Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

Шмидт Светлана Андреевна

Выпускная квалификационная работа

Разработка программного средства для автоматической изоляции тестируемого LLVM кода

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

Научный руководитель:
доцент, кафедра системного программирования,
СПбГУ, к.ф.-м.н., Д. А. Мордвинов

Рецензент:
доцент, кафедра цифровой экономики и информационных технологий, ЮУрГУ, к.т.н., И. А. Прохорова

Санкт-Петербург
2023 г.

Содержание

Введение

Важнейшей характеристикой любого программного продукта является его надёжность. Именно поэтому тестирование — неотъемлемая часть разработки программного обеспечения. Автоматизация процесса тестирования позволяет значительно снизить стоимость разработки компьютерных программ [1]. Одним из распространённых методов автоматической верификации программного обеспечения является символьное исполнение [2].

Идея классического символьного исполнения состоит в выполнении программного кода таким образом, что вместо *конкретных* данных на вход программа получает *символы*, представляющие собой произвольные значения. Такой подход позволяет анализировать даже очень редкие сценарии поведения, чего трудно добиться, используя случайное тестирование.

Одной из проблем, с которой сталкивается символьное исполнение, является анализ взаимодействия программы с внешним окружением [3]. Работа с системными вызовами, сторонними библиотеками, файлами, сетью и другими внешними объектами часто приводит к побочным эффектам, которые крайне трудно исследовать в тех случаях, когда параметры взаимодействия являются символьными.

В основном современные символьные виртуальные машины решают эту задачу, создавая собственные модели для часто используемых стандартных функций и явно вызывая остальные внешние функции с конкретными аргументами. Несмотря на то, что такой подход демонстрирует хорошие результаты при тестировании небольших проектов, он имеет ряд недостатков при работе с промышленным кодом.

Во-первых, таким образом не получится анализировать пользовательские внешние функции, которые невозможно исполнить явно в момент тестирования. Такие ситуации часто возникают в процессе промышленной разработки в силу того, что над кодом программного продукта могут параллельно работать несколько команд. Так, исследование компании Microsoft [4], направленное на изучение инструментов и рабочих привычек сотрудников компании, показало, что 54% опрошенных используют модульное тестирование с целью упрощения взаимодействия между командами.

Во-вторых, конкретизация символьных аргументов перед вызовом внешней функции может приводить к потере некоторых ветвей исполнения, а следовательно, и к пропуску критических уязвимостей. Возможность анализа каждой из достижимых инструкций является важным свойством символьного исполнения, на которое опираются многие приложения [5].

В-третьих, конкретное исполнение внешних вызовов в ходе анализа программы затрудняет воспроизводимость сгенерированных тестов, так как они могут вести себя по-разному в зависимости от внешних условий. Например, если в ходе выполнения теста вызывается функция, возвращающая текущее время, то результат может зависеть от времени на машине, где проводится тестирование. Такие нестабильные тесты негативно влияют на эффективность и надёжность процесса верификации [6].

Таким образом, для внедрения техники символьного исполнения в процессы тестирования программного обеспечения в IT-компаниях необходимо преодолеть упомянутые выше трудности. Данная работа решает эту задачу путём усовершенствования символьной виртуальной машины KLEE [7], которая занимает высокие позиции в соревнованиях по автоматическому тестированию [8], [9], [10] и лежит в основе многих современных инструментов анализа кода, таких как TracerX [11], Symbiotic 8 [12], KLOVER [13], Cloud9 [14], LibKluzzer [15] и многих других.

Работа выполнена при поддержке российского исследовательского института Huawei. Кодовая база компании Huawei насчитывает гигабайты исходного кода на языках C и C++. Для упрощения тестирования такие большие проекты разделяют на модули, тестируемые в отдельности. Отсюда возникает большое количество внешних вызовов. Поэтому данная работа имеет большое значение для процессов автоматизации тестирования компании.

Постановка задачи

Целью данной работы является разработка программного средства, обеспечивающего автоматическую изоляцию тестируемого LLVM кода. Разработанное решение должно удовлетворять следующим требованиям.

- Программный продукт должен позволять автоматически генерировать тесты для пользовательского кода, содержащего внешние вызовы.
- Пользователю программного продукта должны быть доступны различные стратегии изоляции.
- Программный продукт должен предоставлять возможность воспроизводить сгенерированные им тесты.

Исходя из цели работы, были поставлены следующие задачи.

- Провести обзор существующих подходов к автоматической изоляции тестируемого кода.
- Разработать и реализовать стратегии автоматической изоляции на базе существующей символьной виртуальной машины.
- Разработать и реализовать механизм воспроизведения тестов, взаимодействующих с внешним окружением.
- Провести тестирование предложенного решения на промышленном проекте.

1. Обзор предметной области

В этой главе представлены введение в предметную область и обзор существующих решений. На основе проведённого обзора были выбраны основные инструменты для реализации автоматической изоляции кода.

1.1. Символьное исполнение

Символьное исполнение — метод анализа программ, основная идея которого заключается в выполнении кода таким образом, что вместо *конкретных* аргументов программа получает на вход *символы*, представляющие собой произвольные значения.

На каждом шаге алгоритм символьного исполнения поддерживает множество *состояний пути*, каждое из которых содержит следующие данные:

- `instr` — следующая инструкция на соответствующем пути исполнения;
- σ — символьная память, сопоставляющая ячейке памяти её символьное или конкретное значение;
- π — условие пути, представляющее собой формулу логики первого порядка, которая выражает ограничения на символьные переменные из σ для текущего пути исполнения.

Рассмотрим алгоритм 1, демонстрирующий работу символьного исполнения. Изначально в множество `S` *состояний пути* добавляется исходное состояние. Далее на каждом шаге выбирается следующее состояние для исследования $\texttt{state} = (\texttt{instr}, \sigma, \pi)$. После чего некоторым образом, зависящим типа инструкции `instr`, из состояния `state` получается множество состояний `newStates`, для каждого из которых проверяется выполнимость условия пути. В случае, если условие пути выполнимо, состояние добавляется в `S`. Если множество `newStates` пустое, то исполнение завершилось и можно создавать новый тест.

Для генерации нового теста используется SMT-решатель [16], на вход которому подаётся накопленное условие пути π. На выходе SMT-решатель выдаёт набор входных данных, удовлетворяющий формуле π и, следовательно,

Алгоритм 1 Алгоритм символьного исполнения

```
1:  S = {start}
2:  while S ≠ ∅ do
3:      state ← S
4:      S = S \ state
5:      newStates = executeInstuction(state, state.instr)
6:      if newStates = ∅ then
7:          processTestCase(state)
8:      else
9:          for all state' ∈ S do
10:             if SAT(state'.π) then
11:                 S ← state'
12:             end if
13:         end for
14:     end if
15: end while
```

реализующий соответственный путь исполнения. Для определения выполнимости формулы π так же используется SMT-решатель.

```
1  void foo(int a, int b) {
2      int x = 1, y = 0;
3      if (a != 0) {
4          y += 4;
5          if (b == 0) {
6              x = (a + b) * 2;
7          }
8      }
9      assert(x != y);
10 }
```

Листинг 1: Функция `foo` для иллюстрации работы алгоритма символьного исполнения.

