Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

Гаврилов Кирилл Анатольевич

Выпускная квалификационная работа

Разработка фаззера для языков C/C++ на основе универсальной фаззинг платформы

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5156.2019
«Современное программирование»

Научный руководитель:
к.ф.-м.н. Д. С. Шалымов

Рецензент:
ведущий инженер ООО «Техкомпания Хуавэй»
М. С. Пелевин

Санкт-Петербург
2023 г.

Содержание

Введение

С момента начала разработки программного обеспечения стоит проблема тестирования и выявления дефектов и неисправностей. С ростом сложности программ и увеличением размера кодовой базы важность проблемы только растёт.

Почти всегда необходимо не только установить факт того, что программа выдаёт неправильный результат, не завершает исполнение или завершается аварийно, но так же и определить набор входных данных, на которых это произошло. Для этого часто используют фаззинг.

Фаззинг - метод тестирования программного обеспечения, который основан на генерации входных данных для тестируемого программного обеспечения. Фаззеры стараются генерировать такие данные, которые скорее всего выявят проблемы в программе: заведомо некорректные данные, данные неверного формата, краевые случаи данных.

Для генерации новых данных часто используется уже готовый корпус тестовых случаев. Его загружают перед стартом фаззинга и в дальнейшем к нему применяют различные мутации, комбинируют тестовые данные друг с другом, чтобы получить новые входные данные. Для мутаций часто используют случайные величины, эвристики, но нередко и машинное обучение.

После генерации данные передаются на вход тестируемой программе. Поведение программы анализируется: собирается информация о том, по какому пути пошло исполнение программы, завершилась ли программа аварийно, каков результат выполнения программы. Эти данные передаются фаззеру и на их основе принимается решение о том, как продолжать фаззинг и генерировать новые тестовые случаи.

Таким образом, фаззер состоит из двух независимых частей. В первой части происходит генерация и обработка данных об исполнении программы. Во второй части происходит непосредственно запуск программы на переданных входных данных.

Часто обе части разрабатываются вместе для фаззинга программ на одном из языков программирования:

1. LibFuzzer [1] — фаззер для C/C++.

2. Jsfuzz [3] — фаззер для JavaScript.

Разработка общей фаззинг платформы позволила бы не реализовывать алгоритм фаззинга для каждого языка. Чтобы применить платформу к новому языку программирования, достаточно было бы реализовать вторую часть фаззера для этого языка, которая запускает программу и собирает информацию об исполнении.

Постановка задачи

Целью работы является разработка фаззера для C/C++ на основе уже реализованной фаззинг платформы [4].

Были поставлены следующие задачи:

1. Провести обзор и анализ работы уже существующих фаззеров для C/C++, либо поддерживающих несколько языков программирования.
2. Разработать программный аппарат для запуска программ на C/C++ с произвольными входными данными и получения данных об исполнении.
3. Разработать протокол обмена данными с фаззинг платформой.
4. Разработать программный аппарат для запуска универсальной фаззинг платформы и передачи данных между платформой и программой на C/C++.
5. Провести сравнение разработанного фаззера с аналогами.

1. Обзорный раздел по предметной области

В данной главе будут даны базовые определения, представлен обзор фаззеров, которые решают схожие задачи. Описана универсальная фаззинг платформа.

1.1. Фаззинг

Фаззинг — техника тестирования программного обеспечения, часто автоматическая или полуавтоматическая, заключающаяся в передаче приложению на вход неправильных, неожиданных или случайных данных [5].

Фаззеры подразделяют на несколько видов:

1. «BlackBox» — фаззеры, которым доступен только вывод программы. Так как в таком случае не доступна информация о выполнении программы, то невозможно модифицировать входные данные образом, отличным от случайного, что ограничивает возможности таких фазеров.

2. «WhiteBox» — фаззеры, которым доступен исходный код программы. Такие фаззеры анализируют исходный код и с помощью математического аппарата пытаются найти такие входные данные, которые обеспечат полное покрытие.

