Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Метод самовідновлення програмного забезпечення з журналізацією точок відновлення

(тема)

Виконав:

студент II курсу, групи СПм-21-2

Барсуков А.І
(прізвище, ініціали)

Спеціальність

123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма

Системне програмування
(повна назва освітньої програми)

Керівник: проф. Волк М.О.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ _____ Коваленко А.А.
(підпис) (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
"____" _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Барсукову Антону Ігоровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Метод самовідновлення програмного забезпечення з журналізацією точок відновлення

затверджена наказом по університету від "03" квітня 2023 р. № 318 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 17 травня 2023р.

3. Вхідні дані до роботи

Моделі та методи самовідновлення програмного забезпечення.
Механізми сучасних програмних систем щодо збереження стану програм.
Сучасні операційні системи та засоби віртуалізації.
Програмні системи для дослідження: Apache, ISC Bind, MySQL, Squid, OpenLDAP, PostgreSQL.

4. Перелік питань, що потрібно опрацювати у роботі

Аналіз предметної області та постановка задач
Розробка методу самовідновлення програмного забезпечення з журналізацією точок відновлення
Програмна реалізація та проведення експериментів
Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) презентація 12 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної області	03.04.23 – 06.04.23	
2	Розробка методів	06.04.23 – 12.04.23	
3	Реалізація алгоритмів	13.04.23 – 20.04.23	
4	Розробка структури програмних засобів	21.04.23 – 25.04.23	
5	Розробка програмних модулів	26.04.23 – 30.04.23	
6	Оформлення матеріалів кваліфікаційної роботи	01.05.23 – 10.05.23	
7	Подання кваліфікаційної роботи керівникові та	11.05.23 – 14.05.23	
	попередній захист		
8	Подання кваліфікаційної роботи на	14.05.23 – 17.05.23	
	рецензування		

Дата видачі завдання 03 квітня 2023 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Волк М.О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 66 с., 10 рис., 1 табл., 1 дод., 32 джерела.

САМОВІДНОВЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ВІДНОВЛЕННЯ ПІСЛЯ ПОМИЛОК, НАДІЙНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, КОНТРОЛЬНА ТОЧКА.

Програмні збої в серверних програмах є значною проблемою для збереження доступності системи. В роботі запропоновано метод та структуру системи, яка вводить додаткову надлишкову інформацію з метою відновлення програмного забезпечення після зневідомих збоїв, зберігаючи як цілісність системи, так і доступність, імітуючи поведінку системи після виявлення помилки. Точки відновлення — це місця в існуючому коді програми для обробки заданого набору очікуваних програмістом збоїв, які автоматично перевіряються для безпечного відновлення під час збоїв із більшого класу неочікуваних збоїв.

Коли помилка виникає в довільному місці програми, пропонуєма система відновлює виконання до відповідної точки відновлення та спонукає програму відновити виконання шляхом віртуалізації існуючих засобів обробки помилок програми. Впроваджено прототип системи під ОС Linux, який працює без вихідного коду програми та без змін базового ядра операційної системи. Наші експериментальні результати на наборі реальних серверних програм і помилок показують, що запропоновані рішення відновлення працюють для всіх протестованих помилок із швидким часом відновлення, має незначні накладні витрати на продуктивність і забезпечують автоматичне самовідновлення на порядок швидше, ніж поточні, керовані людиною методи розгортання виправлень.

ABSTRACT

Master’s thesis: 66 pages, 10 figures, 1 tables, 1 appendice, 32 sources.

SOFTWARE SELF-HEALING, ERROR RECOVERY, SOFTWARE RELIABLE, CHECKPOINT.

Software failures in server applications are a significant problem in maintaining system availability. The paper proposes a method and system structure that introduces additional redundant information for the purpose of re-oiling the software after unknown failures, preserving both system integrity and availability, simulating system behavior after error detection. Recovery points are places in existing application code to handle a given set of programmer-expected failures that are automatically checked for safe recovery from failures from a larger class of unexpected failures.

When an error occurs at an arbitrary point in the program, the proposed framework resumes execution to the appropriate recovery point and prompts the program to resume execution by virtualizing the program's existing error handlers. A system prototype has been implemented under the Linux OS, which works without the source code of the program and without changes to the basic kernel of the operating system. Our experimental results on a set of real server applications and errors show that the proposed recovery solutions work for all tested errors with fast recovery times, have negligible performance overhead, and provide automatic self-healing orders of magnitude faster than current human-driven patch deployment methods.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ВМ – віртуальна машина

ОС – операційна система

ПВ – помилка віртуалізації

ASLR – Address Space Layout Randomization

COTS – Commercial off-the-shelf or Commercially Available off-the-shelf

ISC – Internet Systems Consortium

HTTP – Hyper Text Transfer Protocol

NASA – National Aeronautics and Space Administration

OODA – Observe, Orient, Decide, and Act

SIGSEGV – Signal и Segmentation Violation

VM – Virtual Machine

ВСТУП

Помилки програмного забезпечення та вразливі місця в серверних програмах є значною проблемою для збереження цілісності та доступності системи. Загальноприйнятою практикою є використання безлічі інструментів, таких як старанна стратегія розробки програмного забезпечення, динамічний пошук помилок і інструменти статичного аналізу, щоб усунути якомога більше помилок.

Однак досвід показує, що програмне забезпечення без помилок дуже важко [18]. Як наслідок, навіть за найкращих обставин розгортається програмне забезпечення з помилками, і розробники стикаються з постійною та довготривалою боротьбою за створення та випуск патчів достатньо швидко, щоб виправити щойно виявлені помилки. Створення виправлень може тривати кілька днів, якщо не тижнів, і нерідко системи продовжують запускати невиправлені програми ще довго після того, як експлойт помилки стає загальновідомим [23].

За відсутності досконалого програмного забезпечення було запропоновано багато підходів, спрямованих на те, щоб виправити помилки програмного забезпечення та відновити їх. Ці підходи розглядають різні варіанти відновлення після несправності, включаючи фільтрацію зловмисного введення [8,16,31], збій для запобігання експлуатації системи [10], перезавантаження або перезапуск системи чи частин системи [6], повертаючи довільні значення для маскування помилок [25], відтворення в зміненому середовищі виконання [22] або вирізання несправних функцій програми [26,27].

Однак попередні підходи страждають від однієї чи кількох проблем, які можуть обмежити їхню ефективність і корисність на практиці. Ці проблеми включають нездатність мати справу з поліморфною поведінкою введення або звичайними сценаріями застосування, що включають шифрування,

нездатність обробляти детерміновані помилки, застосовність лише до помилок пам’яті, а не інших видів помилок, відсутність гарантій правильного виконання програми, нездатність працювати з немодифікованою програмою (двійковими файлами) та потребують модифікації програми або операційної системи, відсутність перевірки під час роботи з більш реалістичними та широко використовуваними розгортаннями багатопроцесорної або багатопотокової програми, значні накладні витрати на продуктивність під час звичайного виконання або відновлення програми та нездатність зберегти цілісність системи та доступність після виникає несправність.

Щоб вирішити ці проблеми, ми представляємо систему, яка забезпечує автоматичне програмне забезпечення самовідновлення. Система проставляє контрольні точки в існуючому коді програми для обробки передбачуваних програмістом збоїв, які автоматично перепрофільовуються та перевіряються для безпечного забезпечення загального відновлення збоїв. Коли помилка виникає в довільному місці програми, система відновлює виконання до найближчої точки відновлення та спонукає програму відновити виконання шляхом віртуалізації та використання існуючих засобів обробки помилок.

Точки відновлення віртуалізують обробку помилок шляхом створення відповідності між (потенційно нескінченним) набором помилок, які можуть виникнути під час виконання програми (наприклад, виявлена атака переповнення буфера або незаконний виняток розіменування пам’яті) і обмежений набір помилок, які може обробити код програми.

Таким чином, збій, який може призвести до збою програми, перетворюється на «повернення з помилкою» від функції обробки помилок під час виконання завдання, на якому сталася помилка. Завдяки повторному використанню існуючих засобів обробки помилок і їх автоматичному тестуванню перед використанням у робочому коді точки відновлення можуть зменшити ймовірність непередбачених шляхів виконання, тим самим роблячи відновлення надійнішим. Точки відновлення не просто маскують помилки. Замість цього вони «телепортують» несправності до відомих або

прогнозованих місць, з високою ймовірністю, для правильної обробки несправностей (включно з правильним станом програми).

Система спочатку визначає можливі точки повернення у програмі в автономному режимі за допомогою фаззингу [15], а потім реалізує, тестує та розгортає точки порятунку в режимі онлайн у відповідь на виникнення несправностей за допомогою циклу зворотного зв'язку «спостереження за східним рішенням» (OODA) [2]. Під час робочого використання, програма відстежує наявність помилок у програмі. Якщо несправність виявляється вперше, система використовує репліку програми (копію програми та весь її стан), щоб визначити, яку точку відновлення можна використовувати найбільш ефективно. Вибрана точка відновлення-кандидата потім реалізується за допомогою обробки винятків і механізму перезапуску контрольної точки операційної системи, який обробляє багатопроцесні та багатопотокові програми. Система підтверджує, що вона, усунув несправність, повторно запустивши програму проти послідовності подій, яка, очевидно, спричинила збій, а також проти вже відомих хороших і поганих вхідних даних.

У разі успіху, система використовує бінарну ін'єкцію під час виконання, щоб вставити точку відновлення в програму, що працює на робочому сервері. Коли помилка знову виникає на робочому сервері, програма використовує точку відновлення, щоб повернути стан до точки відновлення, де програма змушена повертати помилку, імітуючи поведінку, що спостерігається під час фаззингу. Система розроблена для роботи без втручання людини, щоб мінімізувати час реакції.

Впроваджено прототип системи для Linux, який працює без вихідного коду програми та без змін базового ядра операційної системи. Щоб продемонструвати його ефективність, ми оцінили наш прототип на широкому спектрі реальних серверних програм і помилок. Ми зосереджуємося на серверних програмах, оскільки вони зазвичай мають вищі вимоги до доступності, а також, як правило, мають коротке поширення

помилок відстані [25], які підходять для нашого підходу. Результати наших експериментів показують, що система визначила і використала точки відновлення для успішного відновлення після всіх протестованих помилок. На відміну від інших підходів, наша оцінка підтвердила здатність системи відновлюватися за наявності помилок для розгортання додатків у типових багатопроцесних і багатопоточних конфігураціях під час виконання широко використовуваних робочих навантажень для вимірювання продуктивності. Наші вимірювання продуктивності показали, що система відновлювався від помилок лише за кілька мілісекунд для всіх програм і зазнає менше 10% накладних витрат на продуктивність під час нормального виконання. Крім того, наші результати показують, що система забезпечує автоматичне та перевірене самовідновлення застарілих програм за кілька секунд (до хвилин) залежно від бажаного рівня тестування.

Розроблена система має кілька ключових переваг перед іншими підходами: (1) вона працює без втручання людини; (2) для неї не потрібен доступ або модифікація вихідного коду програми чи ядра операційної системи; (3) для розгортання не потрібна додаткова мережева інфраструктура; (4) вона обробляє поведінку поліморфного введення та зашифрований трафік; (5) це виходить за рамки простої обробки помилок пам’яті та краще справляється з детермінованими помилками; (6) вона працює як для багатопоточних, так і для багатопроцесних програм; (7) вона використовує семантику обробки помилок програми та включає фазу тестування, щоб забезпечити більшу впевненість у правильному виконанні програми за наявності помилок; (8) вона несе скромні накладні витрати на продуктивність.

У цієї роб оті представлено дизайн, реалізацію та оцінку роботи системи. Розділ 1 обговорює подібні роботи. Розділ 2 представляє архітектуру системи і детально розглядає концепцію точок відновлення. Розділ 3 наводить результати експерименту.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧ

1.1 Проблема помилок у програмному забезпеченні

За відсутності ідеального програмного забезпечення необхідним доповненням до проактивних підходів стають методи терпимості до помилок і відновлення. Гостра потреба в техніках, які вирішують проблему відновлення виконання за наявності помилок, відображається в нещодавній появі кількох нових дослідницьких ідей [17,20].

Наприклад, помилка віртуалізації працює за припущення, що існує відображення між набором помилок, які відбуваються під час виконання програми (наприклад, спіймана атака переповнення буфера або виняток незаконного посилання на пам’ять) і обмежений набір помилок, які явно обробляються кодом програми. Таким чином, збій, який може спричинити збій програми, перетворюється на повернення з кодом помилки від функції, у якій сталася помилка (або від одного з її предків у стеку). Ці методи, незважаючи на їх новизну у вирішенні цієї гострої проблеми, викликали багато суперечок, насамперед через відсутність гарантій щодо зміни семантики програми, які можна надати. Маскування виникнення помилок завжди матиме це клеймо, оскільки воно змушує програми зупинятися несподівані шляхи виконання. Проте ми вважаємо, що основна передумова маскування збоїв для забезпечення продовження виконання програми є багатообіцяючою, і наша мета полягає в мінімізації ймовірності небажаних побічних ефектів.