Для лучшего понимания алгоритма символьного исполнения рассмотрим его работу на примере (листинг 1). На рисунке 1 показано дерево состояний символьного исполнения, которое было получено в ходе анализа функции `foo`.

На вход функция `foo` получает два аргумента `a` и `b`, которым соответствуют символы α_a и α_b. Когда в ходе символьного исполнения встречается оператор присваивания, изменяется символьная память. Если в ходе символь-

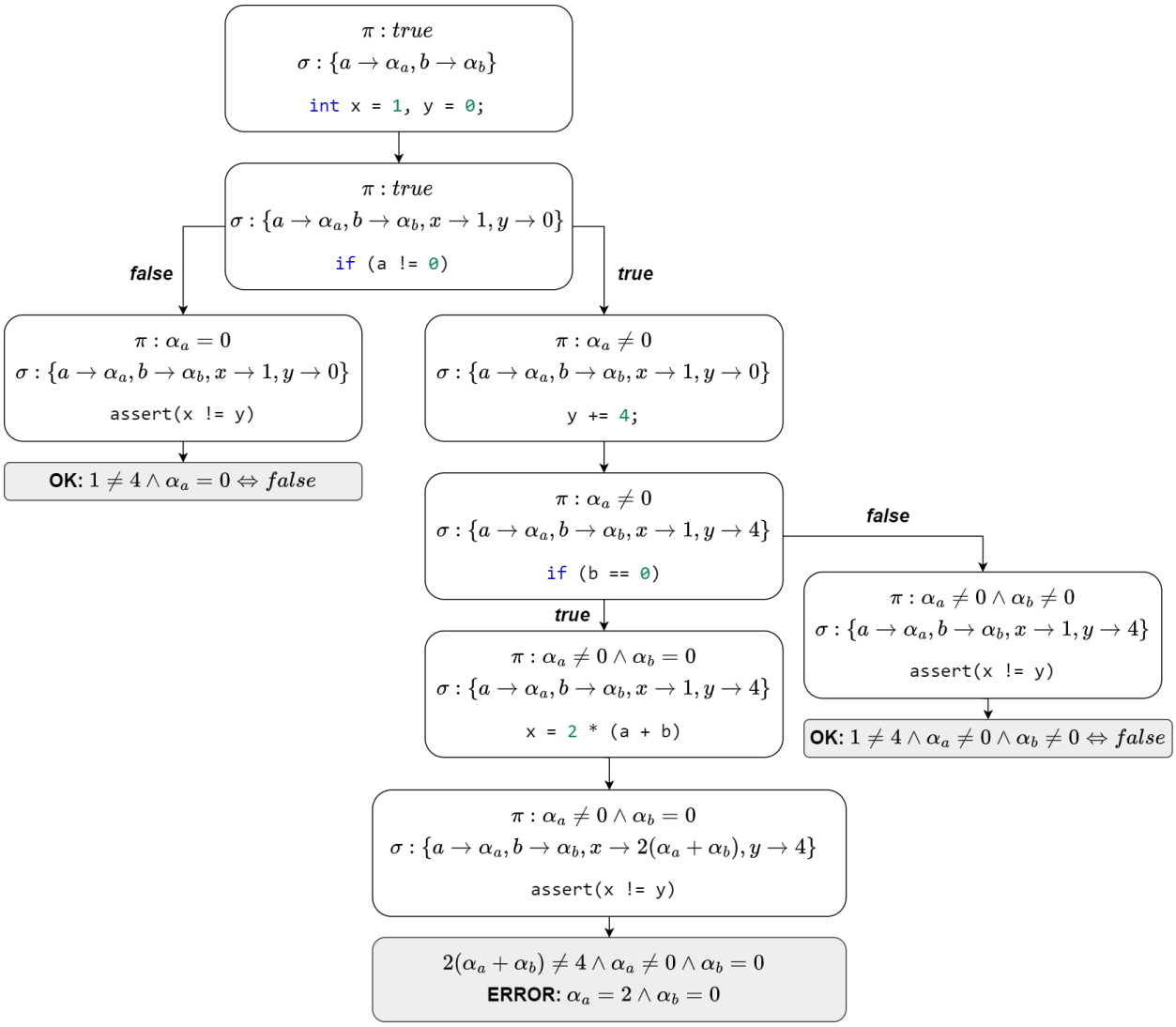

Рис. 1: Дерево состояний символьного исполнения функции `foo`, листинг 1 (на основе [3]).

ного исполнения встретился оператор ветвления, создаются два новых состояния, условия пути которых получаются при помощи пополнения условия пути состояния предка условием попадания в соответствующую ветвь исполнения. Каждая из ветвей исполнения завершается вызовом SMT-решателя, который проверяет выполнимость конъюнкции формул $x \neq y$ и π с целью определить, какие входные аргументы приводят к падению функции `assert`.

Таким образом, из рисунка 1 видно, что в ходе анализа функции `foo` были исследованы три ветви исполнения. Входные аргументы $a = 2, b = 0$ приводят к завершению программы с ошибкой из-за падения функции `assert`.

1.2. Проблема анализа внешних вызовов с использованием символьного исполнения

Одной из сложностей, с которой сталкивается любая реализация алгоритма символьного исполнения, является анализ программного кода, содержащего вызовы внешних функций [3]. Здесь и далее под внешними будем понимать те функции, исходный код которых недоступен для анализа. Таким образом, суть проблемы состоит в невозможности символьно исполнить внешний вызов в силу недоступности исходного кода для анализа. Одним из широко распространённых способов решения данной проблемы является явный вызов внешней функции с использованием конкретных аргументов. Однако такое решение также имеет ряд недостатков.

1. **Невозможность конкретно исполнить неприлинкованные функции.** Внешние функции делятся на две категории: прилинкованные и неприлинкованные. Прилинкованными будем называть те функции, которые можно исполнить явно. К ним относятся, например, системные вызовы. В свою очередь, неприлинкованными будем называть те функции, что невозможно вызвать явно, так как во время анализа недоступен не только их код на некотором высокоуровневом языке программирования, но и их представление в виде машинного кода. К неприлинкованным функциям можно отнести функции из сторонних пользовательских библиотек, которые не были использованы в процессе сборки анализируемого модуля. Таким образом, предложенное выше решение никак не обрабатывает неприлинкованные внешние функции, так как их нельзя исполнить конкретно.

2. **Воспроизводимость тестов**. Результаты внешних вызовов могут зависеть от состояния системы. Следовательно, поведение тестов, в ходе генерации которых внешние функции исполнялись конкретно, может быть недетерминированным. Подобные тесты являются нестабильными и негативно влияют на эффективность и надёжность процесса тестирования в целом [6].

3. **Пропуск некоторых ветвей исполнения**. Из-за конкретизации символьных аргументов перед вызовом внешней функции и конкретного вызова самой функции, некоторые из ветвей исполнения, а следовательно, и некоторые уязвимости могут быть пропущены.

4. **Побочные эффекты**. Внешние вызовы могут приводить к побочным эффектам, изменяющим состояние внешнего окружения. Ввиду того, что в ходе символьного исполнения различные пути исполнения могут анализироваться параллельно, такие побочные эффекты могут приводить к некорректному состоянию системы.