   Часто можно использовать комбинированный метод: сначала собрать несколько наборов входных данных, которые покрывает основные пути, а затем запустить «WhiteBox» фаззер. Это позволяет сократить сложность вычислений, так как в общем случае задача нахождения входных данных, покрывающих определенный путь в исходном коде, является NP полной.

3. «GreyBox» — фаззеры, которым не доступен исходный код программы, однако они получают полную информацию об исполнении программы. После запуска программы, они получают такие данные как: время исполнения, путь исполнения, уточнения об ограничениях на входные данные.

1.2. LLVM

LLVM — набор инструментов для разработки компиляторов программного кода. С помощью LLVM происходит компиляция программного кода на C++.

При компиляции кода на C++, код сначала преобразовывается в код на языке промежуточного представления (LLVM IR). В таком состоянии, код уже можно запускать и анализировать, а так же изменять. Затем код попадает в оптимизатор LLVM, который применяет различные подходы для улучшения быстродействия кода.

После оптимизации код на языке промежуточного представления транслируется в машинной код, совместимый с платформой, для которой идёт сборка программного кода.

1.3. Универсальная фаззинг платформа

Универсальная фаззинг платформа — «GreyBox» фаззер, реализованный на языке Kotlin [4]. Она позволяет в обобщенном виде проводить фаззинг функций, которые могут иметь аргументами такие типы данных как: примитивы (целые числа различной разрядности, дробные числа), пользовательские структуры с вложенными структурами, списки из пользовательских структур. В упрощенном виде, платформа имеет следующий пользовательский интерфейс

```
suspend fun <T, R, D, F> runFuzzing(
    provider: ValueProvider<T, R, D>,
    description: D,
    random: Random = Random(0),
    configuration: Configuration = Configuration(),
    handle: suspend (description: D, values: List<R>) -> F
)
```

Основным преимуществом платформы является то, что она не зависит от языка программирования, на котором написан код программы. Для того

чтобы применять платформу к разным языкам программирования, нужно лишь менять параметр «handle», внутри которого запускается программа.

В параметре «description» необходимо передать описание метода, который подвергнут фаззингу: его аргументы, возвращаемое значение, иные атрибуты.

В параметре «provider» необходимо передать программный аппарат, который позволяет произвести соответствие между типами данных, с которыми оперирует универсальная фаззинг платформа и типами данных, которыми оперирует язык программирования на котором написана программа подвергнутая фаззингу.

В параметре «configuration» возможна передача параметров для фаззинга: вероятности мутации тестовых данных, вероятность переиспользования тестовых данных, глубина рекурсии, количество запусков программы и др.

В параметре «random» возможна передача конфигурации случайности в программе. Её можно использовать, если необходимо зафиксировать случайность и убрать разницу между двумя последовательными запусками фаззинга.

1.4. Аналоги

В этой секции будет произведен обзор аналогичных фаззеров, которые позволяют работать со многими языками программирования и имеют возможность расширения для начала работы с новым языком программирования.

1. Peach Fuzzer [6] — фаззер для C/C++, Java, Python, Ruby и др. Обладает выделенным ядром, к которому можно подключать новые языки. Пользователю предоставляется набор предопределённых окружений, с помощью которых можно запускать программу, фаззинг которой необходимо провести. Также необходимо описать формат входных данных, которые будут переданы программе в этом окружении. Однако, этот процесс переусложнен несколькими факторами:

   (a) Порог вхождения в использование фаззера очень высок. Предопределённые окружения могут иметь побочные эффекты работы в них, поведение программы может искажаться. Необходимо разбираться не только в специфике работы программы на новом языке

программирования, но и во взаимодействии окружения и программы на этом языке программирования.

(b) Настройка фаззинга происходит посредством XML файлов без каких-либо проверок корректности.

(c) Peach Fuzzer работает с исполняемыми файлами на недостаточно абстрактном уровне. Из-за этого процесс добавления нового языка может быть осложнён необходимостью добавлять в Peach Fuzzer код для сборки информации об исполнении программы, для модификации исполняемой программы.

