DOI: 10.15514/ISPRAS-2022-34(4)-4

Автоматическое тестирование LLVM-программ со сложными входными структурами данных

[1]А.В. Мисонижник, ORCID: 0000-0002-5907-0324 <misonijnik@gmail.com>
[1]А.А. Бабушкин, ORCID: 0000-0002-5661-5800 <ocelaiwo@gmail.com>
[2]С.А. Морозов, ORCID: 0000-0003-1160-5614 <morozov.serg901@gmail.com>
[1]Ю.О. Костюков, ORCID: 0000-0003-4607-039X <kostyukov.yurii@gmail.com>
[1]Д.А. Мордвинов, ORCID: 0000-0002-6437-3020 <mordvinov.dmitry@gmail.com>
[1]Д.В. Кознов, ORCID: 0000-0003-2632-3193 <d.koznov@spbu.ru>

[1]Санкт-Петербургский государственный университет,
199034, Россия, Санкт-Петербург, Университетская наб., д. 7-9
[2]Национальный исследовательский университет "Высшая школа экономики",
190121, Россия, Санкт-Петербург, Союза Печатников ул., д.16

Аннотация. Символьное исполнение является известным подходом для автоматической генерации регрессионных тестов и поиска ошибок/уязвимостей в программах. Данная работа посвящена созданию практичного метода к символьному исполнению LLVM-программ, пригодного для работы со сложными входными структурами данных. Метод основан на известной идее ленивой инициализации, позволяющей избавить пользователя от необходимости вручную создавать ограничения на входные структуры данных и полностью автоматизировать процесс символьного исполнения программы. Предлагается два улучшения ленивой инициализации для сегментированной символьной моделей памяти – использование временных меток и информации о типах. Предложенный метод реализован в символьной виртуальной машине KLEE для платформы LLVM и протестирован на реальных С-структурах данных — списках, биномиальных кучах, AVL-деревьях, красно-чёрных деревьях, двоичных деревьях и борах (префиксных деревьях).

Ключевые слова: автоматическое тестирование; символьное исполнение; ленивая инициализация; платформа LLVM; KLEE; структуры данных

Для цитирования: Мисонижник А.В., Бабушкин А.А., Морозов С.А., Костюков, Д.А. Мордвинов Ю.О., Кознов Д.В. Автоматическое тестирование LLVM-программ со сложными входными структурами данных. Труды ИСП РАН, том 34, вып. 4, 2022 г., стр. 49-62. DOI: 10.15514/ISPRAS-2022-34(4)-4

Благодарности: Данное исследование было поддержано грантом РНФ № 22-21-00697.

Automated testing of LLVM programs with complex input data structures

Automated testing of LLVM programs with complex input data structures

[1]A.V. Misonizhnik, ORCID: 0000-0002-5907-0324 <misonijnik@gmail.com>
[1]A.A. Babushkin, ORCID: 0000-0002-5661-5800 <ocelaiwo@gmail.com>
[2]S.A. Morozov, ORCID: 0000-0003-1160-5614 <morozov.serg901@gmail.com>
[1]Yu.O. Kostyukov, ORCID: 0000-0003-4607-039X <kostyukov.yurii@gmail.com>
[1]D.A. Mordvinov, ORCID: 0000-0002-6437-3020 <mordvinov.dmitry@gmail.com>
D.V. Koznov, ORCID: 0000-0003-2632-3193 <d.koznov@spbu.ru>

[1]Saint Petersburg State University,
7/9 Universitetskaya Emb., Saint Petersburg, 199034, Russia
[2]HSE University,
16 Soyuza Pechatnikov Street, St Petersburg, 190121. Russia

Abstract. Symbolic execution is a widely used approach for automatic regression test generation and bug and vulnerability finding. The main goal of this paper is to present a practical symbolic execution-based approach for LLVM programs with complex input data structures. The approach is based on the well-known idea of lazy initialization, which frees the user from providing constraints on input data structures manually. Thus, it provides us with a fully automatic symbolic execution of even complex program. Two lazy initialization improvements are proposed for segmented memory models: one based on timestamps and one based on type information. The approach is implemented in the KLEE symbolic virtual machine for the LLVM platform and tested on real C data structures — lists, binomial heaps, AVL trees, red-black trees, binary trees, and tries.

Keywords: automated testing; symbolic execution; lazy initialization; LLVM platform; KLEE; data structures

For citation: Misonizhnik A.V., Babushkin A.A., Morozov S.A., Kostyukov Yu.O., Mordvinov D.A., Koznov D.V. Automated testing of LLVM programs with complex input data structures. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 4, 2022. pp. 49-62 (in Russian). DOI: 10.15514/ISPRAS-2022-34(4)-4

Acknowledgements. The work is supported by the grant of RNF № 22-21-00697.

1. Введение

Регрессионное тестирование является важным инструментом повышения надёжности программного обеспечения. Регрессионные тесты фиксируют поведение программных компонент, позволяя обнаруживать ошибки при внесении изменений в исходный код этих компонент. Однако при большом количестве различных сценариев поведения тестируемого кода написание регрессионных тестов является трудоёмкой задачей, что часто ведёт к упущенным тестовым сценариям.

Естественной идеей решения этой проблемы служит автоматизированная генерация регрессионных тестов [1]. Одним из эффективных способов достижения этой цели служит *символьное исполнение* [2], [3], которое исследует различные ветви поведения программы, используя инструменты проверки выполнимости логических формул (SMT-решатели [4]) для автоматического вывода входных данных, приводящих исполнение программы в эти ветви.