Рассмотрим пример, иллюстрирующий вышеперечисленные трудности (листинг 2). Функция `maintain_temp` используется для поддержания комфортной температуры на уровне `comfort_temp` в помещении. Внешние функции `read_temp_sensor`, `radiator_on`, `air_conditioner_on` взаимодействуют с внешним окружением, снимая показания с датчика температуры и включая радиатор и кондиционер соответственно.

```
1  extern int read_temp_sensor();
2  extern void radiator_on(int temp);
3  extern void air_conditioner_on(int temp);
4
5  void maintain_temp(int comfort_temp) {
6      int t = read_temp_sensor();
7      if (t < -273) {
8          assert(0 && "sensor is broken");
9      }
10     if (comfort_temp > t) {
11         radiator_on(comfort_temp);
12     } else {
13         air_conditioner_on(comfort_temp);
14     }
15 }
```

Листинг 2: Пример, иллюстрирующий проблемы анализа внешних вызовов в ходе символьного исполнения.

В случае, если данный код собран без использования библиотеки, реализующей внешние функции, данные функции будут неприлинкованными и

исполнить их конкретно будет невозможно.

Рассмотрим поведение алгоритма символьного исполнения на функции `maintain_temp` в случае, если явно вызвать внешние функции возможно. На первом шаге будет конкретно исполнен вызов функции `read_temp_sensor`, значение переменной `t` станет некоторым конкретным числом (например, 20).

В силу того, что `t` — конкретное число, ветвь исполнения, приводящая к падению функции `assert` может быть не исследована. Это иллюстрирует проблему пропуска некоторых ветвей исполнения в ходе анализа.

Далее при выполнении второго условного оператора будет создано два символьных состояния с соответствующими условиями `comfort_temp` $> t$, `comfort_temp` $\leq t$. В ходе дальнейшего анализа этих состояний будут конкретно вызваны обе функции `radiator_on` и `air_conditioner_on`. Это приведёт к одновременному включению радиатора и кондиционера, что является некорректным состоянием системы. Это иллюстрирует проблему побочных эффектов.

Более того, в силу того, что выполнить внешние функции `radiator_on` и `air_conditioner_on` с символьными аргументами невозможно, перед вызовом придётся конкретизировать переменную `comfort_temp`.

Ещё одна проблема заключается в том, что значение переменной `t` зависит от состояния среды, а именно от температуры. Следовательно, при воспроизведении такого теста значение `t` может получаться разным, а поведение теста недетерминированным. Это иллюстрирует проблему воспроизводимости тестов.

1.3. Обзор существующих решений

В ходе обзора существующих решений проблемы анализа внешних вызовов при использовании символьного исполнения было выделено несколько стратегий.

- **Запрет на использование внешних вызовов в исследуемом программном коде**. Такой подход реализует инструмент CUTE [17]. В действительности этот подход не решает поставленную проблему, а лишь ограничивает область применения инструментов, реализующих

его. Ввиду того, что современное программное обеспечение часто содержит взаимодействие с внешними объектами, такие инструменты неприменимы к промышленным проектам.

- **Символьное моделирование**. Большинство символьных виртуальных машин (DART [18], AEG [19], CLOUD9 [14], KLEE [7]) прибегают к созданию символьных моделей. Символьная модель представляет собой реализацию внешней функции с использованием символьных переменных. Символьные модели могут быть реализованы двумя способами.

    1. *Сгенерированы автоматически*. В таком случае исходя из тех данных, что известны о внешней функции (её имени, типе возвращаемого значения и типах её аргументов), автоматически генерируется некоторая реализация. Такой подход использует инструмент DART. DART предполагает, что внешняя функция может вернуть любое значение подходящее по типу, и автоматически создаёт простую реализацию, возвращающую символьное значение заданного типа. Такой метод применим к любым внешним функциям, однако полученные реализации часто достаточно примитивны из-за ограниченности информации, известной о внешней функции.
    2. *Написаны вручную*. Этот метод применяется для создания символьных реализаций часто используемых функций стандартной библиотеки. Так, например, символьная виртуальная машина KLEE реализует большинство функций, входящих в стандартную библиотеку языка программирования C, а также свою собственную символьную файловую систему. AEG также моделирует файловую систему, работу с переменными среды и сокетами. Основное достоинство данного метода состоит в возможности создавать выразительные символьные модели для часто используемых внешних функций. Однако те внешние функции, которые не получили символьных реализаций, также должны быть обработаны. В этом случае KLEE, AEG и CLOUD9 прибегают к конкретному исполнению внешнего вызова, недостатки которого были рассмотрены

в разделе 1.2.

- **Виртуализация**. Этот подход был предложен авторами инструмента S^2E в противовес символьному моделированию. Авторы S^2E отмечают, что создание и поддержка выразительных символьных моделей — трудоёмкий процесс. В случае изменения стандарта моделируемой функции поведение символьной модели может становиться неактуальным. В связи с этим необходимо явно взаимодействовать с внешним окружением, но при этом не допускать некорректных состояний системы из-за возникновения внешних эффектов, считают создатели S^2E. Для достижения этой цели они прибегают к виртуализации, чтобы предотвратить распространение побочных эффектов по независимым путям исполнения при взаимодействии с реальной средой. QEMU [20] используется для эмуляции всего программного стека. Для символьного исполнения набора инструкций используется символьная виртуальная машина KLEE. Такой подход, хотя и решает проблему побочных эффектов, не создавая при этом символьных моделей, не решает проблемы пропуска некоторых ветвей исполнения и воспроизводимости тестов.

1.4. Выводы

Исходя из рассмотренных способов борьбы с проблемой анализа внешних вызовов, было принято решение расширить функционал символьной виртуальной машины KLEE при помощи автоматической генерации символьных моделей для внешних функций. Это позволит решить проблемы побочных эффектов, пропуска некоторых ветвей исполнения и воспроизводимости тестов, а также сохранить выразительность KLEE в работе с теми функциями стандартной библиотеки, которые имеют написанные вручную символьные реализации. Ключевыми особенностями KLEE, повлиявшими на выбор этой символьной виртуальной машины, на основе которой будет реализован алгоритм обработки внешних вызовов, являются наличие открытого исходного кода, архитектура проекта, позволяющая легко вносить изменения, а также интеграция с платформой LLVM [21], которая значительно упрощает генерацию воспроизводимых тестов для нескольких языков программирования

одновременно.

Таблица 1 демонстрирует сравнение реализаций анализа внешних вызовов в различных инструментах символьного исполнения.

Таблица 1: Сравнение символьных виртуальных машин в контексте анализа внешних вызовов.

	CUTE	S^2E	DART	AEG	CLOUD9	KLEE	**Эта работа**
Как реализованы внешние вызовы?	Запрещены	Виртуализация	Символьное моделирование				
Реализует автоматическую генерацию символьных моделей?	—	—	✓	×	×	×	✓
Реализует символьные модели, написанные вручную?	—	—	×	✓	✓	✓	✓
Анализирует неприлинкованные внешние функции?	—	×	✓	×	×	×	✓
Решает проблему побочных эффектов?	—	✓	✓	×	×	×	✓
Решает проблему с обрезанием некоторых ветвей исполнения?	—	×	✓	×	×	×	✓
Решает проблему с воспроизводимостью тестов?	—	×	✓	×	×	×	✓

2. Архитектура символьной виртуальной машины KLEE

В данной главе рассмотрена архитектура основных компонентов символьной виртуальной машины KLEE [7], выбранной в качестве базового решения для внедрения алгоритма анализа внешних вызовов.

