Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

Григорьев Савелий Алексеевич

Выпускная квалификационная работа

Применение двунаправленного исполнения для вывода индуктивных инвариантов в символьной виртуальной машине KLEE

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., доц. Мордвинов Д. А.

Рецензент:
ст. пр. Куликов Е. К.

Санкт-Петербург
2023 г.

Содержание

Введение

Тестирование — одна из обязательных частей разработки программного обеспечения. Целью тестирования являются нахождение ошибок и критических уязвимостей в коде, минимизация усилий и времени, затраченных на проверку надёжности, производительности программы, её соответствия стандартам, правилам и требованиям.

Важным шагом в развитии тестирования программного обеспечения является автоматизация генерации тестов, позволяющая освободить разработчиков от одной из тяжёлых и трудоёмких частей производства продукта [1]. Основная проблема автоматизации тестирования состоит в том, что количество путей исполнения в программе может быть сколь угодно большим. Следовательно, автоматический генератор тестов должен создать столько входных аргументов для тестируемой функции, сколько необходимо для покрытия максимально возможного количества путей исполнения.

Современные подходы к решению этой проблемы являются комбинацией поведенческого (black-box test generation) и структурного (white-box test generation) тестирования [2]. Традиционным примером поведенческого тестирования является генерация случайных входных данных для программы (или же «фаззинг»), а структурного — символьное исполнение. Если сравнить поведенческое и структурное тестирование, то можно отметить, что поведенческое тестирование быстрее анализирует пути исполнения, однако обладает неточностью, поскольку не покрывает редкие, труднодостижимые пути. Наоборот, структурное тестирование отличается своей точностью, но работает медленнее, так как сталкивается с проблемой «взрыва числа путей исполнения» [3]. «Взрыв путей» происходит из-за того, что условные операторы и операторы цикла экспоненциально увеличивают количество возможных путей исполнения программы. На практике большая часть путей не представляет интереса для тестирования, поскольку не добавляет нового поведения в сравнении с уже проанализированными. Проблеме «экспоненциального взрыва» уделяется большое внимание со стороны научного сообщества [4, 5, 6].

Одним из возможных подходов структурного анализа является *прямое символьное* исследование кода программы, а именно символьное исполнение

от точки входа. Но такой способ не может осуществить направленное исследование к конкретной инструкции в коде. В дополнение к прямому можно использовать *обратное символьное* исполнение, позволяющее запустить исследование в направлении от целевой инструкции к точке входа [7]. И в том, и в другом случае существует проблема анализа большого числа абстрактных путей, не реализующихся в реальности. Эффективной комбинацией этих двух подходов является *двунаправленное символьное исполнение* [8].

Двунаправленное исполнение может помочь в решении основной проблемы символьного исполнения — проблемы «взрыва путей». При анализе вопроса «экспоненциального взрыва» можно обнаружить, что не всегда различное количество итераций цикла будет приводить к отличающимся результатам работы программы. Существуют ситуации, в которых программа сохраняет свойство некоторого состояния вне зависимости от числа итераций цикла, другими словами, поддерживает *инвариант*. Инвариантами можно воспользоваться для уменьшения числа путей анализа программы, чтобы отбрасывать заведомо недостижимые участки кода.

Одним из популярных и широко используемых инструментов с открытым исходным кодом, который занимается автоматической генерацией тестов на основе символьного исполнения, является KLEE [9].

Данная работа посвящена улучшению анализа кода символьной виртуальной машиной KLEE путём внедрения вывода индуктивных инвариантов при помощи стратегии двунаправленного исполнения.

Постановка задачи

Целью данной работы является реализация вывода индуктивных инвариантов с применением двунаправленного исполнения в символьной виртуальной машине KLEE [9]. В контексте данной работы были поставлены следующие задачи:

- Произвести обзор существующих подходов и решений в области вывода индуктивных инвариантов среди инструментов автоматической генерации тестов.
- Разработать эффективный алгоритм проверки потенциальных кандидатов-лемм на индуктивность с применением двунаправленного символьного исполнения.
- Реализовать полученный алгоритм в символьной виртуальной машине KLEE.
- Протестировать реализацию на проверочных данных, проанализировав, в каких ситуациях сгенерированные индуктивные инварианты позволяют KLEE улучшить анализ кода, а в каких — нет.

1. Обзорный раздел по предметной области

В данной главе представлен обзор подходов и решений в области вывода индуктивных инвариантов среди инструментов автоматической генерации тестов. На основе этого анализа были выбраны инструмент и алгоритм для реализации вывода индуктивных инвариантов.

1.1. Подходы к автоматической генерации тестовых данных

Так как в настоящее время методы автоматической генерации тестов являются комбинацией поведенческого и структурного тестирования [2], то далее будут подробнее разобраны оба этих традиционных подхода.

1.1.1. Поведенческое тестирование

Поведенческое тестирование позволяет проверять функциональное поведение приложения без учёта его внутреннего устройства [10]. Поэтому такой подход также называют методом «чёрного ящика» (black-box test generation). Поскольку инструменты, построенные на основе только поведенческого тестирования, не анализируют внутреннее устройство программы, они работают быстрее других аналогов, однако покрывают меньшее количество инструкций исходного кода [3]. Это связано с тем, что данный подход анализирует только информацию о внешнем поведении программы, что не позволяет провести *исчерпывающее тестирование*.

Различия в подходах в данной области заключаются в выборе методов генерации новых тестовых данных на основе ранее сгенерированных входных параметров и результатов работы программы на них. Приведём некоторые из них.

Случайное тестирование. Самый простой метод автоматической генерации тестов, заключающийся в случайном подборе входных аргументов для программы [11]. Выбор значений может зависеть от конкретных констант, соответствующих интересным граничным случаям, например, 0, 1, -1, `MININT`, `MAXINT`, а может зависеть только от вероятностного распределения, определённого заранее.

Фаззинг на основе генетических алгоритмов. Фаззинг при тестировании используется для обнаружения проблем безопасности в компьютерных системах. Целью фаззинга являются аварийные завершения, зависания, утечка памяти и другие нарушения внутренней логики работы программы.