Возможно понадобится добавление нового окружения для запуска программы на новом языке программирования и накопления данных о запуске программ. Из-за массивности проекта такие доработки трудноосуществимы.

2. Radamsa [17] — «BlackBox» фаззер для различных языков программирования. Не имеет ограничений на выбор языка программирования, так как работает непосредственно с исполняемым файлом и данными, подаваемыми ему на вход.

   При этом Radamsa мутирует входные данные опираясь лишь на случайность и результат работы программы, не принимая во внимание путь, по которому пошло исполнение кода. Это накладывает ограничения на мощность фаззера.

3. Honggfuzz [18] — «GreyBox» фаззер, предназначенный преимущественно для программ на C/C++. Обладает возможностью фаззинга программ на других языках программирования, однако этому препятствует ряд факторов:

   (a) Фаззер используется для фаззинга программ преимущественно на языках C/C++/Go/Rust, а так же для фаззинга интернет протоколов, интерфейсов серверов, форматов хранения данных. Таким образом алгоритмы фаззера больше направлены на эти цели, снижая их общность и полезность в иных ситуациях.

(b) Система генерации входных данных является недостаточно абстрактной. Фаззер не предоставляет инструменты для генерации произвольных пользовательских структур. Для каждого нового языка необходимо реализовывать программный аппарат, который позволит генерировать входные данные по заранее описанной схеме, и передавать его фаззеру. В процессе фаззинга этот аппарат будет использоваться с различными параметрами, и структуры, которые он сгенерирует, будут переданы на вход программе, подвергнутой фаззингу.

При этом универсальная фаззинг платформа имеет возможность самостоятельной генерации пользовательских структур по заранее заданной схеме данных. Таким образом, для подключения к универсальной фаззинг платформе нового языка программирования необходимо только разработать программный аппарат, который позволит передавать сгенерированные структуры данных на вход программе, подвергнутой фаззингу.

1.5. Обзор литературы

В этом разделе описаны научные статьи, в которых изучается вопрос разработки фаззера, поддерживающего несколько языков программирования.

1.5.1 PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems

В рамках работы [19] авторы описывают устройство фаззера, который может работать с программой, написанной сразу на нескольких языках программирования. Авторы приводят сравнения производительности и показывают, что их решение оказывается лучше, чем традиционные фаззеры, рассчитанные на работу с одним языком программирования, при сравнении на проектах, написанных на нескольких языках программирования.

Авторы реализовали программный аппарат, который транслирует каждый из исходных файлов, независимо от того, на каком языке он написан, в файл на языке абстрактного представления, описанного в работе. Таким образом, проект, написанный на нескольких языках программирования можно

рассматривать как программу, написанную на языке абстрактного представления и применять к ней как традиционные методы фаззинга, так и новые методы, применимые к языку абстрактного представления.

Теоретически, разработанный фаззер можно было бы подключать к новым языкам программирования. Однако, на практике, это требовало бы разработки программного аппарата, который мог бы транслировать файлы на языке программирования в файлы на языке абстрактного представления. Для многих языков со сложным синтаксисом и большим набором возможностей, коим и является C++, написание такого аппарата крайне затруднительно и практически эквивалентно написанию компилятора языка.

1.6. Вывод

Приведённые аналоги, поддерживающие подключение новых языков программирования, либо имеют недостатки по сравнению с универсальной фаззинг платформой, либо имеют ограниченный функционал.

Приведённая научная статья описывает систему, которая не может быть легко применена к программам, написанным на новом языке программирования.

2. Программная реализация

Программная реализация использует C++, Python, LLVM версии 14 [7], систему сборки CMake версии 3.24 [8], JDK 19 [9], систему сборки Gradle версии 7.4.2 [10].