2.1. LLVM

Платформа LLVM представляет собой программную инфраструктуру, позволяющую создавать компиляторы и другие инструменты анализа и синтеза кода [21]. LLVM разработан на основе языка промежуточного представления LLVM IR. Использование LLVM IR позволяет сделать доступными преобразования и анализ произвольного программного обеспечения. Платформа LLVM поддерживает многие современные языки программирования, такие как C, C++, C#, Fortran, Haskell, Julia, Kotlin, Ruby, Rust, Scala, Swift и другие.

2.2. Основные компоненты KLEE

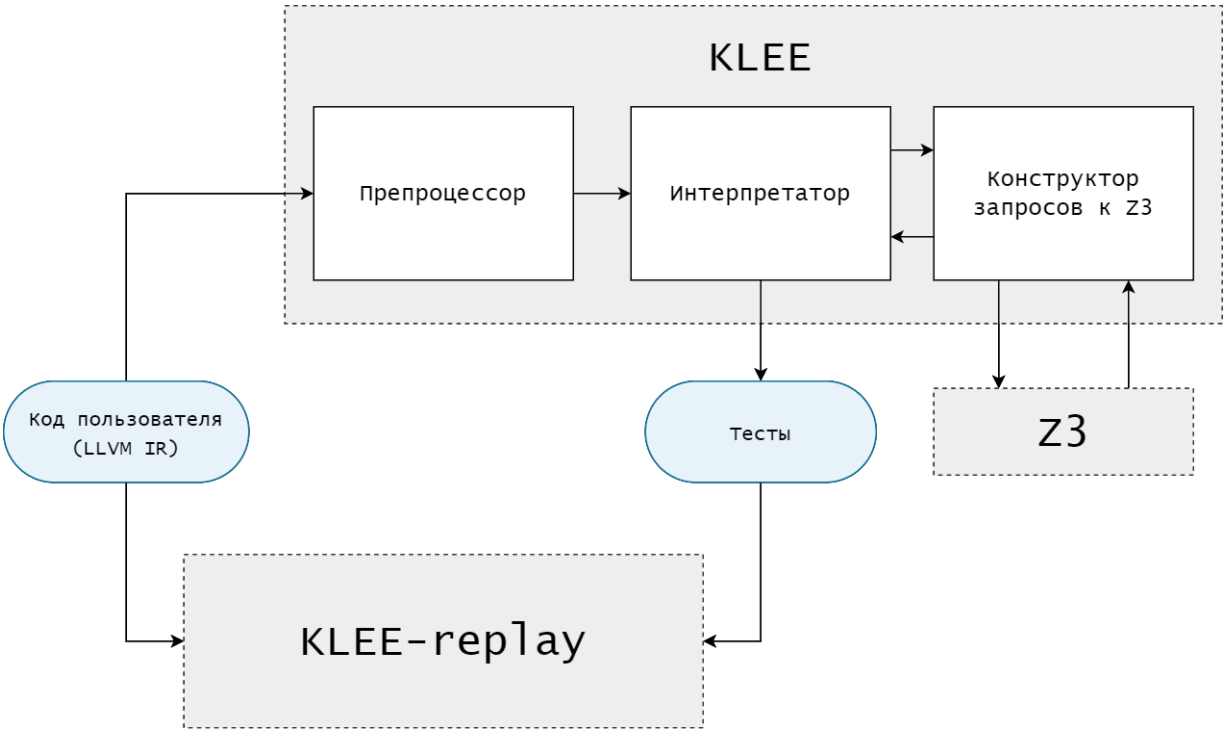

Рис. 2: Архитектура символьной виртуальной машины KLEE.

Упрощённая архитектура символьной виртуальной машины KLEE представлена на рисунке 2. Компонента KLEE представляет собой саму символьную виртуальную машину, которая на вход получает пользовательский код в формате LLVM IR, а на выход отдаёт сгенерированные тесты. В качестве SMT-решателя KLEE использует Z3 [22]. KLEE-replay — инструмент воспроизведения сгенерированных KLEE тестов.

Рассмотрим подробно работу KLEE на примере функции `bar` (листинг 3). В функции `bar` объявляется переменная `a`. Затем вызывается функция `klee_make_symbolic` — специальная функция KLEE, которая делает переменную символьной. После чего вызывается внешняя функция `abs`, и в зависимости от её результатов возвращается 0 или 1.

```
#include "stdlib.h"

int bar() {
    int a;
    klee_make_symbolic(&a, sizeof(a), "a");
    if (abs(a) == 0) {
        return 1;
    }
    return 0;
}
```

Листинг 3: Пример для иллюстрации работы KLEE.

Рассмотрим поведение каждого из компонентов, представленного на рисунке 2, в процессе анализа листинга 3.

2.2.1 Препроцессор

Первым этапом работы KLEE является препроцессинг пользовательского кода. KLEE получает на вход пользовательский код, представленный в формате LLVM IR. Листинг 4 иллюстрирует представление листинга 3 в формате LLVM IR. Как видно из данного примера, KLEE не получает на вход код внешней функции `abs`, а значит, не может символьно его исполнить.

На этапе препроцессинга KLEE определённым образом инструментирует код. В частности, на этом этапе происходит линковка символьных моделей

для библиотечных внешних функций. В нашем случае к коду, представленному на листинге 4 будет добавлено тело функции abs, реализованное в KLEE.

```
@.str = private unnamed_addr constant [2 x i8] c"a\00", align 1

define dso_local i32 @bar() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  call void @klee_make_symbolic(ptr %2, i64 4, ptr @.str)
  %3 = load i32, ptr %2, align 4
  %4 = call i32 @abs(i32 %3) #3
  %5 = icmp eq i32 %4, 0
  br i1 %5, label %6, label %7

6:                                                ; preds = %0
  store i32 1, ptr %1, align 4
  br label %8

7:                                                ; preds = %0
  store i32 0, ptr %1, align 4
  br label %8

8:                                                ; preds = %7, %6
  %9 = load i32, ptr %1, align 4
  ret i32 %9
}

declare dso_local void @klee_make_symbolic(ptr, i64, ptr) #1

declare dso_local i32 @abs(i32) #2
```

Листинг 4: Представление листинга 3 в формате LLVM IR.

2.2.2 Интерпретатор

После препроцессинга подготовленный LLVM модуль попадает в символьный интерпретатор. Интерпретатор в соответствии с алгоритмом, рассмотренным в главе 1, исполняет инструкции LLVM модуля. В случае, если в ходе анализа встретился вызов специальной функции klee_make_symbolic,

интерпретатор обрабатывает этот вызов, добавляя в символьную память соответствующую символьную переменную.

Результатом работы интерпретатора являются сгенерированные тесты, каждый из которых представлен в специальном формате `ktest` и содержит значения всех символьных переменных, встретившихся на конкретном пути, в том порядке, в котором они были созданы при помощи функции `klee_make_symbolic`.

2.2.3 Конструктор запросов к Z3

В соответствии с алгоритмом символьного исполнения, рассмотренным в главе 1, символьная виртуальная машина часто взаимодействует с SMT-решателем. В качестве SMT-решателя в KLEE используется Z3 [22].

Конструктор запросов к Z3 служит прослойкой между непосредственно SMT-решателем и символьным интерпретатором.