Ми описуємо нову техніку для модернізації застарілих програм із можливостями обробки винятків. Наш підхід полягає в загальному механізмі відновлення програмного забезпечення після помилок, який використовує методи віртуалізації операційної системи для забезпечення «точок порятунку», до яких програма може відновити виконання за наявності

помилок [13]. Коли помилка виникає в довільному місці програми, ми відновлюємо виконання програми до «точки відновлення» та імітуємо її спостережувану поведінку, щоб поширювати помилки та відновлювати виконання. Використання точок відновлення зменшує ймовірність непередбачених шляхів виконання, тим самим роблячи відновлення більш надійним, імітуючи поведінку системи в умовах контрольованої помилки. Ці контрольовані умови помилки можна розглядати як набір помилкових вхідних даних, подібних до тих, що використовуються більшістю команд із забезпечення якості під час розробки програмного забезпечення, призначених для тестування програми. Для виявлення «точок відновлення», додатки профілюються та контролюються під час тестів, які бомбардують програму «поганим введенням». Ситуація полягає в тому, що, відстежуючи поведінку додатків під час цих запусків, ми отримуємо уявлення про те, як перевірені програмістом точки програми поширюють помилки.

Ключова відмінність між цією роботою та попередніми методами, які намагаються не помічати виникнення несправностей [17,19,20] – тип впливу на семантику програми. Пункти порятунку не намагаються маскувати помилки в демонстрації сліпої віри. Насправді точки порятунку викликають прямо протилежну поведінку: вони викликають збої в місцях, які є відомими(або є серйозні підозри), щоб правильно усунути несправності. Ми досягаємо цього за допомогою етапу офлайн-аналізу, під час якого ми профілюємо програми під час помилкових тестів, щоб побудувати модель поведінки для програми.

Використовуючи цю модель, ми виявляємо можливі точки порятунку. Потім ми використовуємо набір програмних зондів, які контролюють програму на наявність певних типів несправностей. Після виявлення несправності, запускається механізм відновлення операційної системи, який дозволяє програмі відкотити стан програми до точки відновлення та відтворити виконання, вдаючи, що сталася помилка. Використовуючи безперервну перевірку гіпотезу, ми підтверджуємо, що наші дії усунули

помилку шляхом повторного запуску програми з послідовністю подій, яка, очевидно, спричинила помилку. Ми зосереджуємося на автоматичному "лікуванні" служб від нових виявлених несправностей, включаючи (але не обмежуючись ними) атаки на програмне забезпечення. Ремонт конструкції [8], де обговорюються механізми для виявлення пошкоджених структур даних і їх виправлення відповідно до деяких попередньо визначених обмежень. Хоча точність виправлень щодо семантики програми не гарантована, їхні тестові випадки продовжували працювати, коли помилки вводилися випадковим чином. Подібні результати показані в Y-гілках [21]: коли виконання програми змушене вибрати «неправильний» шлях у інструкції переходу, поведінка програми залишається незмінною більш ніж у половині випадків.

Ця робота зосереджена на додатках серверного типу з двох причин: вони зазвичай мають вищі вимоги до доступності, ніж орієнтовані на користувача додатки, і вони, як правило, мають короткі відстані поширення помилок [17], тобто, помилка, яка може виникнути під час обробки запиту, мало або взагалі не впливає на обслуговування майбутніх запитів. Провівши аналогію, якщо розглядати таку організацію, як NASA, історія показала, що не має сенсу використовувати такий механізм, як віртуалізація помилок, у програмній системі, яка обчислює траєкторії космічних подорожей, оскільки правильність результатів не може бути гарантована. Проте програмне забезпечення на марсоході значно виграє від техніки, яка дозволяє продовжувати виконання за наявності збоїв [18]. Наші плани щодо майбутньої роботи включають дослідження застосовності цієї техніки до клієнтських програм.

1.2 Аналіз підходів к самовідновленню програмного забезпечення

Було запропоновано багато підходів для усунення та відновлення після помилок програмного забезпечення. Такі схеми, як StackGuard [10] і ASLR

[20] фокусуються на захисті від атак із впровадженням коду та запобіганні використанню системи через помилку. Вони зберігають цілісність системи, припиняючи роботу програми, коли виникає помилка, але не можуть підтримувати доступність системи.

Методи перезавантаження, включаючи перезапуск усієї програми [29], омолодження програмного забезпечення [13] і мікроперезавантаження [6], спробуйте повернути систему до чистого стану до або після виявлення несправності. Повний перезапуск програми може зайняти багато часу, що призведе до значного простою програми. Мікроперезавантаження може бути швидшим, якщо перезапустити лише частини системи, але вимагає повного перезапису програм для порівняння розуміти невдачі. Жоден із цих методів не ефективно справляється з детермінованими помилками, оскільки вони можуть повторюватися після перезапуску.

Методи перезапуску контрольної точки [3,11] можна використовувати у спосіб, подібний до перезапуску всієї програми, але може забезпечити швидший час перезапуску, оскільки перезапуски виконуються з контрольної точки. При такому використанні ці методи все ще не обробляють детерміновані помилки, оскільки ці помилки все одно виникатимуть після перезапуску. Також були запропоновані інші варіанти використання контрольної точки-перезапуску в поєднанні з запуском кількох версій програми [3], які можуть пережити детерміновані помилки, якщо збої відбуваються незалежно. Однак вони несуть непомірні витрати для більшості програм з точки зору розробки, підтримки та запуску кількох версій програми одночасно.

Автоматична генерація сигнатур для систем виявлення вторгнень у мережу [21,17] захищає від вразливостей, фільтруючи вхідні дані для відсівання атак. Ключова проблема полягає в тому, що такі сигнатури досить сприйнятливі до помилкових спрацьовувань, особливо для поліморфних атак. Крім того, було показано, що поліморфна поведінка надто різноманітна, щоб її можна було ефективно моделювати за допомогою сигнатур [28].

Vigilante [8] покращено мережеву фільтрацію вхідних даних завдяки автоматичному створенню вхідних фільтрів на основі хоста. Фільтри на основі хоста пропонують покращену точність і вищу толерантність для виявлення семантично еквівалентних вхідних даних. На жаль, вони потребують специфічних для протоколу парсерів і не можуть працювати зі складними правилами, шифруванням і конкретним станом програми.

Shield [32] і VSEF [4] генерують підписи для вразливості замість підписів для введення. Вони забезпечують здатність обробляти певний стан програми та зашифрований трафік і мають набагато менше помилкових спрацьовувань, ніж фільтрація вхідних даних на основі мережі чи хосту.

Однак єдиний доступний варіант у разі виявлення зловмисного введення – це припинити виконання.

Rx [22] використовує механізм перезапуску контрольної точки в поєднанні з механізмами для зміни середовища виконання з метою відновлення після помилок. Однак попередні роботи [7] виявили, що понад 86% помилок програм не залежать від операційного середовища, є повністю детермінованими та повторюваними, і що відновлення, ймовірно, буде успішним лише за допомогою методів, пов’язаних із програмою або програмою. У той час як Rx розглядає ширші можливості для зміни середовища, включаючи відкидання шкідливих запитів на введення, відкидання запитів на практиці виявилося неефективним через поліморфну поведінку [28]. Rx намагається замаскувати прояв несправностей для клієнта, але потребує використання проксі-сервера додатка з підтримкою протоколу, який має бути здатний відфільтрувати інформацію, таку як мітки часу, яка могла б заплутати клієнтську програму.

Використання проксі-сервера ускладнює використання все більшої кількості програм, які використовують шифрування. Rx вимагає змін ядра операційної системи, що є ще однією перешкодою для розгортання. Нарешті, Rx не вирішує проблеми узгодженості під час встановлення контрольних точок і перезапуску програм, що включають кілька процесів.

Підмітальна машина [31] поєднує механізм перезапуску контрольної точки Rx і проксі з VSEF. Якщо виникає помилка, Sweeper використовує аналіз забруднення та зворотне зрізання, щоб визначити вхідні дані, які призвели до збою, генерує вхідний фільтр для видалення цього та подібних майбутніх запитів, а потім повертається до попередньої контрольної точки та повторно відтворює вхідні дані.

Оскільки Sweeper зменшує VSEF до використання для генерації вхідного підпису, він страждає від тих самих обмежень фільтрації введення, які описані раніше (поліморфізм і зашифрований трафік).

Обчислення, орієнтовані на прийнятність [9,24,25] просуває ідею про те, що поточні зусилля з розробки програмного забезпечення можуть бути неправильно спрямовані, ґрунтуючись на спостереженні про те, що можна знехтувати певними регіонами програми без негативного впливу на загальну доступність системи.

Обчислення без збоїв [25] — це спекулятивна техніка відновлення, яка базується на компіляторі для вставки коду для роботи із записами в нерозподілену пам’ять шляхом віртуального розширення цільового буфера. Така можливість має на меті забезпечити більш надійну реакцію на помилку, ніж просто збій, хоча й із значними накладними витратами на продуктивність, які варіюються від 80% до 500% для різноманітних програм.

Вибіркова трансакційна емуляція (STEM), яка використовується в реактивній імунній системі [27] — це спекулятивна техніка відновлення, розроблена двома авторами, яка визначає функцію, у якій сталася помилка, а потім вибірково емулює цю функцію та, можливо, інші в більшому діапазоні, щоб повернути значення помилок у спробі відновлення після помилки. STEM використовує поняття віртуалізації помилок, щоб означати повернення евристичного значення помилки від функції, у якій сталася помилка. Це дуже відрізняється від поняття віртуалізації помилок точки порятунку, яке використовується в системе, яке повторно використовує існуючий код обробки помилок у програмах і повертає значення на основі профілювання

цих функцій для імітації поведінки системи в умовах контрольованих і очікуваних помилок. На відміну від STEM, наша система не потребує вихідного коду, працює з багатопроцесорними та багатопоточними програмами, забезпечує суттєві покращення продуктивності системи.

1.3 Обчислення без збоїв

Ряд робот в області недосконалих, але прийнятних систем програмного забезпечення, які пропонують досконалу систему, як ввів Рінард [15] просувають ідею про те, що поточні зусилля з розробки програмного забезпечення можуть бути неправильно спрямовані, ґрунтуючись на спостереженні про те, що можна знехтувати певними регіонами програми без негативного впливу на загальну доступність системи. Щоб підтвердити ці ствердження, наведено низку прикладів, де введення помилок, таких як помилка «off-by-one», не призводить до неприйнятної поведінки. Ця робота підтверджує наше твердження, що більшість складних систем містять необхідну структуру для ефективного поширення помилок, а допустимість помилок, дозволена нашою системою, розширює діапазон прийнятності даної програми.

Тот же Рінард та інші[16,17] розробили обчислення без збоїв, екземпляром якого є компілятор, який вставляє код для обробки записів у нерозподілену пам’ять шляхом віртуального розширення цільового буфера. Така можливість має на меті ту саму мету, що й наша система: забезпечити більш надійну реакцію на помилку, а не просто збій. Оскільки програмний код значно переписується, щоб включити необхідні перевірки для кожного доступу до пам’яті їхня система спричиняє накладні витрати в діапазоні від 80% до 500% для різноманітних програм. Подібно до нашої попередньої роботи [19,20] існує лише обмежене емпіричне дослідження побічних ефектів на виконання програми. Нарешті, ця техніка застосовна лише до помилок пам’яті, тоді як наша техніка може бути застосована до

різноманітних помилок.

Однією з найважливіших проблем, пов’язаних із відновленням після збоїв програмного забезпечення та використання вразливостей, є забезпечення узгодженості та правильності даних і стану програми. Важливим внеском у цій галузі є автоматичні дані реконструкції [8], де обговорюються механізми для виявлення пошкоджених структур даних і їх виправлення відповідно до всіх визначених обмежень. І точність виправлень не зовсім гарантована, їх тестові випадки продовжують працювати, коли помилки визначались випадковим чином. Подібні результати показані вінній роботі[21]: коли виконання програми змушене вибрати «неправильний» шлях у інструкції переходу, поведінка програми залишається незмінною у значної частині випадків.

У системі Rx [14] програми періодично перевіряються та постійно перевіряються на наявність помилок. Коли виникає помилка, стан процесу відкочується і відтворюється в новому «середовищі». Якщо зміни в середовищі не призвели до виявлення помилки, програма пережила цей конкретний програмний збій. Однак попередні роботи [6,7] виявили, що понад 86% помилок програм не залежать від операційного середовища, повністю детерміновані та повторювані, і що відновлення, ймовірно, буде успішним лише за допомогою методів, що стосуються конкретної програми (або з урахуванням програми). Таким чином, здається ймовірним, що Rx застосовний лише в невеликій кількості випадків.

2 РОЗРОБКА МЕТОДУ САМОВІДНОВЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ З ЖУРНАЛІЗАЦІЄЮ ТОЧОК ВІДНОВЛЕННЯ

2.1 Життєвий цикл системи з самовідновленням

Система забезпечує архітектурну підтримку для самовідновлення програми за наявності непередбачених збоїв у повністю автоматизований спосіб. Система постійно стежить за програмою на наявність збоїв і визначає стратегії, використовуючи точки порятунку для реагування на майбутні випадки таких самих або подібних збоїв. Після вибору стратегії, система динамічно модифікує програму, використовуючи динамічну бінарну ін'єкцію, щоб вона могла виявити та оминути ту саму помилку в майбутньому. Метою нашої системи є автоматичне створення тимчасового виправлення певної проблеми, доки не стане доступним рішення користувача.

Рисунок 2.1 ілюструє роботу системи на високому рівні. Перед розгортанням, програма профілюється для виявлення потенційних точок відновлення. Після завершення профілювання, програма розгортається у своєму робочому середовищі.