2.1. Основные компоненты

Назовём **исполнителем C++** программу, позволяющую принимать на вход набор LLVM IR файлов и запускать любую заданную функцию, находящуюся в файлах, с любым заданным набором входных данных. А так же собирать информацию о поведении функции: времени её выполнения, пути выполнения, результате выполнения.

Назовём **предобработчиком исходного кода** программную систему, позволяющую проект, собираемый при помощи CMake, модифицировать таким образом, чтобы **исполнитель C++** при запуске целевой функции мог собирать информацию о поведении функции.

Назовём **исполнителем Java** программу, позволяющую, при наличии **исполнителя C++**, передавать последнему входные данные для целевой функции, полученные от универсальной фаззинг платформы. Затем, считывать результат выполнения функции, информацию о выполнении функции и передавать эти данные универсальной фаззинг платформе.

Назовём **протоколом обмена данными** программную систему описывающую взаимодействие **исполнителя Java** с **исполнителем C++** в рамках обмена входными данными и результатами выполнения функции.

2.2. Реализация предобработчика исходного кода

Задачу реализации предобработчика можно разбить на две части:

1. Создание LLVM Pass [14], инструментирующего исходный код требуемым образом.

2. Создание системы для встраивания предобработчика в сборку CMake.

2.2.1 Реализация LLVM Pass

Для реализации LLVM Pass используется интерфейс **llvm::FunctionPass**. В этом LLVM Pass перед каждой инструкцией вставляется сигнальная функция, которая позволяет собирать информацию об исполнении программы. Эту функцию затем необходимо поставить отдельным LLVM IR модулем.

Полученный LLVM Pass можно использовать так:

```
clang -emit-llvm -c test.cpp -o test.ll
opt -load libAstPass.so -ast test.ll > modified_test.ll
clang -emit-llvm -c logop.cpp -o logop.ll
llvm-link logop.ll modified_test.ll -o result.ll
lli result.ll
```

Например, при имеющемся файле main.cpp со следующим содержимым:

```
#include <iostream>

int sum(int a, int b) {
    return a + b;
}

int main() {
    std::cout << sum(5, 3);
    return 0;
}
```

после вызова команд

```
clang --emit-llvm -c main.cpp -o main.ll
opt -load libAstPass.so -ast main.ll > modified_main.ll
clang -emit-llvm -c logop.cpp -o logop.ll
llvm-link logop.ll modified_main.ll -o result.ll
lli result.ll
```

в выводе можно увидеть

```
Function called: main, instruction: 0;
Function called: main, instruction: 1;
Function called: main, instruction: 2;
Function called: _Z3sumii, instruction: 0;
Function called: _Z3sumii, instruction: 1;
Function called: _Z3sumii, instruction: 2;
Function called: _Z3sumii, instruction: 3;
Function called: _Z3sumii, instruction: 4;
Function called: _Z3sumii, instruction: 5;
Function called: _Z3sumii, instruction: 6;
Function called: _Z3sumii, instruction: 7;
Function called: main, instruction: 3;
7
Function called: main, instruction: 4;
```

что соответствует пути исполнения программы.

Реализованный LLVM Pass находится в пакете pass [22], в файле AstPass.cpp.

2.2.2 Встраивание LLVM Pass в систему сборки CMake

Описанную выше процедуру применения LLVM Pass необходимо проводить с каждым из LLVM IR файлов, полученных в результате сборки. Это не всегда возможно вручную, особенно для крупных проектов.

Если проект собирается при помощи CMake и поддерживает флаг

CMAKE_EXPORT_COMPILE_COMMANDS [11]

то после сборки проекта становится доступен файл compile_commands.json, в котором, в том числе можно обнаружить все команды компилятора.

Необходимо для каждой команды компилятора clang добавить флаг -emit-llvm.

Затем, добавить команды, которые к каждому из полученных LLVM IR файлов применят LLVM Pass.