2.3. Алгоритм воспроизведения тестов

Чтобы воспроизвести тесты, сгенерированные символьной виртуальной машиной KLEE, необходимо собрать исходный LLVM модуль вместе с библиотекой `KLEE-Runtest` в исполняемый файл.

Библиотека `KLEE-Runtest` содержит в себе определения для специальных функций KLEE, в частности для функции `klee_make_symbolic`. Функция `klee_make_symbolic` реализована в библиотеке `KLEE-Runtest` таким образом, что каждый вызов данной функции приводит к чтению очередного значения из выбранного файла с тестом.

Таким образом, за счёт того, что значения символьных переменных записываются в файл с тестом в том порядке, в котором вызывалась функция `klee_make_symbolic` для их создания, при запуске теста данные значения будут считаны и подставлены в том же порядке. Следовательно, чтобы воспроизвести тест, необходимо лишь запустить собранный исполняемый файл с указанием пути до файла с тестом.

3. Реализация алгоритма анализа внешних вызовов

В данной главе подробно описано, как алгоритм анализа внешних вызовов был внедрён в символьную виртуальную машину KLEE с учётом её архитектуры, рассмотренной в главе 2. Реализация алгоритма доступна по ссылке: `https://github.com/UnitTestBot/klee/pull/80`.

3.1. Основная идея алгоритма

В ходе разработки алгоритма анализа внешних вызовов образующей частью была разработка механизма воспроизведения тестов, содержащих внешние вызовы. Чтобы воспроизвести тесты, необходимо собрать исполняемый файл, а, значит, нужно собрать пользовательский модуль вместе с определениями для внешних функций.

Таким образом, первым шагом к реализации алгоритма было добавить автоматическую генерацию символьных моделей в формате LLVM IR. Общий вид автоматически сгенерированной символьной модели представлен на листинге 5. Здесь `T` — тип возвращаемого значения моделируемой функции, `func` — имя моделируемой функции, `args` — её аргументы.

```
1 T func(args...) {
2     T return_value;
3     klee_make_mock(&return_value, sizeof(T), "func");
4     return return_value;
5 }
```

Листинг 5: Псевдокод автоматически сгенерированной символьной модели.

В рамках работы в KLEE была добавлена новая специальная функция `klee_make_mock`, работающая по аналогии с функцией `klee_make_symbolic`. В процессе воспроизведения тестов функция `klee_make_mock` производит чтение возвращаемых значений из того же файла с тестом, откуда читает данные функция `klee_make_symbolic`. В свою очередь, в ходе генерации тестов функция `klee_make_mock` обрабатывается некоторым особым образом в зависимости от выбранной стратегии изоляции. Подробно реализованные стратегии изоляции рассмотрены в главе 3.2.

3.2. Стратегии изоляции

В рамках работы были реализованы две стратегии изоляции, соответствующие недетерминированным и детерминированным символьным моделям.

3.2.1 Недетерминированные символьные модели

Под недетерминированным будем понимать такое поведение функции, что при каждом вызове она может вернуть произвольное значение, никак не зависящее от значений её аргументов.

В случае, если мы предполагаем, что поведение внешней функции недетерминировано, её символьная модель при каждом вызове может возвращать новую символьную переменную. Таким образом, если нами выбрана недетерминированная стратегия изоляции, при обработке каждого вызова функции `klee_make_mock` символьный интерпретатор будет добавлять в символьную память новую символьную переменную, соответствующую результату текущего вызова, что полностью совпадает с поведением функции `klee_make_symbolic`.

3.2.2 Детерминированные символьные модели

Будем говорить, что функция детерминированная, если она возвращает равные значения для каждого вызова с одинаковым набором аргументов.

Для реализации данной стратегии необходимо было не только изменить поведение интерпретатора, но и модифицировать запросы к Z3-решателю, так как именно SMT-решатель в символьном исполнении ответственен за решение любого рода ограничений, в том числе и тех, что описаны выше для детерминированной функции. Для достижения данной цели в конструкторе запросов к Z3 была поддержана теория неинтерпретируемых функций. В математической логике неинтерпретируемая функция — это та функция, о которой известны только её имя и арность. При этом неинтерпретируемая функция является функцией с математической точки зрения, то есть сопоставляет одному набору аргументов единственное значение.

Таким образом, если пользователь выбирает детерминированную стратегию изоляции, то при вызове функции `klee_make_mock` интерпретатор возвращает символ, равный применению соответствующей неинтерпретируемой функции к тем аргументам, с которыми вызывалась внешняя функция. Когда все ограничения пути будут собраны и переданы в Z3, Z3 вернёт значения всех символов с учётом того, что некоторые из них представляют собой применение неинтерпретируемой функции к набору аргументов.

3.2.3 Пример работы реализованных стратегий изоляции

Несмотря на то, что детерминированные символьные модели накладывают большее количество ограничений на моделируемую функцию, такая стратегия позволяет получить более разумные тесты, а также снизить число недостижимых уязвимостей, которые символьная виртуальная машина сочтёт достижимыми. Рассмотрим иллюстрирующий пример, представленный на листинге 6. Заметим, что если функция `foo` детерминированная, `assert` является недостижимым.

```
1  extern int foo(int x, int y);
2
3  int bar(int a, int b) {
4      if (a == b) {
5          if (foo(a + b, a) != foo(2 * b, b)) {
6              assert("unreachable point" && 0);
7          } else {
8              return 1;
9          }
10     }
11     return 0;
12 }
```

Листинг 6: Пример, иллюстрирующий разницу в поведении детерминированных и недетерминированных символьных моделей.

Рассмотрим пошагово поведение предложенного решения при символьном исполнении функции `bar`, если переменные `a` и `b` являются символьными.

На этапе препроцессинга вне зависимости от выбранной стратегии изоляции будет сгенерировано тело функции `foo` (листинг 7). После чего полу-

ченный LLVM модуль будет передан в интерпретатор.

```
define i32 @foo(i32 %0, i32 %1) {
entry:
  %2 = alloca i32
  %3 = bitcast i32* %2 to i8*
  call void @klee_make_mock(i8* %3, i64 4)
  %klee_var = load i32, i32* %2
  ret i32 %klee_var
}
```

Листинг 7: Автоматически сгенерированное тело функции `foo`.

В свою очередь, интерпретатор будет обрабатывать инструкции в привычном режиме, но когда он встретит вызов функции `klee_make_mock`, поведение будет отличаться в зависимости от выбранной стратегии.

В случае если выбрана стратегия генерации недетерминированных моделей, интерпретатор создаст новую символьную переменную на каждый вызов. В таком случае в итоге будет сгенерировано 3 теста, из которых 2 завершаются успешно, а третий приводит к падению функции `assert`.

В случае если выбрана стратегия генерации детерминированных моделей, интерпретатор, пытаясь исследовать ветвь исполнения, приводящую к падению функции `assert`, соберёт следующее условие пути: $a = b \wedge foo(a + b, a) \neq foo(2b, b)$. Данное условие будет передано в Z3-решатель, который ответит интерпретатору, что данное условие является невыполнимым. Таким образом, в результате будет сгенерировано 2 теста, оба из которых завершаются успешно.