Генетические алгоритмы создавались по образу процесса естественного отбора, где выживали только лучшие представители. Отличительной особенностью этих алгоритмов является акцент на «скрещивании» — рекомбинации различных кандидатов с целью получения нового кандидата, превосходящего предков [12].

Фаззинг на основе генетических алгоритмов, упрощая, можно представить как цикл, где на каждой итерации происходит следующее [13]:

1. Из двух наборов тестов по некоторой метрике выбираются лучшие, которые затем при помощи «скрещивания» порождают два новых тестовых набора.

2. В тестовые наборы вносятся «мутации» — случайные изменения тестов, либо добавление новых.

3. Если изменения не привели к увеличению метрики, то изменения отклоняются, в противном случае они используются для дальнейшего процесса.

Фаззинг на основе машинного обучения. В отличие от предыдущего пункта в данном методе новые тесты, новые «мутации» вносятся при помощи машинного обучения [14, 15]. Модель предобучают, какие «мутации» наиболее успешны для конкретных типов входных аргументов, и на основе этой модели генерируют новые модификации для тестового набора. Также решение о том, оставлять ли новые тесты в наборе, то есть считать ли «мутации» успешными, тоже можно передать машинному обучению.

1.1.2. Структурное тестирование

Структурное тестирование позволяет проверять функциональное поведение приложения с учётом его внутреннего устройства [16]. Соответственно,

по аналогии с методом «чёрного ящика» структурное тестирование называют методом «белого ящика» (white-box test generation). В отличие от поведенческого структурное тестирование может обеспечить *исчерпывающее тестирование* исходного кода. Однако большинство программ имеют сколь угодно много путей исполнения, а интересных с точки зрения анализа путей там может быть лишь малая часть. Поэтому инструменты, построенные на структурном тестировании, страдают от большого числа путей исполнения, в том числе от проблемы «экспоненциального взрыва» путей, порождённых условными операторами и циклами.

Рассмотрим различные подходы в области структурного тестирования.

Структурный фаззинг. Структурный фаззинг отличается от уже описанных выше подходов в фаззинге тем, что при генерации новых тестовых данных использует информацию о коде программы, например, извлекает константы из кода в качестве потенциальных кандидатов, которые могут соответствовать различным случаям работы программы. Структурный фаззинг может показывать результаты выше, чем у поведенческого тестирования, однако он тратит больше времени на анализ программы. Поэтому в ситуациях, когда время исполнения критично, стоит использовать поведенческий фаззинг, а если жёстких ограничений на время нет, то можно использовать структурный фаззинг [17].

Прямое символьное исполнение. Символьное исполнение (англ. *symbolic execution*), в отличие от предыдущих методов, не требует запуска тестируемой программы, следовательно, анализаторы на его основе являются в большей части статическими. Хотя существуют анализаторы, основанные на *динамическом символьном исполнении*, сочетающим в себе чистое символьное исполнение и конкретное исполнение (в английском языке такой подход называют *concolic* как результат слияния слов *concrete* и *symbolic*) [18].

Основной идеей символьного исполнения является работа не с конкретными значениями переменных, а с *символьными*. Рассмотрим подробнее устройство классического прямого символьного исполнения. Алгоритм символьного исполнения начинает свою работу, принимая на вход программу для анализа. В течение всего процесса исполнения алгоритм поддерживает информацию об анализе в виде *состояний исполнения* программы. *Состоя-*

ние исполнения, чаще всего, включает в себя следующую информацию [3]:

- путь исполнения $path$, соответствующий данному состоянию.
- условие пути π, представляющее из себя логическую формулу, обозначающую условие на символьные переменные вдоль текущего пути исполнения. Другими словами, чтобы пройти по данному пути исполнения, необходимо и достаточно, чтобы входные аргументы программы удовлетворяли условию π.
- текущую исполняемую инструкцию $stmt$.
- символьную память σ, хранящую сопоставление текущих переменных программы и их значений, выраженных через символьные переменные.

Алгоритм 1: Символьное исполнение

Исходные данные: $P, start$
Результат: $tests$

```
1  execute()
2      Q ← {start};
3      while Q ≠ ∅ do
4          s ← pickNext(Q);
5          Q ← Q \ {s};
6          forward(s, Q);
7      end
8      return;
9  forward(s, Q)
10     S ← execInstr(s.curr, s);
11     if S ≠ ∅ then
12         forall s' ∈ S ∧ SAT(s'.pc) do
13             Q ← Q ∪ {s'};
14         end
15     else
16         processTestCase(s);
17     end
18     return;
```

В алгоритме 1 представлен упрощённый псевдокод процедуры прямого символьного исполнения. На вход алгоритму подаются программа P и стартовое состояние исполнения $start$. В процессе работы алгоритм поддерживает очередь Q всех текущих состояний, которую изначально инициализирует стартовым состоянием $start$. На каждом шаге работы алгоритм выбирает состояние s, удаляет его из очереди, выполняет соответствующую этому состоянию инструкцию $s.curr$, а затем все новые достижимые состояния s' добавляет в очередь. Состояние является достижимым, если выполнима логическая формула $s'.pc$. Пустое множество новых состояний при исполнении инструкции соответствует ситуации, когда алгоритм дошёл до конечной точки исполнения программы, а значит, может использовать текущее состояние для генерации теста. Сгенерированный из состояния s тест пойдёт по такому же пути, что и состояние s.

Прямое символьное исполнение анализирует все возможные пути исполнения, поэтому подвержено проблеме «экспоненциального взрыва числа путей». Например, при запуске на рекурсивной программе, либо на программе с циклом, количество итераций которого не фиксировано, символьное исполнение не завершится, поскольку множество достижимых состояний велико. В прямом символьном исполнении стратегия выбора следующего состояния определяется функцией $\texttt{pickNext}(Q)$, но поскольку при исполнении от старта невозможно предугадать, дойдёт или нет состояние до конкретной локации кода, сделать анализ программы направленным не представляется возможным, что влияет на качество генерируемых тестовых данных.

Обратное символьное исполнение. Обратное символьное исполнение является двойственным к прямому. Основное отличие заключается в том, что обратное исполнение начинает анализ не из старта программы, а с некоторой заданной инструкции, запуская процесс в обратном направлении [7]. Поскольку стартовую инструкцию для обратного исполнения можно выбрать любую, то это позволяет осуществлять *направленный* анализ кода программы.

