, чи для цієї адреси вже існує готовий варіант рекомпільованого коду. Якщо він є — функція

одразу повертає відповідний покажчик, і нова рекомпіляція не виконується. Це реалізує описану раніше ідею кешування згенерованих блоків. Початкок функції, що реалізує цю логіку, показано в лістингу 3.15.

ЛІстинг 3.15 — Початок функціїї get_recompile_code

```
uint64_t c_x64_code_emulator::get_recompile_code (uint64_t address, bool& executed_first_time) {
	if (address < m_emu_range_start || address >= (m_emu_range_start + m_emu_range_size)) {
		EMU_LOG_ERROR ("Address is out of range!");
		return 0;
	}

	const c_simple_lock_guard lock (m_global_table_mutex);

	auto recompiled_ptr = get_recompiled_entry_with_ptr (address);

	if (recompiled_ptr) {
		executed_first_time = false;
		return *recompiled_ptr;
	}
```

У випадку, якщо інструкція ще не була рекомпільована, створюється об'єкт будівника шеллкоду та ініціалізується декодер інструкцій бібліотеки Zydis. Основна робота відбувається у циклі, який обробляє інструкції одну за одною, починаючи з початкової адреси. Кожна інструкція декодується, її тип визначається, і в залежності від цього викликається відповідна функція обробки: для RIP-залежних інструкцій — handle_rip, для REL-адресацій — handle_rel, для всіх інших — handle_default. Початок циклу показаний в лістингу 3.16.

Лістинг 3.16 — Початок циклу обробки інструкцій

```
c_x64_shellcode_builder recompiled_code_builder {};
std::map<uint64_t, uint64_t> internal_offset_to_recompiled_offset {};
std::map<uint64_t, std::pair<uint64_t, size_t>> internal_offset_to_recompiled_offset_and_size {};
ZydisDecoder decoder;
ZydisDecoderInit (&decoder, ZYDIS_MACHINE_MODE_LONG_64, ZYDIS_STACK_WIDTH_64);
uint64_t current_internal_offset = 0;
ZydisDecodedInstruction instruction {};
ZydisDecodedOperand operands[ZYDIS_MAX_OPERAND_COUNT] {};
while (true) {
	uint64_t current_address = address + current_internal_offset;
```

```
uint64_t begin_recompiled_instruction_offset = recompiled_code_builder.current_offset
();
if (current_internal_offset) {
	auto recompiled_ptr = get_recompiled_entry_with_ptr (current_address);
	if (recompiled_ptr) {
		recompiled_code_builder.jump_ptr_address (recompiled_ptr);
		break;
	}
	internal_offset_to_recompiled_offset[current_internal_offset] =
begin_recompiled_instruction_offset;
}
if (!ZYAN_SUCCESS (ZydisDecoderDecodeFull (&decoder, (void*)current_address,
ZYDIS_MAX_INSTRUCTION_LENGTH, &instruction, operands))) {
	EMU_LOG_ERROR ("Failed to decode instruction at %llx", current_address);
	return 0;}
```

Механізм, що відповідає за попередньо описаний концепцт поділу інструкцій на три категорії, показаний в лістингу 3.17. Важливо зазначити, що до обробки інструкції перевіряється ще одна важлива умова — чи була вона вже раніше рекомпільована окремо (див. лістинг 3.16).

Лістинг 3.17 — Код, відповідальний за обробку різних типів інструкцій

```
bool instruction_processed = false;
bool should_stop = false;
if (!instruction_processed) { // RIP instructions
	for (int i = 0; i < instruction.operand_count; ++i) {
		auto& operand = operands[i];
		if (operand.type == ZYDIS_OPERAND_TYPE_REGISTER &&
operand.reg.value == ZYDIS_REGISTER_RIP) {
			if (!handle_rip (current_address, instruction, operands,
recompiled_code_builder, should_stop, begin_recompiled_instruction_offset)) {
				EMU_LOG_ERROR ("Failed to handle RIP instruction at %llx",
current_address);
				return 0;
			}
			instruction_processed = true;
			break;}}}
if (!instruction_processed) { // REL instructions
	for (int i = 0; i < instruction.operand_count; ++i) {
		auto& operand = operands[i];
		if (instruction.attributes & ZYDIS_ATTRIB_IS_RELATIVE || (operand.type ==
ZYDIS_OPERAND_TYPE_MEMORY && operand.mem.base == ZYDIS_REGISTER_RIP)) {
			if (!handle_rel (current_address, instruction, operands,
recompiled_code_builder)) {
				EMU_LOG_ERROR ("Failed to handle REL instruction at %llx",
current_address);
				return 0;
			}
			instruction_processed = true;break;}}}
```

```
if (!instruction_processed) {
        if (!handle_default (current_address, instruction, operands, recompiled_code_builder, begin_recompiled_instruction_offset)) {
                EMU_LOG_ERROR ("Failed to handle DEFAULT instruction at %llx", current_address);
                return 0;}

        instruction_processed = true;}
```

Якщо так, то замість повторної генерації коду, до поточного буфера вставляється перехід на вже згенерований код, що дозволяє уникати дублювання роботи. Також у структурі даних фіксується відповідність між внутрішнім зсувом відносно початкової адреси та положенням у буфері скомпільованого коду. Це дозволяє у подальшому вставити мапінг назад на оригінальну адресу, що використовується для налагодження чи додаткового аналізу виконання коду.

Після обробки інструкції, функція перевіряє змінну should_stop, яка може бути встановлена у випадку, коли була зустрінута інструкція, що змінює хід виконання (наприклад, безумовний перехід або повернення). У цьому випадку цикл завершується, адже наступні інструкції не гарантують виконання і не можуть бути оброблені з достатньою точністю.

Завершальний етап роботи функції полягає у виклику методу build, який будує фінальний код, розміщує його в пам’яті та повертає вказівник на нього. Одразу після цього формується таблиця відповідностей між адресами в скомпільованому коді та їх оригінальними аналогами. Також додаються додаткові записи у глобальну таблицю рекомпільованих блоків, що дозволяє у майбутньому уникати повторного аналізу вже оброблених фрагментів. Ці два процеси показані в лістингу 3.18. Блок-схема алгоритму функції зображена в додатку В.

Лістинг 3.18 — Код формування таблиці відповідності з оригінальним кодом та доповнення глобальної таблиці

```
auto recompiled = recompiled_code_builder.build ();
if (!recompiled) {
        EMU_LOG_ERROR ("Failed to recompile code for %llx", address);
        return 0;
}
for (auto& entry : internal_offset_to_recompiled_offset_and_size) {
        m_recompiled_info_table.insert_values (uint64_t (recompiled + entry.second.first), entry.second.second, address + entry.first);}
```

```
auto result = insert_recompiled_entry (address, recompiled);
for (auto& entry : internal_offset_to_recompiled_offset) {
        insert_recompiled_entry (address + entry.first, recompiled + entry.second);}
```

3.4.4 Реалізація рекомпіляції RIP-залежних інструкцій

Функція handle_rip відповідає за обробку інструкцій, що прямо або опосередковано впливають на регістр RIP, і є одним із ключових компонентів динамічного рекомпілятора. Її завдання полягає у правильному формуванні нового коду для таких інструкцій, з урахуванням можливості переходів, викликів, повернень тощо. Від типу інструкції та її операндів залежить те, який саме код буде згенерований, і якою буде подальша логіка виконання.

Для інструкції ret функція створює код, який зчитує адресу повернення зі стеку, тимчасово зберігає її у регістрі rax, після чого значення переноситься у потоково-залежне сховище за індексом gs_index_next_rip. Після цього викликається функція optimize_rip, яка може оптимізувати це значення. Завершується цей фрагмент емульованою модифікацією стеку, відповідно до значення, заданого у самій інструкції (включаючи можливий оффсет), та безумовним переходом за збереженою адресою. Реалізація обробки інструкції повернення показана в лістингу 3.19.

При обробці інструкцій jmp та call виконується окрема гілка, яка залежить від типу операнда. Якщо ціль вказується через регістр, його значення зберігається у gs_index_next_rip. Якщо операнд — це безпосереднє значення або адресація через пам’ять, то спочатку обчислюється абсолютна адреса. У випадку, коли ця адреса знаходиться за межами області емуляції, вона просто зберігається як є. Якщо ж вона належить до області, яку обслуговує рекомпілятор, то здійснюється спроба отримати вже рекомпільований блок через глобальну таблицю.

Лістинг 3.19 — Код для обробки інструкції повернення

```
case ZYDIS_MNEMONIC_RET:{
        ZyanI64 additional_offset = 0;
        if (instruction.operand_count_visible > 0) {
                if (operands->type != ZYDIS_OPERAND_TYPE_IMMEDIATE) {
                        EMU_LOG_ERROR ("Invalid ret at %llx", runtime_address);
                        return false;}
```

```
                additional_offset = operands->imm.value.s;
        }
        builder.store_reg_gs (gs_index_next_rip, ZYDIS_REGISTER_RAX)
                .mov_reg_mem (ZYDIS_REGISTER_RAX, ZYDIS_REGISTER_RSP)
                .xchg_reg_gs (gs_index_next_rip, ZYDIS_REGISTER_RAX);
        check_emulation_exit (builder, runtime_address, instruction, operands);
        optimize_rip (builder);
        builder.lea_reg_mem (ZYDIS_REGISTER_RSP, ZYDIS_REGISTER_RSP, 8 +
additional_offset)
                .jump_gs (gs_index_next_rip);
        return true;}
```

Якщо такий блок існує, зберігається його адреса, якщо ні — зберігається оригінальна адреса. У всіх варіантах, окрім одного специфічного випадку, після обробки викликається optimize_rip. У разі, якщо мова йде про call, додатково в стек записується адреса повернення (тобто адреса інструкції, яка йде слідом за call). Наприкінці генерується безумовний перехід за оптимізованою адресою. Код для обробки інструкцій call та jmp показані в лістингу 3.20.

Умовні переходи (jcc) мають окрему логіку, яка дозволяє забезпечити дві можливі гілки виконання залежно від виконання умови. Для цього спочатку обчислюється абсолютна адреса переходу, а потім створюється умовна конструкція: перша частина виконується у разі, якщо умова не виконалась, друга — якщо виконалась. У кожній з гілок відбувається спроба звернення до глобальної таблиці: якщо знайдено відповідний рекомпільований блок, переходить до нього, інакше — до оригінальної адреси. Таким чином, забезпечується коректне та гнучке виконання умовного переходу без необхідності повторної рекомпіляції. Реалізація коду для обробки jcc інструкцій показана в лістингу 3.21. Окремо варто зупинитися на функції optimize_rip, яка викликається у більшості випадків після обчислення значення gs_index_next_rip.

Лістинг 3.20 — Код для обробки інструкцій call та jmp

```
case ZYDIS_MNEMONIC_CALL:
case ZYDIS_MNEMONIC_JMP:{
        bool is_call = instruction.mnemonic == ZYDIS_MNEMONIC_CALL;
        bool should_optimize_rip = true;
        switch (operands->type) {
                case ZYDIS_OPERAND_TYPE_REGISTER:{
                        builder.store_reg_gs (gs_index_next_rip, operands->reg.value);break;}
                case ZYDIS_OPERAND_TYPE_IMMEDIATE:{
```

```
                                auto abs_address = calc_absolute_address (operands);
                                if (!abs_address) {
                                        EMU_LOG_ERROR ("Failed to get abs address for %llx!", runtime_address);
                                        return 0;}
                                if (abs_address < m_emu_range_start || abs_address >= (m_emu_range_start + m_emu_range_size)) {
                                        builder.store_qword_gs (gs_index_next_rip, abs_address);
                                        break;}
                                auto global_entry_ptr = get_recompiled_entry_ptr (abs_address);
                                if (!global_entry_ptr) {
                                        EMU_LOG_ERROR ("Failed to get destination global entry for %llx!", runtime_address);
                                        return 0;}
                                builder.store_ptr_gs (gs_index_next_rip, uint64_t (global_entry_ptr));
                                should_optimize_rip = false;
                                break;}
                        case ZYDIS_OPERAND_TYPE_MEMORY:{
                                if (operands->mem.base != ZYDIS_REGISTER_RIP) {
                                        builder.store_mem_gs (gs_index_next_rip, operands->mem.base, operands->mem.disp.has_displacement ? operands->mem.disp.value : 0, operands->mem.index, operands->mem.scale);
                                        break;}
                                auto abs_address = calc_absolute_address (operands);
                                if (!abs_address) {
                                        EMU_LOG_ERROR ("Failed to get abs address for %llx!", runtime_address);
                                        return 0;}
                                builder.store_ptr_gs (gs_index_next_rip, abs_address);
                                break;}
                        default:
                        {
                                EMU_LOG_ERROR ("Invalid operand type for CALL/JMP at %llx", runtime_address);
                                return false;}}
                if (should_optimize_rip) {
                        check_emulation_exit (builder, runtime_address, instruction, operands);
                        optimize_rip (builder);
                }
                if (is_call) {
                        builder.push_qword (runtime_address + instruction.length);
                }
                builder.jump_gs (gs_index_next_rip);
                return true;}
```

Лістинг 3.21 — Реалізація коду для обробки jcc інструкцій

```
case ZYDIS_MNEMONIC_JB:
case ZYDIS_MNEMONIC_JBE:
case ZYDIS_MNEMONIC_JCXZ:
case ZYDIS_MNEMONIC_JECXZ:
case ZYDIS_MNEMONIC_JL:
```

```
        case ZYDIS_MNEMONIC_JLE:
        case ZYDIS_MNEMONIC_JNB:
        case ZYDIS_MNEMONIC_JNBE:
        case ZYDIS_MNEMONIC_JNL:
        case ZYDIS_MNEMONIC_JNLE:
        case ZYDIS_MNEMONIC_JNO:
        case ZYDIS_MNEMONIC_JNP:
        case ZYDIS_MNEMONIC_JNS:
        case ZYDIS_MNEMONIC_JNZ:
        case ZYDIS_MNEMONIC_JO:
        case ZYDIS_MNEMONIC_JP:
        case ZYDIS_MNEMONIC_JRCXZ:
        case ZYDIS_MNEMONIC_JS:
        case ZYDIS_MNEMONIC_JZ:{
                auto abs_address = calc_absolute_address (operands);
                if (!abs_address) {
                        EMU_LOG_ERROR ("Failed to get abs address for %llx!", runtime_address);
                        return 0;}
                auto global_entry_default = get_recompiled_entry_ptr (runtime_address +
instruction.length);
                auto jump = builder.conditional_jump (instruction.mnemonic);
                if (global_entry_default) {
                        builder.store_ptr_gs (gs_index_next_rip, uint64_t (global_entry_default));
                } else {
                        builder.store_qword_gs (gs_index_next_rip, runtime_address +
instruction.length);
                        check_emulation_exit (builder, runtime_address, instruction, operands);}
                builder.jump_gs (gs_index_next_rip);
                jump.link (builder.current_offset ());
                auto global_entry_on_jump = get_recompiled_entry_ptr (abs_address);
                if (global_entry_on_jump) {
                        builder.store_ptr_gs (gs_index_next_rip, uint64_t (global_entry_on_jump));
                } else {
                        builder.store_qword_gs (gs_index_next_rip, abs_address);
                        check_emulation_exit (builder, runtime_address, instruction, operands);}
                builder.jump_gs (gs_index_next_rip);
                return true;}
```

Її задача полягає в оптимізації цього значення перед переходом. Конкретно, вона додає до коду логіку, яка перевіряє, чи знаходиться адреса у межах області емульованого коду. Якщо так — виконується доступ до глобальної таблиці відповідності між оригінальними адресами та рекомпільованими. Замість того, щоб виконуватись за оригінальною адресою, програма отримає можливість одразу перейти до вже згенерованого фрагмента коду. Якщо відповідного запису у таблиці немає, то значення залишиться без змін. Саме завдяки цьому підходу зменшується

кількість повторних рекомпіляцій і підвищується загальна ефективність емулятора. Реалізація функції optimize_rip показана в лістингу 3.22.