Таблица 2 демонстрирует, какие тесты будут сгенерированы при выборе различных стратегий изоляции. Из таблицы видно, что оригинальный KLEE генерирует только тесты для путей, не содержащих внешние вызовы, однако даже такие тесты нельзя воспроизвести в силу того, что не удаётся собрать исполняемый файл из-за отсутствия определения функции `foo`. KLEE с детерминированными моделями генерирует на один тест меньше, чем KLEE с недетерминированными моделями, тем самым снижается покрытие кода тестами. В то же время KLEE с детерминированными моделями не сообщает об уязвимости, которая недостижима в случае, если `foo` — детерминированная

функция. Таким образом, в зависимости от своих целей пользователь может выбирать различные стратегии изоляции.

Таблица 2: Сравнение тестов к функции `bar`, сгенерированных KLEE.

	Тест 1	Тест 2	Тест 3
`a`	0	0	0
`b`	1	0	0
`foo(a + b, a)`	—	0	0
`foo(2 * b, b)`	—	0	1
Результат выполнения	return 0	return 1	assertion failed
Генерируется оригинальным KLEE?	✓	×	×
Генерируется изменённым KLEE с детерминированными моделями?	✓	✓	×
Генерируется изменённым KLEE с недетерминированными моделями?	✓	✓	✓
Воспроизводится оригинальным KLEE?	×	—	—
Воспроизводится изменённым KLEE?	✓	✓	✓

3.3. Реализация символьных моделей для аллокаторов

В рамках работы так же были реализованы символьные модели для стандартных аллокаторов, используемых в языках программирования C и C++: `malloc`, `calloc`, `realloc`. Известно, что в случае, если на устройстве недостаточно памяти для выделения, такие аллокаторы возвращают нулевой указатель, разыменование которого приведёт к аварийному завершению программы. Однако промоделировать такую ситуацию в тесте достаточно сложно. Эту задачу можно решить путём реализации символьных моделей для таких функций. Для этого необходимо реализовать сами символьные модели, а затем в переданном в KLEE LLVM-модуле подменить вызовы соответствующей функции на вызов функции обёртки, реализующей данную модель. В качестве примера рассмотрим функцию `baz` (листинг 8).

```
1 int baz() {
2     int *array = malloc(sizeof(int));
3     return array[0];
4 }
```

Листинг 8: Пример, иллюстрирующий работу символьных моделей для аллокаторов.

Листинг 9 иллюстрирует символьную модель, разработанную для функции `malloc`. Изначально заводится символьный флаг `retNull`, далее в зависимости от значения этого флага возвращается нулевой указатель, либо выделенная при помощи вызова функции `malloc` память.

```
void *__klee_wrapped_malloc(size_t size) {
  char retNull;
  klee_make_symbolic(&retNull, sizeof(char), "retNullMalloc");
  if (retNull) {
    return 0;
  }
  void *array = malloc(size);
  return array;
}
```

Листинг 9: Символьная модель для функции `malloc`.

Во время тестирования функции `bar` необходимо подменить вызов функции `malloc` на вызов функции `__klee_wrapped_malloc`. В таком случае будет сгенерировано два теста, один из которых приводит к аварийному завершению программы из-за разыменования нулевого указателя.

Таблица 3: Сравнение тестов к функции `baz`, сгенерированных KLEE.

	Тест 1	Тест 2
Что возвращает `malloc`?	Указатель на выделенный участок памяти	Нулевой указатель
Результат выполнения	return array[0]	Segmentation fault (null dereference)
Генерируется оригинальным KLEE?	✓	×
Генерируется изменённым KLEE?	✓	✓
Воспроизводится оригинальным KLEE?	✓	—
Воспроизводится изменённым KLEE?	✓	✓

3.4. Архитектура итогового решения

Рисунок 3 иллюстрирует как изменилась архитектура KLEE после реализации автоматической изоляции. Красным цветом указано, что было реализовано внутри каждого из компонентов в рамках работы над проектом. Как видно из диаграммы, изменения, связанные с внедрением автоматической

изоляции в KLEE, коснулись всех основных этапов алгоритма символьного исполнения.

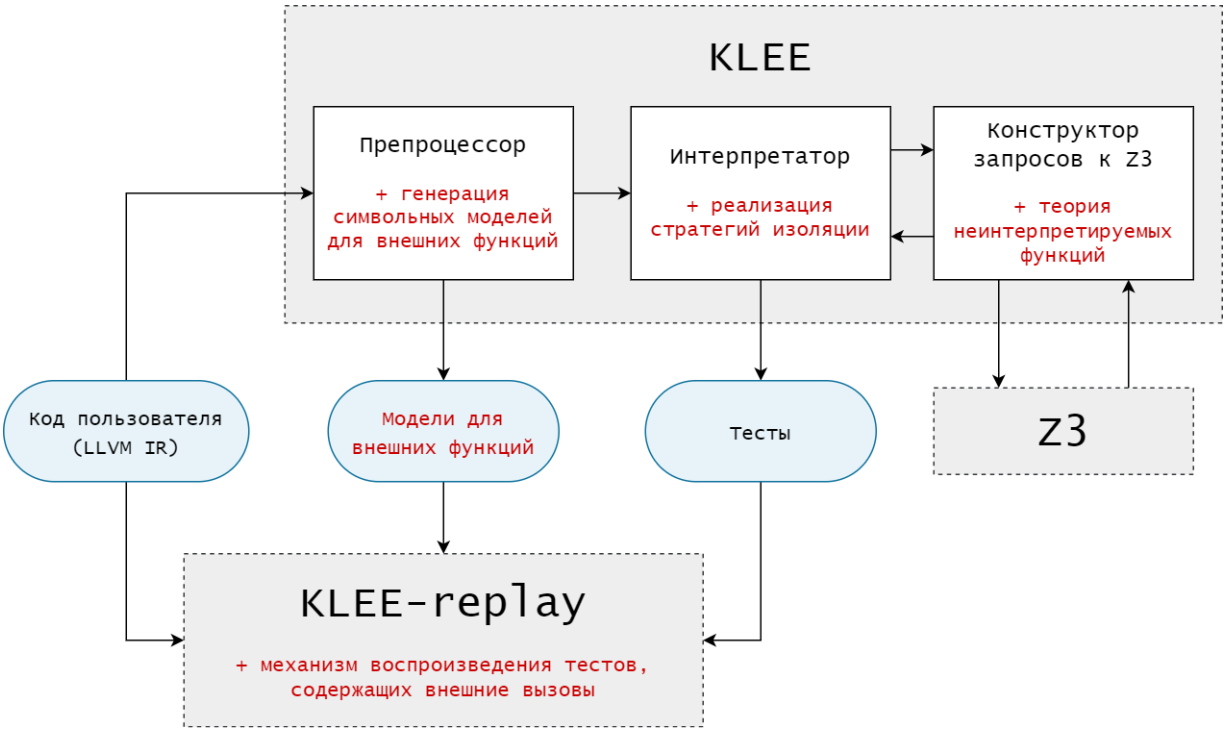

Рис. 3: Архитектура KLEE после внедрения автоматической изоляции.

На этапе препроцессинга добавилась автоматическая генерация символьных моделей в формате LLVM IR. Сгенерированные символьные модели теперь являются артефактами KLEE и используются для воспроизведения тестов.

В интерпретаторе были реализованы две стратегии изоляции, позволяющие генерировать детерминированные или недетерминированные символьные модели в зависимости от нужд пользователя.