В остальном обратное исполнение похоже на прямое: так же на каждом шаге выбирается состояние, которое продвигается в обратном направлении, генерируя новые состояния, часть из которых отсекается по причине невыполнимости ограничений вдоль пути исполнения. Следовательно, обратное

```
1  int func(int x, int y)
2  {
3      int res = 0;
4      if (x + y > 10)
5      {
6          res = x - y;
7      }
8
9      if (res == 1)
10     {
11         return 1;
12     }
13     else
14     {
15         return 2;
16     }
17 }
```

Листинг 1: функция `func` для иллюстрации символьного исполнения

исполнение страдает от тех же проблем, что и прямое, включая «взрыв числа путей». Однако эффективная комбинация этих двух подходов приводит к *двунаправленному исполнению*, позволяющему эффективно анализировать программу и труднодостижимые места кода, минимизируя недостатки обоих подходов [19, 20].

1.2. Механизм работы символьного исполнения

Изучим подробнее механизм работы прямого и обратного символьного исполнения на примере функции `func` (листинг 1).

`foo` принимает на вход два аргумента. Требуется ответить на вопрос: какие значения может вернуть данная функция? Также необходимо для каждого возвращаемого значения $result$ предоставить конкретные значения аргументов x и y, на которых `foo` возвращает $result$.

Перед исполнением каждому из входных аргументов `foo` ставится в соответствие собственная символьная переменная (например, аргументам x и y будут сопоставлены переменные α_x и α_y). Именно это соответствие будет храниться в символьной памяти σ.

