МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧОРНОМОРСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ПЕТРА МОГИЛИ

Хортюк Ярослав Ігорович

УДК 004.89

Автоматизація підвищення надійності програмних інтерфейсів на основі фазинг-тестування

124 - Системний аналіз

Автореферат
магістерської наукової роботи на здобуття освітньої кваліфікації
«Магістр системного аналізу»

Миколаїв – 2020

Магістерська наукова робота є рукопис.

Робота виконана в Чорноморському національному університеті імені Петра Могили Міністерства освіти і науки України на кафедрі інтелектуальних інформаційних систем

Науковий керівник: к.т.н., доцент кафедри інтелектуальних інформаційних систем Кондратенко Галина Володимирівна

Рецензент: доцент, к.ф.-м.н, Пузирьов С. В.

Захист відбудеться «26» лютого 2020 р. о $9^{\underline{30}}$ год. на засіданні екзаменаційної комісії (ауд. 2-403) у Чорноморському національному університеті імені Петра Могили за адресою: 54003, м. Миколаїв, вул. 68-ми Десантників, 10.

З магістерською науковою роботою можна ознайомитися в бібліотеці Чорноморського національного університету імені Петра Могили за адресою: 54003, м. Миколаїв, вул. 68-ми Десантників, 10.

Автореферат представлений «___» лютого 2020 р.

Секретар
екзаменаційної комісії,
к.пед.н., доцент Н. М. Болюбаш

ЗАГАЛЬНА ХАРАКТЕРИСТИКА РОБОТИ

У даній роботі для визначення проблемних програмних інтерфейсів в додатках пропонується використовувати попередній статичний аналіз документації з інтелектуальним виявленням залежностей між окремими запитами, які можуть бути представлені у вигляді незбалансованого дерева. Сукупність проаналізованих характеристик дає можливість знизити кількість помилкових результатів під час проведення аналізу.

Характеристики збираються на основі графового представлення статичної документації, що дозволяє виділити залежність між різними задукоментованими програмними інтерфейсами в додатку.

На основі запропонованого алгоритму виявлення залежностей була реалізована система фазинг-тестування для послідовної інтеграції. Ця система здатна аналізувати документацію, виявляти залежності та використовуючи наявну інформацію виконувати коректну послідовність запитів та регулярно перевірити програмний код на наявність можливих помилок.

Об'єктом дослідження даної роботи є проекти з задукоментованими програмними інтерфейсами

Предметом дослідження автоматизований статичний аналіз документації, фазинг-тестування та алгоритми генерації наборів вхідних даних.

Метою роботи є підвищення ефективності фазинг-тестування за допомогою статичного аналізу та виявлення залежностей між програмними інтерфейсами. Для досягнення даної цілі було поставлено наступні завдання:

1. дослідити наявні підходи до аналізу програмних інтерфейсів на предмет наявності вразливостей;
2. ознайомитись з стандартом документації API;
3. дослідити інтерпретацію програмної документації та розбір у вигляді графа характеристик;
4. визначити набір характеристик коду, комбінацій для аналізу якості коду;

5. розробка програмного засобу, реалізація технологічної ранжирування функцій та виявлення залежностей між програмними інтерфейсами.

Інформаційну базу дослідження складають матеріали, зібрані в процесі проходження навчальної практики, а також знання, отримані при вивченні навчальних дисциплін. **Наукова новизна** представлених досліджень досліджується в розробці автоматизованої системи генерації фазинг-тестів на основі статичного аналізу документації.

Практична значущість магістерської роботи передбачається в застосування практичної реалізації алгоритму у виявленні програмних інтерфейсів, які є найбільш підтвердженими до вразливостей, за допомогою фазинг-тестування.

