НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

До захисту допущено
Завідувач кафедри

_____ **Дмитро ЛАНДЕ**
(підпис)

«_____»_____ 2024 р.

Дипломна робота
на здобуття ступеня бакалавра
за освітньо-професійною програмою «Системи, технології та математичні методи кібербезпеки» спеціальності 125 «Кібербезпека»

на тему: «Методи аналізу недекларованих можливостей Android застосунків на основі фазингу»

Виконав: здобувач вищої освіти **IV** курсу, групи ФБ-03
(шифр групи)

Бондаренко Олексій Юрійович
(прізвище, ім'я, по батькові) (підпис)

Керівник: К.т.н., доц. Ільїн М.І.
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові) (підпис)

Рецензент: К.т.н., доц. Яковлєв С. В.
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище, ім'я, по батькові) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Здобувач вищої освіти _____
(підпис)

Київ - 2024 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 125 «Кібербезпека»
Освітньо-професійна програма «Системи, технології та математичні методи кібербезпеки»

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ Дмитро ЛАНДЕ
(підпис)

«_____» _____ 2024 р.

ЗАВДАННЯ
на дипломну роботу здобувачу вищої освіти

Бондаренко Олексію Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи: «Методи аналізу недекларованих можливостей Android застосунків на основі фазингу»,
керівник роботи Ільїн Микола Іванович, кандидат технічних наук,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

   затверджені наказом по університету від 31 травня 2024 р. № 2251-с

2. Термін подання здобувачем вищої освіти роботи 10 червня 2024 р.

3. Вихідні дані до роботи: література з аналізу недекларованих можливостей методом фазингу, наявні методи фазингу, їх реалізація та технічна документація.

4. Зміст роботи: огляд сучасних методів аналізу недекларованих можливостей на основі фазингу, реалізація моделі автоматизації фазингу Android застосунків, експериментальне дослідження запропонованої моделі аналізу.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): «Методи аналізу недекларованих можливостей Android застосунків на основі фазингу» – презентація.

6. Дата видачі завдання: 25.09.2023

Календарний план

No з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів дипломної роботи	Примітка
1	Отримання завдання та узгодження теми роботи	25.09.2023	Виконано
2	Вивчення алгоритму фазингу	25.09.2023 – 25.10.2023	Виконано
3	Огляд наукової літератури по темі	25.10.2023 – 25.11.2023	Виконано
4	Огляд наявних методів та реалізацій фазингу	25.12.2023 – 21.01.2024	Виконано
5	Налаштування та підготовка інструментів середовища аналізу	21.01.2024 – 20.02.2024	Виконано
6	Реалізація автоматизованого методу розгортання середовища аналізу	20.02.2024 - 02.04.2024	Виконано
7	Проведення аналізу тестових зразків	02.04.2024 – 08.04.2024	Виконано
8	Визначення ефективності методу фазингу	08.04.2024 – 12.04.2024	Виконано
9	Аналіз отриманих результатів	12.04.2024 – 15.04.2024	Виконано
9	Проходження переддипломної практики	15.04.2024 – 19.05.2024	Виконано
10	Фінальне оформлення дипломної роботи	19.05.2024 – 30.05.2024	Виконано
11	Передзахист дипломної роботи	10.06.2024	Виконано
12	Захист дипломної роботи	18.06.2024	Виконано

Здобувач вищої освіти _____ Олексій БОНДАРЕНКО
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

Керівник роботи _____ Микола ІЛЬЇН
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Обсяг дипломної роботи 61 сторінка, 13 ілюстрацій, 1 таблиця, 1 додаток, 42 джерела.

У роботі визначено сучасні методи проведення аналізу недекларованих можливостей Android застосунків на основі фазингу сірої скриньки.

Створено та програмно реалізовано модель автоматизації проведення фазингу нативних бібліотек Android застосунків з використанням інструмента фазингу AFL++ з FRIDA режимом для динамічного інструментування. Програмна реалізація моделі полягає у автоматизованому налаштуванні середовища аналізу шляхом динамічної компіляції всіх необхідних для фазингу компонентів у ізольованому середовищі Docker, та автоматичній підготовці досліджуваного зразка.

За допомогою запропонованої моделі було проаналізовано наявність недекларованих можливостей тестового зразка Android застосунку. Доведено успішність аналізу шляхом знаходження вразливості сегментації пам'яті за 47 секунд, після проходження гілки з 16 умовних операторів.

Досліджено залежність ефективності використання методу фазингу для пошуку недекларованих можливостей від складності досліджуваного зразка Android застосунку. Зроблено висновки про експоненційну залежність швидкості фазингу від складності застосунку.

Запропонована реалізація та отримані результати дослідження стануть корисними в системах автоматизованого аналізу застосунків для операційної системи Android на наявність недекларованих можливостей в рамках діяльності з організації та ведення кіберборотьби та застосуванні у підрозділах кіберзахисту. Зокрема дана модель була використана в рамках науково дослідницької роботи за державним оборонним замовленням.

Ключові слова: **Android, фазинг сірої скриньки, аналіз недекларованих можливостей, AFL++, Docker**

ABSTRACT

The volume of the thesis is 61 pages, 13 illustrations, 1 table, 1 appendix and 42 sources of literature.

The thesis identifies contemporary methods for analyzing undeclared capabilities of Android applications based on gray-box fuzzing.

A model for automating the fuzzing of native libraries of Android applications was created and implemented using the AFL++ fuzzing tool with FRIDA mode for dynamic instrumentation. The implementation involves the automated setup of the analysis environment by dynamically compiling all necessary components for fuzzing in an isolated Docker environment, and automatically preparing the sample under study.

Using the proposed model, the presence of undeclared capabilities in a test sample of an Android application was analyzed. The success of the analysis was demonstrated by finding a memory segmentation vulnerability in 47 seconds after passing through a branch with 16 conditional operators.

The research investigated the dependence of the effectiveness of the fuzzing method for finding undeclared capabilities on the complexity of the Android application sample. Conclusions were drawn about the exponential dependence of fuzzing speed on application complexity.

The proposed implementation and obtained research results will be useful in automated analysis systems for Android applications to detect undeclared capabilities, within the scope of organizing and conducting cyber defense activities and use in cyber protection units. Specifically, this model was utilized in the framework of a scientific research project under a state defense order.

Keywords: **Android, greybox fuzzing, analysis of undeclared capabilities, AFL++, Docker**

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ОС – Операційна система.

Мутація – Випадкова зміна генів. В контексті фазингу – випадкова зміна вхідних даних.

Схрещування – обмін генетичним матеріалом. В контексті фазингу – злиття разом вхідних даних.

Селекція – процес відбору найбільш придатних до виживання осіб. В контексті фазингу – вибір даних, що забезпечують найбільше просування в потоці виконання програми.

Фазер – програма. що виконує фазинг.

API – абревіатура з англійської мови «Application Programming Interface», метод комунікації компонентів комп'ютерної системи.'

Фреймворк – набір інструментів та бібліотек для спрощення виконання програмних операцій.

Креш (англ. crash) – несподіване завершення програми з критичною помилкою.

ВСТУП

На сьогодні, кількість застосунків для платформи Android та їх функціонал збільшується з великими темпами. Кожна людина, що має власний смартфон, зберігає на ньому велику кількість чутливої та конфіденційної інформації. Недекларовані можливості в Android застосунках можуть створювати вразливості, які будуть порушувати роботу програми та дозволятимуть зловмисникам успішно здійснювати атаки, що неодмінно призведе до порушення цілісності, конфіденційності та доступності інформації на девайсі.

Одним з привабливих методів аналізу недекларованих можливостей є фазинг. Його ефективність полягає в широкому охопленні можливих вхідних комбінацій, що допомагає виявляти вразливості, які інші методи можуть пропустити. Цей метод є динамічним та не потребує доступу до вихідного коду програми, не вимагаючи глибокого розуміння внутрішньої реалізації, що робить його використання доступним для дослідників з різним рівнем кваліфікації.

Попри велику поширеність в цілому, тема фазингу конкретно для платформи Android не є достатньо дослідженою через специфіку самої операційної системи. Тому, в даній роботі будуть розглянуті методи аналізу недекларованих можливостей Android застосунків з використанням цього методу.

Актуальність роботи: Вдосконалення методів пошуку недекларованих можливостей Android застосунків в рамках діяльності з організації та ведення кіберборотьби та застосуванні у підрозділах кіберзахисту.

Мета і завдання дослідження: Вдосконалення методів виявлення недекларованих можливостей в Android застосунках.

Мета досягається шляхом розв'язання наступних задач:

1. Проаналізувати методи фазингу, які вже існують.
2. Вдосконалити модель аналізу недекларованих можливостей Android застосунків на основі фазингу.

3. Провести експериментальне дослідження запропонованої моделі.

Об'єкти дослідження: Недекларовані можливості мобільних застосунків для операційної системи Android.

Предмет дослідження: Методи физингу сірої скриньки для задач пошуку недекларованих можливостей Android застосунків.

Методи дослідження:

1. Системний аналіз.

2. Методи об'єктно-орієнтованого програмування.

Наукова новизна одержаних результатів: Запропоновано модель аналізу недекларованих можливостей Android застосунків на основі фазингу, що знижує трудомісткість формулювання вхідних даних.

Практичне значення одержаних результатів: Запропоновану модель доведено до програмної реалізації та успішно використано в рамках науково дослідницької роботи за державним оборонним замовленням.

Апробація результатів роботи: Робота виконана в рамках науково дослідницької роботи за державним оборонним замовленням N 41 від 24.06.20 р., додаткова угода №1 від 29.12.2020 «Розроблення механізмів аналізу комп'ютерних програм та їх оновлень на відсутність не документованих функцій, навмисне використання яких може заподіяти шкоду даних, що обробляються, та комп'ютерним системами, які використовують такі програми» р/к N 0120U103181.

1 СУЧАСНІ МЕТОДИ АНАЛІЗУ НА ОСНОВІ ФАЗИНГУ

На сьогодні, тестування програмного забезпечення є одним з найбільш критичних етапів розробки, оскільки воно дозволяє виявити та виправити помилки, щоб забезпечити надійність, стабільність та безпечність функціонування програми. В умовах сучасних вимог до створення програмного забезпечення, дуже важливим є якомога швидше надати користувачам необхідний продукт, задля збільшення прибутку компанії. Відповідно, в умовах нестачі часу, розробнику набагато легше припуститися помилки, що може породити вразливість у відповідній програмі.

Деякі недоліки програмного коду, такі як наявність неочевидних станів програми, недосконалість обробки користувацького вводу, неправильне керування пам'яттю, та інші логічні та технічні вразливості, можуть бути непомітні під час перевірки та тестування вихідного коду програми, або як це загально називають: тестування білої скриньки.

Натомість існує концепція тестування програмного забезпечення сірої скриньки, в якій перевірка на наявність недекларованих можливостей проводиться в умовах відсутності знання внутрішньої реалізації програми, за наявності лише часткового доступу до структури та коду програми, що часто здійснюється за допомогою процесу зворотної розробки, наприклад методами декомпіляції, дизасемблювання, аналізу графу потоку керування тощо. Такий метод аналізу недекларованих можливостей дозволяє оцінити програму з погляду зовнішнього користувацького інтерфейсу в умовах реального використання програми, що дозволяє виявити можливі проблеми та несправності, як можуть виникнути в реальних умовах експлуатації.

З метою автоматизації та збільшення ефективності аналізу на основі сірої скриньки, було запропоновано низку методів тестування, серед яких, одним з найбільш популярних та розповсюджених [1] є метод фазингу (Fuzzing, Fuzz testing) [2].

Розпочинаючи з 1990 року [3], метод фазингу активно використовувався

для знаходження вразливостей сірої скриньки. За цей час лише за допомогою утиліти AFL++ [4] було знайдено велику кількість критичних вразливостей [5] у поширеному програмному забезпеченні.

Ідея фазингу сірої скриньки також була успішно реалізована у змаганні 2016 DARPA Cyber Grand Challenge [6], метою якого стало створення систем автоматичного захисту, які можуть виявляти та виправляти недоліки програмного забезпечення в реальному часі.

1.1 Загальний огляд алгоритму фазингу

На високому рівні абстракції, фазинг - це виконання досліджуваної програми з використанням вхідних даних, вибраних з вхідної множини, яка виходить за межі множини очікуваних вхідних даних [2].

$$f \in F; F \subseteq \overline{I} \tag{1.1}$$

де f — вхідні дані, F — множина вхідних даних, I — очікувані вхідні дані.

Слід зазначити, що однозначно визначити таку вхідну множину вдається не завжди, через відсутність інформації про досліджувану систему. Тут алгоритм фазингу можна розділити на 3 категорії: фазинг чорної, сірої та білої скриньок [7].

При використанні фазингу чорної скриньки [3, 8], досліднику нічого не відомо про внутрішню структуру та функціонал програми, тому множина вхідних даних є множиною всіх можливих даних. У такому випадку процес фазингу може керуватися даними, які повертає програма [2].

Для білої скриньки, вся структура та вихідних код програми відомий, тому процес фазингу буде керований цим знанням [9, 10].

Найчастішим та найпопулярнішим випадком є сіра скринька [11]. В цьому випадку, у дослідника є обмежений доступ до структури програми, з допомогою якого можна визначити набір вхідних даних. У такому випадку процес фазингу буде керований за допомогою моніторингу за поведінкою

програми та покриттям коду, наприклад за покриттям операторів, шляхів, функцій і в цілому за аналізом потоку виконання програми.

1.2 Модель алгоритму фазингу сірої скриньки

Алгоритм фазингу на основі сірої скриньки зазвичай визначають таким чином:

1. Внесення зразка вводу в множину вхідних даних. Початковий зразок створюється на основі аналізу програми, її відомого функціоналу, аналізу декомпільованого або дизасембльованого коду. За зразок вхідних даних допустимо обирати випадкові дані.

2. Вибір вхідних даних з множини. Цю множину також часто називають «тілом» (англ. «corpus») [12].

3. Мутація вибраних вхідних даних. На цьому етапі вибрані вхідні дані видозмінюються за допомогою мутатора [13], щоб на основі вхідних даних згенерувати зразки, що можуть призвести до зміни потоку виконання програми.

4. Запуск досліджуваної програми з використанням вибраних вхідних даних.

5. Оцінка поведінки програми та результатів її виконання [14]. На цьому етапі оцінюється результат роботи програми, яка опрацювала мутовані вхідні дані. Ця функція визначає чи є поточний випадок «цікавим», і, відповідно, чи варто його зберігати в множину вхідних даних для побудови на його основі нових зразків. Поняття «цікавість» є абстрактним, але зазвичай воно пов'язане з пошуком нових гілок у графі виконання програми. Зазвичай використовують методи оцінювання основані на покритті коду (англ. «Code Coverage»), рідше зважені лічильники базових блоків (VUzzer [15]).

6. Повідомлення про виявлення аномалій, якщо такі були знайдені. Часто, такими аномаліями є помилки пов'язані з пошкодженням пам'яті, що аварійно завершить програму, наприклад в UNIX-подібних операційних системах це завершення програми з сигналами: «SIGSEGV», «SIGBUS», «SIGILL» тощо [16].

7. Зміна бази вхідних даних. Якщо було знайдено «цікавий» мутований зразок, то його необхідно додати в множину вхідних зразків для продовження дослідження на його основі.

8. Повернутися до пункту 2.

1.3 Застосування генетичного алгоритму для задач фазингу сірої скриньки

Найбільш поширені утиліти для фазингу на основі сірої скриньки, такі як AFL [17], AFL++ [4], libFuzzer [18], HonggFuzz [19] тощо, використовують генетичний алгоритм.