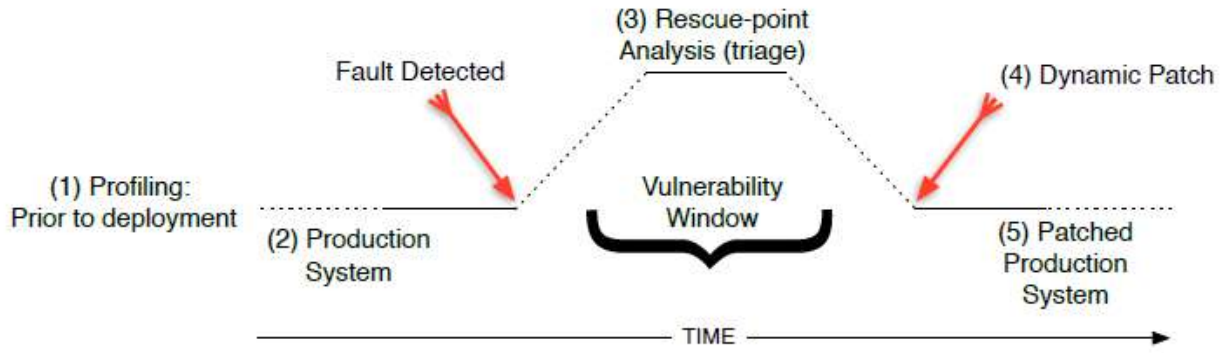

Рисунок 2.1 – Життєвий цикл системи

Під час нормального виконання, система контролює програму за допомогою різноманітних легких інструментальних механізмів, які полегшують виявлення та звітування про неправильну поведінку програми та системи. Крім того, система періодично перевіряє стан програми та веде журнал виконання (включаючи мережевий трафік).

Коли під час виконання виявляється помилка, стан останньої контрольної точки програми разом із журналом усіх вхідних даних, починаючи з цієї контрольної точки, передається в систему сортування, тіньове розгортання програми, де аналізується помилка. Потім система виконує автоматизований процес, метою якого є визначення відповідної точки відновлення, до якої програма може відновити виконання, якщо ця конкретна помилка виявиться повторно. Протягом цього часу робоча система залишається вразливою до повторного виникнення помилки, що призводить до вікна вразливості, у якому програмі може знадобитися вдатися до повного перезапуску програми для відновлення служби. Хоча нашій системі може знадобитися деякий час простою для фази аналізу, це в порядку секунд, і вартість амортизується, оскільки вона виникає один раз за нову помилку. Поєднання нашого підходу з такими методами, як мікроперезавантаження [6] є темою майбутніх досліджень.

Після вибору точки порятунку-кандидата система підтверджує, що вона придатна для розгортання, перевіряючи, що вона задовольняє трьом критеріям: живучість, правильність і продуктивність. Вибрана точка порятунку забезпечує живучість, якщо віртуалізація помилок у цей момент дозволяє програмі вижити після повторення помилки.

Точка відновлення є дійсною, якщо вона не вносить семантичних помилок і якщо програма може правильно обслуговувати майбутні запити. Точка відновлення є справжньою, якщо продуктивність захисту не спричиняє значних витрат часу виконання. Живучість перевіряється шляхом відтворення послідовності подій, які, очевидно, спричинили несправність.

Правильність перевіряється за допомогою ретельного тестування, яке адаптовано для конкретної роботи програми.

Ефективність оптимізується за рахунок використання продуктивності як метрики при виборі точки відновлення. Щойно відповідну точку відновлення буде перевірено, система створює виправлення, яке динамічно застосовується до програмного забезпечення, поки програма виконується в робочій системі. Патч створює точку відновлення всередині програми, щоб захистити програму від повторення конкретної помилки. Змінена програма запускатиме контрольну точку кожного разу, коли виконання досягне точки відновлення, і повертатиме свій стан до цієї точки, якщо помилка повториться.

Після відкату виконання до точки відновлення, віртуалізація помилок використовується для використання існуючих можливостей обробки помилок програми для ефективного усунення помилок. Замість фільтрації певних вхідних даних, які можуть спричинити збої, патч захищає програму від збоїв, які можуть виникнути в певному місці програми. Отриманий механізм відновлення не залежить від вхідних даних і, отже, захищений від ризиків, пов’язаних із поліморфізмом помилок/вхідних даних.

2.2 Розробка механізмів відновлення

Наша система забезпечує загальний механізм, який програми можуть використовувати для відновлення виконання за наявності помилок. Наш підхід можна розділити на наступні етапи офлайнових і онлайнових дій. В автономному режимі додатки профілюються під час «хибних запусків», щоб побудувати модель поведінки додатків. Ці хибні прогони генеруються регресійними тестами, якщо вони доступні, або за допомогою методів фаззингу [3,10], який підкреслює можливості обробки помилок програм.

Дії полягають в тому, що існує набір тестованих програмістами точок

програми, які регулярно використовуються для поширення «очікуваних» помилок. У свою чергу, ці прикладні точки можна використовувати для відновлення після збоїв і таким чином підтримувати доступність системи.

Використовуючи цю модель, ми виділяємо місця розташування програми, які можна використовувати як потенційні точки порятунку. Наша архітектура в режимі он-лайн дозволяє використовувати різноманітні монітори несправностей. Після виявлення помилки стан програми повертається до попередньо визначеного розташування програми, точки відновлення, де програма змушена повертати помилку, імітуючи поведінку, що спостерігається під час помилкових запусків.

Різні компоненти показано на рисунку 2.1: набір датчиків, який постійно відстежує програму на наявність несправностей і бере на себе контроль щоразу, коли вони виявляються; помилка віртуалізації активує компонент, який відповідає за визначення значень для введення в разі несправності; виявлення точки відновлення активує компонент, який використовується для визначення потенційних точок відновлення за допомогою статики та аналізу; активатор, який використовує механізм перезапуску контрольної точки для фіксації стану програми та відкату до збереженого стану; генератор патчів, який створює патчі з підтримкою точки відновлення для nerable аплікації; середовище тестування, в якому запропоновані патчі оцінюються згодом; вставка засобів, які полегшують вставлення затверджених патчів у запущені бінарні файли. На рисунку 2.1 – (1) датчики контролюють програму на наявність несправностей; (2) при виявленні несправності функція, де вона виявлена, захищається за допомогою одного з методів виявлення несправностей (3) визначається точка відновлення і встановлюється у програму; (4) патч, що містить точку відновлення, вставляється у програму (5) згенерований патч тестується за допомогою обставин, які спричинили помилку, і загальна поведінка програми відстежується; (6) робоча версія серверу оновлюється за допомогою патча (7), програма тепер може виявляти та відновлювать

несправності за допомогою віртуалізації помилок.

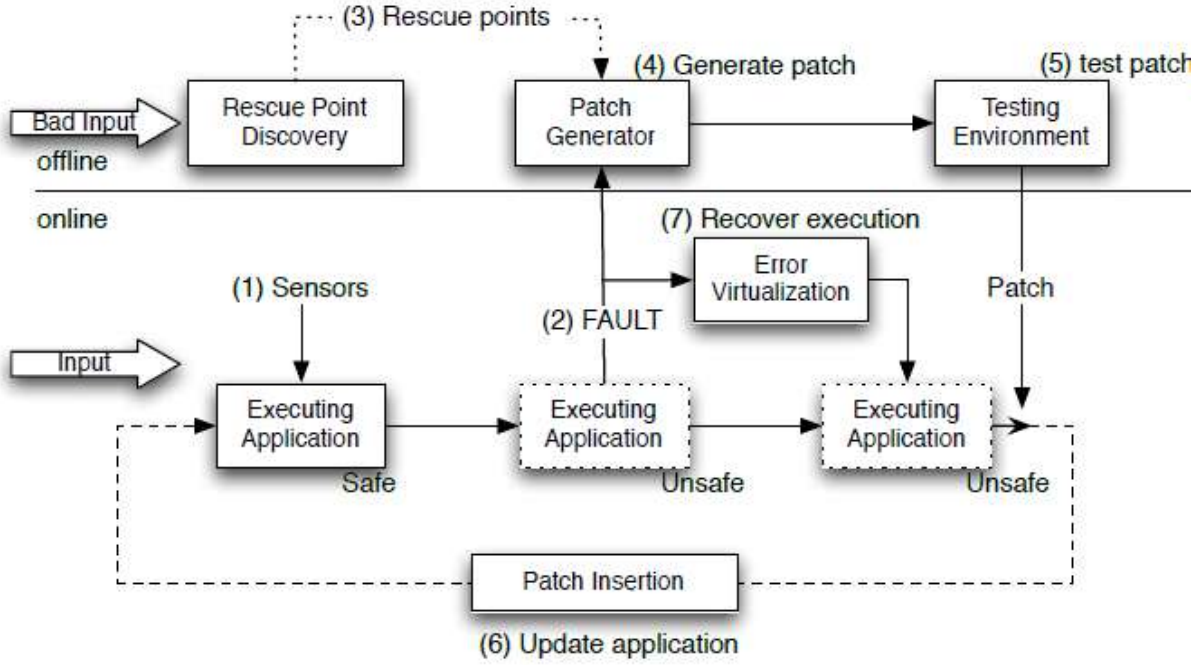

Рисунок 2.2 – Структура системи з самовідновленням

Усі компоненти розроблені для роботи в автоматичному режимі, щоб мінімізувати час реакції. У наступному підрозділі ми докладніше розглянемо кожен із цих компонентів.

2.3 Призначення точок відновлення

Визначення прийнятної точки відновлення має ключове значення для віртуалізації помилок. Це багато в чому визначає ступінь, ймовірність того, що програма продовжить роботу при несправності. Для виявлення відновлення використовуються два механізми: один статичний і один динамічний з більшою поширеністю.

Динамічний аналіз є кращим механізмом для виявлення відповідних точок відновлення, оскільки він надає однозначне розуміння поведінки програми. Зокрема, наша мета — дізнатися, як реагує програма на «поганий вхід» у контрольованих умовах і використовувати це знання, щоб зробити

раніше невидимі дефекти бути прогнозованими. Ситуація полягає в тому, що існує набір перевірених програмістами точок програм, які є регулярними та використовуються для розповсюдження «очікуваних» помилок. Наприклад, можна побачити, як зазвичай поширюється програма помилки під час стрес-тестування тестами забезпечення якості.

Навчання на основі помилок використовувалося в машинному навчанні і згодом системи виявлення помилок, було використано для відновлення програмного забезпечення при несправностях. Зокрема, ми інструментуємо програми шляхом вставки коду моніторингу в точках входу та виходу кожної функції, які можливі впровадження під час виконання dyninst [2], середовище виконання є інструментом бінарної ін'єкції. Прилади записують обидва параметри функції - типи та значення, що повертаються програмой при помилці. З цих записів ми побудували графіки викликів функцій разом із інформацією про тип повернення для кожної точки на графіку. Ми називаємо ці графіки "графіки відновлення". Rescue-графи використовуються як програмні фрагменти на функціональному рівні деталізації, які, у свою чергу, можна використовувати щоб ізолювати контрольний потік несправностей та визначити можливе відновлення.

Статичний аналіз використовується для посилення результатів описаних методів динамічного аналізу. Зокрема, ми використовуємо статичний аналіз для полегшення віртуалізацію помилок і виявленням точки відновлення. Для віртуалізації помилок статичний аналіз може допомогти визначити відповідні значення повернення помилок через перевірку коду та зворотне фрагментації програми. Детально досліджуємо шляхи, що виникають при прояві несправності, де вразлива функція знаходиться під час виклику. На цьому етапі ми перевіряємо, як повертається значення функція використовується в коді обробки. Це забезпечує розуміння того, як ми можемо використовувати під час помилки механізм віртуалізації. Наприклад, коли функція повертає значення оператора керування, за яким слідують оператори виходу – це створює умови для використання цього значення як

відповідного відновлюючого значення під час віртуалізації помилок. Протягом цього процесу ми також приділяємо увагу фрагментам програми для будь-яких проблемних випадків віртуалізації помилок, включаючи введення-виведення, яке виконується разом із використанням глобальних змінних та існування коду обробки сигналів.

2.3 Віртуалізація з використанням точок відновлення

Опишемо основну концепцію віртуалізації помилок за допомогою точок відновлення та розглянемо основні будівельні блоки системи. Як показано на рисунку 2.3, віртуалізацію помилок можна підсумувати такими кроками: стан програми контрольної точки в точці відновлення; контролювати заявку на наявність помилок; коли виникає помилка, скасувати зміни стану, зроблені функцією, аж до точки відновлення; після відновлення виконання до точки відновлення, примусове повернення помилки з використанням спостережуваного значення.

Ми розглядаємо компонент виявлення несправностей як чорний ящик, якому потрібно лише повідомити монітор несправностей про виникнення несправності. На додаток до стандартної обробки помилок операційної системи (напрнезаконне розіменування пам'яті тощо), ми використовуємо додаткові механізми для виявлення помилок пам'яті.

Існує ряд доступних компонентів виявлення несправностей, які можуть виявляти помилки пам’яті (наприклад, ProPolice [9] і TaintCheck [12]) і деякі, які виявляють порушення основних політик безпеки [1,11]. Для цілей нашої системи ми використовуємо два компоненти виявлення несправностей, які були розроблені раніше [19,20], які пропонують компроміс між накладними витратами на продуктивність і діапазоном помилок, які вони можуть виявити. Для цієї реалізації ми припускаємо наявність вихідного коду, але плануємо розглянути можливість застосування до комерційного готового

програмного забезпечення (COTS) у майбутній роботі.