На подготовленном мной примере проекта из нескольких файлов на C++ можно увидеть пример работы. Проект состоит из 2 файлов: main.cpp, mult.cpp. При сборке проекта с помощью CMake с включенным флагом

CMAKE_EXPORT_COMPILE_COMMANDS

будет образован файл compile_commands.json со следующим содержимым (пути, для краткости, сокращены):

```
[
    {
        "directory": "*/test_project",
        "command": "/usr/bin/c++ -o */main.cpp.o -c */main.cpp",
        "file": "*/main.cpp",
        "output": "*/main.cpp.o"
    },
    {
        "directory": "*/test_project",
        "command": "/usr/bin/c++ -o */mult.cpp.o -c */mult.cpp",
        "file": "*/mult.cpp",
        "output": "*/mult.cpp.o"
    }
]
```

что соответствует компиляции двух файлов на C++. После обработки этого файла с помощью системы встраивания LLVM Pass в систему сборки CMake, итоговый набор команд будет иметь вид:

```
[
    {
        "directory": "*/test_project",
        "command": "
            /usr/bin/c++ -o */main.cpp.ll
            -c */main.cpp -emit-llvm
        ",
```

```
    "file": "*/main.cpp",
    "output": "*/main.cpp.ll"
},
{
    "directory": "*/test_project",
    "command": "
        /usr/bin/opt -load */libAstPass.so -ast
        */main.cpp.ll > */main.cpp.ll.modified.ll
    ",
    "file": "*/main.cpp.ll",
    "output": "*/main.cpp.ll.modified.ll"
},
{
    "directory": "*/test_project",
    "command": "
        /usr/bin/llvm-link  */logop.ll */main.cpp.ll.modified.ll
        -o */main.cpp.ll.linked.ll
    ",
    "file": "*/main.cpp.ll.modified.ll",
    "output": "*/main.cpp.ll.linked.ll"
},
{
    "directory": "*/test_project",
    "command": "
        /usr/bin/c++ -o */mult.cpp.ll
        -c */mult.cpp -emit-llvm
    ",
    "file": "*/test_project/mult.cpp",
    "output": "*/mult.cpp.ll"
},
{
    "directory": "*/test_project",
    "command": "
```

```
            /usr/bin/opt -load */libAstPass.so -ast
            */mult.cpp.ll > */mult.cpp.ll.modified.ll
        ",
        "file": "*/mult.cpp.ll",
        "output": "*/mult.cpp.ll.modified.ll"
    },
    {
        "directory": "*/test_project",
        "command": "
            /usr/bin/llvm-link  */logop.ll */mult.cpp.ll.modified.ll
            -o */mult.cpp.ll.linked.ll
        ",
        "file": "*/mult.cpp.ll.modified.ll",
        "output": "*/mult.cpp.ll.linked.ll"
    }
]
```

В приведённом выше примере, для каждого компилируемого файла добавляется получение из него LLVM IR. Затем, LLVM IR файлы проходят через LLVM Pass. Затем, к полученным LLVM IR файлам с помощью llvm-link [20] добавляются вспомогательные функции, которые не находились в исходном коде, но были добавлены при применении LLVM Pass.

Система встраивания LLVM Pass в систему сборки CMake находится в пакете cmake_tools [21] в файлах modifier.py, runner.py.

2.3. Реализация протокола обмена данными

Для обмена данными между исполнителями было решено использовать сокеты [12].

Перед началом работы исполнители устанавливают друг с другом соединение и затем не разрывают его до окончания фаззинга. Но, если соединение прервалось, то как исполнитель C++, так и исполнитель Java поддерживают возможность переподключения.

Таким образом, исполнитель C++ выступает в роли сервера, а исполни-

тель Java — в роли клиента, который подключается к серверу. Из этого так же следует, что оба исполнителя должны использовать один и тот же порт для обмена данными друг с другом.

К тому же, в такой реализации, исполнители могут находиться на разных машинах, что положительно сказывается на применимости системы в разных конфигурациях и операционных системах.

После установки соединения исполнитель Java по соединению передаёт тестовые данные для запуска функции, находящейся под фаззингом. После окончания исполнения функции, исполнитель C++ в ответ отправляет результат выполнения функции, а так же путь исполнения программы.