1.3.1 Модель генетичного алгоритму

Генетичний алгоритм [20] - це алгоритм, заснований на ідеї біологічної еволюції та природного добору, а саме на концепції, що при природному доборі з популяції виживають лише найбільш пристосовані та сильні особи. Він використовується для здач пошуку та оптимізації на біологічно-натхненних методах мутації, схрещування та селекції. Абстрактно цей алгоритм можна описати так:

1. Вибір початкової популяції з деякою кількістю осіб.

2. Оцінка кожної особи за допомогою функції допасованості, яка визначає ймовірність особи на виживання.

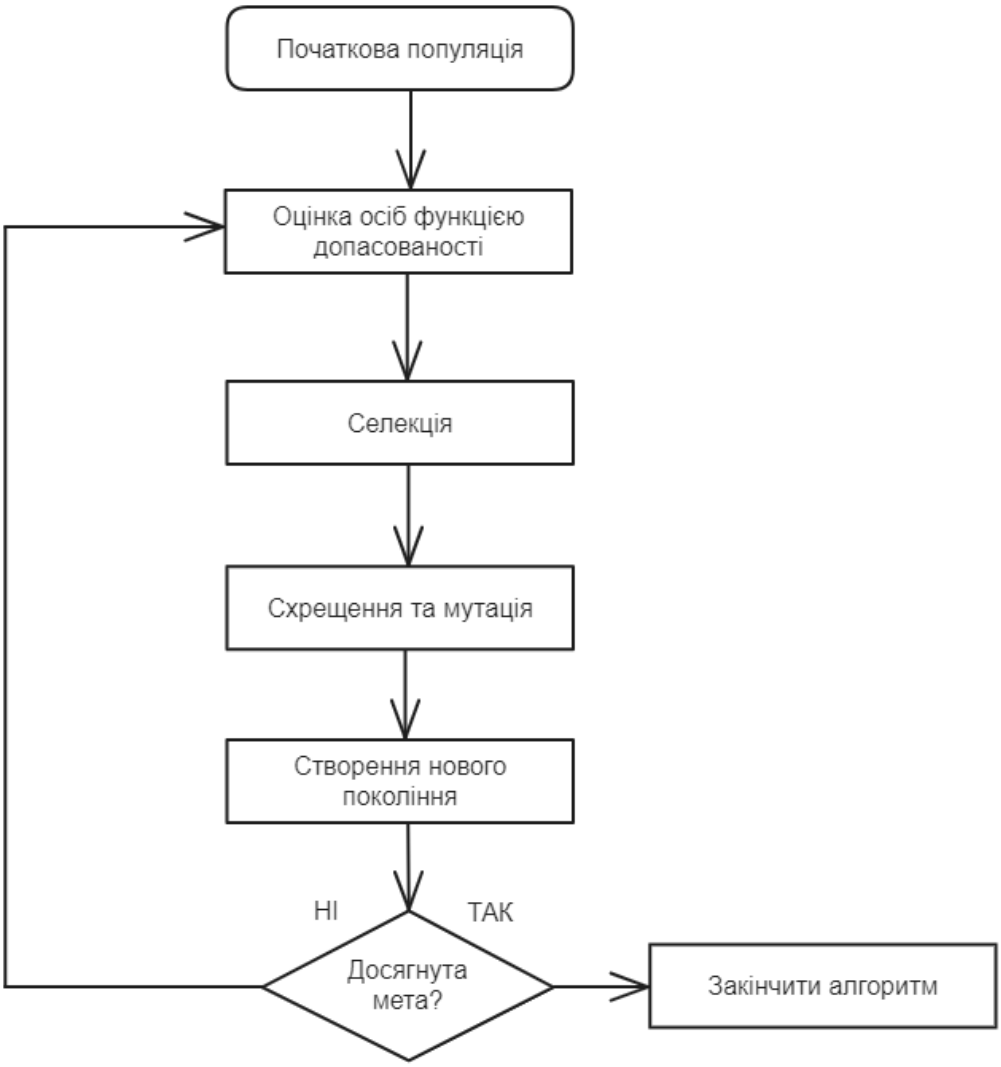

Рисунок 1.1 — Візуалізація генетичного алгоритму

3. Вибір осіб, які здатні на подальше виживання. Цей етап також називають «Селекція».

4. Схрещення та мутація осіб, в результаті яких утворюється нове покоління.

5. Якщо досягнута умова зупинки, то зупинити алгоритм, якщо ні, то повернутися на крок 2.

Для наочності, алгоритм показаний на рисунку 1.1

До умов зупинки генетичного алгоритму відносять:

1. Знаходження потрібного вирішення (генотипу).
2. Досягнення максимальної кількості поколінь, виділених для алгоритму.
3. Досягнення максимального часу, виділеного на алгоритм.
4. Зупинка алгоритму на вимогу.
5. Популяція з найвищим показником функції допасованості припинила розвиток.

1.3.2 Генетичний алгоритм як основна частина сучасних методів фазингу

Безумовно, під час фазингу генетичний алгоритм, набагато покращує ефективність знаходження недекларованих можливостей програмних застосунків. У деяких програмах, щоб фазер дістався до певної вразливості, необхідне виконання великої кількості умов. Фазер не на основі генетичного алгоритму виконує такі задачі надзвичайно довго та не ефективно [21], тому що він не може розпізнати, виконалась та чи інша умова, чи ні. Наприклад, нехай необхідно дослідити наступний код:

```
if (buffer[0] == 'E')
    if (buffer[1] == 'X')
        if (buffer[2] == 'A')
            if (buffer[3] == 'M')
                if (buffer[4] == 'P')
                    if (buffer[5] == 'L')
                        if (buffer[6] == 'E')
                            crash_me();
```

У цьому коді, в масиві символів перевіряється кожен з них по черзі. Якщо всі перевірки успішні, викликається функція crush_me(), яка завершує програму з помилкою. Примітивні фазери будуть перебирати всі можливі рядки довжиною 7, щоб дістатися до виклику функції crush_me(), у такому випадку фазинг не буде відрізнятися від звичайного методу brute-force

search. Враховуючи, що кожен символ може мати значення від 0 до 255, то ймовірність знаходження примітивним фазером такої комбінації за 1 виконання: $\frac{1}{256^7} \approx 1.39 \times 10^{-17}$.

Натомість при використанні генетичного алгоритму, фазер буде «усвідомлений» про стан виконання програми під час кожної ітерації, тобто якщо певна комбінація пройде перевірку першої літери, то такий набір даних (або популяція) буде позначена як «цікавий» та потрапить у набір вхідних даних для наступних ітерацій (corpus). Таким чином, фазер почне підбирати вже наступні літери, не витрачаючи час на підбір першої літери знову і знову, аж допоки не дійде до виклику функції crush_me().

В реальному світі перед методами фазингу стоять розгалуження набагато більші ніж перевірка 7 символів, тому генетичний алгоритм є основною частиною більшості сучасних фазерів.

1.3.3 Сучасні методи мутації в фазингу на основі генетичного алгоритму

У процесі фазингу на основі генетичного алгоритму ключовою ланкою є процес мутацій, який перетворює зразок вхідних даних таким чином, щоб створити нові зразки, які призведуть до відкриття нових шляхів у потоці виконання програми. Оператори мутації можуть бути різні, деякі фазери (наприклад AFL++) навіть дозволяють створювати свої мутатори [22]. Тому розглянемо один з найпопулярніших фазерів: American Fuzzy Lop [17].

Процес мутації AFL має два основних етапи [23]:

1. Детермінований. В цій стадії, оператори мутації змінюють зразок вхідних даних наперед заданим чином. Кількість мутацій визначається довжиною вхідних даних. Ця стадія займає відносно багато часу, але і виконується лише один раз для кожних вхідних даних.

2. Недетермінований. Ця стадія складається з двох етапів: хаотичного перетворення та схрещення. На етапі хаотичних перетворень дані мутують, поки не стануть «цікавими». Якщо ж перший етап закінчується без успіху, запускається етап схрещування, в якому з множини вхідних

даних береться випадковим чином два зразки та комбінуються в один, після чого знову запускається перший етап.

American Fuzzy Lop містить в собі такі види мутацій: зміна біта, зміна байта, арифметичне додавання та віднімання, випадкове видалення, вставляння та заміна байтів.

1.3.4 Оцінка покриття коду як функція допасованості в фазингу на основі генетичного алгоритму

Як було описано вище, процес фазингу на основі генетичного алгоритму вимагає оцінки «цікавості» вхідних даних функцією допасованості. У процесі фазингу такою мірою зазвичай є «Покриття коду». Основна ідея цього методу [24], це оцінити які саме частини програми виконуються залежно від вхідних даних. Якщо зразок який перевіряється в поточний момент часу спричинив виконання в раніше недоступних частинах програми, тобто відкрив новий шлях виконання програми, то його показник покриття коду збільшився, тому такий зразок позначається «цікавим», та потрапляє у множину вхідних даних для подальшої мутації та створення на його основі даних, які зможуть покрити код аж до моменту виявлення недекларованої можливості.

Наприклад, для програмного коду у пункті 1.3.2 зразок даних «EXAMPAB» матиме більший показник функції допасованості, ніж, наприклад «EXUMPLE».

До критеріїв функції допасованості належать покриття операторів, умов, шляхів, функцій, програмного виводу тощо.

До прикладу в AFL++ оцінка покриття коду зазвичай відбувається за допомогою інструментів «PCGUARD» або «LTO» [25]. Це відбувається шляхом компілювання програми спеціальним компілятором, який вводить в програму спеціальні інструкції для відстежування покриття коду [26]. Ці інструкції при їх виконанні отримують доступ до спільної з фазером пам'яті, до так званої «карти покриття коду». Таким чином, фазер знатиме, що щойно було досягнуто конкретної гілки програми.

У випадку відсутності вихідного коду і відповідно можливості скомпілювати програму, у сучасних фазерах є інструментарій для відстежування покриття коду. Одним з найбільш популярних методів є ін'єкція коду, що робить оцінку покриття коду в досліджуваному процесі. Сама оцінка покриття коду відбувається аналогічно оцінці при фазингу білої скриньки, її можна показати наступним псевдокодом:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Як видно, відбувається доступ до спільної пам'яті, яка є картою покриття коду, в якій кожен байт, встановлений у масиві, позначає попадання до певної пари блоків (A, B) в коді, де базовий блок B виконується після базового блоку A [27].

Найбільш поширеною утилітою для такого виду фазингу є AFL++ з QEMU [28] або FRIDA [29] режимами. інструментарій цих режимів дозволяє гнучко отримувати оцінку покриття коду, та ефективно проводити фазинг сірої та чорної скриньок.

QEMU [30] - це популярний емулятор, отже має повний контроль над процесом виконання програми, з його допомогою AFL++ може динамічно вбудовувати необхідний для фазингу інструментарій, наприклад для відстежування покриття коду та інших даних про виконання.

FRIDA [31] - популярний фреймворк для динамічного інструментування, створений для розробників, спеціалістів з кібербезпеки, інженерів зворотної розробки тощо. Основна ідея FRIDA полягає в здатності ін'єкції користувацького Java Script коду в процес програми, що дозволяє перехоплювати, аналізувати, змінювати поведінку програми під час виконання. FRIDA надає зручний та гнучкий для таких задач інтерфейс. Це чудово підходить для задач фазингу, адже з цим функціоналом можна робити ін'єкції коду для фазингу, що дозволяє виконувати фазинг чорної та сірої скриньок. Ще однією особливістю FRIDA є підтримка різних операційних систем: Windows, macOS, GNU/Linux, iOS, watchOS, tvOS, Android, FreeBSD, and QNX.

У цій роботі буде використовуватись цей фреймворк для задач фазин-

гу сірої скриньки, адже він чудово працює на ОС Android, на якій іншими методами важко досягнути такого ж результату через особливості самої операційної системи, її особливого API та методів захисту. FRIDA дозволить процес фазингу на справжньому девайсі, що максимально наблизить процес виявлення недекларованих можливостей до реальних умов експлуатації програми та збільшить швидкодію (у порівнянні з QEMU, який втрачає швидкодію через емуляцію).

Візуалізація основних етапів сучасного процесу фазингу показана на рисунку 1.2.

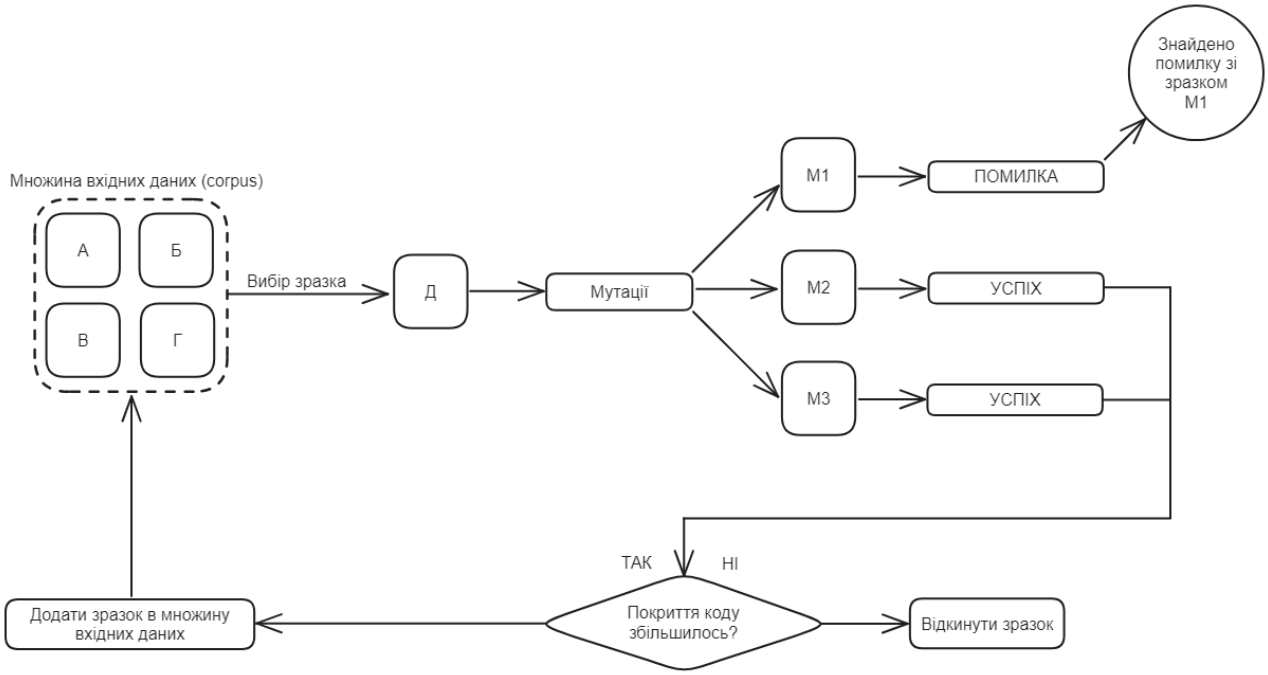

Рисунок 1.2 — Візуалізація процесу фазингу на основі генетичного алгоритму та оцінки покриття коду

Отже, процес фазингу на основі сірої скриньки вимагає специфічних знань та навичок від дослідника, зокрема вміння проводити процес зворотної розробки та аналізу Android застосунків, збірки компонентів фазера для архітектур процесорів Android пристроїв (зокрема armeabi-v7a, arm64-v8a), налаштування динамічного інструментування FRIDA. Тому задля зменшення цих вимог та часу аналізу, актуальним є вдосконалення даної моделі шляхом впровадження автоматизованого налаштування середовища для фазингу.