Лістинг 3.22 — Реалізація функції optimize_rip

```
void c_x64_code_emulator::optimize_rip (c_x64_shellcode_builder& builder) {
        auto backup_state = dynamic_shellcode_begin (builder, true, { ZYDIS_REGISTER_RAX, ZYDIS_REGISTER_RBX });{
                builder.load_reg_gs (ZYDIS_REGISTER_RAX, gs_index_next_rip)
                        .cmp_reg_gs (ZYDIS_REGISTER_RAX, gs_index_emu_end);
                auto jge = builder.conditional_jump (ZYDIS_MNEMONIC_JNL);
                builder.cmp_reg_gs (ZYDIS_REGISTER_RAX, gs_index_emu_begin);
                auto jl = builder.conditional_jump (ZYDIS_MNEMONIC_JL);
                builder.sub_reg_gs (ZYDIS_REGISTER_RAX, gs_index_emu_begin)
                        .load_reg_gs (ZYDIS_REGISTER_RBX, gs_index_global_table)
                        .lea_reg_mem (ZYDIS_REGISTER_RAX, ZYDIS_REGISTER_RBX, 0, ZYDIS_REGISTER_RAX, 8)
                        .bytes ({ 0x48, 0x8B, 0x00 }) // mov rax, qword ptr ds:[rax]
                        .store_reg_gs (gs_index_next_rip, ZYDIS_REGISTER_RAX);
                jge.link (builder.current_offset ());
                jl.link (builder.current_offset ());}
        dynamic_shellcode_end (builder, backup_state, false);}
```

3.4.5 Реалізація рекомпіляції інструкцій з відносною адресацією

Функція handle_rel реалізує логіку обробки інструкцій з відносною адресацією (REL), які базуються на зміщенні від регістра RIP. Такий тип інструкцій часто зустрічається у x64 архітектурі, зокрема в командах доступу до пам’яті з використанням поточного положення виконання. Ці інструкції, хоча й не змінюють значення RIP напряму, все ж залежать від нього, і тому потребують спеціального підходу при трансформації.

На початку функції реалізовано пошук операнду, який використовує базовий регістр RIP, що дозволяє виявити саме ту частину інструкції, яка містить відносне зміщення. За допомогою бібліотеки Zydis обчислюється абсолютна адреса, на яку буде звертатися інструкція під час виконання. Якщо така адреса не може бути обчислена, функція завершується з помилкою, оскільки це унеможливлює коректну реконструкцію поведінки оригінального коду. Початок функції з реалізацію проходження по операндам показано в лістингу 3.23.

Лістинг 3.23 — Початок функції handle_rel

```
bool c_x64_code_emulator::handle_rel (uint64_t runtime_address, ZydisDecodedInstruction& instruction, ZydisDecodedOperand* operands, c_x64_shellcode_builder& builder) {
	auto calc_abs_address = [&] ()->std::optional<uint64_t> {
		for (size_t i = 0; i < instruction.operand_count_visible; ++i) {
			auto& operand = operands[i];
			if (operand.type == ZYDIS_OPERAND_TYPE_MEMORY && operand.mem.base == ZYDIS_REGISTER_RIP) {
				ZyanU64 abs_address = 0;
				if (!ZYAN_SUCCESS (ZydisCalcAbsoluteAddress (&instruction, &operand, runtime_address, &abs_address))) {
					EMU_LOG_ERROR ("Failed to calculate absolute address for REL instruction at %llx", runtime_address);
					return std::nullopt;}
				return abs_address;}}
		return std::nullopt;};
	auto abs_address = calc_abs_address ();
	if (!abs_address.has_value ()) {
		EMU_LOG_ERROR ("Failed to get abs address for REL instruction at %llx", runtime_address);
		return false;}
```

Далі реалізація розгалужується залежно від типу інструкції. Якщо це інструкція lea, тобто обчислення ефективної адреси, то її семантика дозволяє просто замінити її на mov з абсолютним значенням. Такий варіант значно спрощує трансформацію, адже не потребує збереження або модифікації контексту. Якщо ж мова йде про будь-яку іншу інструкцію з відносною адресацією, процес стає складнішим. Код для обробки lea інструкції показано в лістингу 3.24.

Лістинг 3.24 — Код для обробки інструкції lea

```
if (instruction.mnemonic == ZYDIS_MNEMONIC_LEA) {
	if (operands->type != ZYDIS_OPERAND_TYPE_REGISTER) {
		EMU_LOG_ERROR ("Failed to resolve LEA instruction at %llx", runtime_address);
		return false;}
	builder.mov_reg_qword (operands->reg.value, abs_address.value ());}
```

Для таких загальних випадків функція знаходить вільний регістр, який не використовується поточною інструкцією. Цей регістр буде тимчасово використано для заміни RIP у структурі інструкції. З оригінальної інструкції будується новий запит на кодування, і в цьому запиті відповідний операнд змінюється: замість RIP встановлюється обраний тимчасовий регістр, зміщення обнуляється, а всі індексні параметри видаляються. Таким чином, інструкція втрачає свою залежність від

положення у коді, і стає позиційно незалежною. Після цього формується нова інструкція в байтах і додається до згенерованого коду. Перед тим як завантажити абсолютну адресу в тимчасовий регістр, його значення зберігається у спеціальному полі потоково-залежного сховища. Це гарантує, що по завершенню виконання трансформованої інструкції регістр буде відновлено до початкового стану, що дозволяє уникнути конфліктів з оригінальною логікою програми. Реалізація трансформації REL інструкції показано в лістингу 3.25.

Лістинг 3.25 — Реалізація трансформації REL інструкції

```
auto unused_register = get_first_unused_register (instruction, operands);
if (unused_register == ZYDIS_REGISTER_NONE) {
	EMU_LOG_ERROR ("Failed to get unused register for REL instruction at %llx", runtime_address);
	return false;}
ZydisEncoderRequest enc_req {};
if (!ZYAN_SUCCESS (ZydisEncoderDecodedInstructionToEncoderRequest (&instruction, operands, instruction.operand_count_visible, &enc_req))) {
	EMU_LOG_ERROR ("Failed to convert instruction to encoder request for REL instruction at %llx", runtime_address);
	return false;}
bool operand_replaced = false;
for (int i = 0; i < enc_req.operand_count; ++i) {
	auto& operand = enc_req.operands[i];
	if (operand.type == ZYDIS_OPERAND_TYPE_MEMORY && operand.mem.base == ZYDIS_REGISTER_RIP) {
		operand.mem.base = unused_register;
		operand.mem.displacement = 0;
		operand.mem.index = ZYDIS_REGISTER_NONE;
		operand.mem.scale = 0;
		operand_replaced = true;
		break;}}
if (!operand_replaced) {
	EMU_LOG_ERROR ("Failed to replace RIP operand for RIP-relative instruction at %llx", runtime_address);
	return false;}
uint8_t new_bytes[ZYDIS_MAX_INSTRUCTION_LENGTH];
ZyanUSize new_instr_length = sizeof (new_bytes);
if (!ZYAN_SUCCESS (ZydisEncoderEncodeInstruction (&enc_req, new_bytes, &new_instr_length))) {
	EMU_LOG_ERROR ("Failed to encode instruction for RIP-relative instruction at %llx", runtime_address);
	return false;}
builder.store_reg_gs (gs_index_backup_rel, unused_register)
	.mov_reg_qword (unused_register, abs_address.value ())
	.bytes (new_bytes, new_instr_length)
	.load_reg_gs (unused_register, gs_index_backup_rel);
```

3.4.6 Реалізація рекомпіляції стандартних інструкцій

Функція handle_default відповідає за обробку інструкцій, які не належать до класу RIP-залежних або відносних інструкцій. Основна мета цієї функції — скопіювати інструкцію у рекомпільований код без змін, зберігаючи її оригінальну поведінку. Однак, навіть у таких простих випадках існує низка винятків, які потребують додаткової уваги під час трансформації.

Одним із таких винятків є інструкції, які містять префікси REP, REPE або REPNE [25]. Ці префікси змінюють поведінку інструкції, змушуючи її повторно виконуватись певну кількість разів — зазвичай значенням у регістрі RCX. Проблема полягає в тому, що така інструкція виконується "всередині себе", і з точки зору системи не існує очевидного способу відслідкувати кожну окрему ітерацію. Саме тому під час рекомпіляції інструкція з REP префіксом розкладається у вигляді циклу з умовними переходами, які повторюють семантику багаторазового виконання. В результаті такого перетворення інструкція, яка мала один REP префікс, фактично стає циклічною конструкцією, що перевіряє значення RCX, зменшує його, і при потребі повторює тіло циклу. Такий підхід дозволяє в майбутньому інтегрувати виклики зворотніх функцій на доступ до пам’яті або інші події на кожному кроці, гарантуючи повноцінне логування або аналіз кожної ітерації. Код для обробки інструкцій, що мають один з таких префіксів, разом з початком функції показано в лістингу 3.26.

Лістинг 3.26 — Код обробки інструкцій, що мають REP, REPE або REPNE префікс

```
bool c_x64_code_emulator::handle_default (uint64_t runtime_address, ZydisDecodedInstruction& instruction, ZydisDecodedOperand* operands, c_x64_shellcode_builder& builder, uint64_t shellcode_begin_offset) {
    const auto any_rep_mask = ZYDIS_ATTRIB_HAS_REP | ZYDIS_ATTRIB_HAS_REPE | ZYDIS_ATTRIB_HAS_REPNE;
    if (instruction.attributes & any_rep_mask) {
        EMU_LOG_INFO ("REP%s instruction at %llx", (instruction.attributes & ZYDIS_ATTRIB_HAS_REPE) ? "E" : ((instruction.attributes & ZYDIS_ATTRIB_HAS_REPNE) ? "NE" : ""), runtime_address);
        ZydisEncoderRequest enc_req {};
        if (!ZYAN_SUCCESS (ZydisEncoderDecodedInstructionToEncoderRequest (&instruction, operands, instruction.operand_count_visible, &enc_req))) {
```

```
                    EMU_LOG_ERROR ("Failed to convert decoded instruction to encoder request at %llx", runtime_address);
                    return false;}
                enc_req.prefixes &= ~any_rep_mask;
                uint8_t new_bytes[ZYDIS_MAX_INSTRUCTION_LENGTH];
                ZyanUSize new_instr_length = sizeof (new_bytes);
                if (!ZYAN_SUCCESS (ZydisEncoderEncodeInstruction (&enc_req, new_bytes, &new_instr_length))) {
                    EMU_LOG_ERROR ("Failed to encode instruction w/o REP... at %llx", runtime_address);
                    return false;}
                builder.bytes ({ 0x9C }); // pushfq
                builder.bytes ({ 0x48, 0x85, 0xC9 }); // test rcx, rcx
                auto jz_end = builder.conditional_jump (ZYDIS_MNEMONIC_JZ);
                std::optional<c_x64_shellcode_builder::relative_insn_t> repe_repne_exit = std::nullopt;
                builder.bytes ({ 0x9D }); // popfq
                builder.bytes (new_bytes, new_instr_length)
                    .bytes ({ 0x48, 0x8D, 0x49, 0xFF }); // lea rcx, ds:[rcx-0x01]
                if (instruction.attributes & ZYDIS_ATTRIB_HAS_REPE) {
                    repe_repne_exit = builder.conditional_jump (ZYDIS_MNEMONIC_JNZ);
                } else if (instruction.attributes & ZYDIS_ATTRIB_HAS_REPNE) {
                    repe_repne_exit = builder.conditional_jump (ZYDIS_MNEMONIC_JZ);
                }
                builder.conditional_jump (ZYDIS_MNEMONIC_JMP).link (shellcode_begin_offset);
                jz_end.link (builder.current_offset ());
                builder.bytes ({ 0x9D }); // popfq
                if (repe_repne_exit.has_value ()) {
                    repe_repne_exit->link (builder.current_offset ());}}
```

Ще одним важливим винятком є інструкція POPFQ та її аналоги (POPF, POPFD). Особливість цих інструкцій полягає в тому, що вони відновлюють стан прапорців процесора [26], зокрема прапор Trace Flag (TF), який відповідає за режим покрокового виконання. У випадку, коли цей прапор встановлений, Windows генерує виключення після виконання кожної інструкції. Якщо не вжити спеціальних заходів, виключення, що виникає після виконання POPFQ, буде зафіксоване вже на наступній інструкції, яка може належати до іншого фрагмента коду. Це унеможливлює точну прив’язку виключення до конкретної інструкції POPFQ. Щоб уникнути цієї ситуації, після кожної інструкції POPFQ у рекомпільований код додається одна інструкція NOP. Вона виконується одразу після POPFQ, і якщо буде активовано Trace Flag, то саме NOP стане джерелом

виключення. Таким чином, можна буде точно ідентифікувати, яка саме інструкція призвела до виключення, і правильно перенаправити обробку цієї події. Реалізацію обрбки інструкції POPFQ, включаючи всі інші, показано в лістингу 3.27.