В конструкторе запросов к Z3 была поддержана теория неинтерпретируемых функций, необходимая для создания детерминированных символьных моделей.

Механизм KLEE-replay был модифицирован таким образом, чтобы тесты, вызывающие внешние функции, могли быть воспроизведены с учётом сгенерированных символьных моделей.

4. Тестирование

В данной главе описаны тестовая инфраструктура для проведения экспериментов и результаты, которые были достигнуты.

4.1. Тестовая инфраструктура

В ходе экспериментов сравнивалась работа оригинального KLEE и KLEE с реализованной автоматической изоляцией. В качестве метрик для сравнения были выбраны количество сгенерированных тестов, количество обнаруженных уязвимостей, а также покрытие строк исходного кода тестами. Будем говорить, что строка кода покрывается набором тестов, если в результате его запуска данная строка была исполнена. Соответственно, покрытие исходного кода — это процентное соотношение покрытых сгенерированными тестами строк кода к их общему количеству.

В качестве проекта для сравнения поведения оригинального KLEE и KLEE с реализованной автоматической изоляцией был выбран проект с открытым исходным кодом FRRouting [23]. FRRouting реализует различные протоколы маршрутизации IPv4 и IPv6 и управляет ими. Проект содержит большое число внешних функций и вызовов, именно поэтому он был выбран для тестирования реализованной функциональности.

Для измерения покрытия кода использован инструмент llvm-cov [24]. Стоит отметить, что данный инструмент учитывает в итоговом покрытии только те тесты, исполнение которых завершилось успешно.

Тестирование проводилось в контейнере с операционной системой Ubuntu 18.04 внутри виртуальной машины с процессором Intel Xeon E5-2690 с выделенными 16 потоками, 32 Гб оперативной памяти и 388 Гб HDD.

4.2. Результаты

4.2.1 Недетерминированные символьные модели

Ввиду наличия в исходном коде проекта FRRouting внешних переменных, оригинальный KLEE не смог сгенерировать ни одного теста.

Для тестирования модифицированного KLEE каждый из тестируемых файлов проекта собирался в отдельности при помощи инструментов LLVM. Файлы для тестирования отбирались по принципу наличия в них функции `main`, так как это необходимо для запуска тестов и корректной работы KLEE. Всего был отобран 21 файл. На каждом из файлов был запущен KLEE с выбранной недетерминированной стратегией изоляции и символьными моделями для аллокаторов. Время одного запуска было ограничено сверху одной минутой.

В итоге модифицированному KLEE удалось сгенерировать 97 тестов. Из них 50 приводят к аварийному завершению программы, то есть обнаруживают потенциальные уязвимости. Среди обнаруженных уязвимостей 33 являются разыменованием нулевого указателя, а оставшиеся относятся к падениям функции `assert` или завершению программы с ненулевым кодом возврата.

Однако обнаруженные уязвимости ещё не говорят о реальных ошибках в программе, а лишь указывают на то, что авторы FRRouting использовали некоторые предположения о поведении внешних функций, которые не доступны KLEE в ходе анализа. Несмотря на это, данные тесты могут быть полезны в улучшении качества кода, так как использование предположений о поведении внешних функций может приводить к потенциальным ошибкам в ходе дальнейшей разработки.

Исполнение остальных сгенерированных тестов завершается успешно. Для этого набора тестов было вычислено покрытие на различных утилитах проекта. График на рисунке 4 представляет собой распределение полученного на каждом из файлов покрытия. Например, из графика видно, что на трёх утилитах было получено покрытие от 70% до 80%. В итоге было протестировано более 4200 строк исходного кода, из которых покрыто 1700. Таким образом, полученное покрытие составило 40%. Оставшиеся строки не были покрыты в основном по следующим причинам.

- **Неизменяемые аргументы в символьных моделях.** Семантика многих внешних функций тестируемого проекта подразумевает изменение или инициализацию переданных по указателю переменных. В то же время реализованные в рамках данной работы символьные модели не

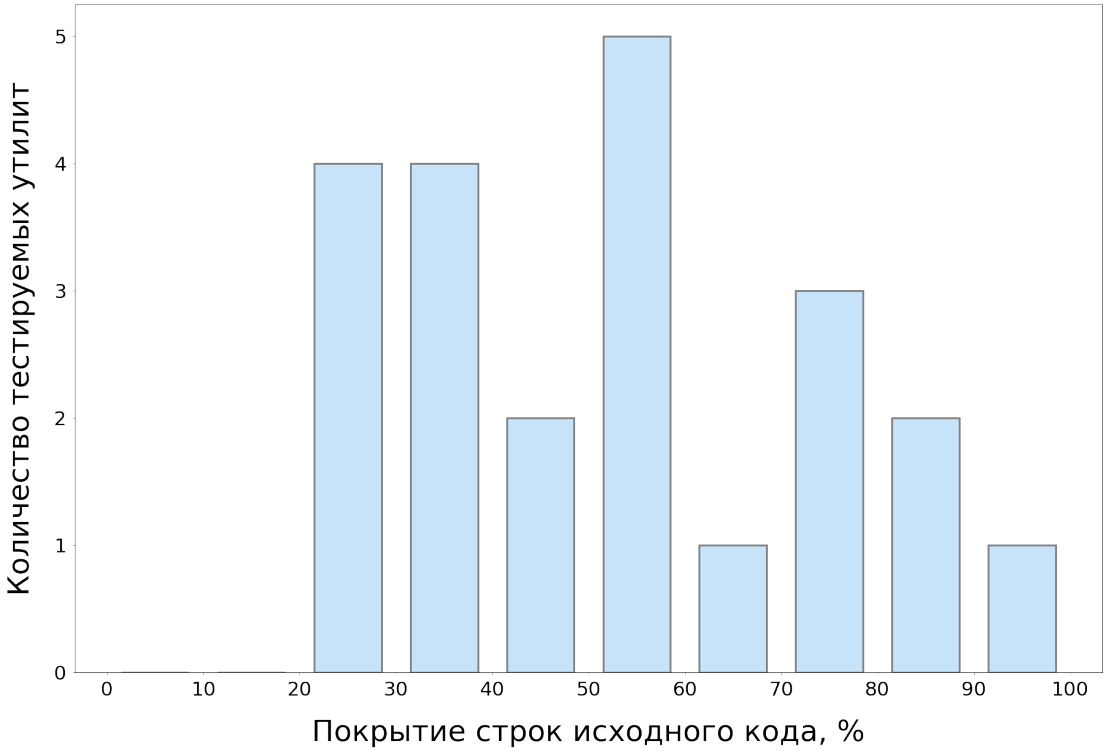

Рис. 4: Покрытие утилит проекта FRRouting.

подразумевают такого рода изменений из-за сложности их моделирования. Таким образом, некоторые из ветвей исполнения остаются непокрытыми в связи с тем, что переменные не были инициализированы или изменены. Генерация более сложных символьных моделей является предметом будущих исследований.

- **Экспоненциальный взрыв числа путей исполнения.** Из-за наличия в проекте циклов и рекурсий, число исследуемых путей исполнения растёт экспоненциально. Из-за этого KLEE может застревать в некоторых ветках и не исследовать другие. Данная проблема релевантна для символьного исполнения в целом.