Висновки до розділу 1

В першому розділі був розглянутий процес фазингу, його основні етапи та концепції, а саме застосування генетичного алгоритму для задач фазингу, оцінка покриття коду як функція допасованості в генетичному алгоритмі для методу фазингу сірої скриньки. Були розглянуті та описані види мутацій вхідних даних у сучасний фазерах на прикладі «American Fuzzy Lop».

З'ясовано, що процес фазингу застосунків для операційної системи Android є трудомістким та вимагає від дослідника специфічних знань та навичок, таких як вміння проводити аналіз та зворотну розробку Android застосунків, збирати компоненти фазера для типових архітектур процесорів Android пристроїв (зокрема armeabi-v7a, arm64-v8a) та налаштування динамічного інструментування FRIDA. Тому є актуальним вдосконалення наявних моделей, шляхом впровадження моделі автоматизованого розгортання середовища фазингу.

Проаналізувавши літературні джерела та дослідивши сучасні методи аналізу недекларованих можливостей, для реалізації фазингу було обрано утиліту AFL++ з використанням FRIDA режиму, як одне з найпоширеніших рішень для фазингу сірої скриньки Android застосунків.

2 МОДЕЛЬ СИСТЕМИ ПОШУКУ НЕДЕКЛАРОВАНИХ МОЖЛИВОСТЕЙ ANDROID ЗАСТОСУНКІВ НА ОСНОВІ ФАЗИНГУ

Розглянувши літературні джерела та сучасні моделі та методи пошуку недекларованих можливостей на основі фазингу сірої скриньки, було розроблено програмну реалізацію автоматизації фазингу Android застосунків, а саме нативних бібліотек, що є скомпільованими програмами, часто написаними мовою C або C++, що використовуються Android додатками, зокрема з розширенням «.apk», для виконання задач, що потребують низькорівневої взаємодії з ОС та збільшення продуктивності програми в алгоритмах, що потребують високої швидкодії.

Для програмної реалізації було обрано мову програмування Python, адже вона чудово підходить для задач автоматизації, має простий для розуміння синтаксис, та весь потрібний функціонал.

Реалізований процес фазингу, складається з кількох етапів, їх візуалізація показана на рисунку 2.1.

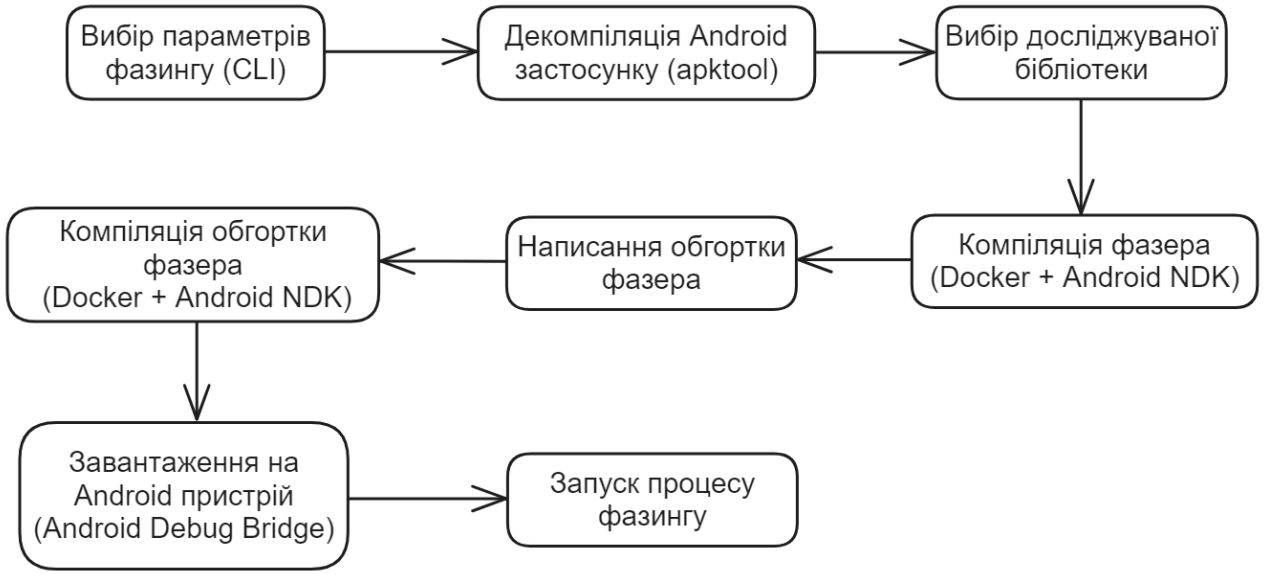

Рисунок 2.1 — Візуалізація процесу автоматизованого фазингу Android застосунку

2.1 Модуль декомпіляції Android застосунку

Для того, щоб з Android застосунку отримати всі необхідні нативні бібліотеки та вихідний код для аналізу, його потрібно декомпілювати. Найпопулярнішою утилітою для такої задачі є «apktool» [32]. Ця програма має інтерфейс командного рядка, тому її можна використовувати просто породивши процес який запустить її. Це виконується за допомогою модуля `subprocess` в функції `decompile_apk()` (Додаток А):

```
def decompile_apk():
    ilog(f"Decompiling {FuzzParams.file}")
    decompile_dir = ensure_dir(FuzzParams.folder + '/' + DECOMPILE_FOLDER)
    cmd = ["apktool", "d", FuzzParams.file, "-f", "-o", decompile_dir]
    process = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.
        STDOUT, text=True)
    for line in process.stdout:
        dlog(line[:-1])
    process.wait()
    ilog(f"Decompiled apk in {decompile_dir}")
```

В результаті виконання даного інструменту, створюється каталог з декомпільованим Android додатком. Шукані нативні бібліотеки знаходяться у папці `lib` і розділені на різні каталоги відповідно до архітектури для якої вони були побудовані, як показано на рисунку 2.2.

Після цього, основний скрипт запропонує вибрати досліднику бібліотеку, яку він прагне дослідити.

2.2 Модуль компіляції фазера

2.2.1 Динамічне інструментування з FRIDA

За основу фазера було обрано одне з найпопулярніших рішень: AFL++ [4]. Цей фазер є сучасним, ефективним, а головне добре задокументованим та надійним рішенням. Як було описано в першому розділі, AFL++ має можливість інтеграції з фреймворком динамічного інструментування FRIDA [31],

```
[/tmp/thesis/google-chrome-125-0-6422-53/lib]$ tree
.
├── arm64-v8a
│   ├── libcrashpad_handler_trampoline.so
│   ├── libmonochrome.so
│   └── libplaceholder.so
└── armeabi-v7a
    ├── libarcore_sdk_c.so
    ├── libcrashpad_handler_trampoline.so
    ├── libelements.so
    ├── libmonochrome_cablev2_authenticator_partition.so
    ├── libmonochrome.so
    ├── libmonochrome_stack_unwinder_partition.so
    └── libmonochrome_test_dummy_partition.so
```

Рисунок 2.2 — Демонстрація структури розміщення нативних бібліотек в .apk файлах на прикладі Google Chrome

що дозволить ефективно проводити фазинг сірої скриньки виконуючи ін'єкції інструментарію фазера напряму в процес. FRIDA має можливість виконувати Java Script код під час виконання програми, що дозволить досліднику вбудовувати весь необхідний функціонал для фазингу за допомогою високорівневої мови програмування. Особливістю FRIDA є також можливість перехоплювання викликів функцій, що дозволить обходити певні обмеження, які можуть виникнути під час фазингу, наприклад наявність виклику функції `Sleep()`, що змусить програму перейти в режим сну на певну кількість часу, що погано вплине на ефективність фазингу. Оскільки, FRIDA надає функціонал перехоплення викликів функцій, дослідник може з легкістю підмінити виклик проблемної функції таким чином, що в ньому буде відсутній виклик `Sleep()`, що значно спростить наступний аналіз.

Задля інтеграції AFL++ та FRIDA, перший має бути скомпільований з основними компонентами FRIDA - frida-gumjs та frida-gum [33]. Ці модулі дозволять виконувати Java Script код в під час виконання процесу, а також найважливішим є те, що вони дозволяють доповнювати та перехоплювати виклики функцій, компілюючи власний користувацький код написаний мовою C, що значно покращить швидкодію фазингу. Детальніше про це в

розділі 2.4.

Оскільки дослідження недекларованих можливостей може проводитися на абсолютно різних девайсах, з різними архітектурами та особливостями, було прийняте рішення динамічно компілювати фазер на початку процесу фазингу, це додасть гнучкості до програмної реалізації.

Для компіляції програм для ОС Android, найпопулярнішою утилітою є Android NDK [34]. За допомогою цього набору інструментів створено переважну більшість бібліотек для Android застосунків, цей набір є найпопулярнішим рішенням для компіляції низькорівневих програм для Android.

2.2.2 Компіляція фазера в ізольованому середовищі Docker

Щоб не обтяжувати систему дослідника, на якій ведеться дослідження, зайвими пакетами, а також заради надійності та повторюваності результату, було прийняте рішення використовувати реалізацію процесу компіляції на основі ізольованих контейнерів з використанням технології Docker [35]. З його допомогою, всі необхідні для компіляції компоненти та залежності будуть встановлені в образі docker, що схожий на мінімальну віртуальну машину з наперед встановленими утилітами.

Для цього, напишемо файл конфігурації, що називається Dockerfile, який є набором інструкцій для побудови docker образа. В цей файл необхідно помістити команди для встановлення всіх необхідних інструментів для збирання фазера. Тоді, основний скрипт буде запускати контейнер на основі створеного образу, який компілюватиме фазер та надаватиме вже готові бінарні виконувані файли, необхідні для фазингу.

Вміст Dockerfile наведений нижче:

```
FROM debian:12.5
# Set working directory
WORKDIR /app
# Create result directory
RUN mkdir -p /result
# Update packages and sources
RUN apt update -y && apt upgrade -y
```

```
# Install tools
RUN apt install -y curl wget cmake zip unzip xxd
# Download AFL++
RUN wget https://github.com/AFLplusplus/AFLplusplus/archive/refs/tags/v4.20c.zip -O
    afl.zip && unzip afl.zip
# Download NDK
RUN wget https://dl.google.com/android/repository/android-ndk-r26d-linux.zip -O ndk.
    zip && unzip ndk.zip
# Create build directory
RUN mkdir -p AFLplusplus-4.20c/build
# Copy CMakeLists.txt
COPY CMakeLists.txt "AFLplusplus-4.20c/CMakeLists.txt"
# Copy entrypoint script
COPY entrypoint.sh .
# Make entrypoint script executable
RUN chmod +x entrypoint.sh
ENTRYPOINT ["./entrypoint.sh"]
```

За основу образу, використовується ОС Debian 12.5, на неї встановлюються всі необхідні інструменти, та копіюються конфігураційні файли. Файл CMakeLists.txt містить інструкції для правильної збірки фазера. Також у Dockerfile, інструкцією «ENTRYPOINT» зазначено, що при запуску контейнера, буде запускатися bash скрипт `./entrypoint.sh`. Цей скрипт написаний для запуску процесу компіляції фазера, та містить в собі наступні команди:

```
#!/bin/bash

# Enter build directory
cd AFLplusplus-4.20c/build || exit

# Choose FRIDA_GUMJS_ARCH
if [ "$ANDROID_ARCH" = "arm64-v8a" ]; then
    FRIDA_GUMJS_ARCH="arm64"
elif [ "$ANDROID_ARCH" = "armeabi-v7a" ]; then
    FRIDA_GUMJS_ARCH="arm"
elif [ "$ANDROID_ARCH" = "x86_64" ]; then
    FRIDA_GUMJS_ARCH="x86_64"
elif [ "$ANDROID_ARCH" = "x86" ]; then
    FRIDA_GUMJS_ARCH="x86"
```

```
else
    echo "[!] WRONG ANDROID_ARCH: ${ANDROID_ARCH}"
    exit
fi

# Run cmake with target params
echo "[*] Using ANDROID_PLATFORM: ${ANDROID_PLATFORM}, ANDROID_ABI:
    ${ANDROID_ARCH}, running cmake..."
cmake -DANDROID_PLATFORM="${ANDROID_PLATFORM}" \
        -DCMAKE_TOOLCHAIN_FILE=/app/android-ndk-r26d/build/cmake/android
            .toolchain.cmake \
        -DANDROID_ABI="${ANDROID_ARCH}" \
        -DFRIDA_GUMJS_ARCH="${FRIDA_GUMJS_ARCH}" ..

# Run make to build AFL
echo "[*] Running make..."
make

# Copy files to destination folder
echo "[*] Copying afl-fuzz afl-frida-trace.so to /result"
cp afl-fuzz /result
cp afl-frida-trace.so /result

# Done message
echo "[*] Done, bye!"
```

У першій частині цього скрипта знаходиться розгалуження, за допомогою якого програма, залежно від вибраної архітектури цільової системи, обирає архітектуру для компіляції. Така необхідність пов'язана з тим, що назви архітектур в компіляторі та структурі Android застосунків трохи відрізняються, наприклад на рисунку 2.2 зазначена архітектура arm64-v8a, а компілятор знає лише про arm64, це пов'язано з тим, що arm64-v8a це специфічна архітектура, яку найчастіше використовують для ОС Android, коли arm64, більш загальна. Однак, програми скомпільовані на arm64 мають працювати на arm64-v8a.

За допомогою утиліти cmake [36] читається файл CMakeLists.txt та налаштовується середовище для наступної компіляції, яке здійснюється за до-

помогою команди make [37]. Після чого, скомпільовані файли afl-fuzz та afl-frida-trace.so копіюються в спільну з ОС дослідника папку.

Запуск контейнера з наступним процесом компіляції, ініціює функція `build_afl()` (Додаток А) основна частина якої представлена нижче:

```
cmd = ["docker", "run", "--rm",
       "-e", f"ANDROID_PLATFORM={ANDROID_PLATFORM_VERSION}",
       "-e", f"ANDROID_ARCH={FuzzParams.arch}",
       "-v",
       f"{FuzzParams.folder + '/' + AFL_BUILD_FOLDER}:/result",
       AFL_BUILDER_DOCKER_IMAGE]
dlog(f"Starting docker image for AFL++ build: {' '.join(cmd)}")

process = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.STDOUT,
    text=True)
for line in process.stdout:
    dlog(line[:-1])
process.wait()
```

Дана функція за допомогою командного інтерфейсу docker та модуля subprocess запускає контейнер для компіляції фазера, створює спільну папку у файловій системі за допомогою інструмента docker volume [38], в яку скрипт компіляції, помістить скомпільовані файли фазера.

2.3 Модуль компіляції обгортки фазера

2.3.1 Реалізація обгортки фазера

За замовчуванням, AFL++ буде передавати вхідні дані для фазингу програмі через стандартний потік введення (stdin). Цей функціонал створений для забезпечення гнучкості фазингу, без необхідності перекомпільовувати фазер під кожну програму окремо.

Оскільки буде проводитись фазинг вже скомпільованих нативних бібліотек Android застосунків, необхідно створити обгортку для фазера, яка є проміжною програмою, що читає ввід зі стандартного потоку вводу від фазера та, імпортуючи нативну бібліотеку, викликати досліджувану функцію,

а як параметр передавати отриманий ввід від фазера. Проміжними етапами цього функціоналу можуть бути певні перетворення вхідних даних від AFL++ за необхідності. Просту візуалізацію цього процесу можна спостерігати на рисунку 2.3.