Лістинг 3.27 — Код для обробки POPFQ та інших інструкцій

```
else { builder.bytes ((void*)runtime_address, instruction.length);
                if (instruction.mnemonic == ZYDIS_MNEMONIC_POPF || instruction.mnemonic == ZYDIS_MNEMONIC_POPFD || instruction.mnemonic == ZYDIS_MNEMONIC_POPFQ) {
                        builder.bytes ({ 0x90 });}}
```

3.4.7 Реалізація області динамічного коду

У межах реалізації системи динамічної емуляції виникла потреба у створенні так званої області динамічного коду — спеціального механізму, що дозволяє гнучко додавати в код фрагменти, які виконують певні допоміжні дії з тимчасовим збереженням і відновленням стану регістрів. Основна ідея полягає в тому, щоб частини коду, які можуть викликатися з багатьох різних місць, не створювали побічних ефектів при виконанні, тобто не змінювали критично важливі регістри чи прапорці процесора. Для цього реалізовано три основні допоміжні функції: dynamic_shellcode_begin, dynamic_shellcode_end та call_dynamic_shellcode.

Функція dynamic_shellcode_begin використовується для збереження поточного стану CPU-контексту, зокрема заданих регістрів та, за потреби, прапорців RFLAGS. У межах цієї функції надається можливість задати список регістрів, які слід зберегти, і якщо прапорець rflags увімкнено, буде збережено й значення прапорців. Дані зберігаються в GS-орієнтованому сховищі у відповідні індекси резервних регістрів. Результатом виконання dynamic_shellcode_begin є структура dynamic_shellcode_backup_state_t, яка зберігає, які саме значення були збережені, і згодом використовується для їх відновлення. Реалізацію наведено в лістингу 3.28.

Лістинг 3.28 — Реалізація функції dynamic_shellcode_begin

```
dynamic_shellcode_backup_state_t c_x64_code_emulator::dynamic_shellcode_begin (c_x64_shellcode_builder& builder, bool rflags, const std::vector<ZydisRegister>& registers) {
        dynamic_shellcode_backup_state_t result {};
        if (rflags) {
```

```
                builder.store_rflags_gs (gs_index_backup_rflags);}
        result.m_rflags = rflags;
        std::vector<int> free_slots_prepared { gs_index_backup_gp_register_1, gs_index_backup_gp_register_2, gs_index_backup_gp_register_3 };
        std::stack<int> free_slots_stack { std::deque (free_slots_prepared.begin (), free_slots_prepared.end ()) };
        for (auto reg : registers) {
                if (free_slots_stack.empty ()) {
                        EMU_LOG_ERROR ("Failed to get free slot!");
                        utils::abort_execution ();}
                auto index = free_slots_stack.top ();
                result.m_index_to_register[index] = reg;
                builder.store_reg_gs (index, reg);
                free_slots_stack.pop ();}
        return result;}
```

Зі свого боку dynamic_shellcode_end відповідає за відновлення раніше збережених регістрів, прапорців, і, якщо зазначено в аргументі ret, генерує інструкцію повернення до точки виклику. Це повернення реалізовано як перехід за адресою, збереженою в GS-сегменті за індексом gs_index_called_from, куди попередньо записується адреса повернення. Реалізацію функції dynamic_shellcode_end показано в лістингу 3.29.

Лістинг 3.29 — Реалізація функції dynamic_shellcode_end

```
void c_x64_code_emulator::dynamic_shellcode_end (c_x64_shellcode_builder& builder, dynamic_shellcode_backup_state_t& state, bool ret) {
        if (state.m_rflags) {
                builder.load_rflags_gs (gs_index_backup_rflags);
        }
        for (auto& entry : state.m_index_to_register) {
                builder.load_reg_gs (entry.second, entry.first);
        }
        if (ret) {
                builder.jump_gs (gs_index_called_from);}}
```

Окрему увагу варто приділити двом різним сценаріям використання області динамічного коду. Перший — інлайновий сценарій, у якому динамічний код розміщується безпосередньо в тілі рекомпільованого блоку. У цьому випадку dynamic_shellcode_end викликається з параметром ret = false, і жодного повернення до початкової точки виклику не відбувається — виконання просто переходить далі

по потоку. Такий варіант використовується, коли вставка динамічного коду є одноразовою та локалізованою.

Другий сценарій — відокремлений (або загальний), коли динамічний код генерується один раз наперед, зазвичай під час ініціалізації емулятора, і зберігається в окрему область пам’яті. Пізніше, за потреби, з різних частин емулятора можна звернутись до цієї області через виклик call_dynamic_shellcode. Перед тим як зробити перехід до динамічного коду, ця функція зберігає адресу повернення (в обхідну змінну gs_index_called_from), записує адресу виклику в gs_index_call_destination і виконує безумовний перехід. Такий підхід дозволяє заощадити значну кількість пам’яті, оскільки замість генерації одного й того ж фрагменту кілька разів, досить зробити його один раз і викликати при потребі. Код фукції call_dynamic_shellcode наведено в лістингу 3.30.

Лістинг 3.30 — Код функції call_dynamic_shellcode

```
void c_x64_code_emulator::call_dynamic_shellcode (c_x64_shellcode_builder& builder, uint8_t* shellcode) {
	builder.store_reg_gs (gs_index_called_from, ZYDIS_REGISTER_RAX);
	auto lea = builder.lea_rax_rip ();
	builder.xchg_reg_gs (gs_index_called_from, ZYDIS_REGISTER_RAX)
		.store_qword_gs (gs_index_call_destination, uint64_t (shellcode))
		.jump_gs (gs_index_call_destination);
	lea.link (builder.current_offset ());}
```

3.5 Реалізація центрального контролеру

Центральний контролер є головним елементом, що забезпечує управління виконанням емуляції на рівні виключень. Його основна роль полягає у перехопленні виключень, які виникають під час виконання коду в захищеному регіоні пам’яті, та прийнятті рішень — чи слід ці виключення обробляти як частину емуляції, чи передати назад системі. Для цього в центральному контролері реалізована віртуальна функція handle_exception, яка є перевизначенням відповідного методу базового класу — менеджера виключень.

При надходженні виключення функція насамперед перевіряє, чи знаходиться вказівник RIP у межах області, яка підлягає емуляції. Це робиться за допомогою методу should_emulate. Якщо так, то відбувається спроба отримати або згенерувати

відповідний блок рекомпільованого коду, який відповідає цій адресі. Для цього викликається функція get_recompile_code, яка повертає вказівник на рекомпільовану версію коду. Якщо код був згенерований вперше, логування може повідомити про це для відлагоджувальних цілей. Код обробки виключення з рекомпіляцією показано в лістингу 3.31.

Лістинг 3.31 — Код обробки виключення з рекомпіляцією

```
if (context->Rip >= m_emu_range_start && context->Rip < (m_emu_range_start + m_emu_range_size)) {
        bool executed_first_time = false;
        auto recompiled = c_x64_code_emulator::get_recompile_code (context->Rip, executed_first_time);
        if (!recompiled) {
                EMU_LOG_ERROR ("Failed to get/recompile code! for %llx", context->Rip);
                utils::abort_execution ();}
        if (m_first_execution_log_enabled && executed_first_time) {
                EMU_LOG_INFO ("Starting emulation of %llx [%llx]", context->Rip, recompiled);}
        c_x64_code_emulator::setup_current_thread_storage ();
        context->Rip = recompiled;
        RtlRestoreContext (context, 0);}
```

Перед передачею управління рекомпільованому коду, обов’язково викликається метод setup_current_thread_storage (див. лістинг 3.31), який ініціалізує потоково-залежне сховище. Воно розміщене в сегменті GS, і містить критичну інформацію для роботи системи, зокрема межі області емуляції, покажчики на глобальні таблиці відповідності та інші важливі дані. Лише після ініціалізації цих структур функція модифікує регістр RIP у контексті та передає управління рекомпільованому коду через виклик RtlRestoreContext, що повністю імітує природне продовження виконання програми.

Якщо ж вхідна адреса RIP не підлягає локальній обробці, функція перевіряє, чи може цим виключенням опікуватись інший активний екземпляр центрального контролера. Така перевірка здійснюється через should_emulate_global, яка обходить список усіх активних екземплярів і перевіряє, чи підпадає адреса під контроль хоча б одного з них. У випадку позитивного результату функція просто виходить, дозволяючи іншому екземпляру перехопити виключення.

Однак, існує також третій варіант — коли виключення сталося у вже рекомпільованому коді. У цьому випадку потрібно переконатись, що система зможе обробити це виключення так, ніби воно сталося у вихідному оригінальному коді. Для цього використовуються функції recompiled_to_original, які дозволяють встановити відповідність між адресами в рекомпільованому та оригінальному коді. Якщо така відповідність виявляється, значення полів RIP та ExceptionAddress замінюються на оригінальні. Особливої уваги потребує випадок з виключенням EXCEPTION_SINGLE_STEP. У цьому випадку також проводиться аналіз розміру інструкції, яка виконувалась, щоб скоректувати значення RIP, і зробити вигляд, що виняток стався після завершення інструкції, як того вимагає логіка покрокового трасування. Окрім заміни базових полів, також змінюється вміст додаткової структури, розміщеної в пам’яті одразу після EXCEPTION_RECORD. Ця структура, хоч і не документована офіційно, впливає на те, як система Windows ідентифікує місце виникнення виключення. Якщо не внести в неї відповідні зміни, система може помилково вирішити, що виключення сталося у "невідомій" області пам’яті, що призведе до неправильного або непередбачуваного сценарію завершення процесу. Саме тому функція проходиться по ключовим полям і замінює значення, що збігаються з адресами в рекомпільованому коді, на їхні оригінальні відповідники. Код для обробки виключення, що сталось в рекомпільованому коді, показано в лістингу 3.32.

Лістинг 3.32 — Код для обробки виключення, що сталось в рекомпільованому коді

```
        else {
            if (should_emulate_global (context->Rip)) {
               return;
            }
            auto original_rip = context->Rip;
            auto original_exception_address = uint64_t (record->ExceptionAddress);
            auto relocated_rip = c_x64_code_emulator::recompiled_to_original (original_rip);
                 auto relocated_exception_address = c_x64_code_emulator::recompiled_to_original
(original_exception_address);
             bool should_try_relocate_exception = (relocated_rip != 0) || (relocated_exception_address
!= 0);
            auto get_instruction_size = [] (uint64_t address) -> std::optional<int> {
               ZydisDecoder decoder;
```

```
                ZydisDecoderInit (&decoder, ZYDIS_MACHINE_MODE_LONG_64,
ZYDIS_STACK_WIDTH_64);
            ZydisDecoderContext zcontext {};
            ZydisDecodedInstruction instruction {};
            if (!ZYAN_SUCCESS (ZydisDecoderDecodeInstruction (
                                &decoder, &zcontext, reinterpret_cast<void*>(address),
ZYDIS_MAX_INSTRUCTION_LENGTH, &instruction))) {
              EMU_LOG_ERROR ("get_instruction_size failed at %llx", address);
              return std::nullopt;
            }
            return instruction.length;
          };
          if (should_try_relocate_exception) {
            EMU_LOG_INTERNAL ("--- Internal exception ---");
              EMU_LOG_INTERNAL ("Exception %x occured inside recompiled code at %llx.
Exception address: %llx",
                static_cast <unsigned int> (record->ExceptionCode), original_rip,
original_exception_address);
          } else {
            EMU_LOG_INTERNAL ("Unknown exception %x occured at %llx. Exception address:
%llx",
                static_cast <unsigned int> (record->ExceptionCode), original_rip,
original_exception_address);
          }
          if (relocated_rip) {
            if (record->ExceptionCode == EXCEPTION_SINGLE_STEP) {
              auto instruction_size = get_instruction_size (relocated_rip);
              if (!instruction_size.has_value ()) {
                EMU_LOG_ERROR ("Failed to get instruction size at %llx", relocated_rip);
                utils::abort_execution ();
              }
               EMU_LOG_INTERNAL ("Exception: adding extra offset %d to relocated pointers
due to single step exception!",
                 instruction_size.value ());
              relocated_rip += instruction_size.value ();
            }
             EMU_LOG_INTERNAL ("Exception: context->Rip relocated to the original code =>
%llx", relocated_rip);
            context->Rip = relocated_rip;
          }
          if (relocated_exception_address) {
            if (original_rip == original_exception_address) {
              relocated_exception_address = relocated_rip;
            }
              EMU_LOG_INTERNAL ("Exception: record->ExceptionAddress relocated to the
original code => %llx",
              relocated_exception_address);
            record->ExceptionAddress = reinterpret_cast<PVOID>(relocated_exception_address);}
          if (should_try_relocate_exception) {
            uint64_t* after_record_array = reinterpret_cast<uint64_t*>(
              reinterpret_cast<uint8_t*>(record) + sizeof (*record));
```

```
                for (int i = 0; i < 5; ++i) {
                   if (relocated_rip && after_record_array [i] == original_rip) {
                      after_record_array [i] = relocated_rip;
                     EMU_LOG_INTERNAL ("Exception: rip relocated to the original code in unknown
structure => %llx", relocated_rip);}
                                    if (relocated_exception_address && after_record_array [i] ==
original_exception_address) {
                      after_record_array [i] = relocated_exception_address;
                       EMU_LOG_INTERNAL ("Exception: exception address relocated to the original
code in unknown structure => %llx",
                         relocated_exception_address);}}}}
```

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ФУНКЦІОНАЛЬНОСТІ СИСТЕМИ ДИНАМІЧНОЇ ЕМУЛЯЦІЇ

4.1 Проєктування та реалізація системи зворотних викликів

4.1.1 Концепція та види callback-функцій в системі емуляції

У процесі розширення функціональності системи динамічної емуляції було закладено підтримку механізму зворотних викликів (callback-функцій) [27], що дозволяє отримати повний контроль над тим, як саме виконується машинний код в межах емуляції. Такий підхід дозволяє втручатися у виконання інструкцій, відстежувати звернення до пам’яті, а також контролювати момент виходу з області емуляції. У рамках системи реалізовано три основних типи коллбеків: instruction_callback_t, memory_callback_t та emulation_exit_callback_t. Усі вони мають спільну мету — перехоплення та можливість впливу на ключові події у процесі виконання коду, проте застосовуються у різних ситуаціях та відрізняються за набором параметрів.

Коллбеки типу instruction_callback_t викликаються перед виконанням будь-якої інструкції, що потрапила у межі області емуляції. Такий коллбек отримує вхідні параметри: об’єкт емулятора, контекст поточного процесора (cpu_context_t), а також інформацію про інструкцію (instruction_info_t). У такий спосіб можна відстежувати виконання окремих інструкцій, впливати на значення регістрів та реалізовувати складну поведінку, наприклад, зупинку виконання або модифікацію потоку команд.

Коллбеки типу memory_callback_t використовуються у випадках, коли певна інструкція звертається до визначеного діапазону пам’яті (або до області gs-сегменту). Крім усіх параметрів, характерних для instruction_callback_t, такі коллбеки додатково отримують адресу доступу, розмір та ознаки операції (наприклад, читання чи запис). Це дозволяє реалізовувати логіку моніторингу доступу до критичних структур або реалізовувати обмеження при роботі з певними регіонами пам’яті, не змінюючи при цьому логіку виконання решти коду.