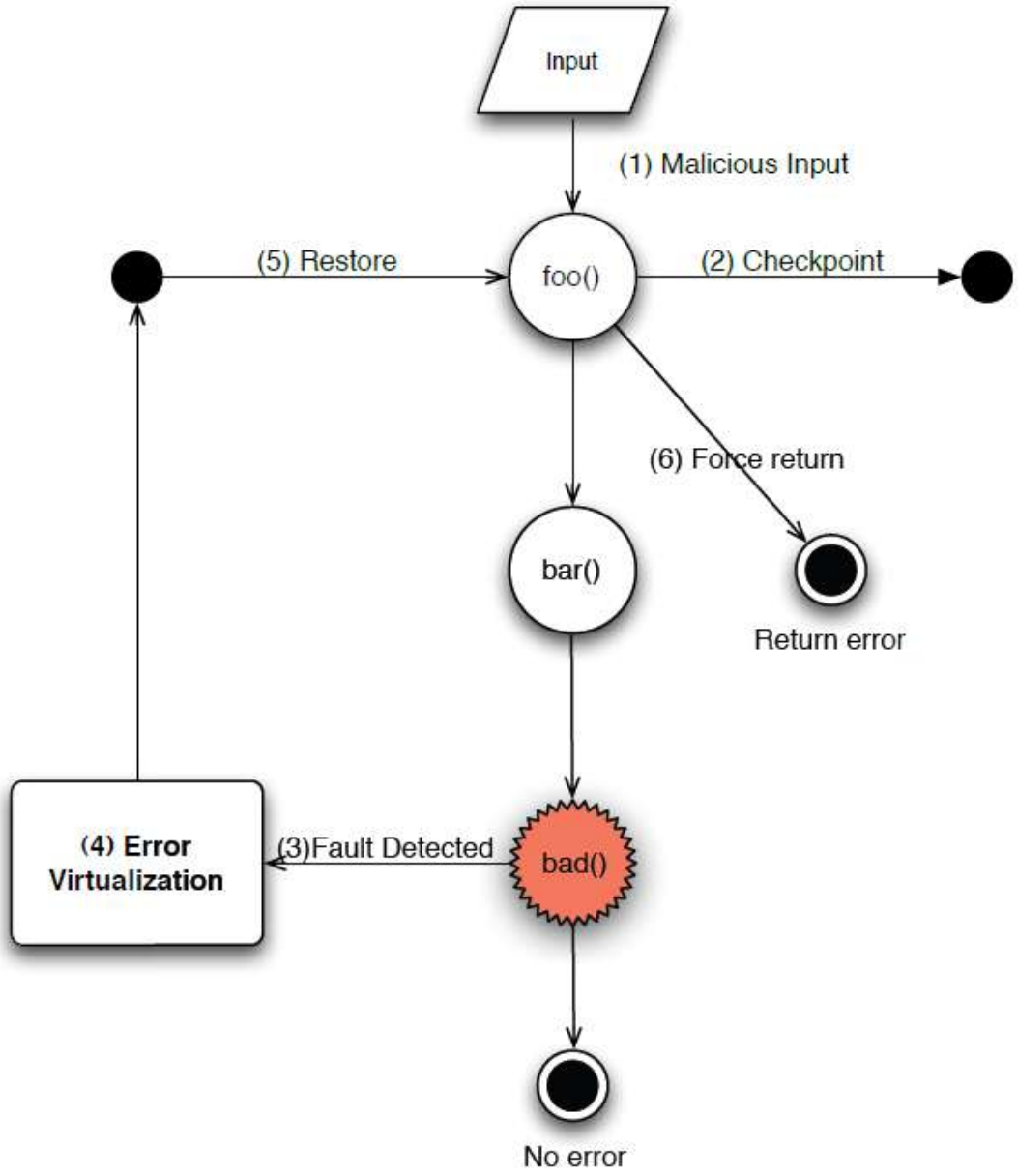

Рисунок 2.3 – Помилка віртуалізації (EV) з обробкою

Помилка віртуалізації (ПВ) – разі виявлення несправності за допомогою однієї з доступних методик виявлення несправності перевіряється стек викликів, щоб отримати послідовність функцій, які призвели до несправності. У цей момент ми порівнюємо стек викликів із графом відновлення, щоб отримати загальні вузли. Загальні вузли утворюють набір потенційних точок порятунку. Потім точки порятунку-кандидати

фільтруються відповідно до типу повернення. Для цієї конкретної реалізації можливі точки порятунку – це функції з типами повернення без вказівників або функції, які повертають покажчики, але спостережуване значення повернення є НУЛЬ.Функції, які повертають покажчики, вимагають глибшої перевірки структур даних, щоб визначити значення їхніх типів повернення за межами простого випадку повернення. Попереднє емпіричне дослідження показує, що обстежуваний програми, як правило, віддають перевагу використанню цілочисельних типів повернення як індикаторів помилок. Після фільтрації функцій ми маємо остаточний набір точок порятунку-кандидатів. У цей момент для визначення потенційних точок відновлення використовується графік потенційних точок відновлення.

Функція, де сталася помилка, утворює корінь дерева. Для кожного вузла в графі відновлення ми відтворюємо вхідні дані, які спричинили збій, і послідовно намагаємося віртуалізувати помилки на кожному з вузлів. За наявності механізму захисту несправності «виловлюються» монітором додатків, а стан програми повертається до точки відновлення. Точка відновлення перевіряє граф відновлення, щоб визначити, яке значення примусово повернути. Це значення отримано шляхом аналізу повернених значень функції точки відновлення.

Використовуючи приклад на рисунку 2.3, коли у функції виявлено помилку, ми спочатку витягуємо стек викликів, який містить відповідні функції. Припускаючи, що rescue-graph містить усі функції, знайдені в стеку викликів, ми ініціюємо віртуалізацію помилки з кореневим вузлом.

Ми повторюємо граф відновлення, намагаючись віртуалізувати помилку на вузлах, поки ми не досягнемо функції, що відновлює виконання програми без побічних ефектів. У випадку, коли немає перекриття між графіками викликів відновлення та функції, можна використовувати альтернативні механізми відновлення. По-перше, ми можемо відновитися до точки, анотованої програмістом, або ми перебираємо стек викликів уразливої функції (потенційно аж до основної), доки не знайдемо відповідну точку

відновлення, тобто, таку, яка не призводить до збою програми, застосовуючи евристику, описану в роботі [20].

2.4 Формування точок для самовідновлення

Рисунок 2.4 ілюструє самовідновлення програмної системи у разі реальної помилки веб-сервера Apache. Опис помилки наведено в таблиці 3.1. Сценарій передбачав виконання трьох функцій: ap_proxy_ftp_handler(), ap_pass_brigade() і ap_proxy_send_dir_filter().

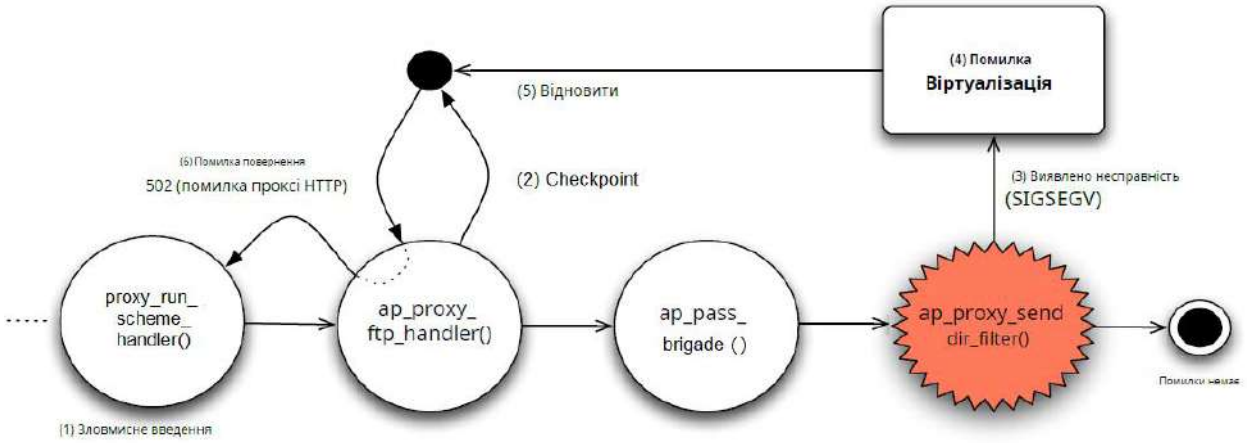

Рисунок 2.4 – Самовідновлення за допомогою точок відновлення: приклад із реальною помилкою сервера Apache

Через неправильне введення помилка проявляється на етапі виконання ap_proxy_send_dir_filter(), і призводить до помилки пам’яті (SIGSEGV). Система перехоплює перше виникнення несправності, ідентифікує ap_proxy_ftp handler() як відповідну точку відновлення та змінює сервер Apache наступним чином. Щоразу, коли виправлений сервер переходить у функцію ap_proxy_ftp handler(), система приймає контрольну точку стану сервера та дозволяє серверу продовжити виконання. Якщо така сама (або схожа) несправність трапляється, ap_proxy_send_dir_filter(), компонент виявлення помилок на виправленому сервері виявляє помилку та сповіщає

про помилку компонент віртуалізації.

Компонент віртуалізації аналізує інформацію про помилку та повертає стан сервер назад до точки відновлення ap_proxy_ftp handler(). Замість того, щоб дозволити виконанню продовжити той самий шлях, який спричинив помилку, система використовує віртуалізацію помилок, щоб примусово запустити ap_proxy_ftp_handler() з поверненням коду помилки, визначеним на етапі профілювання програми, а саме 502 (HTTP «Помилка проксі»). Використовуючи цей приклад, ми зараз опишемо детальніше як система виявляє, обирає, створює, випробовує та розгортає точки відновлення.

2.5 Виявлення несправностей та створення точок відновлення

Ми розглядаємо компонент виявлення несправностей як чорний ящик, якому потрібно лише повідомити монітор про виникнення несправності. На додаток до стандартної обробки помилок операційної системи, ми використовуємо додаткові механізми для виявлення помилок пам'яті.

Існує ряд доступних компонентів виявлення несправностей, які можуть виявляти помилки пам’яті (наприклад, ProPolice [9] і TaintCheck [12]) і деякі, які виявляють порушення основних політик безпеки [1,11]. Для цілей нашої системи ми використовуємо два компоненти виявлення несправностей, які були розроблені раніше [19,20], які пропонують компроміс між накладними витратами на продуктивність і діапазоном помилок, які вони можуть виявити. Для цієї реалізації ми припускаємо наявність вихідного коду, але плануємо розглянути можливість застосування до комерційного готового програмного забезпечення (COTS) у майбутній роботі.

Щоб виявити можливі резервні точки, система створює профіль програми перед розгортанням за допомогою динамічного аналізу з фаззингом. Ситуація полягає в тому, що існує набір тестових точок програми (заданих програмістами), які регулярно використовуються для обробки очікуваних помилок, які можна виявити, дізнавшись, як програма реагує на

«поганий» ввід у контрольованих умовах.

Наприклад, ми хотіли б побачити, як програма зазвичай обробляє помилки під час стрес-тестування за допомогою тестів забезпечення якості. Потім ці знання використовуються в майбутньому, щоб зіставити раніше невидимі несправності з набором спостережуваних несправностей.

Інструменти програмної системи в автономному режимі для виявлення потенційних точок порятунку вставляють код моніторингу в точки входу та виходу кожної функції, використовуючи можливості Dyninst [5], що є інструментом бінарної ін’єкції під час виконання. Інструменти записують значення, що повертаються, параметри функції та типи повернення (останні два доступні лише тоді, коли двійковий файл не видалено), у той час як програма бомбардується помилками (через ін’єкцію помилок) і нечіткими введеннями (наприклад, неправильно сформовані запити протоколу). З цих трасувань система витягує графіки викликів функцій разом із історією повернених значень, які використовуються в кожній точці на графіку. Ми називаємо ці графіки трасуванням точок відновлення.

Рисунок 2.5 ілюструє частину трасування відновлення для прикладу в розділі 2.4. Він показує зведене трасування виконання, яке включає три функції: ap_proxy_ftp_handler(), ap_pass_brigade() і ap_roxy_send_dir_filter(), а також спостережувані значення помилок, пов’язані з кожною функцією, які становлять 502 (код помилки), 0 і 0 відповідно.

Система постійно контролює виконання програми у робочій системі, щоб виявити збої програми та неправильну поведінку, а також записує достатньо інформації про помилку, щоб її можна було відтворити для визначення відповідної точки відновлення. Щоб виявити збої та неправильну поведінку, система використовує різноманітні механізми виявлення несправностей. Він не прив’язаний до будь-якого конкретного механізму виявлення несправностей і сумісний з будь-яким таким механізмом, який просто сповіщає систему про виникнення несправності.