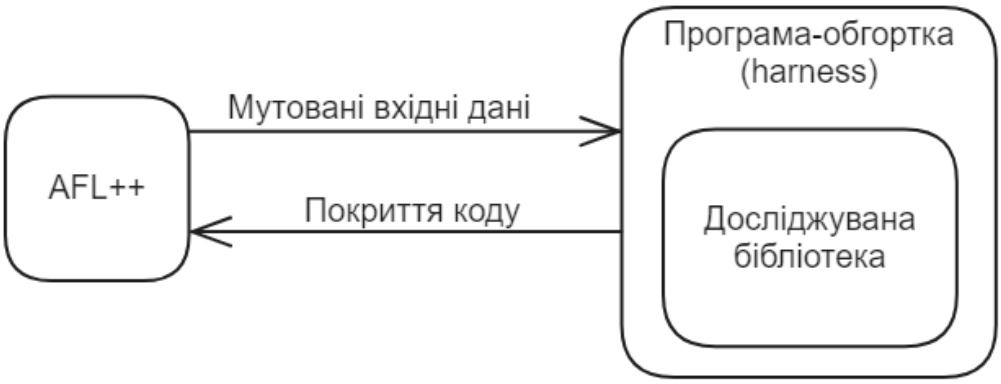

Рисунок 2.3 — Демонстрація роботи програми-обгортки (harness)

Програма обгортка пишеться мовою C та компілюється окремо від фазера, разом з досліджуваною бібліотекою, звичайним компілятором. Оскільки код для такої обгортки буде унікальний для кожної окремої функції, було створено шаблон, з допомогою якого дослідник може ефективніше писати обгортки під потреби свого дослідження. Код-шаблон програми-обгортки для виклику функції, яку необхідно дослідити, пропонується наступний [39]:

```
#include <stdio.h>
#include <stdint.h>
#include <errno.h>

#define BUFFER_SIZE 256

/* Target function */
extern void <function_to_fuzz>(const uint8_t*, uint64_t);

/* harness function */
void fuzz_one_input(const uint8_t *buf, int len) {
    <function_to_fuzz>(buf, len);
}

int main(void) {
```

```
    const uint8_t fuzz_buffer[BUFFER_SIZE];
    ssize_t fuzz_length = fread((void*) fuzz_buffer, 1, BUFFER_SIZE, stdin);
    if (fuzz_length == -1){
        return errno;
    }
    fuzz_one_input(fuzz_buffer, fuzz_length);

    return 0;
}
```

В цьому коді імпортується функція `function_to_fuzz()`, дослідження якої і є метою фазингу. Функція `fuzz_one_input()` приймає як параметри буфер з вхідними даними та його довжину, після чого викликається цільова функція з цими ж параметрами. В `main()` функції ініціюється буфер, розмір якого заданий на початку програми, після чого за допомогою функції `fread()` читається стандартний потік вводу та зберігається в буфер, наступним кроком викликаємо функцію-обгортку `fuzz_one_input()`. Функція-обгортка може здатися непотрібною, адже вона просто викликає цільову функцію з тими ж параметрами, але її наявність спрощує налаштування середовища, тому що досліднику варто лише змінити назву цільової функції в одному файлі, а в інших (наприклад Java Script коді для FRIDA) буде записана завжди однакова функція, що спростить процес дослідження та попередить ненавмисні помилки дослідника.

2.3.2 Компіляція обгортки в ізольованому середовищі Docker

Компіляція програми-обгортки для фазера відбувається з використанням Android NDK, отже задля забезпечення безпеки, надійності та простоти, буде використовуватися Docker. Dockerfile для створення образу буде виглядати так:

```
FROM debian:12.5
# Set working directory
WORKDIR /app
# Create result directory
RUN mkdir -p /result
```

```
# Update packages and sources
RUN apt update -y && apt upgrade -y
# Install tools
RUN apt install -y curl wget cmake zip unzip xxd
# Download NDK
RUN wget https://dl.google.com/android/repository/android-ndk-r26d-linux.zip -O ndk.
    zip && unzip ndk.zip
# Create build directory
RUN mkdir -p build
# Copy entrypoint script
COPY entrypoint.sh .
# Make entrypoint script executable
RUN chmod +x entrypoint.sh
ENTRYPOINT ["./entrypoint.sh"]
```

Як і в Dockefile на етапі компіляції фазера, тут використовується ОС Debian 12.5, на яку встановлюються всі необхідні для збірки утиліти, зокрема Android NDK. В якості ENTRYPOINT скрипта, реалізований наступний код:

```
#!/bin/bash
# Enter build directory
cd build || exit
# Run cmake with target params
echo "[*] Using ANDROID_PLATFORM: ${ANDROID_PLATFORM}, ANDROID_ABI:
    ${ANDROID_ARCH}, running cmake..."
cmake -DANDROID_PLATFORM="${ANDROID_PLATFORM}" \
        -DCMAKE_TOOLCHAIN_FILE=/app/android-ndk-r26d/build/cmake/android
            .toolchain.cmake \
        -DANDROID_ABI="${ANDROID_ARCH}" .
# Run make to build AFL
echo "[*] Running make..."
make
# Copy files to destination folder
echo "[*] Copying fuzz to /result"
cp fuzz /result
# Done message
echo "[*] Done, bye!"
```

В ньому, за допомогою утиліт cmake та make, конфігурується та компілюється обгортка для фазера, що потім копіюється у спільну з ОС дослід-

ника папку. CMakeLists.txt файл копіюється за допомогою основного скрипта автоматизації, та має наступний вміст:

```
project(FuzzWagon)
cmake_minimum_required(VERSION 3.8)
link_directories(${CMAKE_SOURCE_DIR}/lib)
add_executable(fuzz "fuzz.c")
set_property(TARGET fuzz APPEND_STRING PROPERTY LINK_FLAGS " -Wl,-
    rpath=$ORIGIN")
target_link_libraries(fuzz LIBNAME_CHANGE_ME)
```

В даному файлі описано, що компілятор має шукати бібліотеки для зв'язування у каталозі lib, там знаходиться досліджувана бібліотека, функцію з якої імпортуватиме програма-обгортка. Важливою конфігурацією є «-Wl,-rpath=$ORIGIN», вона компілює обгортку таким чином, що під час виконання, пошук динамічних бібліотек, необхідних для виконання програми (зокрема досліджувана бібліотека), буде відбуватися в каталозі, в якому знаходиться виконувана програма. Тобто, під час процесу фазингу, необхідно, щоб програма-обгортка та досліджувана бібліотека знаходились в одній папці.

Структура файлів, що створюється за допомогою скрипта автоматизації показана на рисунку 2.4:

```
[~/.fuzzwagon/db5e56f325722172_1716463348/afl_harness]$ tree
.
├── afl.js
├── CMakeLists.txt
├── fuzz.c
├── include
│   └── jenv.h
└── lib
    ├── libblogfuzz.so
    └── libjenv.so
```

Рисунок 2.4 — Структура файлів, що створюється скриптом автоматизації, для компіляції програми-обгортки

Заголовковий файл jenv.h та бібліотека libjenv.so необхідні для взаємодії нативного низькорівневого коду C з Java [40].

Запуск процесу компіляції програми-обгортки здійснюється функцією `build_afl_harness()` (Додаток А), за допомогою наступного коду:

```
copyfile(templates_path + 'afl.js', destination_path + 'afl.js')
copyfile(templates_path + 'fuzz.c', destination_path + 'fuzz.c')
copyfile(templates_path + 'libjenv.so', destination_path + 'lib/libjenv.so')
copyfile(templates_path + 'jenv.h', destination_path + 'include/jenv.h')
copyfile(templates_path + 'CMakeLists.txt', destination_path + 'CMakeLists.txt')
copyfile(FuzzParams.target, destination_path + 'lib/' + os.path.basename(FuzzParams.
    target))

ilog(f"Fuzzing files placed in {destination_path}, edit them, then press ENTER")

input(get_colored_text("***** Press ENTER *****", Fore.GREEN))

ilog(f"Compiling harness binaries...")
cmd = ["docker", "run", "--rm",
        "-e", f"ANDROID_PLATFORM={ANDROID_PLATFORM_VERSION}",
        "-e", f"ANDROID_ARCH={FuzzParams.arch}",
        "-v",
        f"{FuzzParams.folder + '/' + AFL_HARNESS_FOLDER}:/app/build",
        "-v",
        f"{FuzzParams.folder + '/' + AFL_FINAL_FOLDER}:/result",
        AFL_HARNESS_DOCKER_IMAGE]
dlog(f"Starting docker image for harness build: {' '.join(cmd)}")

process = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.STDOUT,
    text=True)
for line in process.stdout:
    dlog(line[:-1])
process.wait()
```

Цей код за допомогою функції `copyfile()` копіює всі необхідні для компіляції файли у папку проекту. Далі за допомогою бібліотеки subprocess, створюється процес, що викликає docker, запускаючи контейнер та створюючи спільні каталоги для передачі коду обгортки та результатів компіляції.

2.4 Реалізація динамічного інструментування FRIDA

Оскільки з використанням фреймворку FRIDA, є можливість використовувати Java Script задля налаштування та інструментування процесу фазингу, було створено шаблон який використовується разом з модулем обгортки, для керування процесом фазингу. В ньому використовується можливість frida-gum компілювати та вбудовувати користувацький код, написаний мовою програмування C, в якому описано як фазер має викликати функцію-обгортку задля ефективного фазингу в пам'яті процесу.

Код-шаблон конфігурації на Java Script має наступний вигляд [39]:

```
Afl.print(`[*] Starting FRIDA instrumentation for PID: ${Process.id}`);

const cm = new CModule(`
  #include <string.h>
  #include <gum/gumdefs.h>

  #define BUF_LEN 256

  void afl_persistent_hook(GumCpuContext *regs, uint8_t *input_buf, uint32_t
      input_buf_len) {
    uint32_t length;
    if (input_buf_len > BUF_LEN) {
        length = BUF_LEN;
    } else {
        length = input_buf_len;
    }
    memcpy((void *)regs->x[0], input_buf, length);
    regs->x[1] = length;
  }
  `,
  {
    memcpy: Module.getExportByName(null, "memcpy")
  }
);


const pStartAddr = DebugSymbol.fromName("fuzz_one_input").address;
```

```
Afl.setPersistentHook(cm.afl_persistent_hook);
Afl.setPersistentAddress(pStartAddr);
Afl.setEntryPoint(pStartAddr);
Afl.setInMemoryFuzzing();
Afl.setInstrumentLibraries();

Afl.done();
Afl.print("[*] All done!");
```

За допомогою класу `CModule`, FRIDA ініціює користувацький код на C, як посередника між фазером та досліджуваною програмою. В цьому коді створюється функція `afl_persistent_hook()`, в яку передається структура з регістрами `GumCpuContext *regs`, яка дозволяє керувати ними, записуючи в них дані, буфер вхідних даних `uint8_t *input_buf` та його довжина `uint32_t input_buf_len`. В тілі цієї функції перевіряється довжина вхідних даних, встановлюючи обмеження в кількість байт, задану в макросі `BUF_LEN`. Далі, за допомогою стандартної функції `memcpy()`, що копіює дані з одного буфера в інший, вміст буфера з вхідними даними копіюється за адресою в регістрі x0, а довжина в x1. Дані регістри використовуються в архітектурі arm64 для передачі аргументів при виклику функцій [41]. Відповідно для процесу фазингу на інших архітектурах, досліднику необхідно поміняти ці регістри на відповідні для своєї архітектури. Функція `memcpy()` імпортується з з поточного процесу за допомогою методу `Module.getExportByName()`.

Методом `DebugSymbol.fromName()` FRIDA знаходить адресу цільової функції, після чого встановлюється перехоплення виконання цієї функції, новоствореною функцією `afl_persistent_hook()`, після чого починається процес фазингу в пам'яті процесу.

Даним налаштуванням створюється «Постійний перехоплювач» (англ. Persistent Hook), який необхідний для підвищення ефективності процесу фазингу, адже він дозволяє замість постійного запуску програми, що є трудомістким, постійно викликати лише цільову функцію, не перестворюючи весь процес наново.

У цьому скрипті дослідник також може перехоплювати та підмінюва-

ти інші функції, такі як `Sleep()`, задля збільшення ефективності фазингу. Приклад такого перехоплення може бути наступним [39]:

```
const pInitJenv = Module.getExportByName("libjenv.so", "init_java_env");
const interceptor = Interceptor.attach(pInitJenv, {
  onLeave: function(retval) {
    Java.perform(function() {
      Java.use("java.lang.Thread").sleep
        .overload("long")
        .implementation = function() { /* Bypass me */ }
      interceptor.detach();
    });
  }
});
```

В даному коді, використовуючи `libjenv.so` для взаємодії з нативним інтерфейсом Java, перехоплюється виклик `Sleep()`, та замість нього викликається порожня функція, що повністю обходить блокування процесу і робить можливим наступний процес фазингу.

2.5 Модуль доставлення фазера в середовище аналізу

Для доставлення всіх необхідних для фазингу файлів, використовується Android Debug Bridge (adb) [42], що надає можливість копіювати файли з комп'ютера дослідника на Android девайс, а також надавати інтерфейс командного рядка для взаємодії з системою.

Задля надійності роботи, спочатку програма впевнюється чи присутній підключений девайс за допомогою функції `check_adb_devices()` (Додаток А):

```
def check_adb_devices():
    try:
        result = subprocess.run(['adb', 'devices'], capture_output=True, text=True, check=
            True)
        output = result.stdout
        devices = [line.split('\t')[0] for line in output.splitlines()[1:] if line.strip() != '']
        if devices:
            ilog("Found adb devices:")
```

```
            for device in devices:
                ilog(device)
        else:
            elog("No devices connected via ADB.")
            exit_handler()
    except subprocess.CalledProcessError:
        elog("ADB is not installed or not properly configured.")
        exit_handler()
```

В цій функції, за допомогою модуля subprocess, викликається команда «adb devices», що виводить список підключених Android пристроїв. Далі скрипт парсить та перевіряє її вивід, на основі чого і робить висновок про присутність пристрою для дослідження.

За допомогою функції `push_afl()` (Додаток А), скрипт копіює всі необхідні файли на смартфон, це досягається за допомогою виклику команди adb push в наступному коді:

```
files_to_push = os.listdir(FuzzParams.folder + '/' + AFL_FINAL_FOLDER)
for file_name in files_to_push:
    dlog(f"Pushing {FuzzParams.folder + '/' + AFL_FINAL_FOLDER + '/' +
        file_name}")
    cmd = ["adb", "push", FuzzParams.folder + '/' + AFL_FINAL_FOLDER + '/' +
        file_name,
            f"/data/local/tmp/{os.path.basename(FuzzParams.folder)}/{file_name}"]
    try:
        subprocess.run(cmd, check=True, stdout=subprocess.DEVNULL, stderr=
            subprocess.DEVNULL)
    except subprocess.CalledProcessError:
        elog(f"Can't push fuzzing files to device!")
        exit_handler()
```

Копіювання файлів для фазингу відбувається в каталог девайсу /data/local/tmp/PROJECT_NAME, де PROJECT_NAME - це назва поточного проекту дослідження. Цей каталог, що зазвичай використовується Android, використовується для зберігання тимчасових файлів, до нього завжди є доступ через adb, тож це є чудовим місцем для проведення фазингу.

На Android пристрої також створюється зразок вхідних даних за допомогою команд adb shell, що дозволяє виконувати команди на пристрої, та

команди dd:

```
dd if=/dev/urandom of=/data/local/tmp/{os.path.basename(FuzzParams.folder)}/in/
    sample.bin bs=1 count=16
```

Ця команда створить зразок вхідних даних, з випадкових байтів довжиною 16. також, на пристрій буде скопійовано мануал користувача, задля швидкого пошуку необхідних команд та параметрів.