Третім видом є emulation_exit_callback_t, який викликається у момент, коли інструкція призводить до виходу за межі зони емуляції. Це може бути, наприклад, виклик або перехід на адресу, що не покрита середовищем емуляції. У таких випадках коллбек отримує усю стандартну інформацію, а також змінну exit_address, яка дозволяє змінити адресу, куди відбудеться фактичний вихід з емуляції.

Сама система підтримує п’ять різних способів прив’язки коллбеків до подій:

— через add_instruction_callback, який дозволяє встановити коллбек за предикатом інструкції;

— через add_rip_callback, який викликає коллбек при попаданні в точну адресу;

— через add_memory_callback, для моніторингу доступу до пам’яті;

— через add_emulation_exit_callback, який викликається при будь-якому виході з області емуляції;

— а також через add_emulation_exit_range_callback та add_emulation_exit_exit_address_callback, які дозволяють задати вихідні коллбеки по конкретному діапазону або точці.

Усі коллбеки повертають значення типу e_callback_status, яке визначає подальшу поведінку емулятора після виклику коллбеку. Доступні три варіанти: cs_handled, cs_handled_rip_changed та cs_pass_through. Для instruction_callback_t та memory_callback_t їх трактування наступне. Якщо повертається cs_handled, це означає, що подальші коллбеки для цієї інструкції не будуть викликані, і оригінальна інструкція продовжить виконання. Якщо повертається cs_handled_rip_changed, то також припиняється виклик інших коллбеків, але виконання переходить на адресу, задану в полі rip структури cpu_context_t. У випадку повернення cs_pass_through усі інші коллбеки, які ще не були викликані, будуть виконані, а зміни в cpu_context_t можуть бути проігноровані або спричинити некоректну поведінку.

Особливість полягає у тому, що для emulation_exit_callback_t статус cs_handled також сигналізує про завершення виклику коллбеків, однак має додаткову специфіку: якщо у момент виклику коллбек змінює exit_address, то після

завершення обробки інструкція виконає свій звичний ефект (наприклад, call збереже адресу повернення у стек), але перехід відбудеться на змінену адресу. У випадку cs_handled_rip_changed та cs_pass_through поведінка аналогічна тій, що й для інших типів коллбеків.

4.1.2 Алгоритм додавання користувацьких callbacks в чергу

У межах реалізації механізму зворотних викликів (callback-функцій) важливу роль відіграє функція enqueue_callback, яка відповідає за генерацію коду додавання callback-запису до внутрішньої черги. Ця черга використовується для відкладеного виконання коллбеків, пов’язаних із конкретною інструкцією, доступом до пам’яті або подією виходу з області емуляції. Саме завдяки цьому підходу система зберігає контроль над порядком виклику коллбеків та забезпечує, щоб кожен із них був виконаний із переданим при реєстрації контекстом.

Функція enqueue_callback приймає три параметри: екземпляр c_x64_shellcode_builder, в який буде додано машинні інструкції, вказівник на функцію-обробник (ptr), та вказівник на структуру контексту (context), яка зберігає необхідну інформацію для виконання callback-функції. Усі ці дані упаковуються у структуру callbacks_queue_entry_t, що містить лише два поля — сам коллбек і його контекст.

Алгоритм роботи функції складається з кількох послідовних етапів. На першому кроці створюється область динамічного коду (dynamic shellcode), в якій буде розміщений згенерований машинний код для додавання коллбеку в чергу. Під час цього етапу резервуються два регістри — RAX та RBX, які використовуються як робочі. Далі з GS-сегменту зчитується вказівник на саму чергу коллбеків (через gs_index_callbacks_queue) та її поточний розмір або індекс (через gs_index_callbacks_queue_size). Після цього відбувається розрахунок місця у черзі, куди буде записано новий елемент. Для цього індекс множиться на розмір одного елементу (8 байт, бо кожен елемент — це вказівник), і до початкової адреси черги додається зсув. Потім у обчислену позицію записується вказівник на функцію-обробник (m_callback), а також вказівник на структуру контексту (m_context).

Завершальним етапом є оновлення розміру черги — індекс збільшується на одиницю і нове значення записується назад до gs_index_callbacks_queue_size. Реалізацію функції enqueue_callback показано в лістингу 4.1.

Лістинг 4.1 — Реалізація функції enqueue_callback

```
void c_x64_code_emulator::enqueue_callback (c_x64_shellcode_builder& builder, void* ptr, void* context) {
	auto backup_state = dynamic_shellcode_begin (builder, false, { ZYDIS_REGISTER_RAX, ZYDIS_REGISTER_RBX });{
		builder.load_reg_gs (ZYDIS_REGISTER_RAX, gs_index_callbacks_queue)
			.load_reg_gs (ZYDIS_REGISTER_RBX, gs_index_callbacks_queue_size)
			.mov_reg_mem (ZYDIS_REGISTER_RAX, ZYDIS_REGISTER_RAX, 0, ZYDIS_REGISTER_RBX, 8)
			.lea_reg_mem (ZYDIS_REGISTER_RBX, ZYDIS_REGISTER_RBX, 1)
			.store_reg_gs (gs_index_callbacks_queue_size, ZYDIS_REGISTER_RBX)
			.mov_reg_qword (ZYDIS_REGISTER_RBX, uint64_t (ptr))
			.mov_mem_reg (ZYDIS_REGISTER_RBX, ZYDIS_REGISTER_RAX, offsetof (callbacks_queue_entry_t, m_callback))
			.mov_reg_qword (ZYDIS_REGISTER_RBX, uint64_t (context))
			.mov_mem_reg (ZYDIS_REGISTER_RBX, ZYDIS_REGISTER_RAX, offsetof (callbacks_queue_entry_t, m_context));}
	dynamic_shellcode_end (builder, backup_state, false);}
```

У результаті, коли пізніше система обробляє інструкцію, до якої було прив’язано коллбек, вона проходиться по сформованій черзі й викликає кожен обробник, передаючи йому саме той контекст, який був вказаний при реєстрації. Таким чином забезпечується точність, із якою коллбек-функція виконується саме в тому стані та з тими даними, з якими вона була додана, що критично важливо для правильної роботи системи емуляції.

4.1.3 Аглоритм виклику користувацьких callbacks з рекомпільованого коду

У системі динамічної емуляції зворотні виклики (callbacks), зареєстровані користувачем, не виконуються одразу, а додаються до черги. Реалізація механізму обробки цієї черги передбачає виділення окремого динамічного шеллкоду, який створюється один раз під час ініціалізації емулятора, а потім багаторазово викликається з різних місць. Цей код генерується у функції setup і зберігається у змінній m_process_callbacks_shellcode. Реалізація функції process_callbacks

наведена в лістингу 4.2. Реалізація функції зберігання контексту наведена в лістингу 4.3.

Сама функція setup будує внутрішній динамічний шеллкод, який циклічно проходить по черзі зворотних викликів. У кожній ітерації вона читає елемент з черги, витягує з нього вказівник на функцію зворотного виклику та відповідний контекст.

Лістинг 4.2 — Реалізація функції process_callbacks

```
void c_x64_code_emulator::process_callbacks (c_x64_shellcode_builder& builder, uint64_t original_rip) {
	auto backup_state = dynamic_shellcode_begin (builder, true, { ZYDIS_REGISTER_RAX });
	builder.load_reg_gs (ZYDIS_REGISTER_RAX, gs_index_callbacks_queue_size)
		.bytes ({ 0x48, 0x85, 0xC0 }); // test rax, rax
	auto jz_end = builder.conditional_jump (ZYDIS_MNEMONIC_JZ);
	dynamic_shellcode_end (builder, backup_state, false);
	store_context (builder, original_rip);
	call_dynamic_shellcode (builder, m_process_callbacks_shellcode);
	auto jmp_end = builder.conditional_jump (ZYDIS_MNEMONIC_JMP);
	jz_end.link (builder.current_offset ());
	dynamic_shellcode_end (builder, backup_state, false);
	jmp_end.link (builder.current_offset ());}
```

Лістинг 4.3 — Реалізація функції збереження контексту

```
void c_x64_code_emulator::store_context (c_x64_shellcode_builder& builder, uint64_t original_rip) {
	builder.store_reg_gs (gs_index_backup_gp_register_1, ZYDIS_REGISTER_RAX)
		.store_reg_gs (gs_index_backup_gp_register_2, ZYDIS_REGISTER_RBX)
		.store_reg_gs (gs_index_backup_gp_register_3, ZYDIS_REGISTER_RDX)
		.load_reg_gs (ZYDIS_REGISTER_RAX, gs_index_callback_cpu_context)
		.mov_mem_reg (ZYDIS_REGISTER_RBX, ZYDIS_REGISTER_RAX, offsetof (cpu_context_t, m_rbx))
		.load_reg_gs (ZYDIS_REGISTER_RBX, gs_index_backup_gp_register_1)
		.mov_mem_reg (ZYDIS_REGISTER_RBX, ZYDIS_REGISTER_RAX, offsetof (cpu_context_t, m_rax))
		.mov_mem_reg (ZYDIS_REGISTER_RCX, ZYDIS_REGISTER_RAX, offsetof (cpu_context_t, m_rcx))
		.mov_mem_reg (ZYDIS_REGISTER_RDX, ZYDIS_REGISTER_RAX, offsetof (cpu_context_t, m_rdx))
		.mov_mem_reg (ZYDIS_REGISTER_RBP, ZYDIS_REGISTER_RAX, offsetof (cpu_context_t, m_rbp))
		.mov_mem_reg (ZYDIS_REGISTER_RSP, ZYDIS_REGISTER_RAX, offsetof (cpu_context_t, m_rsp))
		.mov_mem_reg (ZYDIS_REGISTER_RSI, ZYDIS_REGISTER_RAX, offsetof (cpu_context_t, m_rsi))
		.mov_mem_reg (ZYDIS_REGISTER_RDI, ZYDIS_REGISTER_RAX, offsetof (cpu_context_t, m_rdi))
```

```
                .mov_mem_reg (ZYDIS_REGISTER_R8, ZYDIS_REGISTER_RAX, offsetof
(cpu_context_t, m_r8))
                .mov_mem_reg (ZYDIS_REGISTER_R9, ZYDIS_REGISTER_RAX, offsetof
(cpu_context_t, m_r9))
                .mov_mem_reg (ZYDIS_REGISTER_R10, ZYDIS_REGISTER_RAX, offsetof
(cpu_context_t, m_r10))
                .mov_mem_reg (ZYDIS_REGISTER_R11, ZYDIS_REGISTER_RAX, offsetof
(cpu_context_t, m_r11))
                .mov_mem_reg (ZYDIS_REGISTER_R12, ZYDIS_REGISTER_RAX, offsetof
(cpu_context_t, m_r12))
                .mov_mem_reg (ZYDIS_REGISTER_R13, ZYDIS_REGISTER_RAX, offsetof
(cpu_context_t, m_r13))
                .mov_mem_reg (ZYDIS_REGISTER_R14, ZYDIS_REGISTER_RAX, offsetof
(cpu_context_t, m_r14))
                .mov_mem_reg (ZYDIS_REGISTER_R15, ZYDIS_REGISTER_RAX, offsetof
(cpu_context_t, m_r15))
                .store_rflags_rbx ()
                .mov_mem_reg (ZYDIS_REGISTER_RBX, ZYDIS_REGISTER_RAX, offsetof
(cpu_context_t, m_rflags))
                .mov_reg_qword (ZYDIS_REGISTER_RBX, original_rip)
                .mov_mem_reg (ZYDIS_REGISTER_RBX, ZYDIS_REGISTER_RAX, offsetof
(cpu_context_t, m_rip))
                .bytes ({ 0x48, 0x89, 0xC3 }) // mov rbx, rax
                .bytes ({ 0x48, 0xC7, 0xC0, 0xFF, 0xFF, 0xFF, 0xFF }) // mov rax,
0xFFFFFFFFFFFFFFFF
                .bytes ({ 0x48, 0xC7, 0xC2, 0xFF, 0xFF, 0xFF, 0xFF }) // mov rdx,
0xFFFFFFFFFFFFFFFF
                .bytes ({ 0x0F, 0xAE, 0xA3 }).dword (offsetof (cpu_context_t, m_xsave_area)) //
xsave [rbx+m_xsave_area]
                .load_reg_gs (ZYDIS_REGISTER_RAX, gs_index_backup_gp_register_1)
                .load_reg_gs (ZYDIS_REGISTER_RBX, gs_index_backup_gp_register_2)
                .load_reg_gs (ZYDIS_REGISTER_RDX, gs_index_backup_gp_register_3);}
```

Потім відбувається виклик цієї функції (через call rdx) з правильно підготовленими аргументами. Після виклику оцінюється результат, який має тип e_callback_status. Якщо функція повертає cs_handled, виконання просто триває далі, переходячи до наступного елементу черги. Якщо ж результат — cs_handled_rip_changed, це означає, що коллбек змінив адресу інструкції, на яку слід передати виконання, тож контекст емуляції відновлюється і виконується перехід на нову адресу. Якщо повертається cs_pass_through або черга завершується, виконується відновлення оригінального контексту і емуляція триває звичайним чином. Код, який генерує цей шеллкод, наведено в лістингу 4.4.