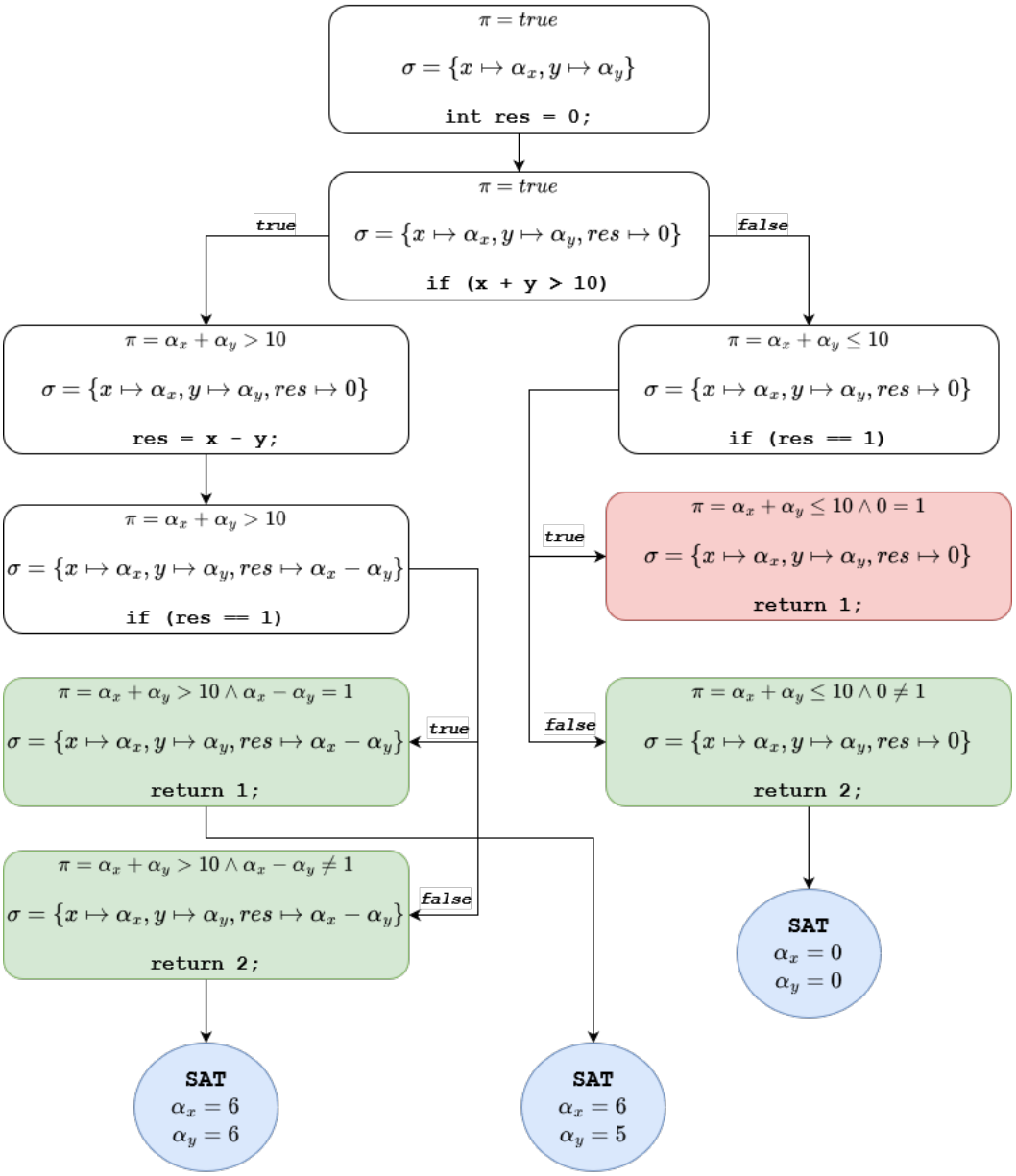

Рисунок 1: схема прямого символьного исполнения функции `func`

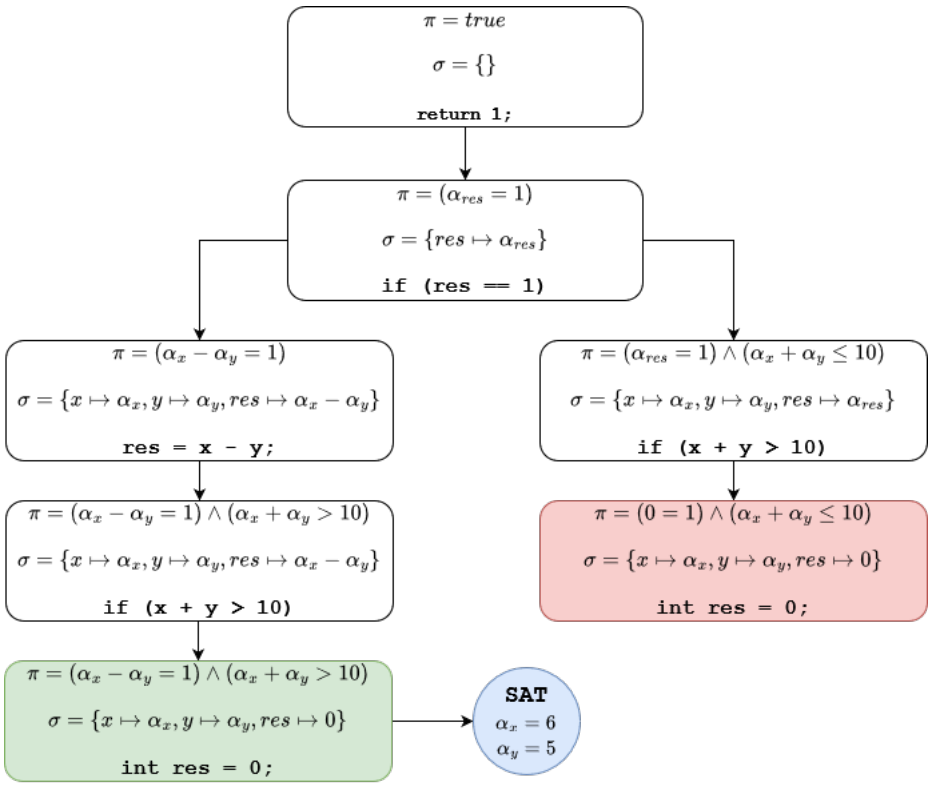

Рисунок 2: схема обратного символьного исполнения функции `func`

Схема работы прямого символьного исполнения представлена на рисунке 1. Исполняя инструкции по очереди, символьная машина обновляет символьную память σ и условие пути π. Проходя через условный оператор, она создаёт два состояния, в которых π равняется конъюнкции текущего значения и условия вхождения в соответствующую ветку исполнения. В дальнейшем анализ этих состояний происходит независимо.

В результате работы четыре состояния дошли до выхода из функции. Однако одно из них недостижимо, а для трёх других SMT-решатель найдёт удовлетворяющие наборы входных аргументов функции. Такой набор называется **моделью**.

Теперь взглянем на работу обратного символьного исполнения. Поскольку обратное исполнение позволяет выполнять *направленный* анализ, то рассмотрим в качестве целевой инструкцию `return` на 11 строке.

Схема работы обратного символьного исполнения представлена на рисунке 2. Исполняя инструкции по очереди в обратном порядке, символьная машина так же, как и в прямом случае, обновляет символьную память σ и

```
1  void func(int n)
2  {
3      int x = 0;
4      int i = 0;
5      while (i < n)
6      {
7          ++x;
8          ++i;
9          assert(x == i);
10     }
11 }
```

Листинг 2: функция `func` для иллюстрации лемм в `assert`-инструкциях

условие пути π. Поскольку в инструкцию `if (res == 1)` можно прийти из двух различных инструкций, а именно из `res = x - y;` и `if (x + y > 10)` в случае ветки `else`, то символьная машина создаёт два новых состояния, которые в дальнейшем исполняются независимо.

В результате одно из двух состояний будет недостижимо, а второе найдёт возможные значения входных аргументов $x = 6, y = 5$.

1.3. Подходы к выводу лемм и их проверке на инвариантность

Существует несколько подходов к выводу лемм как в фаззинге, так и в символьном исполнении.

- Самый простой — использование в качестве лемм выражений в коде, которые упоминаются в качестве `assert`-инструкций. В листинге 2 в качестве такой леммы выступает выражение `x == i`, которое действительно является инвариантом относительно итерации цикла.

- В символьном исполнении в случае, если продвижение анализа за пределы цикла постоянно невыполнимо, в качестве леммы можно рассматривать как отрицание формулы пути, так и некоторое *совместно невыполнимое множество конъюнктов* из исходной формулы (*англ. unsat core*) [21].

- Самым продвинутым подходом в данной области является алгоритм IC3

```
1  void func(int n)
2  {
3      int i = 0, a = 1, b = 2, c = 3;
4      while (i < n)
5      {
6          assert(a != b);
7          int tmp = a;
8          a = b;
9          b = c;
10         c = tmp;
11         i++;
12     }
13 }
```

Листинг 3: функция `func` для иллюстрации `k`-индукции

[22]. Он позволяет с помощью специальной иерархической структуры лемм OARS находить инварианты, «просеивая» более сильные леммы на верхние уровни (то есть продвигая вверх более потенциальных кандидатов на инвариант), оставляя на нижних неподходящие леммы.

После нахождения потенциальных кандидатов-инвариантов требуется проверить, является ли данная лемма инвариантом. В частном случае достаточно выполнить одну итерацию цикла, чтобы удостовериться в выполнимости инварианта. Такой инвариант называют **индуктивным**.

Однако существуют ситуации, когда одной итерации цикла недостаточно, чтобы доказать корректность инварианта. В таком случае необходимо сделать несколько итераций. Данный подход называют `k`-индукцией [23]. Например, в листинге 3 видно, что для доказательства инварианта `a != b` требуется выполнить три итерации цикла.

1.4. Инструменты с открытым исходным кодом

В данном разделе рассматриваются только инструменты с открытым исходным кодом, поскольку целью данного обзора является выбор подходящего инструмента автоматической генерации тестов для реализации алгоритма вывода индуктивных инвариантов.

Выбор инструмента для реализации алгоритма основывался на модульности архитектуры, возможности добавления нового кода в уже существующую кодовую базу проекта, а также на подходах и дополнительных важных свойствах инструмента.

В качестве потенциальных кандидатов были рассмотрены инструменты 2LS [24], Angr [25], KLEE [9].