2.6 Модуль автоматизації розгортання середовища аналізу

Задля здійснення автоматизації процесу аналізу недекларованих можливостей на ОС Android, було написано два скрипти, що керують попередньо описаними модулями та надають досліднику зручний інтерфейс. Основний скрипт (main.py) на високому рівні керує програмою, імпортуючи функції зі скрипта (utils.py), та має наступний код:

```
#!/usr/bin/env python3
from utils import *

def main():
    # Print logo
    print_colored(PROJECT_TITLE, Fore.GREEN)
    # Parse program arguments
    parse_arguments()
    # Ensure main program directory is present
    ensure_dir(MAIN_DIRECTORY)
    # Create directory for files
    FuzzParams.folder = ensure_dir(MAIN_DIRECTORY + '/' + get_random_dir_name
        ())
    if FuzzParams.type == "APK":
        decompile_apk()
        FuzzParams.target = get_target_lib()
    else:
        FuzzParams.target = FuzzParams.file
    run_afl()

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

Функція `main()` на початку друкує кольорове привітання з дослідником за допомогою функції `print_colored()`. Наступним кроком є парсинг аргументів командного рядка функцією parse_arguments(), що використовує популярний модуль argparse, у ній ініційовані наступні аргументи:

1. «-w» або «–wizard», при наявності цього параметра, запускається функція опитування користувача, задля вибору ним необхідних аргументів. Даний параметр не є сумісним з аргументами «-a» та «-f», дослідник має вибрати щось одне.

2. «-a» або «–arch», цей прапорець необхідний для вибору архітектури, для якої збиратиметься фазер («x86», «x86_64», «armeabi-v7a», «arm64-v8a»).

3. «-f» або «–file», цей прапорець дозволяє вибрати шлях до досліджуваного файлу, ним може бути файл з розширенням .apk або .so. Валідність вибраного файлу перевіряється функцією `validate_file()`, що перевіряє чи файл існує і чи правильне у нього розширення, після чого заносить цю інформацію в клас FuzzParams.

4. «-v» або «–verbose». Даний прапорець відповідає за ввімкнення режиму детальних логів, використовується задля відлагодження програми.

Використовуючи функцію `ensure_dir()` (Додаток А) створюються каталоги проекту, в яких будуть зберігатися файли проекту. Створення цих каталогів за замовчуванням відбувається в директорії користувача за допомогою наступного коду:

```
def ensure_dir(dir_path):
    new_directory = os.path.join(os.environ['HOME'], dir_path)
    if not os.path.exists(new_directory):
        dlog(f"Creating directory {new_directory}")
        os.mkdir(new_directory)
    elif os.path.isfile(new_directory):
        elog(f"{new_directory} should be a directory")
        exit_handler()
    return new_directory
```

Отримання шляху до директорії користувача відбувається за допомогою отримування змінної оточення «HOME».

Наступним кроком є перевірка розширення досліджуваного файлу, якщо це .apk, то викликається функція декомпіляції `decompile_apk()`, а потім функція `get_target_lib()`, що отримує потрібну бібліотеку з декомпільованого застосунку, враховуючи архітектуру задану дослідником. За наявності кількох бібліотек, застосунок питає користувача, яку саме бібліотеку він хоче дослідити використовуючи функцію `choice_menu()`.

Після цього, викликається функція `run_afl()` (Додаток А), яка викликає модулі описані в минулих розділах, вона має наступний код:

```
def run_afl():
    build_afl()
    build_afl_harness()
    check_adb_devices()
    push_afl()
```

Приклад виконання реалізованого застосунку, показаний на рисунку 2.5.

```
(.venv) [main][~/FuzzWagon]$ ./main.py --arch arm64-v8a --file ../Downloads/afl.apk

 _____           __       __
|  ___|   _ _____\ \     / /_ _  __ _  ___  _ __
| |_ | | | |_  /\ \ /\ / / _` |/ _` |/ _ \| '_ \
|  _|| |_| |/ / / /  \ V  V / (_| | (_| | (_) | | | |
|_|   \__,_/___/___|  \_/\_/ \__,_|\__, |\___/|_| |_|
                                    |___/

[INFO] (2024-05-23 16:04:16,429): Decompiling /home/user/Downloads/afl.apk
[INFO] (2024-05-23 16:04:19,726): Decompiled apk in /home/user/.fuzzwagon/458855c078df8650_1716469456/decompiled
[INFO] (2024-05-23 16:04:19,726): Target arch arm64-v8a found!
[INFO] (2024-05-23 16:04:19,726): Target lib: libblogfuzz.so
[INFO] (2024-05-23 16:04:19,726): Building AFL++ binaries...
[INFO] (2024-05-23 16:04:35,102): Built AFL++ binaries in /home/user/.fuzzwagon/458855c078df8650_1716469456/afl_build
[INFO] (2024-05-23 16:04:35,102): Copying final AFL++ files to /home/user/.fuzzwagon/458855c078df8650_1716469456/final
[INFO] (2024-05-23 16:04:35,123): Building Harness binaries...
[INFO] (2024-05-23 16:04:35,123): Copying Harness source code templates to /home/user/.fuzzwagon/458855c078df8650_1716469456/afl_harness/
[INFO] (2024-05-23 16:04:35,123): Fuzzing files placed in /home/user/.fuzzwagon/458855c078df8650_1716469456/afl_harness/, edit them, then press ENTER
***** Press ENTER *****
[INFO] (2024-05-23 16:05:58,749): Compiling harness binaries...
[INFO] (2024-05-23 16:05:59,590): Built harness binaries in /home/user/.fuzzwagon/458855c078df8650_1716469456/final
[INFO] (2024-05-23 16:05:59,590): Copying final harness files to /home/user/.fuzzwagon/458855c078df8650_1716469456/final
[INFO] (2024-05-23 16:05:59,594): Found adb devices:
[INFO] (2024-05-23 16:05:59,595): 88VX0337G
[INFO] (2024-05-23 16:05:59,595): Pushing fuzzing binaries to device...
[INFO] (2024-05-23 16:06:01,853): That all, now you need to execute 'adb shell' (make sure, that user is root)
[INFO] (2024-05-23 16:06:01,853): Execute 'cd /data/local/tmp/458855c078df8650_1716469456' to navigate to fuzzing folder
[INFO] (2024-05-23 16:06:01,853): Read instructions in README.md to start fuzzing
[INFO] (2024-05-23 16:06:01,853): Good luck =)
(.venv) [main][~/FuzzWagon]$
```

Рисунок 2.5 — Демонстрація роботи реалізованого застосунку автоматизації процесу фазингу

Висновки до розділу 2

В другому розділі була запропонована програмна реалізація моделі автоматизованого процесу розгортання середовища аналізу недекларованих можливостей Android застосунків на основі фазингу сірої скриньки з використанням AFL++ з FRIDA режимом.

Реалізація містить модулі: декомпіляції Android застосунків з використанням утиліти apktool, компіляції фазера з інтегрованим фреймворком динамічного інструментування FRIDA в ізольованому середовищі Docker, компіляції обгортки фазера в ізольованому середовищі Docker, доставлення зібраних компонентів на Android пристрій, користувацького інтерфейсу в командному рядку.

Наведені шаблони обгорток та налаштувань FRIDA для типових зразків функцій нативних бібліотек Android застосунків. Для збірки обгортки та фазера запропоновані налаштування компілятора Android NDK для типових архітектур Android пристроїв, а саме x86, x86_64, armeabi-v7a, arm64-v8a.

3 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

Для дослідження був використаний смартфон Google Pixel 3, що має 8 ядер процесора, 4 гігабайти оперативної пам'яті, та архітектуру процесора arm64-v8a. Для більш ефективного фазингу, в режимі суперкористувача було виконано наступні команди:

```
# cd /sys/devices/system/cpu
# echo performance | tee cpu*/cpufreq/scaling_governor
```

Даний набір команд, використовується для зміни політики керування частотою процесора (scaling governor) на «продуктивність» (performance) для всіх ядер процесора, що надасть необхідну потужність для короткострокових процесів фазера.

3.1 Фазинг зразка Android застосунку з недекларованою можливістю

Для початку дослідження ефективності розробленого методу автоматизованого фазингу, дослідимо тестовий зразок [39] Android застосунку з недекларованою можливістю. Він спеціально створений, для демонстрації ефективності роботи фазингу, нативна бібліотека, що міститься в ньому має вразливу функцію, що містить умовні оператори, пройшовши які, викликається функція `crash()`, що намагається викликати код за адресою 0x0, що призводить до термінового завершення програми з помилкою.

Граф потоку керування даної функції показаний на рисунку 3.1.

Дана програма приймає на вхід буфер, та побайтово перевіряє його еквівалентність рядку `Quarksl4bfuzzMe!`. Як видно, дана фраза має букви у верхньому та нижньому регістрах, цифри та спеціальні символи.

Для її аналізу шляхом фазингу, запустимо запропоновану реалізацію, вказавши за цільовий застосунок даний зразок, а архітектуру вкажемо arm64-v8a. Результати запуску показані на рисунку 3.2.

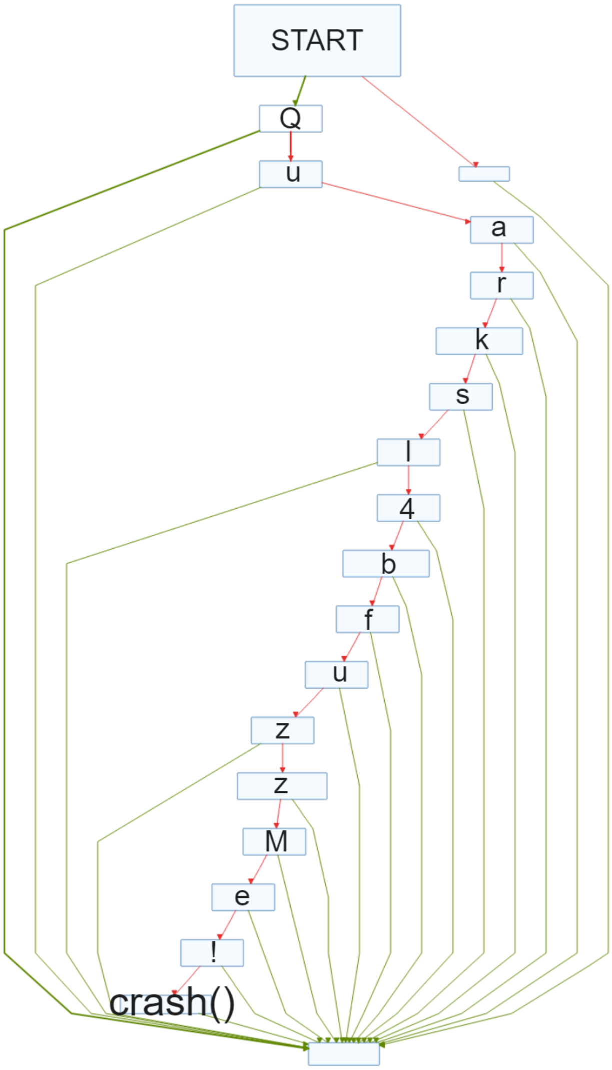

Рисунок 3.1 — Граф потоку керування зразка досліджуваної програми

```
(.venv) [main][~/FuzzWagon]$ ./main.py --arch arm64-v8a --file /home/user/Downloads/afl.apk

 _____              __        __
|  ___|   _ _______\ \      / /_ _  __ _  ___  _ __
| |_ | | | |_  /_  /\ \ /\ / / _` |/ _` |/ _ \| '_ \
|  _|| |_| |/ / / /  \ V  V / (_| | (_| | (_) | | | |
|_|   \__,_/___/___|  \_/\_/ \__,_|\__, |\___/|_| |_|
                                   |___/

[INFO] (2024-05-25 13:58:05,780): Decompiling /home/user/Downloads/afl.apk
[INFO] (2024-05-25 13:58:10,699): Decompiled apk in /home/user/.fuzzwagon/219a07de78ccaec3_1716634686/decompiled
[INFO] (2024-05-25 13:58:10,699): Target arch arm64-v8a found!
[INFO] (2024-05-25 13:58:10,699): Target lib: libblogfuzz.so
[INFO] (2024-05-25 13:58:10,699): Building AFL++ binaries...
[INFO] (2024-05-25 13:58:29,286): Built AFL++ binaries in /home/user/.fuzzwagon/219a07de78ccaec3_1716634686/afl_build
[INFO] (2024-05-25 13:58:29,287): Copying final AFL++ files to /home/user/.fuzzwagon/219a07de78ccaec3_1716634686/final
[INFO] (2024-05-25 13:58:29,308): Building Harness binaries...
[INFO] (2024-05-25 13:58:29,308): Copying Harness source code templates to /home/user/.fuzzwagon/219a07de78ccaec3_1716634686/afl_harness/
[INFO] (2024-05-25 13:58:29,309): Fuzzing files placed in /home/user/.fuzzwagon/219a07de78ccaec3_1716634686/afl_harness/, edit them, then press ENTER
***** Press ENTER *****
```

Рисунок 3.2 — Демонстрація запуску запропонованого методу автоматизації фазингу

Після запуску, скрипт запускає процес декомпіляції застосунку, читає вміст директорії з бібліотеками, та враховуючи задану архітектуру, обирає відповідний каталог lib/arm64-v8a. В цьому каталозі знаходиться лише одна бібліотека libblogfuzz.so, тому скрипт автоматично вибрав її як цільову. Далі програма пропонує перейти до каталогу проекту та написати код програми-обгортки. Шаблон, що програма пропонує досліднику майже повністю підходить для досліджуваної бібліотеки, потрібно лише вписати назву функції, що потрібно проаналізувати. Знайти повну назву цієї функції можна за допомогою наступної команди:

```
$ nm -D libblogfuzz.so | grep ' T '
0000000000000a10 T Java_qb_blogfuzz_NativeHelper_fuzzMeArray
0000000000000c44 T Java_qb_blogfuzz_NativeHelper_fuzzMeWrapper
0000000000000b08 T _Z6fuzzMePKai <-------------------- TAGET !
```

Отже, справжня назва цільової функції `_Z6fuzzMePKai`, впишемо її в програму обгортку. Після чого, в програмі натиснемо кнопку ENTER, та продовжимо процес збірки середовища, який показаний на рисунку 3.3.