Апробація:

1. Хортюк Я.І., Кондратенко Г.В. Автоматизація підвищення надійності програмних інтерфейсів на основі фазинг-тестування. Могилянські читання – 2019: тези доповідей: Комп’ютерні науки. Технічні науки, Миколаїв: ЧНУ, 11-16 листопада, 2019. – С. 9-11.
2. Хортюк Я.І., Кондратенко Г.В. Автоматизація підвищення надійності програмних інтерфейсів на основі фазинг-тестування. Інтелектуальні інформаційні системи – 2020: тези доповідей, Миколаїв: ЧНУ, 28-31 січня, 2020. – С. 29-31.
3. Подано англомовну публікацію "Automated Improvement of Programming Interfaces Based On Fuzz Testing" на конференцію DESSERT’2020 (The 11th International Conference on Dependable Systems, Services and Technologies)

Структура магістерської наукової роботи. Магістерська наукова робота складається із вступу, 3 розділів, висновків, 3 додатків. Загальний обсяг роботи складає 121 сторінки, 18 рисунків, 6 таблиць та 65 посилань на літературні джерела.

ОСНОВНИЙ ЗМІСТ РОБОТИ

У вступі подано загальну характеристику досліджуваної теми, обґрунтовано актуальність магістерської наукової роботи, сформульовано мету, завдання досліджень, відзначено наукову новизну та практичну цінність отриманих результатів, подано інформацію про апробацію, структуру та обсяг роботи.

У першому розділі висвітлена проблема виявлення помилок в програмних інтерфейсах. Наведені основні терміни та визначення, що стосуються фазинг-тестування. Розглянуто нормативні документи щодо захисту інформації та можливості їх виконання при роботі з ПО з закритим вихідним кодом. Актуальність аналізу програм без вихідного коду багато в чому обумовлена високим рівнем поширення комерційного програмного забезпечення, вихідний код якого являє собою інтелектуальну власність тієї чи іншої компанії, які як правило не зацікавлені в публікації або передачі своїх розробок третім особам. Отже, така ситуація сприяє появі як випадкових помилок і вразливостей, так і навмисно внесених в код недекларованих можливостей. Це особливо актуально для України у зв'язку із застосуванням комерційного ПЗ закордонного виробництва для обробки і захисту інформації обмеженого доступу (інформація, доступ до якої обмежується законами).

У другому розділі розглядаються попередні дослідження в області фазинг тестувавання. Наводиться опис моделі фазерів та алгоритми передбачення дефектів. Детально вивчено підхід з тестування за допомогою направленого фазингу сірої скриньки.

Фазинг - повторюваний процес модифікації даних, що обробляються програмою. Метою є виявлення таких входів, які призводять до виявлення вразливостей через некоректну обробки даних. Прикладом такої вразливості служить переповнення буфера. По контексту аналізу досліджуваної програми фазингу ділиться на фазинг чорної скриньки, фазинг білої скриньки і, щось середнє – фазинг сірої скриньки. Фазинг чорної скриньки - мутує дані випадковим чином, а інформацію про процес отримує тільки з виведення досліджуваної програми. З іншого боку, фазинг білої скриньки дозволяє отримувати набагато більше

інформації, наприклад, за рахунок інструментації вихідного коду, символьного виконання і інших технік.

Направлений фазинг сірої скриньки відрізняється від звичайного алгоритмом роботи на етапі планування. На ньому пріоритет надається на такі входи, які дозволять за найменшу кількість мутацій отримати покриття потрібного фрагменту програмного інтерфейсу.

Направлений сірої скриньки фазинг дозволяє вирішити такі завдання:

- пошук вразливостей в програмному інтерфейсі проекту;
- відтворення помилок по трасі виконання програми;
- підтвердження результатів роботи статичних аналізаторів.

Для вирішення цих завдань існує підхід з використанням символьного виконання, який дозволяє через рішення складних рівнянь підібрати дані, які дозволять досягти і проаналізувати задані ділянки коду. Однак даний підхід є дуже витратним з погляду ресурсів і часу. Для вирішення цього недоліку існує метод фазингу з додатковим аналізом інструментації, який дозволяє перебирати на порядок більшу кількість вхідних даних на етапі аналізу та формувати більш точні входи.

Проведений в рамках даного розділу аналіз існуючих моделей, алгоритмів і програмних реалізацій для динамічного і статичного аналізу коду дозволив виділити основні підходи, які використовуються для пошуку вразливостей в умовах відсутності вихідного коду, а також недоліки, які притаманні цим підходам. Так, для автоматизованого статичного аналізу можуть використовуватися системи пошуку потенційних вразливостей по шаблонами або символічне виконання програми. Однак фундаментальні проблеми, пов'язані з декомпіляцією і аналізом коду не дозволяють повністю відновити високорівневий аналог скомпільованого модуля, що створює труднощі для статичного аналізу, а зростання числа потенційних шляхів виконання в рамках символьного виконання веде до проблеми «Експоненціального вибуху».