2.4. Реализация исполнителя C++

Исполнитель C++ реализован при помощи LLVM API и его работу можно разделить на несколько этапов:

1. Обработка LLVM IR файлов поданных на вход [15]. Необходимо их считать, загрузить память, преобразовать в единый LLVM IR модуль.

2. Создание ядра llvm::ExecutionEngine [13] запуска функции из LLVM IR модуля.

3. Поиск функции в ядре, подготовка к запуску.

4. Подключение к интерфейсу обмена данными с исполнителем Java.

5. Запуск в цикле целевой функции из этапа 3, обмен данными с исполнителем Java.

Непосредственно во время работы функции так же необходимо собирать данные о пути исполнения. Это достигается за счёт предварительного прохода LLVM Pass, где перед каждой инструкцией функции вставляется вызов логирующей функции, в которую передаются имя текущей функции и номер текущей инструкции. Логирующая функция в свою очередь делает вывод.

Затем, непосредственно перед запуском с помощью подмены буффера канала вывода логирующая функция настраивается так, что все её вызовы

окажутся отражены в структуре под контролем исполнителя C++. Затем эта структура передается исполнителю Java, который интерпретирует её как путь исполнения.

Запустить исполнитель C++ можно следующим образом:

```
./runner _Z3sumii 6004 int32 result.ll llvm_files.txt
```

Реализованный исполнитель C++ находится в пакете runner [23], в файле runner.cpp.

2.5. Реализация исполнителя Java

Исполнитель Java реализован на языке Kotlin и имеет зависимость на универсальную фаззинг платформу. В рамках запуска исполнителя, он выполняет следующие шаги:

1. Установка связи с исполнителем C++.
2. Запуск универсальной фаззинг платформы.
3. В цикле обмен данными с исполнителем C++, передача ему входных данных, получение от него информации об исполнении, передача информации универсальной фаззинг платформе.

Запустить Java исполнитель можно следующим образом:

```
java -jar fuzzing.jar
    --portCpp 6004
    --functionDescriptionPath description.txt
```

Реализованный исполнитель Java находится в пакете fuzzing-java [24].

2.6. Пример использования

В этой секции будет изложен пример использования фаззера на проекте, который собирается при помощи CMake.

1. Запустить сборку при помощи CMake с установленным флагом

CMAKE_EXPORT_COMPILE_COMMANDS

2. Полученный compile_commands.json модифицировать, получить LLVM IR файлы на выходе, прошедшие LLVM Pass.

3. Запустить исполнитель C++, передав ему список LLVM IR файлов, а также порт и при необходимости IP, через который будет происходить обмен данными с исполнителем Java.

4. Запустить исполнитель Java, передав ему те же порт и IP.

5. Анализировать вывод исполнителя Java, в котором будут содержаться тестовые случаи, предложенные универсальной фаззинг платформой.

Приведу пример применения описанных шагов к тестовому проекту: двум файлам на C++ (main.cpp, mult.cpp). В файле main.cpp находится функция

```
int sum(int a, int b) {
    return a + b;
}
```

которая и подвергается фаззингу. Исполнители Java, C++ собраны при помощи CMake. Файл со вспомогательными функциями, использующимися в LLVM Pass, собран. Подготовлен конфигурационный файл description.txt, описывающий функцию для универсальной фаззинг платформы:

```
int32,int32,int32
```

Пути к файлам для краткости опущены. Изначально пользователь находится в папке с проектом. Тогда, фаззинг запускается следующими командами:

```
cmake ./
make
python3.10 */modifier.py compile_commands.json llvm_files_list.txt
    new_commands.json */libAstPass.so */logop.ll
```

```
python3.10 */runner.py new_commands.json
*/runner _Z3sumii 6004 int32 llvm_files_list.txt
java -jar */fuzzing.jar --portCpp 6004
    --functionDescriptionPath description.txt
```

Фаззинг быстро завершается и исполнитель Java выводит тестовые данные, соответствующие единственному пути исполнения:

```
case 1:
    arguments:
        int32: 0
        int32: 0
    result:
        Int32(value=0)
```

3. Тестирование

В этой главе описана инфраструктура для тестирования разработанного фаззера с аналогами. Так как разработанный фаззер является применением универсальной фаззинг платформы к языку C++, то нельзя ожидать, что он будет работать лучше, чем фаззеры, разработанные специально для работы с программами на C++. Однако, в сравнительно простых программах, фаззер должен показывать сравнимые результаты как по качеству, так и по времени работы.