2LS. Инструмент на основе символьного исполнения, предназначенный для верификации программ, написанных на языке программирования C. В своей работе наибольший акцент делает на выводе k-инвариантов при помощи k-индукции [26]. Также разработчики инструмента в качестве достоинств отмечают нахождение ошибок, связанных с некорректной работой с памятью (*англ. memory safety*), а также работу с вещественными числами, но указывают, что рекурсивные программы плохо поддаются анализу инструмента.

Angr. Данный инструмент, разработанный на языке Python, предназначен для многофункционального анализа двоичных файлов. Angr может комбинировать методы фаззинга и динамического символьного исполнения (то есть комбинацию конкретного и символьного исполнения программы). Также инструмент способен проводить трансляцию кода в промежуточное представление, анализ потока управления и инструментацию исходного кода. Особого внимания выводу инвариантов инструмент не уделяет, концентрируясь на других свойствах, позволяющих ему улучшать анализ программ.

KLEE. Данный инструмент на основе символьного исполнения предназначен для автоматической генерации тестов для программ, написанных на языках C и C++, и нахождения уязвимостей в них. KLEE использует множество оптимизаций для хранения состояний программы, эффективно проверяет на выполнимость условия пути исполнения, обеспечивая качественное покрытие кода программы тестами. Всё это улучшает производительность KLEE по сравнению с аналогичными инструментами и позволяет генерировать тесты для множества самых разных программ.

Тесты, сгенерированные KLEE, имеют высокий процент покрытия кода на реальных крупных проектах [27]. Например, KLEE успешно запускался на программах из пакета GNU Coreutils [28], состоящего из примерно $61\,000$ строк кода утилит и $80\,000$ строк библиотечного кода.

KLEE внутри себя имеет удобную модульную архитектуру, что позволяет легко модифицировать код, а также добавлять новые расширения при помощи уже реализованных модулей.

В компании Huawei разрабатывается модифицированная версия KLEE с различными дополнениями и улучшениями [29]. В частности, одной из особенностей данной версии является реализованный модуль двунаправленного символьного исполнения, позволяющий эффективно комбинировать преимущества прямого и обратного символьного исполнения.

Модуль двунаправленного исполнения имеет непосредственное отношение к выводу инвариантов, поскольку благодаря направленному анализу, появляется возможность исследовать заданные участки кода, следовательно, в случае если какая-то инструкция остаётся недостижимой после множества итераций цикла, может возникнуть запрос на проверку индуктивности некоторой леммы, из-за которой инструкция остаётся недостижимой.

1.5. Выводы

Результатом проведения обзора стал выбор инструмента KLEE для реализации алгоритма вывода индуктивных инвариантов, важной составляющей деталью которого является уже реализованный модуль двунаправленного исполнения. Ключевой особенностью этого инструмента, повлиявшей на выбор, является хорошо выстроенная модульная архитектура проекта, позволяющая легко модифицировать старый код и добавлять новый.

2. Алгоритм вывода индуктивных инвариантов

В этой главе будет подробно разобран алгоритм формирования лемм и их проверки на индуктивность, предложенный автором данной дипломной работы.

2.1. Индуктивный инвариант

Пусть задана некоторая система, описывающаяся тройкой $(V, Init, T)$. Здесь V — множество пропозициональных переменных, $Init(V)$ — формула, обозначающая начальное состояние, а $T(V, V')$ — формула, определяющая множество переходов. Тогда формула P называется *индуктивным инвариантом*, если для неё выполняются два свойства:

- $\models Init(V) \rightarrow P(V)$
- $\models P(V) \wedge T(V, V') \rightarrow P(V')$

Другими словами, P — индуктивный инвариант, если это условие достижимо из начального состояния программы и сохраняется при переходе.

Рассмотрим пример функции `max_elem` для иллюстрации индуктивных инвариантов (листинг 3). В данной функции в переменную `max` записывается максимальное по модулю значение элемента массива `array`. Можно заметить, что условие $\texttt{max} \geq 0$ является индуктивным инвариантом, поскольку достижимо из начального состояния программы (как указано в строке 4, начальное значение `max` равняется 0), а также сохраняется при переходе (внутри одной итерации цикла значение переменной `max` не уменьшается, поэтому не может стать отрицательным). Следовательно, вывод индуктивного инварианта $\texttt{max} \geq 0$ позволил бы упростить анализ данной программы, так как можно было бы не исследовать условный оператор, условие которого ($\texttt{max} < 0$) невыполнимо ни при каких значениях входных аргументов.

2.2. Двунаправленное исполнение

Рассмотрим детальнее алгоритм двунаправленного символьного исполнения (алгоритм 2) [8].

```
void max_elem(std::vector<int> array)
{
    int n = array.size();
    int max = 0;
    for (int i = 0; i < n; i++)
    {
        max = std::max(max, abs(array[i]));
    }
    if (max < 0)
    {
        assert(0);
    }
    return;
}
```

Листинг 3: функция `max_elem` для иллюстрации индуктивных инвариантов

В процессе работы алгоритма поддерживаются две очереди состояний: очередь *прямых состояний* Q_{front} (очередь состояний, которые движутся от стартовой инструкции программы в прямом направлении) и очередь *обратных состояний* Q_{back} (очередь состояний, которые движутся от целевой инструкции в обратном направлении). Обратные состояния называются ***proof obligation*** и описываются тремя характеристиками (loc, lvl, pc): loc — инструкция кода, в которой находится данное состояние; lvl — ограничение на количество посещений одной и той же инструкции; pc — условие пути исполнения.

На каждом шаге основного цикла символьная машина выбирает одно из трёх возможных действий:

1. `Start` s — добавить новое прямое состояние в соответствующую очередь.

2. `GoFront` s — выполнить для прямого состояния s следующую инструкцию $s.curr$, добавив в очередь все появившиеся новые состояния.