Лістинг 4.4 — Код, який генерує шеллкод виклику коллбеків

```
bool c_x64_code_emulator::setup () {
```

```
	m_initialization_started = true;
	if (!m_process_callbacks_shellcode) {
		c_x64_shellcode_builder builder {};
		auto backup_state = dynamic_shellcode_begin (builder);{
			builder.load_reg_gs (ZYDIS_REGISTER_RSP, gs_index_callback_stack)
				.bytes ({ 0x48, 0x31, 0xDB }); // xor rbx, rbx
			auto retry_offset = builder.current_offset ();
			builder.load_reg_gs (ZYDIS_REGISTER_RAX, gs_index_callbacks_queue)
				.store_reg_gs (gs_index_backup_gp_register_1, ZYDIS_REGISTER_RBX);
			builder.mov_reg_mem (ZYDIS_REGISTER_RDX, ZYDIS_REGISTER_RAX, 0, ZYDIS_REGISTER_RBX, 8)
				.mov_reg_mem (ZYDIS_REGISTER_RCX, ZYDIS_REGISTER_RDX, offsetof (callbacks_queue_entry_t, m_context))
				.mov_reg_mem (ZYDIS_REGISTER_RDX, ZYDIS_REGISTER_RDX, offsetof (callbacks_queue_entry_t, m_callback))
				.bytes ({ 0xFF, 0xD2 }); // call rdx
			builder.bytes ({ 0x83, 0xF8, uint8_t (cs_handled) }); // cmp eax, cs_handled
			auto jz_handled = builder.conditional_jump (ZYDIS_MNEMONIC_JZ);
			builder.bytes ({ 0x83, 0xF8, uint8_t (cs_handled_rip_changed) }); // cmp eax, cs_handled_rip_changed
			auto jz_handled_and_changed_rip = builder.conditional_jump (ZYDIS_MNEMONIC_JZ);
			builder.load_reg_gs (ZYDIS_REGISTER_RBX, gs_index_backup_gp_register_1)
				.bytes ({ 0x48, 0xFF, 0xC3 }) // inc rbx
				.cmp_reg_gs (ZYDIS_REGISTER_RBX, gs_index_callbacks_queue_size);
			builder.conditional_jump (ZYDIS_MNEMONIC_JL).link (retry_offset);
			auto jmp_not_handled = builder.conditional_jump (ZYDIS_MNEMONIC_JMP);
			jz_handled_and_changed_rip.link (builder.current_offset ());
			restore_context (builder);
			builder.store_qword_gs (gs_index_callbacks_queue_size, 0);
			auto changed_rip_backup_state = dynamic_shellcode_begin (builder, false, { ZYDIS_REGISTER_RAX });{
				builder.load_reg_gs (ZYDIS_REGISTER_RAX, gs_index_callback_cpu_context)
					.mov_reg_mem (ZYDIS_REGISTER_RAX, ZYDIS_REGISTER_RAX, offsetof (cpu_context_t, m_rip))
					.store_reg_gs (gs_index_next_rip, ZYDIS_REGISTER_RAX);}
			dynamic_shellcode_end (builder, changed_rip_backup_state, false);
			optimize_rip (builder);
			builder.jump_gs (gs_index_next_rip);
			jz_handled.link (builder.current_offset ());
			jmp_not_handled.link (builder.current_offset ());
			restore_context (builder);
```

```
                builder.store_qword_gs (gs_index_callbacks_queue_size, 0);}
            dynamic_shellcode_end (builder, backup_state, true);
            m_process_callbacks_shellcode = builder.build ();}
        return m_process_callbacks_shellcode != nullptr;}
```

Функція process_callbacks не виконує обробку черги безпосередньо, а лише генерує код, який цю обробку реалізує. Її основна задача — вставити у рекомпільований код логіку виклику динамічного шеллкоду, який був попередньо створений під час ініціалізації. На початку функція зберігає поточний стан контексту процесора, після чого перевіряє, чи є щось у черзі зворотних викликів. Якщо черга порожня, вставляється умовний перехід, який минає подальшу логіку. Якщо ж у черзі є елементи, генерується виклик функції call_dynamic_shellcode, яка вбудовує у код інструкції для переходу на динамічний шеллкод, що зберігається в m_process_callbacks_shellcode. Після цього вставляється ще один умовний перехід, щоб забезпечити коректне повернення до основного потоку виконання після завершення обробки черги.

4.2 Реалізація зворотних викликів за предикатом та адресою інструкції

У межах підрозділу 4.2 розглядається реалізація механізму зворотних викликів, що активуються або за конкретною адресою інструкції, або при виконанні інструкції, яка задовольняє певний предикат. У системі динамічної емуляції ці два механізми реалізовано через окремі функції: add_rip_callback та add_instruction_callback. Реалізації цих функцій наведено в лістингу 4.5.

Лістинг 4.5 — Реалізація функцій add_instruction_callback та add_rip_callback

```
    void c_x64_code_emulator::add_instruction_callback (instruction_callback_predicate_t
predicate, instruction_callback_t callback) {
        if (m_initialization_started) {
            EMU_LOG_ERROR ("Function called after initialization!");
            utils::abort_execution ();}
        m_instruction_callbacks.push_back ({ predicate, callback });}
    void c_x64_code_emulator::add_rip_callback (uint64_t address, instruction_callback_t
callback) {
        if (m_initialization_started) {
            EMU_LOG_ERROR ("add_rip_callback called after initialization!");
            utils::abort_execution ();}
        m_rip_callbacks.insert_value (address, callback);}
```

Функція add_rip_callback дозволяє зареєструвати коллбек, який буде викликано для інструкції за конкретною віртуальною адресою. Всі такі коллбеки зберігаються у спеціалізованій структурі m_rip_callbacks, яка реалізована на основі c_page_table_storage. Цей контейнер забезпечує швидкий доступ до коллбеку за адресою, дозволяючи ефективно перевіряти, чи існує зворотний виклик для певної інструкції на етапі рекомпіляції. Функція вставляє відповідність між адресою та коллбеком, але передбачено, що її не можна викликати після початку ініціалізації, оскільки вся реєстрація має завершитися до старту роботи емулятора.

Функція add_instruction_callback, своєю чергою, дозволяє зареєструвати зворотний виклик, який буде активовано для будь-якої інструкції, що задовольняє заданий предикат. Ці коллбеки зберігаються в m_instruction_callbacks, який є вектором структур instruction_callback_info_t, де зберігаються предикат та відповідний коллбек. У процесі рекомпіляції кожна інструкція порівнюється з усіма зареєстрованими предикатами. Якщо хоча б один з них повертає true, до коду додається логіка виклику відповідного зворотного виклику.

Цей процес реалізовано через два окремі фрагменти коду в тілі функції get_recompile_code. Перший — це обробка rip-коллбеків: тут за поточною адресою інструкції через m_rip_callbacks.get_entry перевіряється, чи існує відповідний зворотний виклик. Якщо так, то створюється структура rip_callback_extended_info_t, яка заповнюється даними про інструкцію та вказівником на коллбек, після чого через функцію enqueue_callback у рекомпільований код вставляється логіка додавання цього коллбеку в чергу. Аналогічно працює і друга частина, що стосується предикатів. Для кожного елемента з m_instruction_callbacks викликається предикат з поточною інструкцією та її операндами. Якщо предикат виконується, створюється об’єкт типу instruction_callback_extended_info_t, який також додається в чергу через enqueue_callback. Код, який виконує додавання коллбеків в чергу, показано в лістингу 4.6.

Лістинг 4.6 — Код, який виконує додавання коллбеків в чергу

```
bool should_process_callbacks = false;
auto rip_callback_pointer = m_rip_callbacks.get_entry (current_address);
if (rip_callback_pointer && *rip_callback_pointer) {
	auto rip_callback_info = new rip_callback_extended_info_t ();
	rip_callback_info->m_emulator = this;
	rip_callback_info->m_instruction_info.m_info = instruction;
	memcpy (rip_callback_info->m_instruction_info.m_operands, operands, sizeof (operands));
	rip_callback_info->m_callback = *rip_callback_pointer;
	enqueue_callback (recompiled_code_builder, rip_callback_proxy, rip_callback_info);
	should_process_callbacks = true;}
for (auto& entry : m_instruction_callbacks) {
	if (entry.m_predicate (&instruction, operands)) {
		auto extended_info = new instruction_callback_extended_info_t ();
		extended_info->m_emulator = this;
		extended_info->m_instruction_info.m_info = instruction;
		memcpy (extended_info->m_instruction_info.m_operands, operands, sizeof (operands));
		extended_info->m_callback_info = &entry;
		enqueue_callback (recompiled_code_builder, instruction_callback_proxy, extended_info);
		should_process_callbacks = true;}}
if (instruction.mnemonic != ZYDIS_MNEMONIC_NOP) {
	std::vector<ZydisDecodedOperand*> memory_operands {};
	for (int i = 0; i < instruction.operand_count; ++i) {
		auto& op = operands[i];
		if (op.type == ZYDIS_OPERAND_TYPE_MEMORY && op.actions != 0
			) {
			memory_operands.push_back (&op);
			should_process_callbacks = true;}}
	process_memory_callbacks (recompiled_code_builder, current_address, instruction, operands, memory_operands);}
if (should_process_callbacks) {
	process_callbacks (recompiled_code_builder, current_address);}
```

Важливо зазначити, що обробка самої черги коллбеків не виконується одразу після вставки. Натомість використовується прапорець should_process_callbacks, який встановлюється в true у разі, якщо хоча б один зворотний виклик було зареєстровано під час проходження обробки інструкції. І лише якщо цей прапорець встановлено, у рекомпільований код додається виклик функції process_callbacks, яка генерує динамічний код обробки черги (див. лістинг 4.6).

Коли врешті-решт виконується динамічно згенерований код обробки черги, кожен коллбек викликається через відповідний проксі. У випадку коллбеків, прив’язаних до конкретних інструкцій за предикатом, використовується функція instruction_callback_proxy, тоді як для коллбеків, зареєстрованих за RIP-адресою, викликається rip_callback_proxy. Обидві ці функції отримують відповідні структури з даними, з яких вилучається покажчик на контекст потоку (cpu_context_t), збережений у GS-сегменті. Далі відбувається виклик користувацького коллбеку з передачею поточного емулятора, контексту процесора та інформації про інструкцію. Такий підхід забезпечує максимально повний контроль з боку користувача над процесом виконання кожної інструкції.

4.3 Реалізація механізму зворотних викликів доступу до пам’яті

У системі динамічної емуляції підтримується повноцінний механізм зворотних викликів на доступ до пам’яті. Така функціональність дозволяє відслідковувати будь-які звернення до пам’яті зі сторони рекомпільованого коду, включаючи читання, запис та навіть специфічні доступи до сегментованих областей, наприклад GS-сегменту. Для цього у системі передбачено окремий тип callback-функцій — memory_callback_t, а також відповідний набір механізмів їх реєстрації, перевірки та виклику.

Реєстрація коллбеків виконується за допомогою функції add_memory_callback. Вона приймає як аргументи маску прапорців доступу (читання, запис, GS-сегмент), адресу пам’яті, розмір області, яку потрібно моніторити, а також саму callback-функцію. Внутрішньо, крім збереження вектору коллбеків m_memory_callbacks, виконується ще й відмітка сторінок пам’яті в спеціалізованій структурі m_memory_pages_table. Це зроблено для того, щоб уже в рекомпіляцільованому коді швидко визначати, чи є взагалі потреба викликати коллбеки. Реалізація функції add_memory_callback наведена в лістингу 4.7.

Лістинг 4.7 — Реалізація функції add_memory_callback

```
void c_x64_code_emulator::add_memory_callback (uint64_t flags, uint64_t address, size_t size, memory_callback_t callback) {
```

```
uint64_t page_address = address & ~0xfff;
size_t validated_size = (size % 0x1000) == 0 ? size : ((size + 0x1000) & ~0xfff);
m_memory_pages_table.insert_page_value (page_address - 0x1000, true);
for (int i = 0; i < (validated_size / 0x1000); ++i) {
        m_memory_pages_table.insert_page_value (page_address + (i * 0x1000), true);
}
m_memory_pages_table.insert_page_value (page_address + validated_size, true);
{
        const std::unique_lock lock (m_memory_callbacks_mutex);
        m_memory_callbacks.push_back (new memory_callback_info_t { address, size,
flags, callback });}}
```

Під час рекомпіляції кожної інструкції, якщо вона містить хоча б один операнд типу memory, ця інструкція вважається потенційно цікавою для моніторингу доступу до пам’яті. У такому випадку викликається функція process_memory_callbacks, яка генерує відповідний фрагмент шеллкоду. Код виконує багаторівневу перевірку наявності відповідного прапорця в m_memory_pages_table, яка структурована за принципом таблиці сторінок у x64 архітектурі. Якщо жоден із рівнів не відмічений як активний, то згенерований код переходить далі, і ніякого виклику не відбувається.

Якщо ж сторінка активна, генерується інструкція, що зберігає адресу доступу, розмір і прапорці в GS-сховищі. Також створюється структура instruction_info_t, яка містить інформацію про саму інструкцію, та її покажчик зберігається в GS-сегменті. Після цього виконується виклик enqueue_callback, який додає в чергу відповідний виклик функції memory_callback_proxy. Реалізація memory_callback_proxy наведена в лістингу 4.8.

У момент виконання цього згенерованого коду в runtime-режимі, відбувається обробка черги. Сам memory_callback_proxy отримує з GS-сегменту всю необхідну інформацію: адресу доступу, розмір, прапори, контекст процесора та структуру інструкції. Далі, у циклі перевіряються всі зареєстровані коллбеки в m_memory_callbacks. Для кожного з них виконується перевірка відповідності адреси доступу до визначеного діапазону, а також збіг флагів доступу. Якщо умова виконується, викликається відповідний користувацький коллбек. Якщо він повертає статус cs_handled або cs_handled_rip_changed, виконання зупиняється і

результат одразу повертається в черговий процес. Якщо ж повернуто cs_pass_through, обробка триває для наступних коллбеків.

Лістинг 4.8 — Реалізація функції memory_callback_proxy

```
e_callback_status c_x64_code_emulator::memory_callback_proxy (c_x64_code_emulator* emulator) {
	uint64_t access_address = __readgsqword (GS_STORAGE_VALUE_OFFSET (gs_index_memory_access_address));
	size_t access_size = __readgsqword (GS_STORAGE_VALUE_OFFSET (gs_index_memory_access_size));
	uint64_t access_flags = __readgsqword (GS_STORAGE_VALUE_OFFSET (gs_index_memory_access_flags));
	cpu_context_t* context = (cpu_context_t*)__readgsqword (GS_STORAGE_VALUE_OFFSET (gs_index_callback_cpu_context));
	instruction_info_t* instruction_info = (instruction_info_t*)__readgsqword (GS_STORAGE_VALUE_OFFSET (gs_index_memory_instruction_info));
	auto do_flags_intersect = [] (uint64_t flags1, uint64_t flags2) -> bool {
		bool gs_segment_match = (flags1 & mcf_gs_segment) == (flags2 & mcf_gs_segment);
		if (!gs_segment_match) {
			return false;}
		int relevant_flags1 = flags1 & (mcf_read | mcf_write);
		int relevant_flags2 = flags2 & (mcf_read | mcf_write);
		return (relevant_flags1 & relevant_flags2) != 0;};
	std::vector<memory_callback_info_t*> copied_callbacks = {};{
		const std::shared_lock lock (emulator->m_memory_callbacks_mutex);
		copied_callbacks = emulator->m_memory_callbacks;}
	for (const auto& callback_info : copied_callbacks) {
		if (!do_flags_intersect (callback_info->m_flags, access_flags)) {
			continue;}
		for (uint64_t i = 0; i < access_size; ++i) {
			uint64_t access_byte = access_address + i;
			if (access_byte >= callback_info->m_memory_address && access_byte < (callback_info->m_memory_address + callback_info->m_memory_size)) {
				auto status = callback_info->m_callback (emulator, context, instruction_info, access_address, access_size, access_flags);
				if (status != cs_pass_through) {
					return status;
				}
				break;}}}
	return cs_pass_through;}
```

4.4 Реалізація механізму зворотних викликів виходу з області емуляції

У рамках розширення можливостей системи динамічної емуляції реалізовано ще один важливий механізм — зворотні виклики виходу з області емуляції, які дозволяють реагувати на ті ситуації, коли інструкція викликає перехід на адресу, що

більше не належить до зони, контрольованої емулятором. Це необхідно для повного контролю над потоком виконання, особливо якщо потрібно виявити момент виходу з емуляції та, наприклад, зафіксувати цей факт або перенаправити виконання.