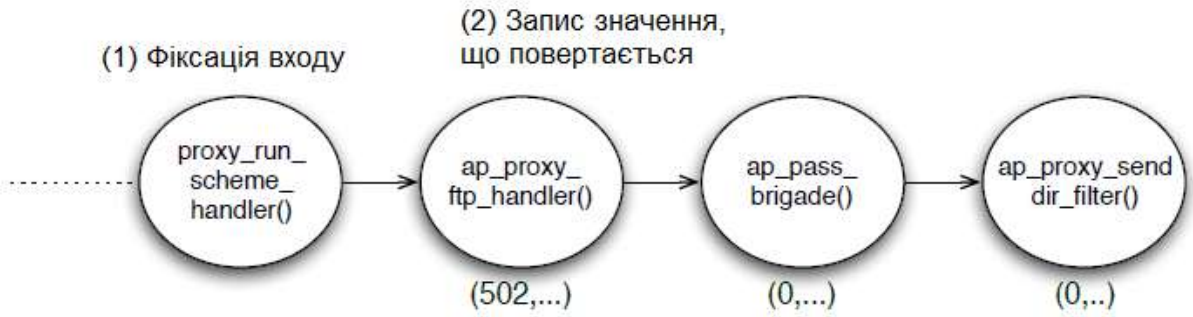

Рисунок 2.5 – Створення траси відновлення

У прикладі в розділі 2.4, індикація порушення сегментації (SIGSEGV) використовувалася як сигнал про неправильну обробку пам'яті. Окрім стандартної обробки помилок операційної системи (наприклад, завершення програми через незаконне звернення до пам'яті), система може використовувати додаткові механізми виявлення помилок пам'яті.

Існує кілька доступних компонентів виявлення несправностей, які можуть виявляти помилки пам'яті, наприклад ProPolice [10], ASLR [20] і TaintCheck [16] і деякі, які можуть виявляти порушення політик безпеки [1,12]. Щоб відтворити помилку, система використовує журнал виконання та періодичні контрольні точки для запису та відтворення послідовності подій, які призвели до прояву помилки. Ми представляємо лише огляд механізму реєстрації через обмеження простору; детальний опис виходить за рамки цієї роботи. Механізм контрольних точок такий самий, як і для реалізації точок порятунку, як описано вище.

Система записує всі вхідні дані (між контрольними точками) до процесів додатків, щоб їх можна було відтворити як для багатоядерних, так і для багатопроцесорних середовищ. Система враховує недетерміноване виконання, точно записуючи всі форми взаємодії між процесами та середовищем їх виконання та, у свою чергу, здатне точно їх відтворювати. Майже всі такі взаємодії включають системні виклики, які можна розділити на дві широкі категорії: з і без побічних ефектів. Системні виклики без побічних ефектів, наприклад getpid і gettimeofday, не потребують повторного

виконання. Їх ефект можна імітувати шляхом перехоплення їхнього значення, що повертається. Навпаки, системні виклики з побічними ефектами (наприклад, brk і fork) потрібно відтворити, щоб отримати бажаний ефект. Серед останніх також системні виклики, такі як конвеєр (pipe) і write, чий ефект може бути видимим після завершення повтору та переходу системи в «живий режим».

З кількома процесами система не має повторює точний порядок планування, як у початковому виконанні; швидше, вона намагається переконатися, що системні виклики та інші події впорядковані правильно, відстежуючи їхні залежності. Система ідентифікує пов’язані системні виклики (а саме, результат одного залежить від виконання іншого) і координує їх виконання за допомогою точок зустрічі.

Порядок системних викликів відстежується під час реєстрації, а потім виконується під час відтворення. Періодичні контрольні точки в системі мають такі переваги. По-перше, вони надають знімок стану програми, який у поєднанні з журналом виконання може відтворити стан програми, коли сталася помилка. Це критична вимога для етапу аналізу, щоб відтворити несправність і згодом забезпечити її усунення. По-друге, він обмежує розмір журналу виконання, який потрібно підтримувати; система відстежує лише виконання, яке відбулося з моменту останньої контрольної точки. По-третє, це мінімізує час, необхідний для відтворення несправності. Системі просто потрібно відтворити запис виконання з останньої контрольної точки. Швидке відтворення помилок є життєво важливим для системи, оскільки воно зменшує вікно вразливості.

2.6 Вибір точки відновлення

Здатність ідентифікувати та, що більш важливо, відтворювати несправності дозволяє нам вибрати найбільш відповідну точку відновлення для кожної виявленої несправності. Коли вперше виявляється помилка в

певній області коду, перевіряється стек викликів, виявляється послідовність функцій, яка призвели до несправності. На у цій точці система порівнює стек викликів із трасуванням відновлення з фази виявлення, щоб отримати спільні вузли. Загальні вузли утворюють набір кандидатів на відновлення, або граф відновлення.

Якщо стек викликів пошкоджений, як у випадку переповнення буфера, він відтворюється під час повторного відтворення вхідних даних, які призвели до збою.

Після визначення потенційних точок відновлення, система намагається визначити їхній тип повернення. Якщо інформація про налагодження доступна, типи повернення функції можна отримати безпосередньо з двійкового файлу. У випадку з розібраними двійковими файлами, як у випадку з більшістю комерційних стандартних програм (COTS), система оцінює фактичний тип повернення функції за допомогою набору евристик, які працюють із спостережуваними значеннями повернення, знайденими в трасах профілювання та за допомогою бінарного аналізу.

Кандидати на точки відновлення фільтруються відповідно до евристик, які враховують як тип повернення (якщо доступний), так і спостережувані значення повернення. Наразі можливі точки відновлення – це функції з типами повернення без вказівників або функції, які повертають покажчики (pointers), але спостережуване значення повернення є null. Функції, які повертають покажчики, вимагають глибшої перевірки структур даних, щоб визначити значення їхніх типів повернення за межами випадків повернення null. Попереднє емпіричне дослідження показує, що обстежувані C програми віддають перевагу використанню цілочисельних типів повернення як індикаторів помилок.

Далі система перевіряє розподіл повернених значень, знайдений у кожній можливій точці відновлення. Мета полягає в тому, щоб знайти значення, яке компонент віртуалізації помилок може використовувати для запуску коду обробки помилок.

Очевидною стратегією є використання значення, що повертається найчастіше, враховуючи, що запуски профілювання складаються з трас виконання, які поширюють помилки. Це особливо актуально за відсутності вихідного коду, коли система не може перевірити, як фактично обробляється код помилки.

Рисунок 2.6 ілюструє, як ідентифікуються можливі точки відновлення для певної несправності. При виявленні несправності в ap_proxy_send_dir filter(), перевіряється стек викликів, щоб визначити шлях виконання, який призвів до спостережуваної помилки.

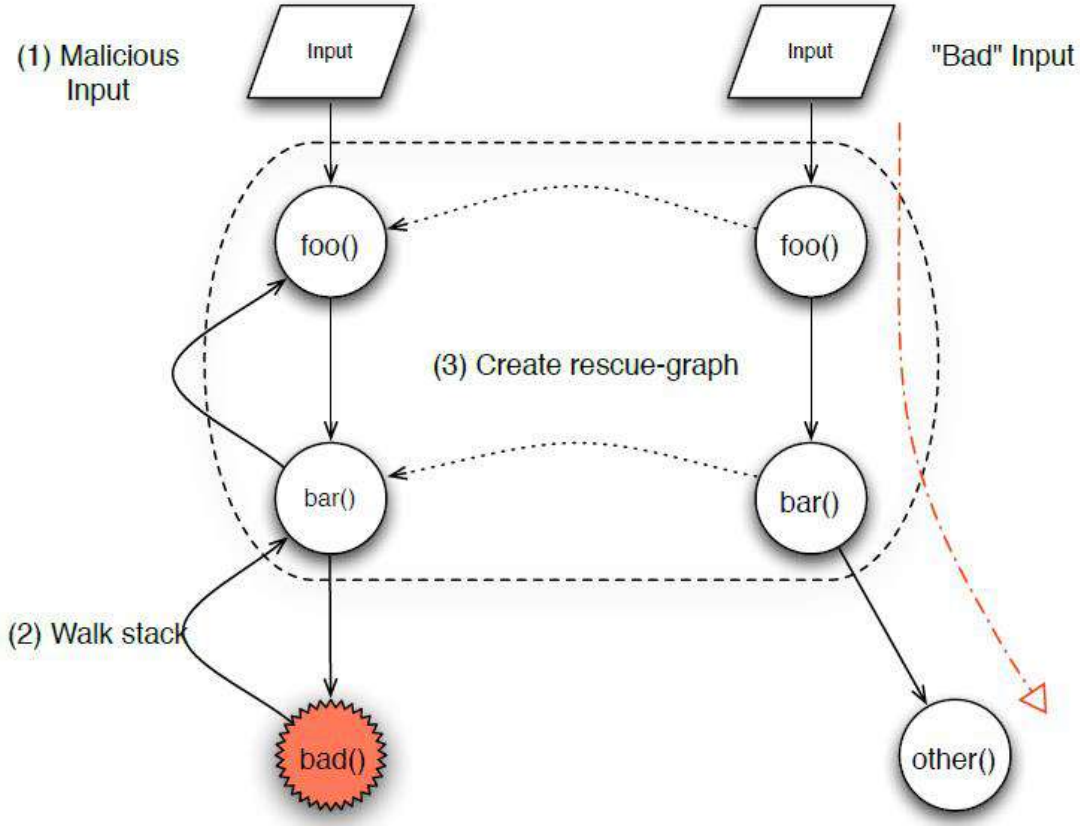

Рисунок 2.6 – Створення графу відновлення

Цей шлях порівнюється зі слідом порятунку щоб визначити функції, що перекриваються, які формують rescue-graph. Користуючись тим же прикладом з рисунку, функції ap_ proxy_ftp_handler() і ap_pass_bigads() формують граф відновлення для конкретного випадку несправності.

Точки порятунку в трасі порятунку можна сортувати за допомогою

різних стратегій вибору. Ми вибрали, мабуть, найпростіший: відсортувати точки порятунку за найкоротшою відстанню до несправного коду, який представляє активну функцію на графі викликів. Ідея полягає в тому, що відповідні точки відновлення, розташовані ближче до несправності, мінімізують накладні витрати на продуктивність, які несуть точки відновлення (через контрольні точки та моніторинг конкретної помилки), оскільки вони можуть уникнути критичних шляхів додатків, які викликаються під час кожного запиту. Ще одна причина для мінімізації відстані між несправністю та точкою відновлення стає очевидною під час роботи з багатопроцесними/багатопоточними програмами. А саме, коротка відстань мінімізує кількість прогресу, який виконують безпомилкові процеси/потоки, таким чином мінімізуючи обсяг роботи, яку потрібно буде повернути у разі збою. Крім того, це зменшує ймовірність того, що протягом цього часу відбулося б будь-яке зовнішній видимий обмін даними. Ми вимірюємо глибину відновлення та відстань від відновлювання до несправності.

Система дотримується цього порядку, щоб створити екземпляр і перевірити точки відновлення, шукаючи ту, яка дозволяє відновлення після даної несправності. Якщо найближча точка порятунку не проходить тест, система вибирає найближчого активного предка та повторює процес.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ПРОВЕДЕННЯ ЕКСПЕРИМЕНТІВ

3.1 Створення контрольних точок відновлення

Визначивши набір потенційних точок відновлення, система тепер може активувати та тестувати ці точки. Щоб активувати точку відновлення, система має вставити код у програму, що працює в середовищі тестування. Це робиться за допомогою того самого механізму для розгортання точки відновлення на робочому сервері. Використовуючи цей механізм, система активує точку відновлення, вставляючи у функцію, призначену як точка відновлення, виклик int rescue_capture(id,fault) як показано в лістингу 3.1.

Лістинг 3.1 – Захват точки відновлення

```
int rescue_point (int id, fault_t fault) {
     int rid = rescue_capture (id, fault);
     if (rid < 0)
          handle_error(id); /* rescue point error */
     else if (rid == 0)
          return get_rescue_return_value(fault);
     /* all ok */
     ...
}
```

Параметр id однозначно визначає точку відновлення; несправність – це структура, яка містить всю додаткову інформацію, що стосується точки відновлення, включаючи код віртуалізації помилок, який буде використано для примусового повернення. У нашому прикладі цей виклик вставлений в ap_proxy_ftp_handler().

Rescue_capture() функція відповідає за фіксацію стану програми під час її виконання через точку відновлення шляхом виконання контрольної точки. Контрольні точки повністю зберігаються в пам’яті за допомогою стандартної семантики копіювання при записі та індексуються відповідними

ідентифікаторами. Rescue_capture() повертає ідентифікатор точки відновлення після успішної контрольної точки або нуль, коли повертається після відкоту стану програми. Типову послідовність викликів наведено у наступному фрагменті коду. Схожу семантику має функція fork(). Значення, що повертається функцією rescue_capture() керує контекстом виконання.

Щоб підтримувати контрольну точку відкату спільних процесів, система розміщує програми у віртуальному середовищі виконання на основі Zap [14,19]. Розроблений на основі Zap, движок системи використовує стандартний інтерфейс між програмами та ОС для прозорої інкапсуляції програм у віртуальному просторі імен. Це важливо для підтримки можливості безперервного встановлення контрольних точок і подальшого відкату багатопроцесорних програм, дозволяючи їм використовувати ті самі імена ресурсів ОС, які використовувалися до встановлення контрольних точок, навіть якщо вони зіставляються з іншими основними ресурсами ОС після відкату.

Точка відновлення повинна задовольняти двом ключовим вимогам. По-перше, вона повинна забезпечувати скоординовану та послідовну контрольну точку кількох процесів і потоків та середовища їх виконання; це суттєво відрізняється від простого встановлення контрольних точок окремого процесу. По-друге, це повинно мати мінімальний вплив на продуктивність програми. Для задоволення цих вимог система використовує глобально узгоджену контрольну точку для всіх процесів (і потоків) програми, коли вони зупинені, щоб нічого не могло змінитися, але потім мінімізує тип і вартість необхідних операцій.