4.2.2 Детерминированные символьные модели

Детерминированные символьные модели также были протестированы на проекте FRRouting, однако их поведение не отличалось от поведения недетерминированных моделей значительно. Это связано с тем, что поведение KLEE с данными стратегиями изоляции отличается только в том случае, если одна внешняя функция несколько раз вызывалась с одинаковым набором аргументов, что не свойственно тестируемому проекту.

Однако стоит подчеркнуть, что, несмотря на наличие более сложных ограничений, KLEE с детерминированными символьными моделями не уступают в производительности KLEE с недетерминированными моделями. Более того, наличие детерминированных моделей позволяет генерировать более релевантные запросу пользователя тесты.

Отдельно стоит отметить, что на текущий момент для всех внешних функций проекта одновременно может быть создан только один тип символьных моделей. В ходе дальнейшего развития продукта планируется добавить поддержку пользовательских аннотаций, при помощи которых пользователь сможет размечать внешние функции проекта в соответствии с их свойствами. Например, если пользователь укажет в конфигурационном файле, что функция является детерминированной, для неё будет создана детерминированная символьная модель.

Другим примером аннотации является возможность функции возвращать нулевой указатель. Если пользователь отмечает, что функция может вернуть нулевой указатель и её можно вызывать конкретно, для неё можно создать символьную модель по аналогии с реализованными моделями для стандартных аллокаторов.

4.3. Выводы

В рамках работы было проведено тестирование реализованной функциональности на промышленном проекте FRRouting. Результаты показывают, что внедрённый алгоритм автоматической изоляции действительно улучшает работу оригинального KLEE и позволяет генерировать тесты для программ, содержащих внешние вызовы. Благодаря этому, появилась возможность те-

стировать при помощи KLEE не только собранные целиком проекты, но и их компоненты собранные в отдельности. Это может значительно упростить применение инструмента в будущем.

Также были выделены области для дальнейшего развития. К ним относятся моделирование изменений переданных по указателю аргументов и поддержка пользовательских аннотаций.

Заключение

В данной работе были выполнены следующие задачи.

- Рассмотрены способы анализа внешних вызовов, реализованные в различных инструментах символьного исполнения. В результате проведённого обзора было принято решение реализовать автоматическую изоляцию тестируемого кода при помощи генерации символьных моделей для внешних функций. В качестве основы для реализации алгоритма была выбрана символьная виртуальная машина KLEE.

- Разработаны два типа символьных моделей: детерминированные и недетерминированные. Их использование позволяет пользователю получать более релевантные его запросу тесты.

- Разработан и реализован алгоритм автоматической генерации символьных моделей для внешних функций. Также реализован механизм воспроизведения тестов, содержащих внешние вызовы, с использованием сгенерированных символьных моделей.

- Разработаны и реализованы символьные модели для стандартных аллокаторов языка C. Также поддержано воспроизведение тестов, полученных в результате применения данных моделей. Данная функциональность позволяет анализировать редкие сценарии выполнения программы, а также детерминировано их воспроизводить.

- Проведена апробация реализованной функциональности на проекте FRRouting, которая показала эффективность модифицированного KLEE в сравнении с оригинальным в случае анализа программ, содержащих внешние вызовы. Модифицированному KLEE удалось покрыть 1700 из 4200 тестируемых строк кода, что составило 40%. В то же время оригинальный KLEE не покрыл ни строки в связи наличием в исходном коде внешних переменных.

Список литературы

[1] Edvardsson J. A survey on automatic test data generation // Proceedings of the 2nd Conference on Computer Science and Engineering. – 1999. – С. 21-28.

[2] Cadar C. et al. Symbolic execution for software testing in practice: preliminary assessment // Proceedings of the 33rd International Conference on Software Engineering. – 2011. – С. 1066-1071.

[3] Baldoni R. et al. A survey of symbolic execution techniques // ACM Computing Surveys (CSUR). – 2018. – Т. 51. – №. 3. – С. 1-39.

[4] Venolia G., DeLine R., LaToza T. Software development at Microsoft observed // Microsoft Research, TR. – 2005.

[5] Ma K. K. et al. Directed symbolic execution // Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18. – Springer Berlin Heidelberg, 2011. – С. 95-111.

[6] Parry O. et al. A survey of flaky tests // ACM Transactions on Software Engineering and Methodology (TOSEM). – 2021. – Т. 31. – №. 1. – С. 1-74.

[7] Cadar C. et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs // OSDI. – 2008. – Т. 8. – С. 209-224.

[8] Beyer D. Software testing: 5th comparative evaluation: Test-Comp 2023 // Fundamental Approaches to Software Engineering: 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings. – Cham : Springer Nature Switzerland, 2023. – С. 309-323.

[9] Beyer D. Advances in Automatic Software Testing: Test-Comp 2022 // FASE. – 2022. – С. 321-335.

[10] Beyer D. Status report on software testing: Test-Comp 2021 // Fundamental Approaches to Software Engineering: 24th International Conference, FASE 2021, Held as Part of the European Joint Conferences on Theory and Practice

of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings 24. – Springer International Publishing, 2021. – C. 341-357.

[11] Jaffar J. et al. TracerX: Dynamic symbolic execution with interpolation (competition contribution) // Fundamental Approaches to Software Engineering. – 2020. – T. 12076. – C. 530.

[12] Chalupa M. et al. Symbiotic 8: Beyond Symbolic Execution: (Competition Contribution) // Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part II 27. – Springer International Publishing, 2021. – C. 453-457.

[13] Li G., Ghosh I., Rajan S. P. KLOVER: A symbolic execution and automatic test generation tool for C++ programs // Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23. – Springer Berlin Heidelberg, 2011. – C. 609-615.

[14] Ciortea L. et al. Cloud9: A software testing service // ACM SIGOPS Operating Systems Review. – 2010. – T. 43. – №. 4. – C. 5-10.

[15] Le H. M. LLVM-based Hybrid Fuzzing with LibKluzzer (Competition Contribution) // FASE. – 2020. – C. 535-539.

[16] Barrett C., Tinelli C. Satisfiability modulo theories. – Springer International Publishing, 2018. – C. 305-343.

[17] Sen K., Marinov D., Agha G. CUTE: A concolic unit testing engine for C // ACM SIGSOFT Software Engineering Notes. – 2005. – T. 30. – №. 5. – C. 263-272.

[18] Godefroid P., Klarlund N., Sen K. DART: Directed automated random testing // Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. – 2005. – C. 213-223.

[19] Avgerinos T. et al. Automatic exploit generation // Communications of the ACM. – 2014. – Т. 57. – №. 2. – С. 74-84.

[20] Bellard F. QEMU, a fast and portable dynamic translator // USENIX annual technical conference, FREENIX Track. – 2005. – Т. 41. – С. 46.

[21] Lattner C., Adve V. LLVM: A compilation framework for lifelong program analysis & transformation // International symposium on code generation and optimization, 2004. CGO 2004. – IEEE, 2004. – С. 75-86.

[22] De Moura L., Bjørner N. Z3: An efficient SMT solver // Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14. – Springer Berlin Heidelberg, 2008. – С. 337-340.

[23] FRRouting Project. URL: `https://frrouting.org/` (дата обр. 24.05.2023).

[24] The LLVM Compiler Infrastructure Project: llvm-cov. URL: `https://llvm.org/docs/CommandGuide/llvm-cov.html` (дата обр. 24.05.2023).