Работа фаззера проверялась на виртуальной машине с процессором Intel Broadwell, 16 ГБ RAM, 200 ГБ SSD с операционной системой Ubuntu 20.04 64-bit.

3.1. Функция isPalindrome

Для теста была выбрана функция isPalindrome из репозитория со многими алгоритмами и функциями на языке C [25]. Интересующая нас функция находится по пути C/math/palindrome.c. Проект собирается при помощи CMake. Исходный код функции выглядит следующим образом:

```
bool isPalindrome(int number)
{
    int reversedNumber = 0;
    int originalNumber = number;
    while (number != 0)
    {
        int remainder = number % 10;
        reversedNumber = reversedNumber * 10 + remainder;
        number /= 10;
    }
    return originalNumber == reversedNumber;
}
```

После запуска разработанного фаззера он находит все 11 возможных тестовых случаев, покрывающих все пути исполнения:

```
case 1:
    arguments:
        int32: 0
    result:
        Bool(value=true)
case 2:
    arguments:
        int32: 2
    result:
        Bool(value=true)
case 3:
    arguments:
        int32: 262144
    result:
        Bool(value=false)
case 4:
    arguments:
        int32: -2147221504
    result:
        Bool(value=false)
case 5:
    arguments:
        int32: 4456448
    result:
        Bool(value=false)
case 6:
    arguments:
        int32: 545259522
    result:
        Bool(value=false)

case 7:
    arguments:
        int32: 514
    result:
        Bool(value=false)
case 8:
    arguments:
        int32: 10
    result:
        Bool(value=false)
case 9:
    arguments:
        int32: 4738
    result:
        Bool(value=false)
case 10:
    arguments:
        int32: 16394
    result:
        Bool(value=false)
case 11:
    arguments:
        int32: 16777219
    result:
        Bool(value=false)
```

за время

```
real    0m4.913s
user    0m2.334s
sys     0m0.295s
```

При этом фаззер Radamsa выполняет ту же работу за время и находит полное покрытие

```
real    0m3.216s
user    0m3.187s
sys     0m0.032s
```

Также было проведено тестирование на других функциях из пакета и получены схожие результаты.

3.2. Выводы

По результатам тестирования видно, что реализованный фаззер обладает схожими характеристиками с популярными фаззерами для C/C++. Разработанный фаззер можно удобно применять на проекте, в случае если проект собирается при помощи CMake.

3.3. Дальнейшей развитие

У проекта есть ряд направлений для развития:

1. Необходимо и дальше улучшать систему интеграции в большие проекты.

   Существуют проекты, которые собирается не при помощи CMake. Так же существуют проекты, которые собираются CMake, но не пригодны для получения LLVM IR файлов. Таким образом, ряд допущений, принятых при разработке фаззера, не выполняются всегда.

   К тому же, необходимо сокращать сложность использования полученного фаззера, в идеале — до одной команды в консоли, как это часто бывает у аналогов.

2. Необходимо расширять поддерживаемые типы пользовательскими структурами. Это в некоторой мере поддержано в универсальной фаззинг платформе, но представляет проблему при переносе на C++.

3. Возможен рефакторинг и добавление полученного фаззера в репозиторий UnitTestBot [16], где уже находятся фаззеры для Python, Java, JavaScript, основанные на универсальной фаззинг платформе.