Алгоритм 2: Двунаправленное символьное исполнение

Исходные данные: Множество целевых инструкций $targets$, начальное состояние $init$

Результат: Тестовые данные, достигающие целевых инструкций из $targets$

```
1  Q_front ← init;
2  Q_back ← {(loc, BOUND, ⊤) | loc ∈ targets};
3  while Q_front ≠ ∅ ∧ Q_back ≠ ∅ do
4      switch searcher.pick(Q_front, Q_back) do
5          case Start s do
6              Q_front ← Q_front ∪ {s};
7          end
8          case GoFront s do
9              Q_front ← Q_front \ {s};
10             forward(s, Q_front);
11         end
12         case GoBack (s, q) do
13             assert(s.curr = q.loc);
14             backward(s, q, Q_back);
15         end
16     end
17 end
18
19 forward(s, Q_front)
20     forall s' ∈ execInstr(s.curr, s) do
21         if s'.pc is SAT ∧ s'.lvl ≤ BOUND then
22             Q_front ← Q_front ∪ {s'};
23         end
24     end
25 backward(s', q', Q_back)
26     if q'.loc = ENTRY_POINT then
27         // Генерируем тест, достигающий корень q'
28         // Удаляем поддерево q' из Q_back
29     end
30     q ← ⟨s'.start, q'.lvl − s'.lvl, WP(s', q')⟩;
31     if q.pc is SAT ∧ q.lvl ≥ 0 then
32         parent(q) ← q';
33         Q_back ← Q_back ∪ {q};
34     end
```

3. `GoBack`(s', q') — обработать ситуацию, когда прямое состояние s' дошло до обратного состояния q'. В случае, если вдоль пути $s'.path$ можно продвинуть обратное состояние q', то создаётся новое обратное состояние q, которое добавляется в очередь. Если же обратное состояние q' успешно продвинулось до точки входа, то символьная машина сгенерирует подходящие значения входных аргументов для нового теста.

2.3. Алгоритм вывода лемм

У алгоритма из предыдущей главы есть недостаток. Он никак не использует информацию о ситуации, когда в `GoBack` продвинуть обратное состояние не получилось. Невозможность продвинуть обратное состояние может как раз сигнализировать о наличии некоторого инварианта, который не позволяет достигнуть заданную инструкцию в коде.

Предлагается модифицировать описанный ранее алгоритм так, чтобы ситуации, когда обратное состояние продвинуть невозможно, также учитывались символьной машиной.

В новой версии алгоритма в ситуации, когда обратное состояние не удалось продвинуть вдоль пути, генерируется лемма φ. Формула для φ получается следующим образом: поскольку условие пути оказалось невыполнимым, то из этого условия можно извлечь минимальное по включению множество совместно невыполнимых конъюнктов (*англ. minimal unsat core*). Дальше из этих конъюнктов остаются только те, которые имеют отношение к обратному состоянию q (поскольку есть ещё конъюнкты, которые были получены из ограничений прямого состояния s). Соответственно, эти конъюнкты можно объединить конъюнкцией в некоторую формулу ψ, которая будет заведомо невыполнимой при продвижении обратного состояния назад. А поскольку инвариант — это формула, которая выполняется «всегда», то итоговый вид инварианта φ — это отрицание ψ, то есть дизъюнкция отрицаний конъюнктов из минимального невыполнимого набора, соответствующих обратному состоянию q.

В дальнейшем, если оказалось, что φ — индуктивный инвариант, то алгоритм запоминает этот инвариант и при следующих попытках продвинуть

обратное состояние вдоль соответствующего пути приписывает к условию пути конъюнкцию всех инвариантов, соответствующих данному участку кода (`summaries` в алгоритме).

Можно заметить, что в некоторый момент для обратного состояния q' могут быть рассмотрены все потенциальные прямые состояния s', которые могли дойти до него, однако ни для одного из них условие пути не выполнилось. В такой ситуации можно удалить q' из очереди обратных состояний, поскольку символьная машина доказала, что это состояние продвинуть назад невозможно, следовательно, само состояние недостижимо.

Алгоритм 3: Модификация двунаправленного символьного исполнения

Исходные данные: Множество целевых инструкций $targets$, начальное состояние $init$

Результат: Тестовые данные, достигающие целевых инструкций из $targets$

```
1  backward(s', q', Q_back)
2      if q'.loc = ENTRY_POINT then
3          // Генерируем тест, достигающий корень q'
4          // Удаляем поддерево q' из Q_back
5      end
6      q ← ⟨s'.start, q'.lvl − s'.lvl, WP(s', q')⟩;
7      if (q.pc ∧ summaries(q.loc, q.lvl)) is SAT ∧ q.lvl ≥ 0 then
8          parent(q) ← q';
9          Q_back ← Q_back ∪ {q};
10     end
11     else
12         φ ← genLemma(s, q, q');
13         addLemma(q', s.lvl, s.path, φ);
14         if не осталось пути до q' then
15             // Удаляем q' из Q_back
16         end
17     end
18 genLemma(s, q, q')
19     c ← MUC(q);
20     return {¬a | a ∈ q', subst(a, s) ∈ c};
```

2.4. Алгоритм проверки индуктивности лемм

Последняя часть алгоритма — проверка леммы на индуктивность. Поскольку основной интерес для символьной машины представляют инварианты в циклах, позволяющие вне зависимости от количества итераций выводить некоторые утверждения, то основной акцент в алгоритме будет сделан на проверке инвариантов в циклах.

Для доказательства индуктивности, как было указано в подглаве 1, требуется показать, что формула $\varphi \wedge T \rightarrow \varphi'$ всегда выполнима, что равносильно невыполнимости $\neg(\varphi \wedge T \rightarrow \varphi') = \varphi \wedge T \wedge \neg\varphi'$ (здесь φ' обозначает исходную формулу φ после одной итерации цикла).

Таким образом, для проверки достаточно будет создать отдельное обратное состояние q_{lemma}, хранящее отрицание леммы (то есть $\neg\varphi$). После успешного продвижения в обратном направлении вдоль цикла, когда состояние q_{lemma} придёт повторно в исходную точку, условие выполнимости необходимо будет дополнить леммой φ (поскольку ограничения одной итерации цикла и $\neg\varphi'$ уже будут записаны в ограничениях пути).

2.5. Реализация алгоритма в символьной машине KLEE

В данном разделе подробно описаны детали реализации алгоритма вывода индуктивных инвариантов в виртуальной символьной машине KLEE. Код реализации доступен по ссылке [30].