Для вирішення цих проблем застосовується динамічний аналіз, що частково вирішує проблеми статичного аналізу, проте не вирішує питання з покриттям коду

при тестуванні, а також породжує проблеми, пов'язані з впливом аналізатору на тестований додаток. При цьому для деяких інструментів тестування може бути характерним високий рівень споживання обчислювальних ресурсів.

У третьому розділі розглянуто проблему та алгоритм реалізації автоматичного фазингу на основі задокументованого програмного інтерфейсу OpenAPI. В розділі детально описані процеси аналізу документації, алгоритм генерації тестів та тестових даних. Проведений невеликий експеримент з метою підтвердження тверджень щодо ефективності розглянутого методу, шляхом імплементації веб-додатку з завчасно закладеними помилками.

Основний алгоритм для генерації тестів, що використовується, показаний на малюнку 3.4 у вигляді псевдокоду нотації Python. Алгоритм починається обробкою специфікації OpenAPI. Результатом цієї обробки є набір типів запитів, позначених reqSet на (рис 1), та їх залежностей. Алгоритм обчислює набір послідовностей запитів, позначається seqSet і спочатку порожній. На кожному етапі ітерації його основного циклу, алгоритм обчислює всі дійсні послідовності запитів seqSet довжини n, починаючи з n = 1 перед переходом до n + 1 і так далі, поки не буде досягнуто визначеного користувачем maxLength.

Обчислення seqSet робиться в два етапи.

По-перше, набір дійсних послідовностей запитів довжиною n – 1 розширюється, додаючи в кінці кожної послідовності один новий запит, залежність якого задовольняється, для всіх можливих запитів, як описано у функції EXTEND. Функція DEPENDENCIES перевіряє, чи наявні всі типи об'єктів в останньому запиті, позначеному CONSUMES (req), які в свою чергу були створені деякою відповіддю в попередній послідовності запитів, позначається PRODUCES (seq). Якщо всі залежності задоволені, нова послідовність довжини n зберігається, інакше вона відкидається.

По-друге, кожен щойно розширений запит, послідовність залежностей якого задовольняється, генерується по одному таким чином, як описано у функції RENDER. Для кожного щойно доданого запиту обчислюється список усіх нечітких примітивних типів. Такі типи параметрів ідентифікуються за допомогою сигнатури

fuzzable у коді (рис. 1). Потім, кожен нечіткий примітивний тип у запиті замінюється на одне конкретне значення цього типу, взяте з скінченного та невеликого словника значень. Функція RENDER генерує всі можливі такі комбінації. Необхідно відзначити, що не кожен з вищеописаних програмних комплексів може виконувати аналіз будь-якого типу програми і визначати будь-який тип помилок.

```
Inputs: openapi spec, maxLength
# Дерево запитів побудоного з  специвікації OpenAPI reqSet = PROCESS(openapi spec)
# Набір ланцюгів виконання (на початку виконання пустий) seqSet = fg
# Головний цикл: виконується до моменту утворення максимально допустимої довжини послідовності
n = 1
while (n =< maxLength):
seqSet = EXTEND(seqSet, reqSet)
    seqSet = RENDER(seqSet)
    n = n + 1
# Розширення всіх послідовностей в seqSet через додавання
# нових запитів всі залежності яких задоволені
def EXTEND(seqSet, reqSet):
    newSeqSet = fg
    for seq in seqSet:
        for req in reqSet:
            if DEPENDENCIES(seq, req):
                newSeqSet = newSeqSet + concat(seq, req)
    return newSeqSet

# Фазинг доданих запитів за домогою словників,
# виконання кожного запиту та збереження успішних
def RENDER(seqSet):
    newSeqSet = fg
    for seq in seqSet:
        req = last request in(seq)
        ~V = tuple of fuzzable types in(req)
        for ~v in ~V :
            newReq = concretize(req, ~v)
            newSeq = concat(seq, newReq)
            response = EXECUTE(newSeq)
            if response has a valid code:
                newSeqSet = newSeqSet + newSeq
            else:
                log error
    return newSeqSet
# Перевірка того, що всі залежності були отримані
# з попередніх запитів перед виконанням запиту з поточної послідовності
def DEPENDENCIES(seq, req):
    if CONSUMES(req)   PRODUCES(seq):
        return True
    else:
        return False
# Затребувані об'єкти запиту
def CONSUMES(req):
    return object types required in(req)
# Об'єкти створені під час виконання попередніх послідовностей
def PRODUCES(seq):
    dynamicObjects = fg
    for req in seq:
        newObjs = objects produced in response of(req)
        dynamicObjects = dynamicObjects + newObjs
  return dynamicObjects
```