```
***** Press ENTER *****
[INFO] (2024-05-25 14:16:57,471): Compiling harness binaries...
[INFO] (2024-05-25 14:16:58,787): Built harness binaries in /home/user/.fuzzwagon/219a07de78ccaec3_1716634686/final
[INFO] (2024-05-25 14:16:58,787): Copying final harness files to /home/user/.fuzzwagon/219a07de78ccaec3_1716634686/final
[INFO] (2024-05-25 14:16:58,793): Found adb devices:
[INFO] (2024-05-25 14:16:58,793): 88VX0337G
[INFO] (2024-05-25 14:16:58,793): Pushing fuzzing binaries to device...
[INFO] (2024-05-25 14:17:00,907): That all, now you need to execute 'adb shell' (make sure, that user is root)
[INFO] (2024-05-25 14:17:00,907): Execute 'cd /data/local/tmp/219a07de78ccaec3_1716634686' to navigate to fuzzing folder
[INFO] (2024-05-25 14:17:00,907): Read instructions in README.md to start fuzzing
[INFO] (2024-05-25 14:17:00,907): Good luck =)
(.venv) [main][~/FuzzWagon]$
```

Рисунок 3.3 — Демонстрація процесу збірки середовища фазингу

Як видно на рисунку 3.3, компіляція програми обгортки пройшла успішно, був знайдений Android пристрій, на який було скопійовано файли для фазингу. Отже, на цьому моменті, середовище для фазингу повністю підготовлене, залишається лише запустити процес дослідження, для цього виконуємо запропоновані програмою команди для отримання інтерфейсу командного рядка на Android пристрої:

```
$ adb shell
$ cd /data/local/tmp/219a07de78ccaec3_1716634686
```

Тепер дослідник може запустити процес фазингу командою:

```
$ ./afl−fuzz −O −G 256 −i in −o out ./fuzz
```

Дана команда запускає виконуваний файл фазера (afl-fuzz) з такими параметрами:

- Параметр «-O» означає запуск фазера у FRIDA режимі.
- Параметр «-G 256» обмежує максимальну довжину вхідних даних, що генерує фазер, на 256 байтів. За замовчуванням довжина 1048576 байтів. В цілому, навіть з дуже великою довжиною вхідних даних фазер буде працювати, але менш ефективно. Обмеження у 256 байтів імітує ситуацію в якій дослідник не знає точну кількість байтів вводу, потрібних для знаходження недекларованої можливості.
- Параметр «-i in» вказує, що каталог «in» має використовуватись як каталог з вхідними даними.
- Параметр «-o out» вказує, що директорія «out» буде використана для знайдених даних під час процесу фазингу.

Після запуску цієї команди, з'явиться інтерфейс AFL++, в якому дослідник може побачити всю статистику та необхідну інформацію щодо процесу фазингу. На рисунку 3.4 продемонстровано як виглядає інтерфейс AFL++.

- У верхньому рядку «run time» показує скільки триває процес фазингу.
- «last new find» вказує на час, що пройшов з моменту знаходження останнього «цікавого» зразка.
- «last saved crash» вказує на час, що пройшов з моменту знаходження останнього креша.
- «corpus count» вказує на кількість знайдених цікавих зразків вхідних даних. В цьому випадку вона корелює з кількістю умовних операторів у досліджуваній функції.

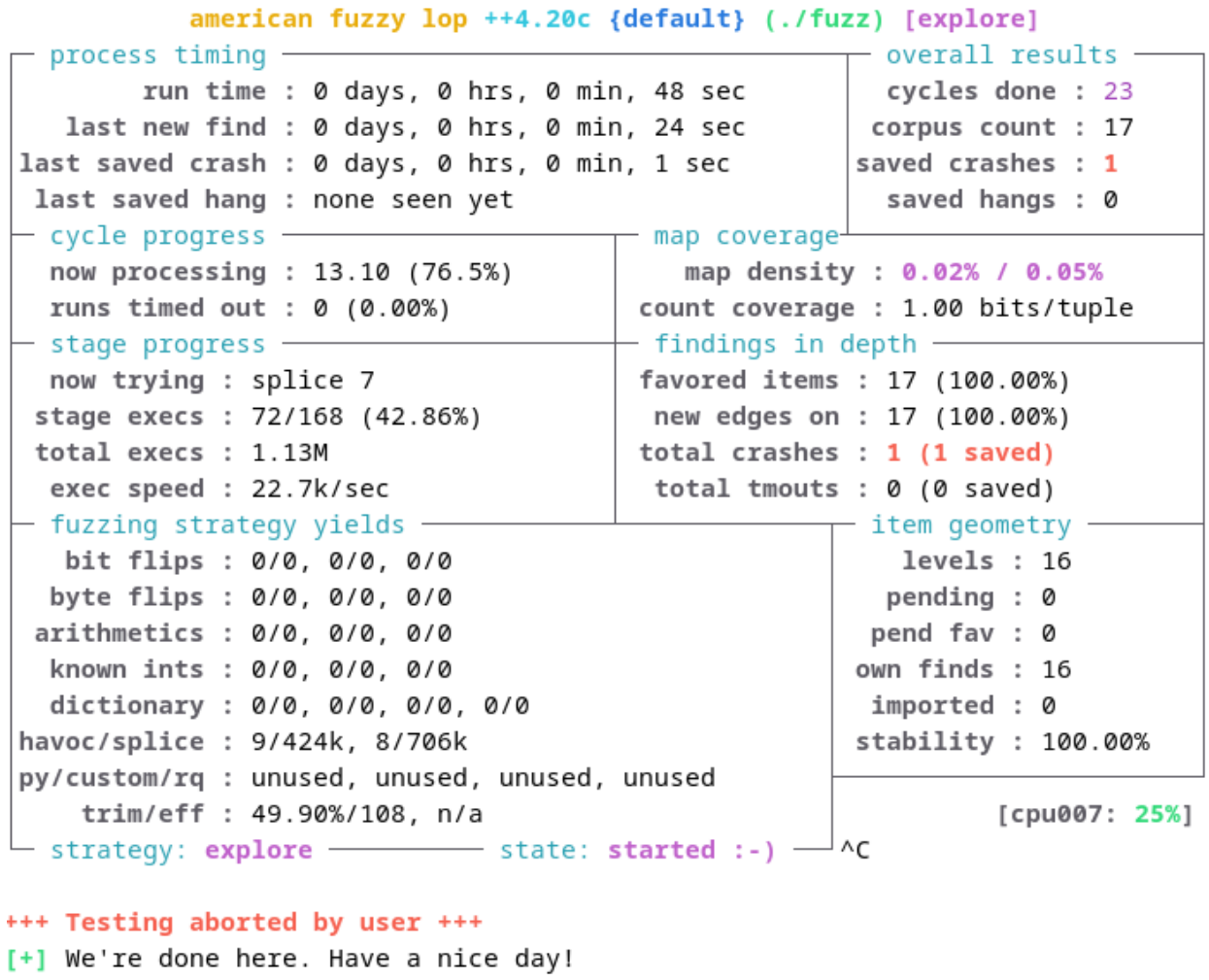

Рисунок 3.4 — Демонстрація знаходження недекларованої можливості у тестовому застосунку за допомогою AFL++

- «saved crashes» показує кількість знайдених крешів програми. Як видно, було знайдено 1 креш.
- «total execs» вказує на кількість виконань функції, що досліджується. Даний параметр за 48 секунд досяг значення 1.14 мільйонів виконань, що можливе завдяки ефективному постійному перехопленню FRIDA.
- «exec speed» вказує на швидкість виконання. В цьому випадку це 22.7 тисяч разів на секунду, що є приголомшливо швидко.

Після закінчення аналізу, результати зберігаються в каталозі «out». Збережені вхідні дані, що призвели до крешу, можна знайти за допомогою команди:

```
$ xxd out/default/crashes/id*
00000000: 5175 6172 6b73 6c34 6266 757a 7a4d 6521 Quarksl4bfuzzMe!
```

Як видно, було знайдено повністю необхідну вхідну фразу «Quarksl4bfuzzMe!», що призвела до виявлення недекларованої можливості. У назві файлу з вхідними даними для крешу також зберігається інформація про час знаходження та кількість виконань програми, наприклад:

```
id:000000,sig:07,src:000016,time:237690,execs:5111326,op:havoc,rep:1
```

Тут, креш було знайдено за 237690 мілісекунд та 5111326 виконань досліджуваної функції.

3.2 Аналіз швидкодії процесу фазингу залежно від складності застосунку

Для аналізу швидкодії процесу фазингу залежно від складності застосунку, за міру складності візьмемо кількість умовних операторів у досліджуваній функції при перевірці вхідних даних, як було описано в розділі 3.1.

3.2.1 Генерація зразків для аналізу

Для цього дослідження, заміряємо час знаходження крешу для кожного зі згенерованих 10 зразків з кількістю умовних операторів від 1 до 15. Генерувати файли з розширенням «.apk» не є доцільним, адже потім буде відбуватися процес декомпіляції, в результаті якого буде отримана нативна бібліотека, тому компілюватимемо одразу бібліотеку.

Для компіляції нативних бібліотек використовуватимемо Android NDK [34].

Файл конфігурації компіляції Application.mk такий:

```
APP_ABI := arm64-v8a
APP_PLATFORM := android-30
APP_BUILD_SCRIPT := Android.mk
```

В ньому описана архітектура процесора arm64-v8a, номер Android API - 30, та скрипт для збірки - Android.mk, код якого має наступний вигляд:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := sample
LOCAL_SRC_FILES := sample.c
LOCAL_CFLAGS := -O0
include $(BUILD_SHARED_LIBRARY)
```

В ньому описана назва програми, що буде скомпільовано - «sample», вказано файл з вихідним кодом - «sample.c» та вказаний прапорець для компілятора «-O0», що вказує на рівень оптимізації компілятора. Якщо його не вказати, компілятор може спотворити програму, оптимізувавши структуру умовних операторів та виклик креш-функції.

Оскільки зразки мають генеруватися випадковим чином, то було написано наступний шаблон для вихідного коду цільової бібліотеки:

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

void fuzz_sample(const char *buffer, size_t length)
{
    void (*crash)() = 0x0;

    if (length < FUZZ_LENGTH)
        return;

FUZZ_IF
}
```

`FUZZ_LENGTH` та `FUZZ_IF` - це шаблони замість яких буде вставлені довжина та тіло умовних операторів відповідно. Також ініціюється функція `crash()`, що вказує на адресу 0x0, її виклик призведе до помилки сегментації пам'яті (Segmentation fault).

Був реалізований скрипт мовою python, що оброблює даний шаблон, його код наступний:

```
#!/usr/bin/env python3
from sys import argv
from random import choice
```

```
def gen_sample(fuzz_length):
    fuzz_length = int(fuzz_length)
    alphabet = "0123456789
        abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!#$&()
        *+,-./:;<=>?@[]^_{|}~"
    tab = " " * 4

    with open("./template.c", "r") as f:
        template = f.read()
    template = template.replace("FUZZ_LENGTH", str(fuzz_length))

    fuzz_if = ""
    for i in range(fuzz_length):
        fuzz_if += tab * (i + 1) + f"if (buffer[{i}] == '{choice(alphabet)}')\n"
    fuzz_if += tab * (fuzz_length + 1 )+ "crash();\n"
    template = template.replace("FUZZ_IF", fuzz_if)

    with open("./sample.c", "w") as f:
        f.write(template)

if __name__ == "__main__":
    if len(argv) != 2:
        print(f"[!] Usage ./gen.py fuzz_length")
        exit()

    gen_sample(argv[1])
```

Він приймає як аргумент командного рядка довжину структури умовних операторів, після чого викликається функція `gen_sample()`, що читає файл-шаблон, потім замінює строку `FUZZ_LENGTH` на довжину структури умовних операторів. Після цього, в циклі, формуються умовні оператори, в праву частину яких потрапляє випадково обраний з алфавіту символ. По закінченню роботи цикла, додається виклик креш-функції. Отриманий рядок вставляється в шаблон замість `FUZZ_IF`, після чого заповнений шаблон записується в файл вихідного коду sample.c.

Для компіляції було створено Makefile, його вміст наведений нижче:

```
NDK_BUILD := NDK_PROJECT_PATH=. ndk-build NDK_APPLICATION_MK=./
    Application.mk
```

```
FUZZ_LENGTH ?= 10
GENERATE := "./gen.py"
UNIQUE := $(shell date +%s%N)

all: prepare build clean
build:
	$(NDK_BUILD)
	@cp libs/arm64-v8a/libsample.so samples_$(FUZZ_LENGTH)/sample_$(
	    FUZZ_LENGTH)_$(UNIQUE).so
prepare:
	@mkdir -p samples_$(FUZZ_LENGTH)
	$(GENERATE) $(FUZZ_LENGTH)
clean:
	@-rm -rf libs obj
```

В ньому задаються команди для компіляції, що поділена на такі стадії:

1. Підготовка (prepare). В цій стадії викликається скрипт для створення вихідного коду з шаблону. Параметр FUZZ_LENGTH приймається з командного рядка. Також створюється папка для зразків з відповідною довжиною структури умовних операторів.

2. Збірка (build). На цьому етапі компілюється бібліотека та переміщується у відповідний каталог.

3. Очищення (clear). Завершальна стадія, на якій видаляються створені на етапі компіляції файли.

Для генерації десяти зразків з довжиною структури умовних операторів 1, виконаємо наступну команду:

```
$ for i in {1..10}; do make FUZZ_LENGTH=1; done
```

Згенеровані зразки виглядають наступним чином:

```
$ ls -l samples_1 | cut -d ' ' -f9

sample_1_1716662896008739933.so
sample_1_1716662896324369868.so
sample_1_1716662896638468793.so
sample_1_1716662896940967843.so
```

sample_1_1716662897254912067.so
sample_1_1716662897558725325.so
sample_1_1716662897881447706.so
sample_1_1716662898194230022.so
sample_1_1716662898509499155.so
sample_1_1716662898826517299.so

3.2.2 Результати вимірювання часу фазингу

Як було описано в розділі 3.1, AFL++ фіксує час знаходження крешів в назві файлу з вхідними даними, з точністю до мілісекунд. Цю інформацію будемо зберігати для подальшої обробки у .csv файл.

В результаті вимірювання часу t_K, де K - порядковий номер зразка в групі з 10 зразків, зі складністю фазингу $N \in \{1, 2, 3, \ldots, 15\}$, було отримано такі результати:

Таблиця 3.1 — Вимірювання часу фазингу залежно від складності зразка

N	t_1, мс	t_2, мс	t_3, мс	t_4, мс	t_5, мс	t_6, мс	t_7, мс	t_8, мс	t_9, мс	t_{10}, мс
1	27	50	17	7	74	80	57	34	40	33
2	147	180	57	57	280	260	280	123	247	60
3	890	3774	3620	1447	637	814	2530	667	2320	563
4	1263	2213	5593	3246	1247	25080	3246	5357	23150	860
5	9617	5306	35567	2437	10316	2026	3320	25893	9137	6216
6	12460	8643	2487	6973	2550	3010	5517	5497	27617	4647
7	3480	22263	17056	35507	17494	36104	6333	21580	20990	11047
8	18650	10946	47520	43850	21990	17444	67700	18726	40420	72266
9	16027	31434	50980	72940	34576	18176	19690	10513	50536	18667
10	91003	115896	70000	63807	57423	24780	11747	30487	124443	83054
11	70340	82167	110903	31906	64323	136630	111943	25374	79473	104560
12	25750	96094	86613	121650	42186	93143	116093	95806	141474	60003
13	205206	152824	217117	252613	52990	66626	97940	85927	41290	43917
14	208630	72554	173486	293060	109206	124493	244016	95430	143650	101090
15	201637	159597	161433	186287	217413	136080	295843	131767	240917	320693

Візуалізуємо отримані результати за допомогою бібліотеки matplotlib та

наступного коду:

```
#!/usr/bin/env python3
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

def exponential(x, a, b):
    return a * np.exp(b * x)

data = pd.read_csv("./samples.csv")
data.drop("Fuzzing Length", axis=1, inplace=True)
exec_times = data.values.tolist()
X, Y = list(), list()
for i in range(len(exec_times)):
    for j in exec_times[i]:
        Y.append(j)
        X.append(i + 1)

x, y = np.array(X), np.array(Y)

plt.figure(figsize=(16, 10))
plt.scatter(x, y, label="Час фазингу зразка складності N")
plt.xlabel("Складність зразка N", fontsize=16)
plt.ylabel("Час фазингу $t_K$, мс", fontsize=16)
plt.legend(fontsize=16)
plt.title("Залежність часу фазингу від складності зразка", fontsize=20)
plt.show()
```

В цьому коді спочатку відбувається парсинг отриманого .csv файлу з результатами вимірювань часу. Відкидається колонка з номерами складності програми, потім створюється два списки з точками на графіку, X - складність зразків, Y - відповідний час фазингу.

Далі відбвається побудова графіка з точками за допомогою інструментів matplotlib зокрема plt.scatter().

Отриманий графік показаний на рисунку 3.5.

Як видно з графіка на рисунку 3.5, залежність схожа не експоненційну. Побудуємо експоненційну модель залежності часу фазингу від його склад-

Рисунок 3.5 — Графік залежності часу фазингу від складності зразка

ності та розрахуємо коефіцієнт детермінації.

Для цього скористаємось наступним кодом:

```
def exponential(x, a, b):
    return a * np.exp(b * x)

popt_exp, _ = curve_fit(exponential, x, y)
exp_a, exp_b = popt_exp
x_fit = np.linspace(min(x), max(x), 100)
y_fit_exp = exponential(x_fit, exp_a, exp_b)
y_exp = exponential(x, exp_a, exp_b)
r2_exp = r2_score(y, y_exp)

plt.figure(figsize=(16, 10))
plt.scatter(x, y, label="Час фазингу зразка складності N")
plt.plot(
    x_fit,
    y_fit_exp,
    label=f"Експоненційна
     залежність: $y = {exp_a:.2f}e^{{{exp_b:.2f}x}}$, $R^2 = {r2_exp:.2f}$",
```

```
    color="red",
)
```

В цьому коді використовується функція curve_fit() з бібліотеки SciPy, що знаходить параметри функції, які дозволяють їй найкраще відповідати заданим даним методом найменших квадратів. Для обрахунку коефіцієнта детермінації використовується, функція r2_score() з бібліотеки scikit-learn. Коефіцієнт детермінації R^2 вказує, наскільки добре модель підтверджується отриманими даними, чим ближче значення коєфіцієнта до 1, тим кращим є підтвердження моделі. Отримана модель візуалізована на рисунку 3.6.

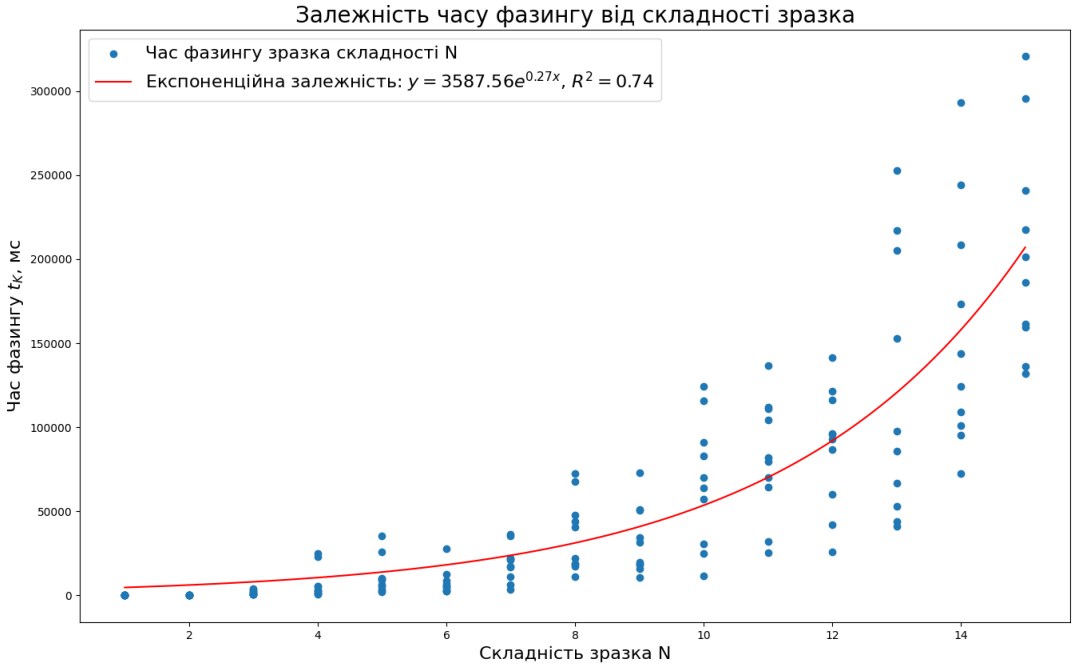

Рисунок 3.6 — Модель експоненційної залежності часу фазингу від складності зразка

Як видно з графіка на рисунку 3.6, залежність дійсно має експоненційну складність з функцією $y = 3587.56e^{0.27x}$, коефіцієнт детермінації $R^2 = 0.74$. Час фазингу стрімко зростає при збільшенні складності зразків.

Варто зауважити, що розкид часу для зразків з однаковою складністю є істотно великим, хоч закономірність все одно прослідковується. Такий ре-

зультат пов'язаний з наявністю хаотичних випадкових перетворень над вхідними даними на етапі мутацій, що призводить до неочікуваних змін зразка, відповідно фазинг не є детермінованим та може займати різні проміжки часу.

Враховуючи цю недетермінованість, рекомендується переривати процес фазингу, якщо не було знайдено нових шляхів за орієнтовний проміжок часу. В AFL++ це можна зробити за допомогою задавання часу в секундах в змінній оточення AFL_EXIT_ON_TIME. В цій ситуації можна або продовжити фазинг, залишивши зразок вхідних даних без змін, або створити новий.

Під час дослідження, було помічено що від постійного інтенсивного фазингу, смартфон нагрівався, а процес фазингу сповільнювався, тому фазинг проводився з перервами для забезпечення охолодження пристрою та отримання точних результатів. Для проведення фазингу рекомендується використовувати більш потужний пристрій та систему активного повітряного або водяного охолодження задля запобігання перегріванню.

Висновки до розділу 3

В третьому розділі було експериментально підтверджено, що програмна реалізація запропонованої моделі автоматизації розгортання середовища аналізу недекларованих можливостей Android застосунків на основі фазингу сірої скриньки, знаходить вразливість пошкодження сегментації пам'яті, що полягає у виклику функції за адресою 0, після проходження гілки побайтової перевірки вхідного параметра розміром не менше 16 байтів в цільовому застосунку, за 47 секунд. Для цього, програмна реалізація надала досліднику можливість не виконувати попередній аналіз та зворотну розробку досліджуваного застосунку, натомість надавши шаблон, в якому дослідник змінивши лише назву досліджуваної функції, отримав готове середовище з усіма зібраними компонентами для фазингу.

Було проведене експериментальне дослідження з визначення залежності часу, необхідного для знаходження вразливості методом фазингу, від складності програми, де за складність було взято кількість перевірок умовними операторами. Після вимірювання часу аналізу для 150 зразків вразливих додатків зі складністю від 1 до 15, було побудовано графік залежності часу фазингу від складності застосунку. Визначено, що залежність має експоненційну складність та швидко зростає, функція має вигляд $y = 3587.56e^{0.27x}$, її коефіцієнт детермінації $R^2 = 0.74$. Розкид часу виявився доволі істотним, що пояснюється хаотичністю процесу фазингу, тому запропоновано використовувати змінну оточення AFL_EXIT_ON_TIME для обмеження часу на пошук нового шляху виконання, враховуючи отримані результати залежності часу фазингу від складності застосунку.

Помічено, що пристрій на якому відбувався фазинг сильно нагрівається, що сповільнює процес фазингу, тому запропоновано використовувати активне охолодження при тривалому аналізі.

ВИСНОВКИ

В роботі був розглянутий процес аналізу недекларованих можливостей Android застосунків на основі фазингу сірої скриньки. Визначено значення генетичного алгоритму з оцінкою покриття коду як функції допасованості в сучасних методах фазингу. Розглянуті типові методи мутацій вхідних зразків на прикладі програми для фазингу «American Fuzzy Lop».

Процес фазингу застосунків для операційної системи
Android визначено трудомістким. Він вимагає від дослідника наявності специфічних знань та навичок, таких як вміння проводити аналіз та зворотну розробку Android застосунків, збирати компоненти фазера для типових архітектур процесорів Android пристроїв (зокрема armeabi-v7a, arm64-v8a) та налаштування динамічного інструментування FRIDA.

Підтверджено актуальність вдосконалення наявних моделей, шляхом впровадження моделі автоматизованого розгортання середовища фазингу.

Запропоновано програмну реалізацію моделі автоматизованого процесу розгортання середовища аналізу недекларованих можливостей Android застосунків на основі фазингу сірої скриньки з використанням AFL++ з FRIDA режимом.

Модель містить наступні модулі: декомпіляції Android застосунків з використанням утиліти apktool, компіляції фазера з інтегрованим фреймворком динамічного інструментування FRIDA та в ізольованому середовищі Docker, компіляції обгортки фазера в ізольованому середовищі Docker, доставлення зібраних компонентів на Android пристрій, користувацького інтерфейсу в командному рядку.

Наведені шаблони обгорток та налаштувань FRIDA для типових зразків функцій нативних бібліотек Android застосунків. Для збірки обгортки та фазера запропоновані налаштування компілятора Android NDK для типових архітектур Android пристроїв, а саме x86, x86_64, armeabi-v7a, arm64-v8a.

Експериментально підтверджено, що програмна реалізація запропоно-

ваної моделі автоматизації розгортання середовища аналізу недекларованих можливостей Android застосунків на основі фазингу сірої скриньки, знаходить вразливість пошкодження сегментації пам'яті, що полягає у виклику функції за адресою 0, після проходження гілки побайтової перевірки вхідного параметра розміром не менше 16 байтів в цільовому застосунку, за 47 секунд.

Було проведене експериментальне дослідження з визначення залежності часу, необхідного для знаходження вразливості методом фазингу, від складності програми, де за складність було взято кількість перевірок умовними операторами. Після вимірювання часу аналізу для 150 зразків вразливих додатків зі складністю від 1 до 15, було побудовано графік залежності часу фазингу від складності застосунку. Визначено, що залежність має експоненційну складність та швидко зростає, функція має вигляд $y = 3587.56e^{0.27x}$, її коефіцієнт детермінації $R^2 = 0.74$. Розкид часу виявився доволі істотним, що пояснюється хаотичністю процесу фазингу, запропоновано використовувати змінну оточення AFL_EXIT_ON_TIME для обмеження часу на пошук нового шляху виконання, враховуючи отримані результати залежності часу фазингу від складності застосунку.

Помічено, що пристрій на якому відбувався фазинг сильно нагрівається, що сповільнює процес фазингу, тому рекомендовано використовувати активне охолодження при тривалому аналізі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. GitHub. Public repositories related to fuzzing. — Режим доступу: https://github.com/search?q=fuzzing&type=Repositories.

2. The Art, Science, and Engineering of Fuzzing: A Survey / Valentin J.M. Manes, HyungSeok Han, Choongwoo Han та ін. — 2018. — Режим доступу: https://arxiv.org/abs/1812.00140.

3. Miller Barton P., Fredriksen Lars, So Bryan. An empirical study of the reliability of UNIX utilities. — 1990. — Режим доступу: https://dl.acm.org/doi/10.1145/96267.96279.

4. "lcamtuf" Zalewski Michał. AFL++ fuzzer. — Режим доступу: https://github.com/AFLplusplus/AFLplusplus.

5. AFL++ Findings. — Режим доступу: https://aflplus.plus/.

6. DARPA Cyber Grand Challenge 2016. — Режим доступу: https://www.darpa.mil/program/cyber-grand-challenge.

7. Greybox Fuzzing of Distributed Systems / Ruijie Meng, George Pîrlea, Abhik Roychoudhury, Ilya Sergey. — Режим доступу: https://arxiv.org/abs/2305.02601.

8. Scheduling Black-box Mutational Fuzzing / Maverick Woo, Sang Kil Cha, Samantha Gottlieb, David Brumley. — Режим доступу: https://www.semanticscholar.org/paper/Scheduling-black-box-mutational-fuzzing-Woo-Cha/7ae091ea6b9221fa8e7fe4c1295557fc1749a9d2.

9. Godefroid Patrice, Kiezun Adam, Levin Michael Y. Grammar-based whitebox fuzzing. — Режим доступу: https://dl.acm.org/doi/10.1145/1375581.1375607.

10. Pham Van-Thuan, Böhme Marcel, Roychoudhury Abhik. Model-Based Whitebox Fuzzing for Program Binaries. — Режим доступу: https://mboehme.github.io/paper/ASE16.pdf.

11. Böhme Marcel, Pham Van-Thuan, Roychoudhury Abhik. Coverage-based Greybox Fuzzing as Markov Chain. — Режим доступу: https://dl.acm.org/doi/10.1145/2976749.2978428.

12. The LibAFL Fuzzing Library, Corpus. — Режим доступу: https://aflplus.plus/libafl-book/core_concepts/corpus.html.

13. The LibAFL Fuzzing Library, Mutator. — Режим доступу: https://aflplus.plus/libafl-book/core_concepts/mutator.html.

14. The LibAFL Fuzzing Library, Feedback. — Режим доступу: https://aflplus.plus/libafl-book/core_concepts/feedback.html.

15. Rawat Sanjay, Jain Vivek, Kumar Ashish та ін. VUzzer: Application-aware Evolutionary Fuzzing. — Режим доступу: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/.

16. signal(7) — Linux manual page. — Режим доступу: https://man7.org/linux/man-pages/man7/signal.7.html.

17. Zalewski Michal. American Fuzzy Lop. — Режим доступу: https://github.com/google/AFL.

18. libFuzzer. — Режим доступу: https://llvm.org/docs/LibFuzzer.html.

19. Google. Honggfuzz. — Режим доступу: https://github.com/google/honggfuzz.

20. AnalytixLabs. A Complete Guide to Genetic Algorithm — Advantages, Limitations and More. — Режим доступу: https://medium.com/@byanalytixlabs/a-complete-guide-to-genetic-algorithm-advantages-limitations-more-738e87427dbb.

21. Bosamiya Jay. Genetic Fuzzing. — Режим доступу: https://www.jaybosamiya.com/blog/2017/05/27/genetic-fuzzing/.

22. Custom Mutators in AFL++. — Режим доступу: https://aflplus.plus/docs/custom_mutators/.

23. GSA-Fuzz: Optimize Seed Mutation with Gravitational Search Algorithm / Mingmin Lin, Yingpei Zeng, Ting Wu та ін. — Режим доступу: https://www.researchgate.net/publication/362034691_GSA-Fuzz_Optimize_Seed_Mutation_with_Gravitational_Search_Algorithm.

24. Khanna Sharad. What is AFL++ and Coverage-Based Fuzzing? — Режим доступу: https://blog.ritsec.club/posts/afl-under-hood/#what-is-afl-and-coverage-based-fuzzing.

25. Fuzzing a target with source code available. — Режим доступу: https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/best_practices.md#fuzzing-a-target-with-source-code-available.

26. Khanna Sharad. Coverage Instrumentation. — Режим доступу: https://blog.ritsec.club/posts/afl-under-hood/#coverage-instrumentation.

27. Technical "whitepaper"for afl-fuzz. — Режим доступу: https://lcamtuf.coredump.cx/afl/technical_details.txt.

28. High-performance binary-only instrumentation for afl-fuzz. — Режим доступу: https://github.com/AFLplusplus/AFLplusplus/blob/stable/qemu_mode/README.md.

29. FRIDA mode. — Режим доступу: https://github.com/AFLplusplus/AFLplusplus/blob/stable/frida_mode/README.md.

30. QEMU - a generic and open source machine emulator and virtualizer. — Режим доступу: https://www.qemu.org/.

31. FRIDA - dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. — Режим доступу: https://frida.re/.

32. Apktool: A tool for reverse engineering Android apk files. — Режим доступу: https://apktool.org/.

33. FRIDA: C API. — Режим доступу: https://frida.re/docs/c-api/.

34. Android NDK. — Режим доступу: https://developer.android.com/ndk.

35. Docker: Accelerated Container Application Development. — Режим доступу: https://www.docker.com/.

36. CMake: A Powerful Software Build System. — Режим доступу: https://cmake.org/.

37. GNU Make. — Режим доступу: https://www.gnu.org/software/make/.

38. Docker Engine: Volumes. — Режим доступу: https://docs.docker.com/storage/volumes/.

39. Guevel Eric Le. Android greybox fuzzing with AFL++ Frida mode. — Режим доступу: https://blog.quarkslab.com/android-greybox-fuzzing-with-afl-frida-mode.html.

40. JENV Library. — Режим доступу: https://github.com/quarkslab/android-fuzzing/tree/main/jenv.

41. Procedure Call Standard. — Режим доступу: https://developer.arm.com/documentation/102374/0102/Procedure-Call-Standard.

42. Android Debug Bridge (adb). — Режим доступу: https://developer.android.com/tools/adb.

ДОДАТОК А

Програмний код скрипта автоматизації фазингу