Ключовою проблемою багатопроцесних (і багатопоточних) програм є те, що контрольні точки завжди ініціюються процесом, оскільки вони мають відбуватися у визначених безпечних місцях. Оскільки процеси спільно використовують стан і середовище виконання, вони повинні погоджувати стан у будь-який момент часу. Однак, коли процес досягає точки відновлення, йому, як правило, доведеться чекати значний проміжок часу,

поки інші процеси також досягнуть відповідного місця. Замість цього наша система використовує привілейований процес поза середовищем виконання для виконання узгодженої контрольної точки всієї програми.

Система зберігає контрольні точки в основній пам’яті, усуваючи необхідність записувати дані на диск. Це зменшує час на контрольній точці завдяки копіюванню блоків пам’яті, а також обсяг пам’яті, необхідний для контрольної точки, завдяки використанню методів копіювання та запису. Контрольні точки пов’язані з контекстом, який ідентифікує відповідну точку відновлення та процес. Кілька контрольних точок можуть співіснувати для однієї точки відновлення або для одного процесу. Сфера застосування контрольної точки є дійсною, доки виконання не повернеться до нормального стану або після відкату, після чого контрольна точка скидається.

Зберігання контрольних точок повністю в пам’яті дозволяє нам не тільки зберігати стан ресурсу, але й зберігати посилання до нього. Система використовує це для збереження вибраних ресурсів як є через відкат програми замість відновлення попереднього стану. Зокрема, система використовує це, щоб усунути необхідність скинути з’єднання між клієнтом і програмою після відкоту, зберігаючи основний сокет без змін. Це особливо корисно для служб, орієнтованих на з’єднання, і для процесів, відмінних від того, у якому сталася помилка.

3.2 Тестування та розгортання точки відновлення

Після вибору потенційної точки відновлення, система переходить до перевірки ефективності запропонованого виправлення шляхом тестування версії програми з підтримкою відновлення. Щоб досягти цього, система перезапускає програму з останнього зображення контрольної точки, доступного в окремому середовищі тестування, а потім відтворює записаний журнал виконання, який спричинив помилку. Коли виникає несправність і

запускається відкат до вибраної точки відновлення, перевіряється її вплив на виконання програми. Якщо програма виходить з ладу, не підтримує доступність служби або не є семантично еквівалентною, створюється нове виправлення з використанням наступної доступної точки відновлення, а етап тестування й аналізу повторюється.

Якщо виправлення не викликає жодних помилок, які призводять до збою програми, програма перевіряється на наявність семантичних помилок за допомогою набору тестів, наданих користувачем. Мета цих тестів — підвищити впевненість у семантичній правильності згенерованого виправлення. Наприклад, онлайн-продавець може запустити тести, які підтвердять, що клієнтські замовлення можуть надсилатися та оброблятися системою. Нарешті, наслідки нашого виправлення для продуктивності під час виконання перевіряються, щоб забезпечити прийнятні робочі характеристики.

Для нашого початкового підходу ми в першу чергу стурбовані відмовами, де існує однозначна відповідність між входами та відмовами, а не тими, які спричинені комбінацією вхідних даних. Зауважте, однак, що багато з останніх типів збоїв насправді вирішуються нашою системою, оскільки останній вхід (і код, що призводить до збою) розпізнається як «проблемний» і обробляється, як ми обговорювали.

Коли ми маємо точку відновлення, ми хочемо без затримки створити її екземпляр у робочій системі. Швидке розгортання виправлень має першочергове значення в «реактивних» системах. По-перше, це зменшує час простою системи, а згодом підвищує доступність системи. По-друге, це дозволяє розгортати критичні виправлення, які можуть зупинити поширення широкомасштабних епідемій, як у випадку з хробаками. Попередня робота покладалася на традиційний цикл розробки програмного забезпечення, в якому вносили зміни у вихідний код (хоча й автоматично через перетворення джерела в джерело), компілювали, зв’язували, тестували та створювали нову версію програми. Для нашого механізму розгортання ми використовуємо

Dyninst [5] завдяки низьким накладним витратам часу виконання та здатності приєднуватися та від’єднуватися від уже запущених процесів. Зауважте, що крім використання для остаточного розгортання виправлення точки відновлення на робочому сервері, той самий механізм ін’єкції під час виконання також використовується для вставки точок відновлення в тіньове розгортання програми під час тестування точки відновлення, а також для впровадження механізму моніторингу помилок у робочий сервер.

3.3 Проведення експериментів

Ми впровадили прототип системи для Linux. Він складається з утиліт у просторі користувача та завантажувальних модулів ядра для стандартного ядра Linux, які забезпечують віртуальне середовище виконання з контрольною точкою перезапуску та відтворення журналу, а також Dyninst 5.2b3 для впровадження коду під час виконання. Використовуючи цей прототип, ми оцінюємо ефективність системи щодо реальних помилок і стандартних робочих навантажень для низки популярних багатопроцесних і багатопоточних серверних програм. Для всіх експериментів процес був повністю автоматизованим, за винятком генерування інформації профілю та ініціювання помилки. Процес профілювання має відбуватися один раз для кожної програми (або надаватися як частина тестового пакету). Усі експерименти проводилися на машинах із двома процесорами Intel Xeon 3,06 ГГц і 4 ГБ оперативної пам’яті, підключено через з’єднання Ethernet 1 Гбіт/с.

Сервери та клієнти працювали на окремих машинах. Ми оцінюємо ефективність системи у вирішенні помилок за трьома напрямками: живучість, правильність і продуктивність.

Живучість перевіряє здатність системи підтримувати доступність служби за наявності збою програмного забезпечення, спричиненого помилкою. Система виявляє збої та автоматично починає процес відновлення. Після відновлення ми відстежуємо сервер на наявність збоїв, які

могли бути спричинені нашим механізмом, і перевіряємо, чи сервер продовжує правильно обслуговувати запити. Оскільки можливо, що механізм відновлення вносить побічні ефекти, ми перевіряємо правильність вихідних даних сервера після відновлення: ми не тільки перевіряємо здатність сервера надавати послуги, але й порівнюємо вихідні дані сервера з попередньо визначеними наборами тестів, щоб підтвердити твердження про семантичну еквівалентність. Нарешті, ми розглянемо різні продуктивності системи з точки зору як повних системних накладних витрат на робочому сервері, так і часткової перевірки компонентів системи.

Таблиця 3.1 містить список помилок і вразливостей, які ми використовували для оцінки системи.

Таблиця 3.1 – Список помилок, використаних в оцінці системи

Програма	Версія	Помилка	Код помилки	Значення	Test
Apache	2.0.59	Переповнення буферу	CVE-2004-0940	NULL	httperf-0.8
Apache	2.2	NULL розіменування	ASF 40733	502	httperf-0.8
Apache	2.4	Вихід за межі	CVE-2006-3747	-1	httperf-0.8
ISC Bind	9.4.2	Вхідних даних	CAN-2002-1220	-1	dnsperf 1.0.0.1
MySQL	8.0	Переповнення буферу	CAN-2002-1373	1	sql-bench 2.15
Squid	5.8	Вхідних даних	CVE-2005-3258	void	WebStone 2.5b3
OpenLDAP	2.6.3	Доступу	CVE-2008-0658	80	DirectoryMark 1.3
PostgreSQL	12.14	Вхідних даних	CVE-2005-0246	0	BenchmarkSQL 2.3.2

Ми використали вісім помилок для шести популярних програм: Apache, названий (ISC Bind), MySQL, Squid, OpenLDAP і PostgreSQL. Помилки варіюються від незаконного розіменування пам’яті до одноразових

помилок і переповнення буфера, як зазначено в стовпці помилка в таблиці 3.1. Ми створюємо помилки, використовуючи існуючий або спеціально створений код експлойту на основі інформації, отриманої з онлайнових баз даних вразливостей. У нас не було доступних помилок для програм Linux із закритим вихідним кодом, оскільки більшість популярних програм Linux є відкритими. Хоча наше дослідження складається з програм із відкритим вихідним кодом, їх розглядали як комерційні готові (COTS), видаляючи двійкові файли всіх символів і видаляючи доступ до вихідного коду.

Таблиця 3.1 демонструє загальну ефективність запуску системи проти ряду реальних помилок і вразливостей. Для кожної помилки в таблиці вказано програму, на яку вплинула помилка, тип помилки та посилання на неї, а також тест, що використовується для перевірки правильності та вимірювання продуктивності. Для кожної перевіреної помилки системі вдалося знайти точку відновлення, яка дозволяє програмі пережити спричинений збій.

Докладніше, помилки запускаються під час виконання тесту для вимірювання відновлення, коли програма завантажується. Программа відстежується, щоб перевірити її здатність успішно пройти контрольний тест. Якщо тест завершується, ми маємо міру живучості та продуктивності. На цьому етапі програма перевіряється на правильність або шляхом перевірки результатів порівняльного тесту (якщо вони повідомляють про правильність), або через додаткові тести, які перевіряють і порівнюють вихідні дані з очікуваним набором результатів. Для кожної помилки ми повідомляємо глибину відновлення та значення відновлення: відстань між несправністю та точкою відновлення та значення віртуалізації помилки, яке використовується для поширення помилок.

Важливі також і глибина відновлення та значення відновлення відповідно для кожної помилки. Середня спостережувана глибина відкату для помилок становить 2. Як згадувалося раніше, ми оцінюємо точки порятунку на живучість, правильність і продуктивність. У випадку MySQL,

система знайшла точку відновлення на глибині відновлення 1, що дозволило програмі пройти тести на живучість і коректність, але саме точка відновлення на глибині 2 забезпечила кращі характеристики продуктивності.

Причина полягала в тому, що точка порятунку на глибині 2 дозволяла виконати еталонний тест без запуску надмірних контрольних точок. Наша система тестування змогла автоматично визначити таку поведінку.

Коротка глибина відновлення є обнадійливою, оскільки вона вказує на те, що точки відновлення мають тенденцію згруповуватися поблизу несправностей, мінімізуючи їхній вплив на продуктивність системи. Для багатопроцесорних (або багатопоточних) серверів це також означає, що кількість процесів (або потоків), відмінних від того, у якому виникла помилка, обмежена, і тому їх відкат з меншою ймовірністю спричинить побічний збиток. Наприклад, невелика глибина відновлення може зменшити ймовірність того, що будь-який видимий для клієнта зв’язок відбудеться між контрольною точкою та відкатом. Рисунок 3.1 представляє відстань між контрольною точкою та відкатом у мілісекундах.

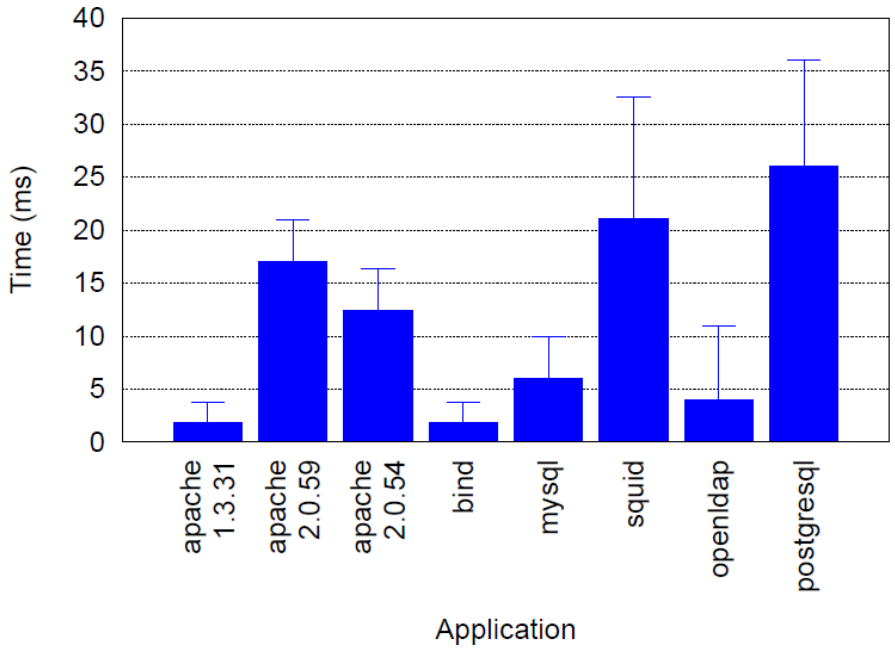

Рисунок 3.1 – Відстань між контрольною точкою та відкатом

Зокрема, ми вимірюємо час між взяттям контрольної точки та наступним відкатом. Смужки помилок показують середній час затримки між поданням команди контрольної точки/відкату та завершенням. Детально вони вказують середній час від завершення контрольної точки до продовження виконання та час, що минув від початку відкату до припинення активності старих процесів. Значення, показані на графіку, коливаються від 1,8 мс для Apache 1.3 до 26 мс у випадку PostgresSQL.

Для більшості колишніх оброблених програм, записаний час від контрольної точки до помилки представляє менше одного запиту, таким чином мінімізуючи вплив на прогрес, досягнутий іншими процесами/потоками.

Діапазон повернених значень, що використовуються точками порятунку, показує ступінь кореляції з раніше спостережуваними результатами [27]. Значення 0 і -1 часто використовуються для розповсюдження помилок, але є випадки, як-от у Apache і openLDAP, де спостережувані значення 502 і 80 відповідно є більш відповідними значеннями для повернення.