Рис. 1 Алгоритм виконання побудови та виконання фазинг-тесту

Функція EXECUTE виконує кожен запит з послідовності по одному, кожен раз перевіряючи коректність відповіді, вилучаючи та запам'ятовуючи динамічні об'єкти (якщо такі є) та підставку їх в наступні запити в послідовності, якщо це потрібно і визначено аналізом залежності. Результатом виконання функції EXECUTE є відповідь отримана для останнього, нещодавно доданого запиту в послідовності. Важливим моментом є те, що якщо послідовність запитів створює більше ніж один динамічний об'єкт заданого типу, функція EXECUTE запам’ятає всі ці об’єкти та якщо буде потрібно, подальшими запитами, їх буде відтворено в точному порядку, в якому вони були створені результатами відповідей на попередні запити.

Точніше кажучи, функція EXECUTE не буде намагатися випадковим чином впорядковувати значення таких об'єктів. Якщо динамічний об'єкт передається як аргумент до наступного запиту і після цього "знищується", тобто він стає непридатним згодом, алгоритм виявить це, отримавши недійсну відповідь з кодом помилки 400 або 500 при спробі повторно використовувати цей непридатний об'єкт, а після цього дана послідовність буде відкинутою.

Для кожного нечіткого примітивного типу алгоритм на рис. 3.4 використовує невеликий набір значень цього типу, який називається словником, і вибирає одне з цих значень для конкретизації. Наприклад, для нечіткого цілого типу, алгоритм може використовувати невеликий словник зі значеннями 0, 1 і -10, тоді як для нечіткого типу рядок, словник може бути визначений зі значеннями "SampleString", порожній рядок і одна дуже довга послідовність символів. Користувач визначає ці словники самостійно в залежності від задач.

За замовчуванням функція RENDER на (рис. 1) генерує всі можливі комбінації значень словника для кожного запиту із кількома нечіткими типами. Для великих словників це може призвести до астрономічно великої кількісті комбінацій. У цьому випадку більш масштабований варіант полягає у випадковій вибірці кожного словника для одного (або кількох) значень, або використовувати комбінаторні

алгоритми тестування для повноцінного покриття кожного значення словника, або кожна пара значень, але не кожен k-кортеж.

При тестуванні алгоритму використовувались невеликі словники та стандартна реалізація функції RENDER (рис 1). Функція EXTEND генерує всі послідовності запитів довжиною n + 1, залежності яких є задоволений. Оскільки n збільшується при кожній ітерації основного циклу, то загальний алгоритм виконує пошук вшир в просторі пошуку, визначеному всіма можливими послідовностями запитів.

Четвертому розділі проаналізовано і з’ясовано вимоги до приміщення, в якому можуть знаходитись працівники ІТ-сфери. Організовано правильне розташування робочих місць та наведені вимоги про розміщення персональних комп’ютерів і периферійних пристроїв. Встановлені правила безпеки під час роботи з персональними комп’ютерами. Висвітлено питання щодо електробезпеки на робочому місці працівника ІТ-сфери.

В п’ятому розділі розроблена лабораторна робота з дисципліни «Введення в фазинг-тестування» з етапами її виконання на тему «Генетичні алгоритми для генерації наборів вхідних даних».