Ниже детально будут разобраны наиболее интересные части реализации, а именно:

- Модуль *ReachabilityTracker*, отвечающий за учёт всех прямых и обратных состояний. Основной его целью является проверка для заданного обратного состояния q существования некоторого прямого состояния s, движущегося к q.
- Проверка сгенерированных лемм на индуктивность путём создания отдельных состояний.

2.5.1. Модуль ReachabilityTracker

В текущей реализации KLEE двунаправленное исполнение устроено следующим образом: исходно прямые состояния, движущиеся к заданной конечной инструкции $finish$, появляются при прохождении некоторой инструкции, создающей несколько путей исполнения (это может быть как условный оператор, так и функции, работающие со внешним окружением, результат работы которых неоднозначен). После того как прямое состояние доходит до целевой инструкции, оно становится готовым для обратного исполнения.

Таким образом, все прямые состояния можно разделить на два больших типа: те, которые ещё движутся вперёд к своим целевым инструкциям, и те, которые уже дошли до назначенного места и ожидают выполнения обратного шага двунаправленного исполнения. Состояния первого типа будем называть `running` состояния, а второго — `waiting` состояния.

В итоге, когда модуль `ReachabilityTracker` проверяет, существует ли какое-то прямое состояние s, которое движется к заданному обратному состоянию q, он должен проверить две вещи:

1. Остались ли `running` состояния, движущиеся к q.
2. Есть ли `waiting` состояния, которые ещё не участвовали в обратном шаге.

Чтобы отвечать на запросы такого характера, `ReachabilityTracker` хранит для каждого обратного состояния q все `running` и `waiting` состояния, относящиеся к q.

2.5.2. Проверка лемм на индуктивность

Как уже описывалось выше, леммы являются дизъюнкцией отрицаний конъюнктов из минимального совместно невыполнимого множества (*minimum unsat core*). Проверка леммы φ на индуктивность равносильна проверке на невыполнимость формулы $\varphi \wedge T \wedge \neg\varphi'$.

Для этой проверки создаётся отдельное обратное состояние q_{lemma}, хранящее отрицание леммы (то есть $\neg\varphi$). Когда это состояние пройдёт итерацию

цикла и вернётся в стартовую точку (чтобы удостовериться, что обратное состояние q' повторно пришло в некоторую инструкцию кода, нужно среди предков q' в дереве обратных состояний найти такой q_{parent}, что $q_{parent}.loc = q'.loc$), нужно усилить условие пути леммой φ, тогда итоговая формула пути будет выглядеть как $\varphi \wedge T \wedge \neg\varphi'$, что и требуется проверить на выполнимость для подтверждения индуктивности инварианта.

3. Тестирование

В этой главе описана инфраструктура для проведения тестирования, результаты тестирования и сделанные по ним выводы. В ходе экспериментов сравнивалась эффективность инструмента автоматической генерации тестовых данных KLEE с модулем двунаправленного исполнения и модифицированного KLEE с модулем вывода индуктивных инвариантов. Результаты этого сравнения покажут, для каких программ вывод индуктивных инвариантов позволил улучшить анализ кода в сравнении с исходной версией символьной машины KLEE.

3.1. Тестовая инфраструктура

Эксперименты проводились на проверочных данных соревнования SVComp [31]. Это соревнование является одним из самых престижных в области тестирования программного обеспечения, позволяющим различным инструментам сравниваться друг с другом на одном общем наборе тестовых данных.

Для экспериментов был выбран набор тестов `loop-invariants`, содержащий различные программы с циклами, анализ каждой из которых символьная машина может завершить, если успешно выведет инвариант. Эксперименты проводились на платформе с операционной системой Ubuntu 20.04 64-bit и процессором Intel Core i7-7700HQ @ 2.80GHz × 8 с ограничением по времени в 120 секунд и ограничением по памяти в 4 ГБ.

3.2. Результаты

Тестирование было проведено на программах из набора `loop-invariants`, состоящего из 12 файлов. На данных задачах символьная машина KLEE без модуля вывода индуктивных инвариантов не могла совершить исчерпывающее тестирование программы, поскольку пошагово разворачивало цикл, анализируя все возможные пути исполнения, что приводило к длительному времени работы программы.

Модуль вывода индуктивных инвариантов позволил завершить анализ

```
1  void sum(int n)
2  {
3      if (n < 10)
4      {
5          int res = 0;
6          for (int i = 0; i < n; i++)
7          {
8              res += 10;
9          }
10         assert(res == 0 || res == 10 * n);
11     }
12 }
```

Листинг 4: функция `sum` для иллюстрации непроанализированного индуктивного инварианта

программы в 8 из 12 случаев, в том числе на функции `max_elem`, представленной в листинге 3.

3.3. Выводы

По результатам тестирования видно, что реализованный алгоритм позволяет KLEE улучшить анализ кода и увеличить спектр программ, на которых символьная машина может провести исчерпывающее тестирование. Повышение эффективности видно для программ с циклами, в которых есть индуктивные инварианты.

Однако далеко не для всех программ с циклами реализованный алгоритм позволяет вывести индуктивный инвариант. В примере из листинга 4 KLEE не может вывести индуктивный инвариант (`res == 0 || res == 10 * i`), поскольку не имеет возможности получить утверждение `res == 10 * i`. Единственное утверждение, которое из кода извлекает символьная машина — это `res == 10 * n`, однако данное утверждение не является индуктивным инвариантом.

3.4. Дальнейшее развитие

Данный алгоритм в будущем можно улучшить, реализовав более эффективный подход к выводу лемм, являющихся потенциальными кандидатами для проверки на индуктивный инвариант. В частности, одним из дальнейших путей исследования в этой области может быть реализация алгоритма IC3 [22].

В области проверки лемм также возможны улучшения, например, можно проанализировать неиндуктивные инварианты и реализовать поддержку k-индукции.