Для цього у системі визначено тип функцій emulation_exit_callback_t, які можуть бути зареєстровані через одну з трьох функцій: add_emulation_exit_callback, add_emulation_exit_range_callback, або add_emulation_exit_address_callback. Перша реєструє загальний коллбек на будь-який вихід з області емуляції, друга — обмежує його діапазоном адрес, а третя — реагує лише на точне співпадіння з вказаною адресою.

Реальна перевірка того, чи відбувається вихід з області, реалізована в окремій функції check_emulation_exit. Ця функція викликається під час рекомпіляції інструкцій, які можуть змінити потік виконання (наприклад, RET, CALL, JMP), і вставляє у згенерований код перевірку на те, чи вказує значення gs_index_next_rip на область, яка розташована всередині межі дозволеного діапазону (між gs_index_emu_begin та gs_index_emu_end). Якщо ні, то шеллкод завершується виконанням коду, який туди додала функція enqueue_callback. Цей код додає в чергу виклик функції emulation_exit_callback_proxy. Реалізацію функції check_emulation_exit наведено в лістингу 4.9.

Лістинг 4.9 — Реалізація функції check_emulation_exit

```
void c_x64_code_emulator::check_emulation_exit (c_x64_shellcode_builder& builder, uint64_t current_address, ZydisDecodedInstruction& instruction, ZydisDecodedOperand* operands) {
	auto backup_state = dynamic_shellcode_begin (builder, true, { ZYDIS_REGISTER_RAX });
	builder.load_reg_gs (ZYDIS_REGISTER_RAX, gs_index_next_rip)
		.cmp_reg_gs (ZYDIS_REGISTER_RAX, gs_index_emu_begin);
	auto jl = builder.conditional_jump (ZYDIS_MNEMONIC_JL);
	builder.cmp_reg_gs (ZYDIS_REGISTER_RAX, gs_index_emu_end);
	auto jge = builder.conditional_jump (ZYDIS_MNEMONIC_JNL);
	auto jmp_exit_end = builder.conditional_jump (ZYDIS_MNEMONIC_JMP);
	jl.link (builder.current_offset ());
	jge.link (builder.current_offset ());
	dynamic_shellcode_end (builder, backup_state, false);
	auto info = new emulation_exit_callback_extended_info_t ();
	info->m_emulator = this;
	info->m_instruction_info.m_info = instruction;
	memcpy (info->m_instruction_info.m_operands, operands, sizeof (info->m_instruction_info.m_operands));
```

```
enqueue_callback (builder, emulation_exit_callback_proxy, info);
process_callbacks (builder, current_address);
auto total_exit = builder.conditional_jump (ZYDIS_MNEMONIC_JMP);
jmp_exit_end.link (builder.current_offset ());
dynamic_shellcode_end (builder, backup_state, false);
total_exit.link (builder.current_offset ());}
```

Сама функція emulation_exit_callback_proxy, що виконується під час виконання згенерованого шеллкоду, перебирає всі зареєстровані callback'и з вектора m_emulation_exit_callbacks. Для кожного такого callback'а перевіряється, чи підпадає поточне значення next_rip (яке зчитується з gs_index_next_rip) під відповідний діапазон або адресу. Якщо так, то викликається відповідна користувацька функція. При цьому після виконання коллбеку оновлене значення next_rip записується назад у сховище, що дозволяє callback'у змінити подальший потік виконання. Якщо якийсь з callback'ів повертає статус, відмінний від cs_pass_through, то виконання припиняється, і повертається відповідний статус. Інакше обробка продовжується далі. Реалізацію функції emulation_exit_callback_proxy показано в лістингу 4.10.

Лістинг 4.10 — Реалізація функції emulation_exit_callback_proxy

```
e_callback_status c_x64_code_emulator::emulation_exit_callback_proxy (emulation_exit_callback_extended_info_t* info) {
    cpu_context_t* context = (cpu_context_t*)__readgsqword (GS_STORAGE_VALUE_OFFSET (gs_index_callback_cpu_context));
    uint64_t next_rip = __readgsqword (GS_STORAGE_VALUE_OFFSET (gs_index_next_rip));
    std::vector<emulation_exit_callback_info_t*> copied_vector {};{
        const std::shared_lock lock (info->m_emulator->m_emulation_exit_callbacks_mutex);
        copied_vector = info->m_emulator->m_emulation_exit_callbacks;}
    for (const auto& callback_info : copied_vector) {
        if (callback_info->m_memory_size) {
            if (!(next_rip >= callback_info->m_memory_address && next_rip < (callback_info->m_memory_address + callback_info->m_memory_size))) {
                continue;}}
        auto status = callback_info->m_callback (info->m_emulator, context, &info->m_instruction_info, next_rip);
        __writegsqword (GS_STORAGE_VALUE_OFFSET (gs_index_next_rip), next_rip);
        if (status != cs_pass_through) {
            return status;}}
    return cs_pass_through;}
```

4.5 Розробка тестової бібліотеки з контрольованими сценаріями виконання

Для повноцінного тестування реалізованого ядра динамічного емулятора було розроблено окрему тестову бібліотеку. Метою створення цієї бібліотеки є перевірка основних функцій емулятора та коректності роботи механізмів зворотних викликів, які відповідають за перехоплення доступу до пам’яті, перехоплення специфічних інструкцій та відстеження виходів з емуляції. Тестова бібліотека має просту структуру та містить одну експортовану функцію з наперед визначеними діями, що дозволяє чітко визначити моменти її роботи, які повинні бути проконтрольовані емулятором.

Функція, що експортується з цієї бібліотеки, називається test_dll_function. Її реалізація складається з кількох ключових операцій, кожна з яких спрямована на тестування конкретних особливостей емулятора. По-перше, здійснюється читання даних з відомої системної структури KUSER_SHARED_DATA [28]. Зокрема, зчитуються поля NtBuildNumber, NtMajorVersion і NtMinorVersion. Ця дія спеціально вибрана через гарантоване розташування структури в пам'яті за адресою 0x7FFE0000. Це дозволить у подальшому перевірити механізм перехоплення доступу до пам'яті, який реалізовано в емуляторі. Код для читання цієї структури, разом з об'явленням тестової функції зображенний в лістингу 4.11.

Лістинг 4.11 — Початок тестової функції

```
extern "C" __declspec(dllexport) void test_dll_function (void) {
    const uint64_t kuser_data_address = 0x7FFE0000;
    uint32_t build_number = *(uint32_t*)(kuser_data_address + 0x260); // NtBuildNumber
    uint32_t major_ver = *(uint32_t*)(kuser_data_address + 0x26c); // NtMajorVersion
    uint32_t minor_ver = *(uint32_t*)(kuser_data_address + 0x270); // NtMinorVersion
    printf ("_KUSER_SHARED_DATA::NtBuildNumber - %x\n", build_number);
    printf ("_KUSER_SHARED_DATA::NtMajorVersion - %d\n", major_ver);
    printf ("_KUSER_SHARED_DATA::NtMinorVersion - %d\n", minor_ver);
```

Другим важливим елементом тестової функції є виклик спеціалізованої процесорної інструкції CPUID [29]. Ця інструкція використовується для отримання інформації про процесор, наприклад назви виробника, а також додаткових параметрів процесора, таких як сімейство, модель і степінг. Використання цієї інструкції у тестовій бібліотеці було зроблено спеціально з метою перевірки роботи

механізму перехоплення специфічних інструкцій, які вимагають додаткової обробки в емуляторі. В лістингу 4.12 зображено код отримання інформації про процесор.

Лістинг 4.12 — Код для отримання інформації про процесор

```
int cpu_info[4] = { 0 };
__cpuid (cpu_info, 0);
int maxLeaf = cpu_info[0];
char vendor[13] = {};
*reinterpret_cast<int*>(vendor) = cpu_info[1];     // EBX
*reinterpret_cast<int*>(vendor + 4) = cpu_info[3]; // EDX
*reinterpret_cast<int*>(vendor + 8) = cpu_info[2]; // ECX
printf ("CPU Vendor: %s\n", vendor);
printf ("Max CPUID Leaf: %d\n", maxLeaf);
__cpuid (cpu_info, 1);
int family = ((cpu_info[0] >> 8) & 0xF) + ((cpu_info[0] >> 20) & 0xFF);
int model = ((cpu_info[0] >> 4) & 0xF) + (((cpu_info[0] >> 16) & 0xF) << 4);
int stepping = cpu_info[0] & 0xF;
printf ("CPU Family: %d\n", family);
printf ("CPU Model: %d\n", model);
printf ("CPU Stepping: %d\n", stepping);
```

Крім того, у функції активно використовується стандартний виклик функції printf, яка розташована у зовнішньому модулі. Використання цього виклику є важливим, оскільки дозволяє перевірити коректність роботи механізму перехоплення виходів за межі області емуляції. При виклику printf виконання виходить за межі емулятора, отже, емулятор має правильно це ідентифікувати та активувати відповідний коллбек.

4.6 Реалізація інструменту моніторингу виконання тестового коду

Для перевірки коректної роботи всіх компонентів системи динамічної емуляції була створена окрема програма, яка використовує можливості реалізованого емулятора для трасування коду, написаного в тестовій бібліотеці. Цей підхід дозволив на практиці оцінити, наскільки ефективно відпрацьовують механізми зворотних викликів у відповідь на доступ до пам’яті, виконання окремих інструкцій, а також вихід за межі емульованого регіону.

У програмі реалізовано повноцінний запуск процесу перевірки, який починається з включення основного заголовочного файлу, що містить контролер

c_x64_emulation_controller. Цей клас є центральною точкою взаємодії з системою динамічної емуляції, оскільки всередині себе поєднує логіку роботи емулятора, менеджера виключень та всі підтримувані callback-механізми. Завдяки цьому, основна програма може централізовано налаштувати всі необхідні параметри.

У головній функції спочатку відбувається завантаження тестової бібліотеки за допомогою стандартної функції LoadLibraryA. Після цього, щоб визначити розмір завантаженої бібліотеки, використовується окрема функція get_image_size_from_module, яка зчитує заголовки PE-файлу. Далі отримується адреса експортованої функції test_dll_function, яка реалізує всі перевірочні сценарії з боку бібліотеки. Код функції для отримання розміру бібліотеки у пам’яті зображено в лістингу 4.13, код завантаження бібліотеки та отримання функції зображено в лістингу 4.14.

Наступним етапом є створення екземпляру класу c_x64_emulation_controller, куди передається базова адреса та розмір завантаженої бібліотеки. Після створення об’єкта, в ньому реєструються всі необхідні callback-функції. Зокрема, додається зворотний виклик на вихід з області емуляції (add_emulation_exit_callback), який дозволяє фіксувати момент, коли виконання виходить за межі заданого регіону. Аналогічно додається зворотний виклик на доступ до пам’яті в області KUSER_SHARED_DATA, що перевіряє можливість відстеження читання системної інформації.

Лістинг 4.13 — Функція для отримання розміру бібліотеки

```
uint32_t get_image_size_from_module (HMODULE module_base) {
	if (!module_base)
		return 0;
	auto dos_header = reinterpret_cast<PIMAGE_DOS_HEADER>(module_base);
	if (dos_header->e_magic != IMAGE_DOS_SIGNATURE)
		return 0;
	auto nt_headers = reinterpret_cast<PIMAGE_NT_HEADERS>(
		reinterpret_cast<BYTE*>(module_base) + dos_header->e_lfanew);
	if (nt_headers->Signature != IMAGE_NT_SIGNATURE)
		return 0;
	return nt_headers->OptionalHeader.SizeOfImage;}
```

Лістинг 4.14 — Код завантаження бібліотеки та пошуку функції

```
auto library = LoadLibraryA ("target-library.dll");
```

```
if (!library) {
	printf ("Failed to load library!\n");
	return;}
printf ("Library loaded at %p\n", library);
auto image_size = get_image_size_from_module (library);
if (!image_size) {
	printf ("Failed to get image size!\n");
	return;}
void(*test_dll_function)() = (decltype(test_dll_function))GetProcAddress (library, "test_dll_function");
if (!test_dll_function) {
	printf ("Failed to get test_dll_function address!\n");
	return;}
printf ("Function test_dll_function found at %p\n", test_dll_function);
```

Останнім зареєстрованим callback'ом є перехоплення інструкції CPUID, що дозволяє перевірити, як система реагує на виконання специфічних команд. В лістингу 4.15 показано код, що реєструє всі зворотні виклики та ініціалізує емулятор.

Після завершення налаштування викликається метод setup, який виконує ініціалізацію всієї системи, зокрема знімає прапори виконання з відповідних сторінок пам’яті, щоб спричинити необхідні виключення при доступі. Якщо ініціалізація пройшла успішно, відбувається виклик функції test_dll_function, що запускає перевірку всіх передбачених сценаріїв і дозволяє в реальному часі спостерігати, як спрацьовують встановлені зворотні виклики.

Лістинг 4.15 — Код додавання зворотніх викликів

```
c_x64_emulation_controller* controller = new c_x64_emulation_controller (uintptr_t (library), image_size);
controller->add_emulation_exit_callback (
	[] (c_x64_code_emulator* emulator, cpu_context_t* context, instruction_info_t* instruction, uint64_t& exit_address)->e_callback_status {
		printf ("- [callback] exit detected! from %llx to %llx\n", context->m_rip, exit_address);
		return cs_pass_through;});
controller->add_memory_callback (mcf_read, 0x7FFE0000, 0x1000,
	[] (c_x64_code_emulator* emulator, cpu_context_t* context, instruction_info_t* instruction,
		uint64_t access_address, size_t access_size, uint64_t access_flags)->e_callback_status {
		printf ("- [callback] memory access detected at %llx! Addess: %llx, size: %llx\n", context->m_rip, access_address, access_size);
		return cs_pass_through;});
controller->add_instruction_callback (
```

```
        [] (ZydisDecodedInstruction* instruction, ZydisDecodedOperand* operands)->bool {
            return instruction->mnemonic == ZYDIS_MNEMONIC_CPUID;
        },
        [] (c_x64_code_emulator* emulator, cpu_context_t* context, instruction_info_t* instruction)->e_callback_status {
            printf ("- [callback] cpuid detected at %llx | rax[0x%llx]\n", context->m_rip, context->m_rax);
            return cs_pass_through;});
```

Усі події при цьому виводяться на консоль, що спрощує процес налагодження та аналізу. Код виклику функції бібліотеки показано в лістингу 4.16.