```
import argparse
import logging
import os
import subprocess
from time import time
from colorama import Fore, Style, init as colorama_init
from constants import *
from shutil import copyfile

colorama_init(autoreset=True)


class FuzzParams:
    arch = None
    file = None
    type = None
    folder = None
    target = None


def get_colored_text(text, color):
    return color + Style.BRIGHT + text + Style.RESET_ALL


def print_colored(text, color):
    print(get_colored_text(text, color))


def exit_handler():
    print_colored(EXIT_TEXT, Fore.RED)
    exit(1)
```

```
def get_random_dir_name():
    return os.urandom(8).hex() + "_" + str(round(time()))


def ensure_dir(dir_path):
    new_directory = os.path.join(os.environ['HOME'], dir_path)

    if not os.path.exists(new_directory):
        dlog(f"Creating directory {new_directory}")
        os.mkdir(new_directory)

    elif os.path.isfile(new_directory):
        elog(f"{new_directory} should be a directory")
        exit_handler()

    return new_directory


def choice_menu(title, options):
    print_colored(f"***** {title.upper()} *****", Fore.GREEN)
    print("Enter option number:")
    options.append("Exit") # Add exit option, so user can exit the program
    for i, option in enumerate(options, start=1):
        print(f"{i}. {option}")

    while True:
        try:
            choice = int(input("> "))
            if 1 <= choice <= len(options):
                if options[choice - 1] == "Exit":
                    exit_handler()
                dlog(f"Choice {options[choice - 1]} selected")
                return options[choice - 1]
            else:
                elog("Invalid choice. Please enter a number within the range of options.")
        except ValueError:
            elog("Invalid input. Please enter a number.")
```

```
        except KeyboardInterrupt:
            exit_handler()


def setup_logger(debug=False):
    if debug:
        logging.basicConfig(level=logging.DEBUG, format=LOG_FORMAT)
        dlog("DEBUG MODE IS ON")
    else:
        logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)


def ilog(message):
    logging.info(get_colored_text(message, Fore.MAGENTA))


def dlog(message):
    logging.debug(get_colored_text(message, Fore.BLUE))


def elog(message):
    logging.error(get_colored_text(message, Fore.RED))


def wizard():
    dlog("Wizard is running...")
    FuzzParams.arch = choice_menu("Select target architecture", ARCH_LIST)
    FuzzParams.file, FuzzParams.type = validate_file(input(get_colored_text("Select .apk
        file > ".upper(), Fore.GREEN)))


def validate_file(file_path):
    abs_path = os.path.abspath(file_path)
    file_type = ""

    # Check if the file exists
    if not os.path.exists(abs_path):
        elog(f"{abs_path} does not exist!")
        exit_handler()
```

```
    # Check if it's a file
    if not os.path.isfile(abs_path):
        elog(f"{abs_path} is not a file!")
        exit_handler()

    # Check if it has the .apk extension
    if abs_path.lower().endswith('.apk'):
        file_type = "APK"
    elif abs_path.lower().endswith('.so'):
        file_type = "SO"
    else:
        elog(f"{abs_path} is not an .apk or .so file!")
        exit_handler()

    dlog(f"{abs_path} selected successfully")
    return abs_path, file_type


def parse_arguments():
    parser = argparse.ArgumentParser(description="Argument parser for project")

    # Define arguments
    parser.add_argument("-w", "--wizard", action="store_true", help="Use wizard to
        configure analysis")
    parser.add_argument("-a", "--arch", choices=ARCH_LIST, help=f"Specify the
        architecture {ARCH_LIST}")
    parser.add_argument("-f", "--file", help="Specify the .apk or .so file path")
    parser.add_argument("-v", "--verbose", action="store_true", help="Verbose output
        ")

    # Parse arguments
    args = parser.parse_args()

    # Setup logger level
    setup_logger(args.verbose)

    if args.wizard:
        wizard()
```

```
    elif args.arch is not None and args.file is not None:
        FuzzParams.arch = args.arch
        FuzzParams.file, FuzzParams.type = validate_file(args.file)
    else:
        elog(f"Please specify either a --wizard or an --arch and --file options!")
        exit_handler()


def decompile_apk():
    ilog(f"Decompiling {FuzzParams.file}")

    decompile_dir = ensure_dir(FuzzParams.folder + '/' + DECOMPILE_FOLDER)
    cmd = ["apktool", "d", FuzzParams.file, "-f", "-o", decompile_dir]
    process = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.
        STDOUT, text=True)
    for line in process.stdout:
        dlog(line[:-1])
    process.wait()

    ilog(f"Decompiled apk in {decompile_dir}")


def get_target_lib():
    decompile_dir = FuzzParams.folder + '/' + DECOMPILE_FOLDER
    dlog(f"Looking for target lib in {decompile_dir}")

    arches = os.listdir(decompile_dir + '/lib')
    dlog(f"Found arches: {arches}")

    if FuzzParams.arch not in arches:
        elog(f"Target arch {FuzzParams.arch} not found in {decompile_dir}. Apk available
            arches: {arches}")
        exit_handler()

    ilog(f"Target arch {FuzzParams.arch} found!")
    libs = os.listdir(decompile_dir + '/lib/' + FuzzParams.arch)

    for lib_name in libs:
        dlog(f"Found lib: {lib_name}")
```

```
    target_lib = ""
    if len(libs) == 0:
        elog("No libs found!")
        exit_handler()
    elif len(libs) == 1:
        target_lib = libs[0]
    else:
        target_lib = choice_menu("Select target library", libs)

    ilog(f"Target lib: {target_lib}")
    target_lib = decompile_dir + '/lib/' + FuzzParams.arch + '/' + target_lib
    return target_lib


def run_afl():
    build_afl()
    build_afl_harness()
    check_adb_devices()
    push_afl()


def build_afl():
    ilog(f"Building AFL++ binaries...")
    ensure_dir(FuzzParams.folder + '/' + AFL_BUILD_FOLDER)
    ensure_dir(FuzzParams.folder + '/' + AFL_FINAL_FOLDER)
    cmd = ["docker", "run", "--rm",
           "-e", f"ANDROID_PLATFORM={ANDROID_PLATFORM_VERSION}",
           "-e", f"ANDROID_ARCH={FuzzParams.arch}",
           "-v",
           f"{FuzzParams.folder + '/' + AFL_BUILD_FOLDER}:/result",
           AFL_BUILDER_DOCKER_IMAGE]
    dlog(f"Starting docker image for AFL++ build: {' '.join(cmd)}")

    process = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.
        STDOUT, text=True)
    for line in process.stdout:
        dlog(line[:-1])
    process.wait()
```

```
    ilog(f"Built AFL++ binaries in {FuzzParams.folder + '/' + AFL_BUILD_FOLDER}")

    ilog(f"Copying final AFL++ files to {FuzzParams.folder + '/' +
        AFL_FINAL_FOLDER}")
    copyfile(FuzzParams.folder + '/' + AFL_BUILD_FOLDER + '/' + 'afl-fuzz',
             FuzzParams.folder + '/' + AFL_FINAL_FOLDER + '/' + 'afl-fuzz')
    copyfile(FuzzParams.folder + '/' + AFL_BUILD_FOLDER + '/' + 'afl-frida-trace.so
        ',
             FuzzParams.folder + '/' + AFL_FINAL_FOLDER + '/' + 'afl-frida-trace.
                 so')


def build_afl_harness():
    ilog(f"Building Harness binaries...")

    ensure_dir(FuzzParams.folder + '/' + AFL_HARNESS_FOLDER)
    ensure_dir(FuzzParams.folder + '/' + AFL_HARNESS_FOLDER + '/lib')
    ensure_dir(FuzzParams.folder + '/' + AFL_HARNESS_FOLDER + '/include')
    templates_path = os.path.dirname(os.path.abspath(__file__)) + '/templates/
        afl_harness/'
    destination_path = FuzzParams.folder + '/' + AFL_HARNESS_FOLDER + '/'

    ilog(f"Copying Harness source code templates to {destination_path}")
    copyfile(templates_path + 'afl.js', destination_path + 'afl.js')
    copyfile(templates_path + 'fuzz.c', destination_path + 'fuzz.c')
    copyfile(templates_path + 'libjenv.so', destination_path + 'lib/libjenv.so')
    copyfile(templates_path + 'jenv.h', destination_path + 'include/jenv.h')
    copyfile(templates_path + 'CMakeLists.txt', destination_path + 'CMakeLists.txt')
    copyfile(FuzzParams.target, destination_path + 'lib/' + os.path.basename(FuzzParams.
        target))

    ilog(f"Fuzzing files placed in {destination_path}, edit them, then press ENTER")

    input(get_colored_text("***** Press ENTER *****", Fore.GREEN))

    ilog(f"Compiling harness binaries...")
    cmd = ["docker", "run", "--rm",
           "-e", f"ANDROID_PLATFORM={ANDROID_PLATFORM_VERSION}",
```

```
            "-e", f"ANDROID_ARCH={FuzzParams.arch}",
            "-v",
            f"{FuzzParams.folder + '/' + AFL_HARNESS_FOLDER}:/app/build",
            "-v",
            f"{FuzzParams.folder + '/' + AFL_FINAL_FOLDER}:/result",
            AFL_HARNESS_DOCKER_IMAGE]
    dlog(f"Starting docker image for harness build: {' '.join(cmd)}")

    process = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.
        STDOUT, text=True)
    for line in process.stdout:
        dlog(line[:-1])
    process.wait()

    ilog(f"Built harness binaries in {FuzzParams.folder + '/' + AFL_FINAL_FOLDER}")

    ilog(f"Copying final harness files to {FuzzParams.folder + '/' +
        AFL_FINAL_FOLDER}")
    copyfile(destination_path + 'afl.js', FuzzParams.folder + '/' + AFL_FINAL_FOLDER
        + '/afl.js')
    copyfile(FuzzParams.target, FuzzParams.folder + '/' + AFL_FINAL_FOLDER + '/' +
        os.path.basename(FuzzParams.target))


def push_afl():
    ilog(f"Pushing fuzzing binaries to device...")

    dlog(f"Creating target directory: /data/local/tmp/{os.path.basename(FuzzParams.folder
        )}")
    cmd = ["adb", "shell", f"mkdir -p /data/local/tmp/{os.path.basename(FuzzParams.
        folder)}"]
    try:
        subprocess.run(cmd, check=True, stdout=subprocess.DEVNULL, stderr=subprocess
            .DEVNULL)
    except subprocess.CalledProcessError:
        elog(f"Can't create target directory: /data/local/tmp/{os.path.basename(
            FuzzParams.folder)}")
        exit_handler()
```

```
dlog(f"Creating in/out directories: /data/local/tmp/{os.path.basename(FuzzParams.
    folder)}/{{in,out}}")
cmd = ["adb", "shell", f"mkdir -p /data/local/tmp/{os.path.basename(FuzzParams.
    folder)}/{{in,out}}"]
try:
    subprocess.run(cmd, check=True, stdout=subprocess.DEVNULL, stderr=subprocess
        .DEVNULL)
except subprocess.CalledProcessError:
    elog(f"Can't create target directories: /data/local/tmp/{os.path.basename(
        FuzzParams.folder)}/{{in,out}}")
    exit_handler()

dlog(f"Creating dummy corpus: /data/local/tmp/{os.path.basename(FuzzParams.folder)
    }/in/sample.bin")
cmd = ["adb", "shell",
       f"dd if=/dev/urandom of=/data/local/tmp/{os.path.basename(FuzzParams.
           folder)}/in/sample.bin bs=1 count=1"]
try:
    subprocess.run(cmd, check=True, stdout=subprocess.DEVNULL, stderr=subprocess
        .DEVNULL)
except subprocess.CalledProcessError:
    elog(f"Can't create dummy corpus: /data/local/tmp/{os.path.basename(FuzzParams
        .folder)}/in/sample.bin")
    exit_handler()

files_to_push = os.listdir(FuzzParams.folder + '/' + AFL_FINAL_FOLDER)
for file_name in files_to_push:
    dlog(f"Pushing {FuzzParams.folder + '/' + AFL_FINAL_FOLDER + '/' +
        file_name}")
    cmd = ["adb", "push", FuzzParams.folder + '/' + AFL_FINAL_FOLDER + '/' +
        file_name,
           f"/data/local/tmp/{os.path.basename(FuzzParams.folder)}/{file_name}"]
    try:
        subprocess.run(cmd, check=True, stdout=subprocess.DEVNULL, stderr=
            subprocess.DEVNULL)
    except subprocess.CalledProcessError:
        elog(f"Can't push fuzzing files to device!")
        exit_handler()
```

```
dlog(f"Granting execution permissions on all files in: /data/local/tmp/{os.path.
    basename(FuzzParams.folder)}")
cmd = ["adb", "shell", f"chmod +x /data/local/tmp/{os.path.basename(FuzzParams.
    folder)}/*"]
try:
    subprocess.run(cmd, check=True, stdout=subprocess.DEVNULL, stderr=subprocess
        .DEVNULL)
except subprocess.CalledProcessError:
    elog(f"Can't create target directory: /data/local/tmp/{os.path.basename(
        FuzzParams.folder)}")
    exit_handler()

dlog(f"Pushing {os.path.dirname(os.path.abspath(__file__)) + '/templates/afl_run/
    README.md'}")
cmd = ["adb", "push", os.path.dirname(os.path.abspath(__file__)) + '/templates/
    afl_run/README.md',
        f"/data/local/tmp/{os.path.basename(FuzzParams.folder)}/README.md"]
try:
    subprocess.run(cmd, check=True, stdout=subprocess.DEVNULL, stderr=subprocess
        .DEVNULL)
except subprocess.CalledProcessError:
    elog(f"Can't push fuzzing files to device!")
    exit_handler()

dlog(f"Pushing {os.path.dirname(os.path.abspath(__file__)) + '/templates/
    afl_harness/libjenv.so'}")
cmd = ["adb", "push", os.path.dirname(os.path.abspath(__file__)) + '/templates/
    afl_harness/libjenv.so',
        f"/data/local/tmp/{os.path.basename(FuzzParams.folder)}/libjenv.so"]
try:
    subprocess.run(cmd, check=True, stdout=subprocess.DEVNULL, stderr=subprocess
        .DEVNULL)
except subprocess.CalledProcessError:
    elog(f"Can't push fuzzing files to device!")
    exit_handler()

ilog("That all, now you need to execute 'adb shell' (make sure, that user is root)")
ilog(f"Execute 'cd /data/local/tmp/{os.path.basename(FuzzParams.folder)}' to navigate
     to fuzzing folder")
```

```
    ilog(f"Read instructions in README.md to start fuzzing")
    ilog(f"Good luck =)")


def check_adb_devices():
    try:
        result = subprocess.run(['adb', 'devices'], capture_output=True, text=True, check=
            True)
        output = result.stdout
        devices = [line.split('\t')[0] for line in output.splitlines()[1:] if line.strip() != '']
        if devices:
            ilog("Found adb devices:")
            for device in devices:
                ilog(device)
        else:
            elog("No devices connected via ADB.")
            exit_handler()
    except subprocess.CalledProcessError:
        elog("ADB is not installed or not properly configured.")
        exit_handler()
```