3.4 Оцінка продуктивності

Щоб оцінити швидкість реагування системи на створення виправлення для нововиявленої помилки, ми виміряли загальний час, необхідний для переходу від помилки до виправлення. Іншими словами, від моменту першого виявлення несправності у виробничій системі до динамічного застосування виправлення.

Рисунок 3.2 показує середній час у секундах для створення робочого, перевіреного патча для кожної з помилок. Вказується загальний час, необхідний для створення, тестування та застосування патча. Загальний час розбивається на дві частини: час, необхідний для підключення до запущеного процесу та введення точки відновлення за допомогою інструментів системи, і

тестовий час, необхідний для виконання тесту живучості, правильності та продуктивності в середньому для успішної точки відновлення. Зауважемо, що час, необхідний для створення графіка відновлення для конкретної помилки, незначний, враховуючи, що траса стека була глибиною менше 15 функцій у всіх випадках.

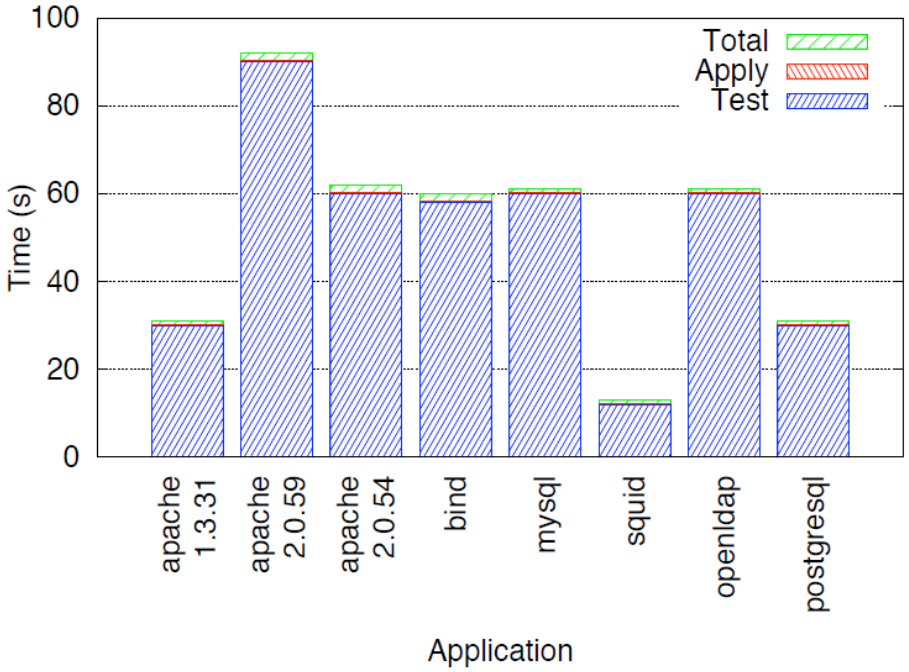

Рисунок 3.2 – Час створення патча

Як показано на рисунку 3.2, середній загальний час генерації патча коливався від 15 до 92 секунд. Для певної програми та набору тестів на правильність загальний час був приблизно лінійним, залежно від глибини відновлення, оскільки тести повторюються для кожної потенційної точки відновлення. Ці часи є консервативними з двох сторін. По-перше, наш прототип дозволяє завершити набір тестів на правильність, перш ніж перейти до наступного кандидата в точку відновлення. Це можна оптимізувати, відхиливши невідповідну точку відновлення під час невдачі тесту, а не аналізуючи результати після завершення тесту. По-друге, наш прототип серіалізує процес вибору та тестування точки порятунку. З типовою глибиною порятунку щонайбільше 3, це можна оптимізувати шляхом

простого паралельного тестування точок відновлення. Також можна виконати інше розпаралелювання, наприклад паралельне виконання різних тестів для певної точки відновлення.

Розподіл загального часу показує, що тестування на живучість і правильність є домінуючим фактором у наскрізній затримці процесу створення виправлення. Для цієї оцінки ми використали набори тестів, які стрес-тестують програми, щоб виявити довгострокові побічні ефекти, такі як витік пам’яті. На практиці організації, які розгортають систему відновлення, можуть мінімізувати тестування, щоб воно охоплювало основні функції, і таким чином скоротити час, необхідний для тестування кожної точки відновлення. Крім того, якщо вони стурбовані правильністю, можна використовувати більш комплексні тести, забезпечуючи компроміс між часом створення виправлення та покриттям тестування.

Час, необхідний ASSURE для створення та динамічного застосування виправлення точки відновлення, становив від 70 мс до Apache 1.3.1 до 120 мс для MySQL. Основний тягар цих витрат полягає в завантаженні та розборі бібліотеки інструментів системи у образ виконання сервера. Цифри вказують на значні покращення в порівнянні з традиційним циклом виправлення, компіляції, зупинки та перезапуску.

Хоча ці результати представляють неоптимізовану реалізацію, вони показують час виконання виправлень, який на порядки швидший, ніж виправлення, створені вручну. За даними Symantec, середній час між виявленням критичної помилки пам’яті та наступним виправленням становить 28 днів [30]. Слід зазначити, що метою системи є не замінити процес створення виправлень, а радше додати проміжний параметр, який адміністратори можуть використовувати для підвищення доступності системи, очікуючи на створені вручну та ретельно перевірені виправлення. Фактично, процес тестування системи може бути використаний творцями патчів для розширення свого існуючого портфоліо.

3.5 Оцінка ефективності відновлення

Для кожної помилки ми оцінюємо ефективність відновлення після несправності. Зокрема, ми вимірюємо час відновлення стану програми до точки відновлення після виявлення несправності. Як і в попередніх експериментах, помилка виникла, коли програма була зайнята виконанням заданих тестів для вимірювання відновлення під навантаженням.

Ми порівняли час відновлення системи із часом перезапуску всієї програми після збою, у якому ми виміряли час, що минув від запуску програми до моменту, коли вона стане працездатною та готовою обслуговувати запити. Повний перезапуск програми не обов’язково дозволяє відновлення, але це може скинути сервер, щоб він міг обслуговувати майбутні запити, навіть якщо він не дозволяє завершити робоче навантаження, яке виконувалося на момент помилки. Хоча він не забезпечує такого ж рівня живучості, як наша система, він забезпечує корисне порівняння часу відновлення. Зауважте, що це порівняння є консервативним, оскільки більшість серверів накопичують стан у виділених кешах, щоб значно покращити свою продуктивність; наші вимірювання не фіксують негативний вплив перезапуску всієї програми на продуктивність через ефективне відкидання цього стану та кешу. Щоб виміряти реалістичний час перезапуску додатка, ми тестуємо перезапуск додатка за допомогою реальних навантажень. Для PostgreSQL ми вимірюємо час, необхідний для перезапуску програми, коли в ній попередньо завантажено набір даних Wisconsin. Для OpenLDAP рисунок 3.3 показує середній час, необхідний для відновлення виконання до точки відновлення, або, іншими словами, відкат колишнього виконання.

Як показано, час перезапуску становив від 135 мс для Apache 1.3, до 388 мс для Postgres. Ціла заявка час перезапуску коливався від 470 мс для прив’язки до 5 секунд для Squid. Ці результати показують, що час відновлення нашої системи на порядки швидше (4x-23x), ніж перезапустити

усю програму. Це справедливо навіть для програм, таких як Apache, якому не потрібно відновлювати значні обсяги до того, як вони стануть функціональними. Зауважимо, що перезапуск програми може зайняти більше часу після помилки, ніж запуск програми в чистій системі через перевірки програма може працювати в результаті збою. У випадку PostgreSQL, повний перезапуск сервера був навіть невдалим у дозволі серверу обслуговувати майбутні запити після помилки.

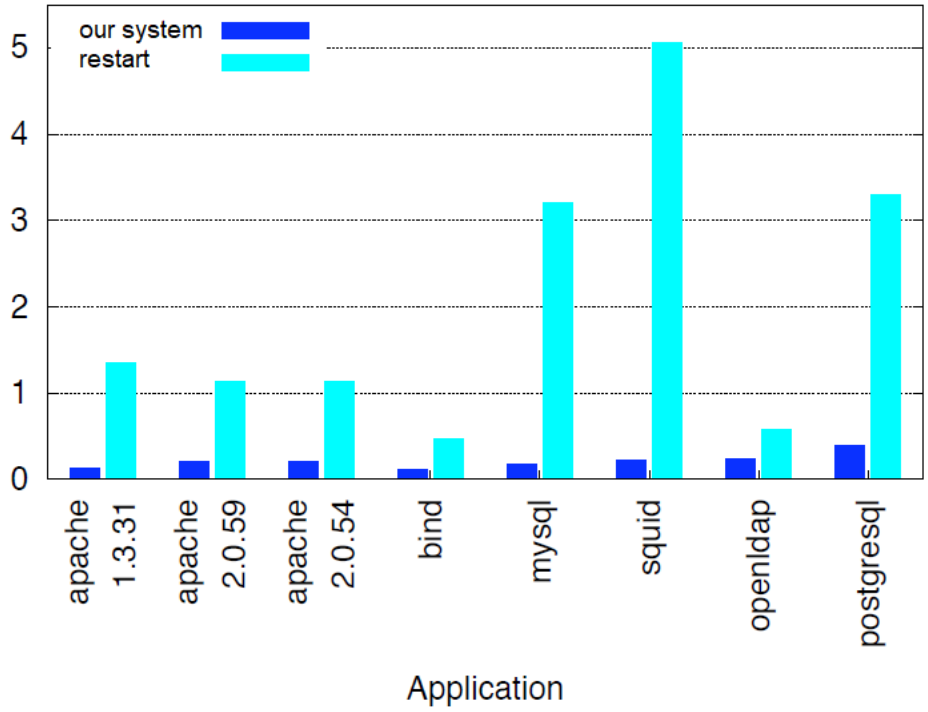

Рисунок 3.3 – Час відновлення

Через помилку тест, який виконувався в фону не вдалося успішно завершити через пошкодження даних. Сервер не працював через пошкодження даних запити на пошкоджені частини бази даних, тому тест навіть не можна було повторно запустити після перезапуску сервера. Навпаки, наша система дозволила програмі успішно завершити тестування, коли сталася помилка.

Щоб оцінити сприйняту клієнтом доступність, ми перевірили кількість помічених клієнтом помилок внаслідок усунення несправностей.

Зокрема, ми впровадили помилки під час виконання тесту, і виміряли

кількість розірваних з'єднань і запитів без відповіді як частину загальної кількості запитів. Ми реалізували різноманітне введення несправності з інтервалами в 10, 20 і 30 секунд. Значення варіюються від 1% до 10% для несправності кожні 30 і 10 секунд відповідно.

Враховуючи отримані успіхи у пошуку точок відновлення, які дозволяють системі відновлювати виконання після введених помилок, ми хотіли вивчити наслідки продуктивності нашого «виправлення». Зокрема, для кожної помилки ми досліджували наслідки наших виправлень на продуктивність системи. Порівняємо виконання продуктивність немодифікованої версії програми порівняно з патчем, створеним системой за допомогою попередньо описаних контрольних тосок. Ми також вимірюємо продуктивність накладних витрат на ініціювання збою під час виконання еталону. Результати, як нормовані показники накладних витрат, показані на рисунку 3.4.

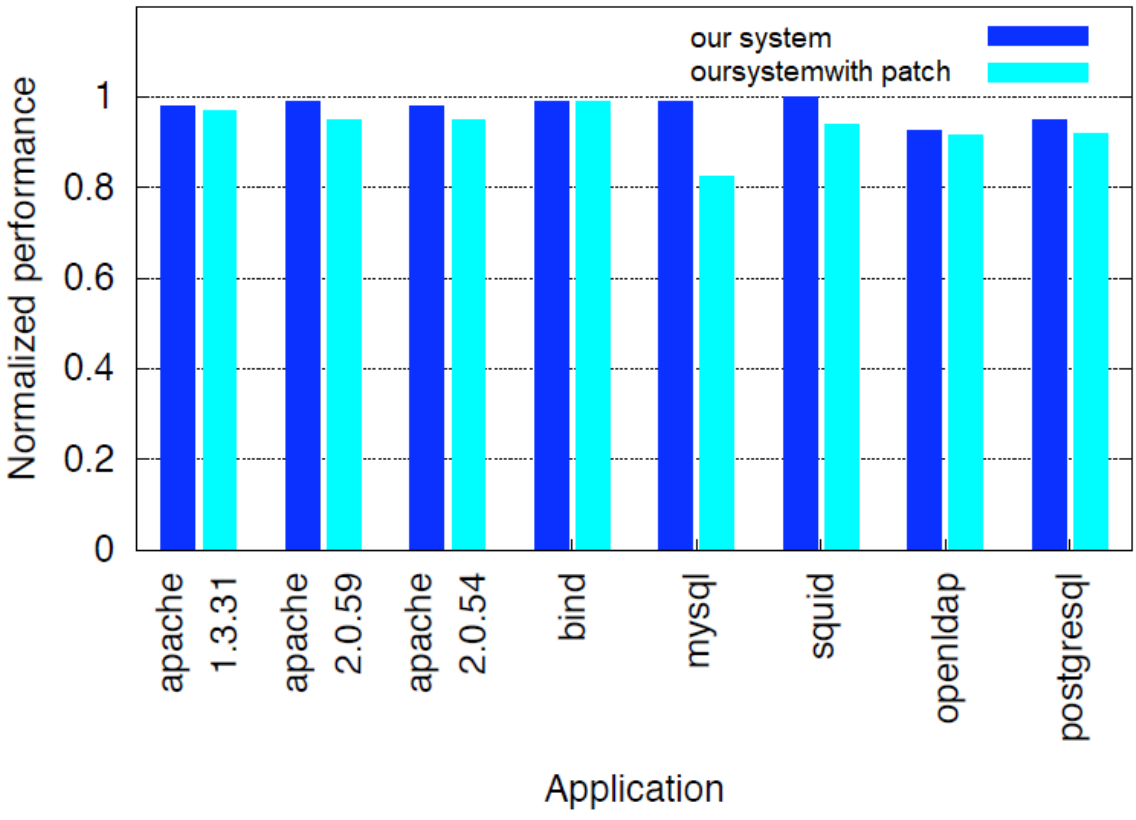

Рисунок 3.4 – Нормована продуктивність

Як показано на рисунку 3.4, наша система має мінімальний вплив на продуктивність. Діапазон від 0% для Squid до 7,6% для OpenLDAP. Ці результати очікуються з двох причин. По-перше, віртуалізація і накладні витрати на прилади невеликі, подібно до того, що повідомляється у попередній роботі [14]. По-друге, для всіх перевірених помилок, збої відбуваються в областях коду, які не є основними шляхами виконання програми. Це дозволяє системі не обтяжувати витрати на відновлення, хіба що це вразливий шлях коду, який використовує програма.