Лістинг 4.16 — Код виклику функції бібліотеки

```
if (!controller->setup ()) {
    printf ("Failed to setup controller!\n");
    return;}
printf ("Calling test_dll_function...\n\n");
test_dll_function ();
```

4.7 Тестування функціональності системи

Після реалізації бібліотеки з контрольованими сценаріями виконання та інструменту моніторингу, наступним етапом стало практичне випробування системи динамічної емуляції з використанням реальних даних. З цією метою було проведено експеримент, в якому одну й ту ж функцію test_dll_function запускали двічі: один раз без активованого емулятора, а другий — з повноцінною активацією динамічної емуляції, що включає всі види зворотних викликів.

На рисунку Г.1 показано вивід програми без використання емулятора. Як видно, функція test_dll_function коректно виводить значення полів з області KUSER_SHARED_DATA, інформацію про CPUID та інші загальні відомості про систему. Жодних додаткових повідомлень про втручання у виконання коду виведено не було, що і підтверджує відсутність будь-яких інструментів трасування в цьому режимі.

У протилежність цьому, рисунок Г.2 демонструє виконання тієї ж функції під управлінням динамічного емулятора. Ще до виводу самих результатів видно лог від контролера, який інформує про ініціалізацію потоку для емуляції. Після цього відбувається серія повідомлень від зареєстрованих callback-функцій, які реагують

на події, що відбуваються в межах інструментованого коду. Так, вже на етапі зчитування значень з KUSER_SHARED_DATA з'являється повідомлення про доступ до пам'яті, що підтверджує правильну роботу системи моніторингу читання із заданої області. Ці події відповідають саме тим рядкам коду в бібліотеці, де відбувається зчитування NtBuildNumber, NtMajorVersion та NtMinorVersion.

Далі у логах фіксуються події, пов'язані з виконанням інструкцій CPUID. Всі ці інструкції були попередньо налаштовані на перехоплення за допомогою instruction callback-а. Повідомлення у консолі включають адресу виконання та значення регістру RAX, що дозволяє точно співвіднести кожен виклик CPUID з відповідною точкою в оригінальному коді. Для повного підтвердження цього факту, рисунок Г.3 демонструє фрагмент виконання в налагоджувачі (debugger), на якому видно, що в точках, відповідних залогованим повідомленням, дійсно розташовані інструкції cpuid. Це додатково підтверджує, що контекст виконання, захоплений під час виклику зворотної функції, повністю відповідає реальному стану виконання коду.

Ще однією ключовою можливістю системи є визначення моментів виходу за межі емульованої області. Такі переходи логуються з використанням emulation exit callback-ів. Вивід у консолі фіксує десятки повідомлень виду "exit detected! from ... to ...", які відповідають викликам стандартної функції printf. Слід зазначити, що у середовищі Windows функція printf реалізована не напряму, а через послідовність внутрішніх викликів, таких як __acrt_iob_func та __stdio_common_vfprintf. Саме ці функції були викликані зсередини бібліотеки під час форматованого виводу, і саме на ці переходи відреагував контролер, позначивши їх як виходи з області емуляції.

Загалом, аналіз результатів показує, що всі основні сценарії, закладені в бібліотеці для тестування — доступ до пам'яті, виконання специфічних інструкцій та переходи між модулями — були успішно перехоплені системою динамічної емуляції. Це свідчить про коректну інтеграцію всіх типів callback-механізмів та точність обробки контексту виконання на кожному етапі.

ВИСНОВКИ

У результаті виконання бакалаврської кваліфікаційної роботи було розроблено й реалізовано ядро системи внутрішньопроцесної динамічної емуляції для архітектури x64. Запропоноване рішення дозволяє здійснювати перехоплення та аналіз інструкцій машинного коду без необхідності повної ізоляції від операційної системи, що забезпечує високу гнучкість і мінімальний вплив на продуктивність цільового процесу.

На основі аналізу сучасних емуляторів та систем віртуалізації було виявлено ключові обмеження традиційних підходів, що зумовило доцільність створення нового механізму. Була спроєктована ефективна архітектура, яка включає підсистему рекомпіляції, систему обробки винятків та розширений механізм callback-функцій для контролю поведінки коду. Запропонована реалізація динамічного рекомпілятора охоплює обробку RIP-залежних інструкцій, відносної адресації та стандартних команд, що дозволяє адаптивно змінювати виконуваний код у режимі реального часу. Реалізація потоково-залежного сховища забезпечує безпечне й ефективне зберігання контексту під час виконання. Механізм callback-функцій забезпечує розширення можливостей моніторингу виконання програми без втрати контролю над потоками виконання.

Практична цінність роботи полягає в можливості застосування створеної системи для задач аналізу програм, виявлення прихованого коду, побудови систем трасування, інструментів реверс-інжинірингу та тестування безпеки.

Отримані результати підтверджують доцільність та ефективність запропонованого підходу та можуть бути основою для подальших досліджень у сфері високопродуктивної динамічної емуляції машинного коду.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Emulation [Електронний ресурс]. — Режим доступу: https://www.imaginationtech.com/glossary/emulation/.

2. What is hardware virtualization? [Електронний ресурс]. — Режим доступу: https://www.techtarget.com/searchitoperations/definition/hardware-virtualization.

3. Hypervisor [Електронний ресурс]. — Режим доступу: https://en.wikipedia.org/wiki/Hypervisor

4. Qemu [Електронний ресурс]. — Режим доступу: https://www.qemu.org/

5. Unicorn [Електронний ресурс]. — Режим доступу: https://www.unicorn-engine.org/.

6. Авраменко В. С., Авраменко А. С. Основи операційних систем. Навчальний посібник. — Черкаси: ЧНУ імені Богдана Хмельницького, 2018. — 524 с.

7. Йосифович П., Іонеску А., Руссинович М. Е., Соломон Д. А. Windows Internals. Part 1: System architecture, processes, threads, memory management, and more. — Редмонд: Microsoft Press, 2017. — 768 с.

8. PE Format [Електронний ресурс]. — Режим доступу: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format

9. Obfuscation [Електронний ресурс]. — Режим доступу: https://en.wikipedia.org/wiki/Obfuscation_(software)

10. CFF Explorer [Електронний ресурс]. — Режим доступу: https://ntcore.com/explorer-suite/

11. Запобігання виконанню даних (DEP) [Електронний ресурс]. — Режим доступу: https://www.vpnunlimited.com/ua/help/cybersecurity/data-execution-prevention?srsltid=AfmBOopCvMWxesg-DZ1s6u9k614FMqcyCnKlYbwRH7u5hTpkK5U3sToI

12. User mode and kernel mode [Електронний ресурс]. — Режим доступу: https://learn.microsoft.com/uk-ua/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode

13. EXCEPTION_RECORD structure [Електронний ресурс]. — Режим доступу: https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-exception_record

14. A journey through KiUserExceptionDispatcher [Електронний ресурс]. — Режим доступу: https://momo5502.com/posts/2024-09-07-a-journey-through-kiuserexceptiondispatcher/

15. IDA Pro [Електронний ресурс]. — Режим доступу: https://hex-rays.com/ida-pro

16. X86 memory segmentation [Електронний ресурс]. — Режим доступу: https://en.wikipedia.org/wiki/X86_memory_segmentation

17. Win32 Thread Information Block [Електронний ресурс]. — Режим доступу: https://en.wikipedia.org/wiki/Win32_Thread_Information_Block

18. Zydis [Електронний ресурс]. — Режим доступу: https://zydis.re/

19. Program counter [Електронний ресурс]. — Режим доступу: https://en.wikipedia.org/wiki/Program_counter

20. Gentle Introduction to x86-64 Assembly [Електронний ресурс]. — Режим доступу: https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp17/cse506/ref/assembly.html

21. Цикли і умовні переходи [Електронний ресурс]. — Режим доступу: http://www.znannya.org/?view=asm-cycles

22. Page Tables [Електронний ресурс]. — Режим доступу: https://wiki.osdev.org/Page_Tables

23. X64 calling convention [Електронний ресурс]. — Режим доступу: https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170

24. Atomic operation [Електронний ресурс]. — Режим доступу: https://wiki.osdev.org/Atomic_operation

25. Repeat String Operation Prefix [Електронний ресурс]. — Режим доступу: https://www.felixcloutier.com/x86/rep:repe:repz:repne:repnz

26. FLAGS register [Електронний ресурс]. — Режим доступу: https://en.wikipedia.org/wiki/FLAGS_register

27. Callback [Електронний ресурс]. — Режим доступу: https://en.wikipedia.org/wiki/Callback_(computer_programming)

28. KUSER_SHARED_DATA [Електронний ресурс]. — Режим доступу: https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/api/ntexapi_x/kuser_shared_data/index.htm

29. CPUID [Електронний ресурс]. — Режим доступу: https://uk.wikipedia.org/wiki/CPUID

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп’ютерної інженерії

Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ

Завідувач кафедри ОТ

проф., д.т.н.. Азаров О.Д.

«___» 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання бакалаврської кваліфікаційної роботи

«Система динамічної емуляції x64 коду в середовищі Windows»

08-54.БКР.016.00.000 ТЗ

Науковий керівник: доцент к.т.н.

Кожем’яко А. В. _____

Студент групи 1СП-21б

Стоколос Ю. М._____

ВНТУ 2025

1. Підстави для виконання бакалаврської кваліфікаційної роботи (БКР)

1.1 Актуальність дослідження обумовлена зростаючою потребою в ефективних засобах аналізу виконання програмного коду в середовищі Windows, зокрема у контексті забезпечення інформаційної безпеки, зворотної інженерії та динамічного моніторингу поведінки застосунків. Ускладнення архітектури сучасних операційних систем і поширення 64-розрядних платформ потребують нових підходів до контролю виконання машинного коду без втрати продуктивності. Особливої актуальності набувають інструменти, здатні здійснювати моніторинг інструкцій у реальному середовищі без повної ізоляції, що дозволяє проводити аналіз більш точно та оперативно.

1.2 Тема бакалаврської кваліфікаційної роботи затверджена наказом ректора університету №97 від 20.03.2025 року.

2. Мета БКР і призначення розробки

2.1 Метою цієї роботи є розробка та реалізація ядра системи внутрішньопроцесної динамічної емуляції для архітектури x64, яка забезпечує гнучке перехоплення, аналіз та модифікацію інструкцій без повної ізоляції середовища.

2.2 Призначення розробки полягає у створенні програмної системи, здатної виконувати динамічну емуляцію x64 інструкцій у середовищі Windows без повної ізоляції процесу. Розроблена система забезпечує перехоплення, аналіз та модифікацію виконання коду в реальному часі, що дозволяє ефективно застосовувати її для зворотної інженерії, дослідження поведінки програм та побудови інструментів безпечного виконання.

3. Вихідні дані для виконання БКР

3.1 Проведення аналізу сучасних методів динамічної емуляції машинного коду, а також огляд інструментів і бібліотек, що використовуються у сфері зворотної інженерії та безпечного виконання програм у середовищі Windows.

3.2 Розробка загальної архітектури системи внутрішньопроцесної динамічної емуляції, визначення основних модулів та принципів їх взаємодії.

3.3 Розробка ядра системи: модулів обробки винятків, динамічного рекомпілятора інструкцій, зберігання потоково-залежних даних, системи зворотних викликів.

3.4 Створення спеціалізованих тестових бібліотек та утиліт для перевірки правильності емуляції, трасування виконання та відлагодження основних сценаріїв роботи системи.

3.5 Забезпечення сумісності з архітектурою Windows x64 та дотримання принципів безпечного доступу до пам’яті, з урахуванням обмежень середовища виконання.

4. Вимоги до виконання БКР

Головна вимога – реалізувати систему внутрішньопроцесної динамічної емуляції машинного коду з архітектурою x64 у середовищі Windows, яка забезпечує перехоплення, аналіз та модифікацію інструкцій під час виконання без повної ізоляції процесу. Особливу увагу слід приділити реалізації рекомпілятора, механізму обробки винятків, зберіганню контексту виконання та організації системи зворотних викликів.

5. Етапи БКР та очікувані результати

5.1 Аналіз літературних джерел, наукових статей та технічної документації з тематики емуляції, зворотної інженерії та моніторингу виконання програмного коду. Вивчення архітектури Windows x64 та механізмів обробки винятків. Очікувані результати: формування теоретичної бази для реалізації системи, виявлення недоліків існуючих рішень та постановка вимог до розробки.

5.2 Формування загальної архітектури системи динамічної емуляції, визначення основних модулів (рекомпілятор, обробник винятків, контролер, система зворотних викликів), їх функцій та взаємозв’язків. Очікувані результати: концептуальна модель системи та логічна схема взаємодії компонентів.

5.3 Розробка програмної структури ядра системи: реалізація обробки виключень, дизасемблювання інструкцій, генерація альтернативного коду, кешування результатів та організація потоково-залежного сховища. Очікувані результати: вихідний код основних компонентів, реалізація механізмів перехоплення виконання коду.

5.4 Розробка тестових бібліотек з контрольованими сценаріями, створення утиліти трасування та моніторингу виконання, інтеграція з системою зворотних викликів. Очікуваний результат: працездатне середовище для перевірки функціональності та відлагодження системи.

5.5 Проведення тестів на різних прикладах виконання коду, перевірка точності роботи механізмів рекомпіляції, фіксація винятків та відповідність початковій логіці програм. Очікувані результати: звіти про тестування, виявлені помилки, підвищення надійності системи.

5.6 Оформлення пояснювальної записки, графічного матеріалу, формулювання висновків та узагальнення отриманих результатів. Очікувані результати: завершений звіт з описом реалізації, підтвердженою працездатністю та аналізом досягнутих цілей.

6. Матеріали, що подаються до захисту БКР

До захисту подаються: пояснювальна записка БКР, графічні та ілюстративні матеріали, протокол попереднього захисту на кафедрі, відгуки наукового керівника та опонента, протоколи складання державних екзаменів, анотації до БКР українською та іноземною мовами, довідка про відповідність оформлення БКР діючим вимогам.

7. Порядок контролю виконання та захисту БКР

Виконання етапів документації БКР контролюється науковим керівником згідно зі встановленими термінами. Захист БКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8. Вимоги до оформлювання та порядок виконання БКР

8.1 Вимоги до оформлювання

При оформлюванні БКР використовуються:

– ДСТУ 3008: 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

– ДСТУ 8302: 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

– міждержавний ГОСТ 2.104-2006 «Єдина система конструкторської документації. Основні написи»;

– методичні вказівки до виконання бакалаврських кваліфікаційних робіт зі спеціальності 123 – «Комп’ютерна інженерія» (освітня програма «Системне програмування»). Кафедра обчислювальної техніки ВНТУ 2023.