Заключение

В данной работе были выполнены следующие задачи:

- Проведён обзор существующих подходов и решений в области вывода индуктивных инвариантов среди инструментов автоматической генерации тестов 2LS, Angr и KLEE. В ходе обзора в качестве основы для реализации алгоритма был выбран инструмент KLEE.
- Разработан эффективный алгоритм проверки лемм на индуктивность с применением двунаправленного символьного исполнения. Описаны необходимые модификации исходного алгоритма двунаправленного исполнения для внедрения возможности проверки лемм на индуктивность.
- Полученный алгоритм реализован в виртуальной символьной машине KLEE. Описаны необходимые структуры данных и детали реализации.
- Проведено тестирование реализации на проверочных данных соревнования SVComp. Модифицированная версия KLEE в 67% случаев позволила вывести индуктивный инвариант, тем самым упростив символьной машине анализ программы. Однако в ситуациях, когда потенциальная лемма-инвариант не встречается в качестве инструкции, алгоритм не улучшает анализ кода.

Список литературы

[1] Edvardsson Jon. A survey on automatic test data generation // Proceedings of the 2nd Conference on Computer Science and Engineering / Citeseer. — 1999. — P. 21–28.

[2] Nidhra Srinivas, Dondeti Jagruthi. Black box and white box testing techniques-a literature review // International Journal of Embedded Systems and Applications (IJESA). — 2012. — Vol. 2, no. 2. — P. 29–50. Algorithms, 12:3, 2016.

[3] A survey of symbolic execution techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia et al. // ACM Computing Surveys (CSUR). — 2018. — Vol. 51, no. 3. — P. 1–39.

[4] Kuznetsov Volodymyr, Kinder Johannes, Bucur Stefan, and Candea George. Efficient state merging in symbolic execution // Acm Sigplan Notices. — 2012. — Vol. 47, no. 6. — P. 193–204.

[5] Avgerinos Thanassis, Rebert Alexandre, Cha Sang Kil, and Brumley David. Enhancing symbolic execution with veritesting // Proceedings of the 36th International Conference on Software Engineering. — 2014. — P. 1083–1094.

[6] Xie Tao, Tillmann Nikolai, De Halleux Jonathan, and Schulte Wolfram. Fitness-guided path exploration in dynamic symbolic execution // 2009 IEEE/IFIP International Conference on Dependable Systems Networks / IEEE. — 2009. — P. 359–368.

[7] Dinges Peter, Agha Gul. Targeted test input generation using symbolic-concrete backward execution // Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. — 2014. — P. 31–36.

[8] Baluda Mauro, Denaro Giovanni, and Pezzè Mauro. Bidirectional symbolic analysis for effective branch testing // IEEE Transactions on Software Engineering. — 2015. — Vol. 42, no. 5. — P. 403–426.

[9] KLEE Symbolic Execution Engine. URL: https://github.com/klee/klee (дата обращения 27.04.2023).

[10] Gao Jerry, Tsao H-SJ, Wu Ye. Testing and quality assurance for component-based software. — Artech House, 2003.

[11] Miller Barton P, Fredriksen Louis, So Bryan. An empirical study of the reliability of UNIX utilities // Communications of the ACM. — 1990. — Vol. 33, no. 12. — P. 32–44.

[12] Fraser G., Arcuri A. Evolutionary Generation of Whole Test Suites //. — 08/2011. — P. 31–40. — DOI: 10.1109/QSIC.2011.19.

[13] Fraser G., Arcuri A. EvoSuite: Automatic test suite generation for objectoriented software //. — 09/2011. — P. 416–419. — DOI: 10.1145/2025113. 2025179.

[14] Klees G. [et al.]. Evaluating Fuzz Testing — 2018.

[15] Saavedra G. J. [et al.]. A Review of Machine Learning Applications in Fuzzing — 2019.

[16] ISO/IEC/IEEE International Standard - Systems and software 36 engineering–Vocabulary // ISO/IEC/IEEE 24765:2017(E). — 2017. — P. 1–541.

[17] Böhme Marcel, Paul Soumya. A probabilistic analysis of the efficiency of automated software testing // IEEE Transactions on Software Engineering. — 2015. — Vol. 42, no. 4. — P. 345–360.

[18] Cadar C., Dunbar D., Engler D. KLEE: Unassisted and Automatic Generation of Highcoverage Tests for Complex Systems Programs // Proceedings of the 8th USENIX conference on Operating systems design and implementation. — 2008. — P. 209–224.

[19] Directed symbolic execution / Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, Michael Hicks // International Static Analysis Symposium / Springer. — 2011. — P. 95–111.

[20] Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach / Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, Alexander Pretschner // Proceedings of the 33rd Annual ACM Symposium on Applied Computing. — 2018. — P. 1475–1482.

[21] Selsam D., Bjørner N. Guiding high-performance SAT solvers with unsat-core predictions //Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings 22. – Springer International Publishing, 2019. – C. 336-353.

[22] Mayuko Kori, Natsuki Urabe, Shin-ya Katsumata, Kohei Suenaga, and Ichiro Hasuo. The Lattice-Theoretic Essence of Property Directed Reachability Analysis / Sigplan Surveys. — 2015.

[23] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rummer. Software Verification Using k-Induction / URL: https://www.cprover.org/kinduction/appendix.pdf (дата обращения 06.05.2023).

[24] 2LS. Verification Tool for C Programs. URL: https://github.com/diffblue/2ls (дата обращения 07.05.2023).

[25] Angr. Open-source binary analysis platform for Python. URL: https://angr.io/ (дата обращения 07.05.2023).

[26] 2LS For Program Analysis. URL: https://www.researchgate.net/publication/368305213_2LS_for_Program_Analysis (дата обращения 07.05.2023).

[27] KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. / Cristian Cadar, Daniel Dunbar, Dawson R Engler et al. // OSDI. — Vol. 8. — 2008. — P. 209–224.

[28] Coreutils - GNU core utilities. URL: https://www.gnu.org/software/coreutils/ (дата обращения 07.05.2023).

[29] Huawei Patch of KLEE Symbolic Execution Engine. URL: https://github.com/UnitTestBot/klee/ (дата обращения 26.05.2023).

[30] KLEE Symbolic Execution Engine With Inductive Invariants. URL: https://github.com/sava-cska/klee/ (дата обращения 26.05.2023).

[31] SVComp Benchmarks. URL: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks (дата обращения 22.05.2023).