3.6 Аналіз результатів експериментів

Однією з найважливіших проблем, пов’язаних із відновленням після збоїв програмного забезпечення та впливу вразливостей, є забезпечення узгодженості та правильності даних і стану програми. Це проблема, яка присутня в більшості спроб відновлення. Наявність точок відновлення, створених автоматично чи за допомогою програміста, може полегшити більшість проблем із непередбачуваними шляхами виконання, але, на жаль, не повністю їх відкинути. Розглядаємо переваги та недоліки відновлення після помилок загалом і віртуалізації помилок зокрема.

Реакція на несправність: механізм усунення несправностей повинен оцінювати та вибирати відповідь із широкого спектру варіантів. Наразі, коли виникає помилка, система може вибрати один із таких варіантів: збій [9], збій і перезапуск монітором [4,5], повернути довільні значення [16,17], змінити середовище та відтворити [14], відрізати функціональність [19,20] або стрибнути до безпечної точки та примусово помилитися.

Попередні підходи були зосереджені на методах, заснованих на збоях, працюючи в припущенні про відсутність прийнятної альтернативи. Остання робота [17,20] показав, що існує набір альтернативних реактивних методів, які, здається, добре працюють на практиці. Ми обираємо останній підхід до відновлення виконання до безпечних точок і примусового виявлення

помилок у виконанні програми. Ранні експерименти показали, що цей вибір працює надзвичайно добре. Це явище також з’являється на рівні машинних команд [21]. Однак існує фундаментальна проблема у виборі конкретної відповіді.

Оскільки високорівнева поведінка будь-якої системи не може бути визначена алгоритмом, система повинна бути обережною, щоб уникнути випадків, коли відповідь призведе до семантично (з точки зору намірів програміста) неправильного шляху. Прикладом такого типу проблеми є пропуск реєстрації sshd що дозволить неаутентифікованому користувачеві отримати доступ до системи. Ми вважаємо, що завдяки використанню точок відновлення ми можемо мінімізувати (на жаль, не усунути) невизначеність того, що програма піде неочікуваним шляхом виконання. Причина, по якій ми можемо робити такі висновки, полягає в тому, що ми вирішили використовувати як точки відновлення позиції в програмі, які відомі й поширювати помилки. Якщо потрібен вищий рівень гарантії, можна покластися на програміста, який надасть анотації щодо того, які частини коду слід використовувати для відновлення, а які не слід обходити.

Програмування з віртуалізацією помилок: у цій роботі ми зосередились на повністю автоматизованих методах для кожного аспекту нашої системи. Однак було б нерозсудливо відмовлятися від використання допомоги програміста у відновленні програми.

Зокрема, програмісти можуть розробляти програмне забезпечення з урахуванням віртуалізації помилок, де певні місця в коді можуть бути призначені, апріорі, як точки порятунку, які плавно поширюють помилки. Інтуїцію програміста важко повторити автоматизовані методи, особливо коли мова йде про очищення та ефективність коду. Ми передбачаємо, що програмування з віртуалізацією помилок буде легшим, ніж мати справу зі специфічними для мови конструкціями, такими як обробка винятків, оскільки увагу можна зосередити на кількох вибраних пунктах програми.

Застосовність до безпечних мов: ми можимо уникнути проблем, які

намагаємося вирішити, використовуючи безпечні мовні конструкції. Але, на жаль, здається, що наявність відповідних мовних конструкцій для обробки помилок (винятків) вирішує деякі проблеми, але це далеко не панацея. Це особливо вірно для великих систем, що розвиваються, де через складність системи дуже важко охопити всі кутові випадки. Наш підхід можна застосувати до таких систем шляхом створення карти між кінцевим набором існуючих можливостей обробки помилок і нескінченним набором майбутніх додаткових можливостей системи.

Наявність: однією з головних цілей нашої роботи є скорочення часу простою системи та, як наслідок, підвищення доступності сервісу в умовах збоїв і атак. Ми очікуємо, що скорочення часу простою буде незначним; віртуалізація помилок покладається на монітори виявлення помилок, які виявляють помилки, знаходять відповідні точки відновлення, створюють виправлення та вставляють виправлення у запущену програму. Цей процес потребує деякого часу простою, але вартість амортизується, оскільки ці витрати сплачуються один раз за кожну виявлену вразливість. Поєднання нашого підходу з такими методами, як мікро перезавантаження [4,5] є темою майбутніх досліджень.

Робота з програмами несерверного типу: успіх нашої системи у відновленні виконання програми можна частково пояснити основними характеристиками типів програм, які ми перевіряємо. Як сформульовано в [17], додатки серверного типу мають тенденцію до коротких відстаней поширення помилок, і форсування помилок в одному запиті практично не впливає на майбутні запити. Хоча серверні програми можуть мати невід’ємну перевагу у поширенні помилок, ми вважаємо, що більшість програм написані з деякими можливостями обробки помилок. Правильне визначення цих точок порятунку має застосувати наш підхід до широкого спектру додатків, хоча, як згадувалося раніше, додаткам, які покладаються на цілісність своїх обчислень, може бути краще використовувати альтернативну стратегію.

ВИСНОВКИ

Розроблена система представляє точки відновлення, нове програмне забезпечення для самовідновлення техніка для виявлення, підхід до відновлення від програмних збоїв у серверних програмах. Контрольні точки – є місцями, які визначені в існуючому коді програми, де обробка помилки виконується щодо заданого набору передбачених (програмістом) збоїв. Ми використовуємо існуючі методи перевірки достовірності для створення відомих поганих вхідних даних програм, щоб визначити потенційні точки відновлення.

При першому виявленні несправності, система використовує репліку програми, щоб визначити, які точки порятунку можна використовувати найбільш ефективно для відновлення майбутнього виконання програми.

Після того, як система перевірить, чи створено виправлення, яке усуває несправність, динамічно виправляє запущене відновлення до програмних засобів, що реалізують це. Якщо помилка виникає знову, система повертає програму до контрольної точки та використовує власний вбудований програмний код обробки помилок для відновлення після несправності та правильного виправлення внутрішнього і зовнішнього стану.

Ми впровадили систему і продемонстрували її ефективність на кількох серверних програмах, включаючи веб, базу даних, сервер доменних імен та проксі-сервери. Наш експериментальні результати, як з реальними помилками, так і з ін'єкціями штучних помилок показали, що наша техніка може бути використана для відновлення виконання у більшості розглянутих випадків зі скромними операційними витратами. Використання неоптимізованого прототипу, який повністю автоматизує процес відновлення програмного забезпечення займає всього пару хвилин, а замовлення на відновлення швидше, ніж поточне розгортання усунення помилок людиною. Крім того, не вимагається наявність вихідного коду програми. Кінцевим

результатом є автоматичне відновлення програмних служб з невідомих і непередбачених програмних збоїв.

Ми окресли сферу віртуалізації помилок за допомогою точок відновлення, запропонували нову техніку самовідновлення програмного забезпечення для виявлення відновлення після помилок програмного забезпечення. Ми використовуємо цільові системи для виявлення помилок програмного забезпечення в додатку, викликаних атаками з метою використання вразливостей програмного забезпечення, і отримання кінцевого стека викликів. Це узгоджується з потенційним набором точок відновлення шляхом відкату та повторення виконання з помилкою, щоб визначити, яку точку відновлення можна використовувати для відновлення після помилки. Наша система динамічно виправляє запущену робочу програму до точки самоперевірки в точці відновлення та, якщо виникає помилка, повертається до контрольної точки і повертає відоме значення, що повертається, якщо використовується для відповіді на неправильний вхід, який використовуєтья власними вбудованими програмами у механізмах обробки помилок для відновлення після несправності.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 2005), pages 340–353, Sept. 2015.

2. J. Boyd. Patterns of Conflict. Unpublished Briefing, http: //www.d-n-i.net/boyd/pdf/poc.pdf, Dec. 2006.

3. T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault Tolerance. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 1995), pages 1–11, Dec. 1995.

4. D. Brumley, H. Wang, S. Jha, and D. Song. Creating Vulnerability Signatures Using Weakest Preconditions. In Proceedings of the 20th IEEE Computer Security Foundations Symposium, pages 311–325, July 2017.

5. B. Buck and J. K. Hollingsworth. An API For Runtime Code Patching. International Journal of High Performance Computing Applications, 14(4):317–329, Nov. 2020.

6. G. Candea and A. Fox. Crash-Only Software. In Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX), pages 12–20, May 2013.

7. S. Chandra. An Evaluation of the Recovery-Related Properties of Software Faults. PhD thesis, University of Michigan, Sept. 2000.

8. M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-To-End Containment of InternetWorms. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005), pages 133–147, Dec. 2015.

9. B. Demsky and M. Rinard. Automatic Detection and Repair of Errors In Data Structures. In Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2003), pages 78–95, Oct. 2021.

10. J. Etoh. GCC Extension for Protecting Ppplications from Stack-smashing Attacks. http://www.trl.ibm.com/projects/security/ssp/.

11. S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems With Time-Traveling Virtual Machines. In Proceedings of the 2015 USENIX Annual Technical Conference (USENIX 2015), pages 1–15, Apr. 2015.

12. V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure Execution Via Program Shepherding. In Proceedings of the 11th USENIX Security Symposium, pages 191–206, Aug. 2022.

13. N. Kolettis and N. D. Fulton. Software Rejuvenation: Analysis, Module and Applications. In Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS- 25), pages 381–395, June 2019.

14. O. Laadan and J. Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In Proceedings of the 2017 USENIX Annual Technical Conference (USENIX 2017), pages 323–336, June 2017.

15. B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. Communications of the ACM, 33(12):32–44, Dec. 1990.

16. J. Newsome, D. Brumley, and D. Song. Vulnerability–Specific Execution Filtering for Exploit Prevention on Commodity Software. In Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS 2016), pages 1–15, Feb. 2016.

17. M. Norton and D. Roelker. Snort 2.0 Protocol Flow Analyzer. Sourcefire White Paper, Apr. 2014.

18. National Vulnerability Database. http://nvd.nist.gov/statistics.cfm, April 2016.

19. S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System For Migrating Computing Environments. In Proceedings of the 15th Symposium on Operating Systems Design and Implementation (OSDI 2022), pages 361–376, Dec. 2022.

20. PaX Team. Address Space Layout Randomization, Mar. 2013.

http://pax.grsecurity.net/docs/aslr.txt.

21. V. Paxson. Bro: A System For Detecting Network Intruders In Real-Time. Computer Networks, 31(23-24):2435–2463, Dec. 2019.

22. F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs As Allergies–A Safe Method To Survive Software Failures. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2015), pages 235–248, Oct. 2015.

23. E. Rescorla. Security Holes... Who Cares? In Proceedings of the 22th USENIX Security Symposium, pages 6–20, Aug. 2013.

24. M. Rinard. Acceptability-Oriented Computing. In Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2013), pages 221–239, Oct. 2013.

25. M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2014), pages 303–316, Dec. 2014.

26. S. Sidiroglou, Y. Giovanidis, and A. Keromytis. A Dynamic Mechanism For Recovery From Buffer Overflow Attacks. In Proceedings of the 8th Information Security Conference (ISC 2015), pages 1–15, Sept. 2015.

27. S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building A Reactive Immune System For Software Services. In Proceedings of the 2015 USENIX Annual Technical Conference (USENIX 2015), pages 149–161, Apr. 2015.

28. Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo. On the Infeasibility of Modeling Polymorphic Shellcode. In Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS 2017), pages 541–551, Oct. 2017.

29. M. Sullivan and R. Chillarege. Software Defects and Their Impact on System Availability–A Study of Field Failures In Operating Systems. In

Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21), pages 2–9, June 2021.

30. Symantec. Internet Security Threat Report. http://www. symantec.com/enterprise/threatreport/index.jsp.

31. J. Tucek, J. Newsome, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song. Sweeper: A Lightweight End-To-End System For Defending Against Fast Worms. In Proceedings of the 2nd European Conference on Computer Systems (EuroSys 2017), pages 115–128, Mar. 2017.

32. H. J.Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters For Preventing Known Vulnerability Exploits. In Proceedings of the 2014 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2014), pages 193–204, Aug. 2014.