8.2 Порядок виконання БКР викладено в «Положення про кваліфікаційні роботи на першому (бакалаврському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21».

ДОДАТОК Б

ПРОТОКОЛ
ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ
НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: Система динамічної емуляції x64 коду в середовищі Windows

Тип роботи: Бакалаврська кваліфікаційна робота
(БДР, МКР)

Підрозділ кафедра обчислювальної техніки
(кафедра, факультет)

Показники звіту подібності StrikePlagiarism

Оригінальність 99,5% Схожість 0,5%

Аналіз звіту подібності (відмітити потрібне):

✓ Запозичення, виявлення у роботі, оформлені коректно і не містять ознак плагіату.

□ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.

□ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку _____ Захарченко С.М.
(підпис) (прізвище, ініціали)

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck щодо роботи.

Автор проекту _____ Стоколос Ю. М.
(підпис) (прізвище, ініціали)

Керівник проекту _____ Кожем'яко А. В.
(підпис) (прізвище, ініціали)

ДОДАТОК В

Графічна частина

СИСТЕМА ДИНАМІЧНОЇ ЕМУЛЯЦІЇ X64 КОДУ В СЕРЕДОВИЩІ WINDOWS

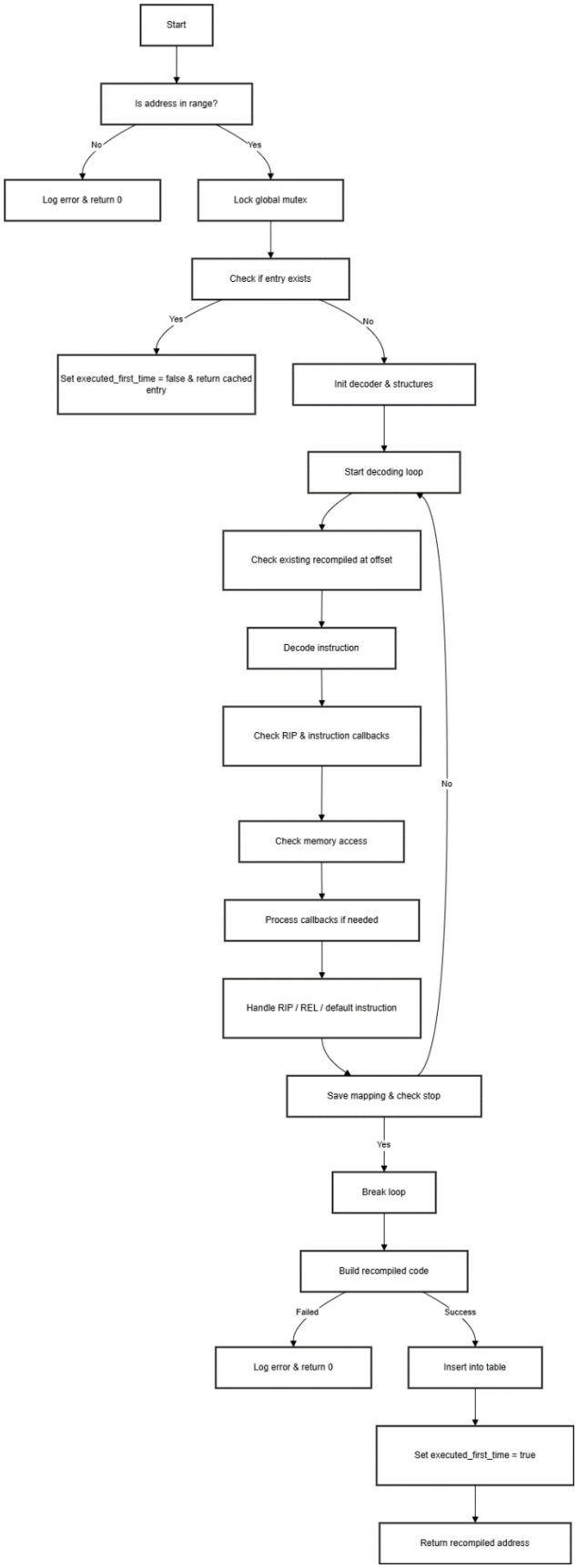

Рисунок В.1 — Блок-схема головного алгоритму обробки інструкцій

ДОДАТОК Г

Ілюстративна частпна

СИСТЕМА ДИНАМІЧНОЇ ЕМУЛЯЦІЇ X64 КОДУ В СЕРЕДОВИЩІ WINDOWS

```
Library loaded at 00007FFD9E4B0000
Function test_dll_function found at 00007FFD9E4B1070
Calling test_dll_function...

_KUSER_SHARED_DATA::NtBuildNumber - 65f4
_KUSER_SHARED_DATA::NtMajorVersion - 10
_KUSER_SHARED_DATA::NtMinorVersion - 0
CPU Vendor: GenuineIntel
Max CPUID Leaf: 32
CPU Family: 6
CPU Model: 183
CPU Stepping: 1
```

Рисунок Г.1 — Результат виконання функції без емуляції

```
Library loaded at 00007FFDDCE80000
Function test_dll_function found at 00007FFDDCE81070
[emulator][32780] Global table: 0000021879FBCF00 [0x8000]
[emulator][32780] Setting up exceptions manager...
[emulator][32780] Setting up code emulator...
[emulator][32780] Setting up protection...
[emulator][32780] Initialized!
Calling test_dll_function...

[emulator][32780] Thread initialized for emulation!
- [callback] memory access detected at 7fffddce81089! Addess: 7ffe0260, size: 4
- [callback] memory access detected at 7fffddce81097! Addess: 7ffe026c, size: 4
- [callback] memory access detected at 7fffddce8109e! Addess: 7ffe0270, size: 4
- [callback] exit detected! from 7fffddce81038 to 7ffde15622a0
- [callback] exit detected! from 7fffddce81057 to 7ffde1524620
_KUSER_SHARED_DATA::NtBuildNumber - 65f4
- [callback] exit detected! from 7fffddce81038 to 7ffde15622a0
- [callback] exit detected! from 7fffddce81057 to 7ffde1524620
_KUSER_SHARED_DATA::NtMajorVersion - 10
- [callback] exit detected! from 7fffddce81038 to 7ffde15622a0
- [callback] exit detected! from 7fffddce81057 to 7ffde1524620
_KUSER_SHARED_DATA::NtMinorVersion - 0
- [callback] cpuid detected at 7fffddce810cf | rax[0x0]
- [callback] exit detected! from 7fffddce81038 to 7ffde15622a0
- [callback] exit detected! from 7fffddce81057 to 7ffde1524620
CPU Vendor: GenuineIntel
- [callback] exit detected! from 7fffddce81038 to 7ffde15622a0
- [callback] exit detected! from 7fffddce81057 to 7ffde1524620
Max CPUID Leaf: 32
- [callback] cpuid detected at 7fffddce81105 | rax[0x1]
- [callback] exit detected! from 7fffddce81038 to 7ffde15622a0
- [callback] exit detected! from 7fffddce81057 to 7ffde1524620
CPU Family: 6
- [callback] exit detected! from 7fffddce81038 to 7ffde15622a0
- [callback] exit detected! from 7fffddce81057 to 7ffde1524620
CPU Model: 183
- [callback] exit detected! from 7fffddce81038 to 7ffde15622a0
- [callback] exit detected! from 7fffddce81057 to 7ffde1524620
CPU Stepping: 1
- [callback] exit detected! from 7fffddce81170 to 7ff6b1c70f12
[emulator][32412] Thread initialized for emulation!
- [callback] exit detected! from 7fffddce814cf to 7ffde3bc8083
```

Рисунок Г.2 — Результат виконання функції з емулятором

```
00007FFDDCE810C8   C64424 3C 00     mov byte ptr ss:[rsp+3C],0          dllmain.cpp:24
00007FFDDCE810CD   33C0             xor eax,eax
00007FFDDCE810CF   0FA2             cpuid
00007FFDDCE810D1   895424 34        mov dword ptr ss:[rsp+34],edx       dllmain.cpp:26
00007FFDDCE810D5   8BF8             mov edi,eax
00007FFDDCE810D7   894C24 38        mov dword ptr ss:[rsp+38],ecx       dllmain.cpp:27
00007FFDDCE810DB   48:8D5424 30     lea rdx,qword ptr ss:[rsp+30]       dllmain.cpp:29
00007FFDDCE810E0   48:8D0D 51210000 lea rcx,qword ptr ds:[<"CPU Vendor: %s\n 00007FFDDCE83238:"CPU Vendor: %s\n"
00007FFDDCE810E7   895C24 30        mov dword ptr ss:[rsp+30],ebx
00007FFDDCE810EB   E8 20FFFFFF      call <target-library.printf>
00007FFDDCE810F0   8BD7             mov edx,edi                         dllmain.cpp:30
00007FFDDCE810F2   48:8D0D 4F210000 lea rcx,qword ptr ds:[<"Max CPUID Leaf: 00007FFDDCE83248:"Max CPUID Leaf: %d\n"
00007FFDDCE810F9   E8 12FFFFFF      call <target-library.printf>
00007FFDDCE810FE   33C9             xor ecx,ecx                         dllmain.cpp:32
00007FFDDCE81100   B8 01000000      mov eax,1
00007FFDDCE81105   0FA2             cpuid
00007FFDDCE81107   8BC8             mov ecx,eax                         dllmain.cpp:33
00007FFDDCE81109   8BF8             mov edi,eax
```

Рисунок Г.3 — Розташування інструкцій cpuid

ДОДАТОК Д

Лістинг програмного забезпечення

dllmain.cpp — Код тестової бібліотеки з контрольованими сценаріями виконання:

```
#include <Windows.h>

#include <iostream>
#include <intrin.h>

extern "C" __declspec(dllexport) void test_dll_function (void) {

   const uint64_t kuser_data_address = 0x7FFE0000;

   uint32_t build_number = *(uint32_t*)(kuser_data_address + 0x260); // NtBuildNumber
   uint32_t major_ver = *(uint32_t*)(kuser_data_address + 0x26c); // NtMajorVersion
   uint32_t minor_ver = *(uint32_t*)(kuser_data_address + 0x270); // NtMinorVersion

   printf ("_KUSER_SHARED_DATA::NtBuildNumber - %x\n", build_number);
   printf ("_KUSER_SHARED_DATA::NtMajorVersion - %d\n", major_ver);
   printf ("_KUSER_SHARED_DATA::NtMinorVersion - %d\n", minor_ver);

   int cpu_info[4] = { 0 };

   __cpuid (cpu_info, 0);

   int maxLeaf = cpu_info[0];

   char vendor[13] = {};
   *reinterpret_cast<int*>(vendor) = cpu_info[1];     // EBX
   *reinterpret_cast<int*>(vendor + 4) = cpu_info[3]; // EDX
   *reinterpret_cast<int*>(vendor + 8) = cpu_info[2]; // ECX

   printf ("CPU Vendor: %s\n", vendor);
   printf ("Max CPUID Leaf: %d\n", maxLeaf);

   __cpuid (cpu_info, 1);
   int family = ((cpu_info[0] >> 8) & 0xF) + ((cpu_info[0] >> 20) & 0xFF);
   int model = ((cpu_info[0] >> 4) & 0xF) + (((cpu_info[0] >> 16) & 0xF) << 4);
   int stepping = cpu_info[0] & 0xF;

   printf ("CPU Family: %d\n", family);
   printf ("CPU Model: %d\n", model);
```

```
    printf ("CPU Stepping: %d\n", stepping);
}

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    return TRUE;
}
```

testing-app.cpp — Код інструменту моніторингу виконання тестового коду:

```
#include <iostream>
#include <Windows.h>

#include "dynamic_emulator/x64_emulation_controller.hpp"

uint32_t get_image_size_from_module (HMODULE module_base) {
        if (!module_base)
                return 0;

        auto dos_header = reinterpret_cast<PIMAGE_DOS_HEADER>(module_base);
        if (dos_header->e_magic != IMAGE_DOS_SIGNATURE)
                return 0;

        auto nt_headers = reinterpret_cast<PIMAGE_NT_HEADERS>(
                reinterpret_cast<BYTE*>(module_base) + dos_header->e_lfanew);

        if (nt_headers->Signature != IMAGE_NT_SIGNATURE)
                return 0;

        return nt_headers->OptionalHeader.SizeOfImage;
}

void entry () {

        auto library = LoadLibraryA ("target-library.dll");

        if (!library) {
                printf ("Failed to load library!\n");
                return;
        }

        printf ("Library loaded at %p\n", library);
```

```
	auto image_size = get_image_size_from_module (library);

	if (!image_size) {
		printf ("Failed to get image size!\n");
		return;
	}

	void(*test_dll_function)() = (decltype(test_dll_function))GetProcAddress (library, "test_dll_function");

	if (!test_dll_function) {
		printf ("Failed to get test_dll_function address!\n");
		return;
	}

	printf ("Function test_dll_function found at %p\n", test_dll_function);

	c_x64_emulation_controller* controller = new c_x64_emulation_controller (uintptr_t (library), image_size);

	controller->add_emulation_exit_callback (
		[] (c_x64_code_emulator* emulator, cpu_context_t* context, instruction_info_t* instruction, uint64_t& exit_address)->e_callback_status {
			printf ("- [callback] exit detected! from %llx to %llx\n", context->m_rip, exit_address);

			return cs_pass_through;
		}
	);

	controller->add_memory_callback (mcf_read, 0x7FFE0000, 0x1000,
		[] (c_x64_code_emulator* emulator, cpu_context_t* context, instruction_info_t* instruction,
			uint64_t access_address, size_t access_size, uint64_t access_flags)->e_callback_status {
			printf ("- [callback] memory access detected at %llx! Addess: %llx, size: %llx\n", context->m_rip, access_address, access_size);

			return cs_pass_through;
		}
	);

	controller->add_instruction_callback (
```

```
            [] (ZydisDecodedInstruction* instruction, ZydisDecodedOperand*
operands)->bool {
                return instruction->mnemonic == ZYDIS_MNEMONIC_CPUID;
            },
            [] (c_x64_code_emulator* emulator, cpu_context_t* context,
instruction_info_t* instruction)->e_callback_status {
                printf ("- [callback] cpuid detected at %llx | rax[0x%llx]\n", context-
>m_rip, context->m_rax);
                return cs_pass_through;
            }
        );

        if (!controller->setup ()) {
            printf ("Failed to setup controller!\n");
            return;
        }

        printf ("Calling test_dll_function...\n\n");

        test_dll_function ();
}

int main()
{
        entry ();
        Sleep (INFINITE);
}
```