Заключение

В данной работе были выполнены следующие задачи:

1. Проведён обзор уже существующих решений для фаззинга программ на C/C++. Были изучены аналоги, позволяющие проводить фаззинг программ на нескольких языках программирования.

2. Разработан исполнитель C++ — программный аппарат для запуска программ, написанных на C/C++. Реализована возможность собирать данные об исполнении программы.

3. Разработан протокол для обмена данными между исполнителем Java и исполнителем C++.

4. Разработан исполнитель Java — программный аппарат для запуска универсальной фаззинг платформы и передачи ей данных об исполнении программы.

5. Разработанный фаззер был сравнён с аналогами.

Список литературы

[1] LibFuzzer. URL: https://llvm.org/docs/LibFuzzer.html (дата обр. 06.05.2023).

[2] Radamsa. URL: https://gitlab.com/akihe/radamsa (дата обр. 06.05.2023).

[3] Jsfuzz. URL: https://github.com/fuzzitdev/jsfuzz (дата обр. 06.05.2023).

[4] Универсальная фаззинг платформа. URL: https://github.com/UnitTestBot/UTBotJava/tree/main/utbot-fuzzing/src/main/kotlin/org/utbot/fuzzing (дата обр. 02.02.2023)

[5] Fuzzing. URL: https://en.wikipedia.org/wiki/Fuzzing (дата обр. 08.05.2023)

[6] Peach Fuzzer. URL: https://peachtech.gitlab.io/peach-fuzzer-community/ (дата обр. 09.12.2022)

[7] LLVM 14. URL: https://github.com/llvm/llvm-project/tree/release/14.x (дата обр. 02.02.2023)

[8] CMake 3.24. URL: https://cmake.org/cmake/help/v3.24/release/3.24.html (дата обр. 02.02.2023)

[9] JDK 19. URL: https://openjdk.org/projects/jdk/19/ (дата обр. 02.02.2023)

[10] Gradle releases. URL: https://gradle.org/releases/ (дата обр. 02.02.2023)

[11] CMAKE_EXPORT_COMPILE_COMMANDS variable. URL: https://cmake.org/cmake/help/latest/variable/CMAKE_EXPORT_COMPILE_COMMANDS.html (дата обр. 02.02.2023)

[12] Socket. URL: https://en.wikipedia.org/wiki/Network_socket (дата обр. 02.02.2023)

[13] LLVM Execution Engine. URL: https://llvm.org/doxygen/classllvm_1_1ExecutionEngine.html (дата обр. 04.02.2023)

[14] LLVM Pass. URL: https://llvm.org/docs/WritingAnLLVMPass.html (дата обр. 15.01.2023)

[15] LLVM IR file parsing. URL: https://llvm.org/doxygen/IRReader_8cpp_source.html (дата обр. 04.02.2023)

[16] UnitTestBot. URL: https://github.com/UnitTestBot (дата обр. 15.11.2022)

[17] Radamsa. URL: https://github.com/Hwangtaewon/radamsa (дата обр. 10.04.2023)

[18] Honggfuzz. URL: https://honggfuzz.dev/ (дата обр. 05.03.2023)

[19] PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems. URL: https://www.usenix.net/conference/usenixsecurity23/presentation/liwen (дата обр. 02.02.2023)

[20] LLVM Link. URL: https://llvm.org/docs/CommandGuide/llvm-link.html (дата обр. 03.03.2023)

[21] CMake tools. URL: https://github.com/mbnexttime/fuzzing/tree/main/cmake_tools (дата обр. 27.05.2023)

[22] LLVM AST pass. URL: https://github.com/mbnexttime/fuzzing/tree/main/pass (дата обр. 27.05.2023)

[23] C++ Runner. URL: https://github.com/mbnexttime/fuzzing/tree/main/runner (дата обр. 28.05.2023)

[24] Java Runner. URL: https://github.com/mbnexttime/fuzzing-java (дата обр. 28.05.2023)

[25] Algorithms on C language. URL: https://github.com/TheAlgorithms/C (дата обр. 26.05.2023)