ЗАГАЛЬНІ ВИСНОВКИ

В процесі виконання магістерської роботи було досліджено існуючі методи та підходи покращення програмних інтерфейсів шляхом фазинг-тестування. Ми дослідили та реалізували алгоритм автоматичного інтелектуального фазингу програмних інтерфейсів через їх REST API. Програмна реалізація аналізує специфікацію OpenAPI REST API та інтелектуально генерує тести, визначаючи залежності між типами запитів та вивчаючи недійсні комбінації запитів на основі відповідей служби. Під час розробки експериментально перевірили, що за рахунок ретельно підібраного словника, стратегії пошуку та схеми агрегації повторних помилок можливо ефективно здійснювати фазинг сірої скриньки для хмарних сервісів одночасно обрізаючи великий обсяг пошуку можливого запиту послідовності. Незважаючи на те, що результати є попередніми та програмна реалізація має певні обмеження, вони обнадіють на покращення та широке застосування подібного підходу в майбутньому.

АНОТАЦІЯ

Хортюк Я.І. Автоматизація підвищення надійності програмних інтерфейсів на основі фазинг-тестування. – На правах рукопису.

Магістерська наукова робота на здобуття освітньої кваліфікації «Магістр системного аналізу». – Чорноморський національний університет імені Петра Могили, Миколаїв, 2020.

У даній роботі для визначення проблемних програмних інтерфейсів в додатках пропонується використовувати попередній статичний аналіз документації з інтелектуальним виявленням залежностей між окремими запитами, які можуть бути представлені у вигляді незбалансованого дерева. Сукупність проаналізованих характеристик дає можливість знизити кількість помилкових результатів під час проведення аналізу.

Об'єктом дослідження даної роботи є проекти з задукоментованими програмними інтерфейсами. Предметом дослідження є автоматизований статичний аналіз документації, фазинг-тестування та алгоритми генерації наборів вхідних даних. Метою роботи є підвищення ефективності фазинг-тестування за допомогою статичного аналізу та виявлення залежностей між програмними інтерфейсами

Інформаційну базу дослідження складають матеріали, зібрані в процесі проходження навчальної практики, а також знання, отримані при вивченні навчальних дисциплін. Наукова новизна представлених досліджень досліджується в розробці автоматизованої системи генерації фазинг-тестів на основі статичного аналізу документації. Практична значущість магістерської роботи передбачається в застосування програмної реалізації алгоритму у виявленні програмних інтерфейсів, які є найбільш підтвердженими до вразливостей, за допомогою фазинг-тестування.

Магістерська робота складається з трьох розділів, спеціальної частини з охорони праці та безпеки у надзвичайних ситуаціях, методичної частини, додатків.

В цілому робота складається із 121 сторінки, 6 таблиць, 18 рисунків, в тому числі фахова частина складається із 48 сторінок, 2 таблиць, 15 рисунків.

Ключові слова: тестування, фазинг, програмні інтерфейси, rest api

ABSTRACT

Khortiuk Yaroslav. Automated Improvement of Programming Interfaces Based On Fuzz Testing – On the rights of the manuscript.

Master's scientific work for obtaining an educational qualification "Master of Systems Analysis". – Petro Mohyla Black Sea National University, Mykolaiv, 2020.

In this masters science work to identify problematic software interfaces in applications proposed to use a preliminary static analysis of documentation with intelligent detection of dependencies between individual queries that can be represented as an unbalanced tree. The combination of the analyzed characteristics makes it possible to reduce the number of erroneous results during the analysis.

The object of study of this work are projects with the recommended software interfaces. The subject of the study is automated static document analysis, file testing and algorithms for generating input data sets. The purpose of this work is to increase the efficiency of fuzz-testing by means of static analysis and to identify dependencies between software interfaces

The information base of the research consists of the materials collected in the course of training, as well as the knowledge gained in the study of disciplines. The scientific novelty of the presented research is investigated in the development of an automated system for the generation of fuzz tests based on a static analysis of the documentation. The practical significance of the master's work is envisaged in the application of the software implementation of the algorithm in the detection of software interfaces that are most vulnerable to vulnerability through fuzz testing.

The master's work consists of three sections, a special part on Occupational Safety and Health, a methodical part, annexes. In total, the work consists of 121 pages, 6 tables, 18 drawings, including the professional part consists of 48 pages, 2 tables, 15 drawings.

Keywords: testing, fuzzing, program interfaces, rest api.