Реальные программы часто содерат *сложные структуры данных*, такие как списки и разные виды деревьев. Далее в этой статье под сложными структурами мы будем иметь в виду структуры, содержащие указатели на другие структуры или на себя. Они широко используются в реальном коде, а их автоматическое тестирование представляет особую сложность. Например, списки используются более чем в 10000 различных точках кода ядра Linux версии 5.6 [5]. Красно-чёрные деревья используются в планировщиках ядра Linux, в драйверах CD/DVD и файловой системе ext3 [6]. Боры (префиксные деревья) используются

для маршрутизации в оборудовании, например, в программном обеспечении компании CISCO [7].

Однако обработка структур данных с указателями является одной из известных проблем символьного исполнения: большинство современных SMT-решателей оказываются неэффективными для работы с динамической памятью и недетерминированными указателями, при помощи которых реализуются сложные структуры данных [8]. Одним из способов решения этой проблемы является использование «ручной» инициализации фрагментов структур данных для конкретизации тестируемого состояния программы [9], что оказывается трудоёмким и неудобным.

Перспективным методом решения этой проблемы является механизм *ленивой инициализации* [10], который наиболее эффективен для *сегментированной* модели символьной памяти [11], в рамках которой вся динамическая память программы разделяется на непересекающиеся блоки. Механизм ленивой инициализации работает следующим образом. Пусть недетерминированный указатель p ссылается на блок памяти размера k, и в динамической памяти уже выделено n объектов размера как минимум k с адресами $a_1, a_2, \ldots, a_n$. При разыменовании указателя p символьное исполнение с ленивой инициализацией рассматривает $n+1$ сценарий поведения программы: n сценариев поведения, где $p = a_i$, плюс ещё один сценарий, где p ссылается на новый блок памяти, чья инициализация отложена (проводится *лениво*). Ленивая инициализация позволяет автоматически инициализировать те фрагменты сложных структур данных, которые в действительности читаются в рассматриваемом пути исполнения программы, и полностью автоматически создать входные экземпляры для сколь угодно сложных структур данных. В итоге все порождённые сценарии дадут соответствующие тесты, однако многие из них не будут воспроизводимы, например, из-за того, что символьный указатель не может указывать на фрагмент памяти, выделенный после его инициализации, или на фрагмент памяти, помеченный другим типом. Наш метод направлен на решение этих проблем.

В данной работе мы разработали улучшенную концепцию ленивой инициализации, добавив в неё *временные метки* и *типы* (разд. 0). Эта концепция может быть использована в различных символьных виртуальных машинах, которые используют сегментированную символьную модель памяти. Далее, мы взяли символьную виртуальную машину KLEE [12], входящую в состав проекта LLVM, поскольку KLEE является эффективным инструментом генерации тестовых данных для программ с примитивными параметрами функций и широко используется в исследовательском и индустриальном сообществах. Мы выполнили собственную реализацию классической ленивой инициализации для KLEE, поскольку нам не удалось найти соответствующей открытой реализации, и улучшили её, добавив временные метки и типы (разд. 4). Эффективность концепции временных меток была исследована в ходе экспериментального исследования, выполненного на наборе реализаций различных структур данных на языке C (разд. 5). Мы также выполнили обзор близких к нашему исследованию работ (разд. 6).

Таким образом, основными результатами данной статьи является следующее.

1) Улучшенный метод ленивой инициализации путём добавления временных меток и типов.
2) Реализация механизма ленивой инициализации с метками времени и типами в символьной виртуальной машине KLEE.
3) Новый формат представления тестов в KLEE, позволяющий воспроизводить сценарии исполнения, в которых была применена ленивая инициализация.
4) Выполненные эксперименты по автоматическому тестированию сложных структур данных, реализованных на языке C — списков, биномиальных куч, AVL-деревьев, красно-чёрных деревьев, двоичных деревьев, боров (префиксных деревьев).

2. Символьное исполнение с ленивой инициализацией

При классическом символьном исполнении функций с простыми входными параметрами, такими как целые числа, в символьной памяти создаются простые символьные значения, и далее при работе с ними строятся символьные термы. Классическое символьное исполнение не масштабируется на функции со сложными входными параметрами (указатели на сложные структуры данных, такие как списки и деревья), т.к. параметр может указывать на любой

```
1. typedef struct List {
2.   struct List *next;
3.   int value;
4. } List;
5.
6. unsigned length(List *list, unsigned bound) {
7.   unsigned len = 0;
8.   for (List *p = list; p && bound; len++, bound--)
9.     p = p->next;
10.  return len;
11.}
12.
13.#define BOUND 2
14.#define SIZE 1
15.
16.int main() {
17.  int *array = make_concrete_array(10, sizeof(int));
18.  List *xs = make_symbolic(List);
19.  List *sing = make_concrete_list(SIZE);
20.
21.  unsigned length1 = length(xs, BOUND);
22.
23.  assert(length1 = 0 || (xs != sing && xs != array));
24.}
```

фрагмент в символьной памяти, который также может содержать символьные указатели. Ленивая инициализация [10] является методом, который решает эту проблему путём учёта (нумерации) возможных адресов в памяти, на которые может указывать символьный указатель.

Листинг 1. Фрагмент кода C-программы с функцией `length`*, имеющей сложный входной параметр (указатель на список)*

Listing 1. Code fragment in C representing `length` *function with a complex input parameter (a list pointer)*

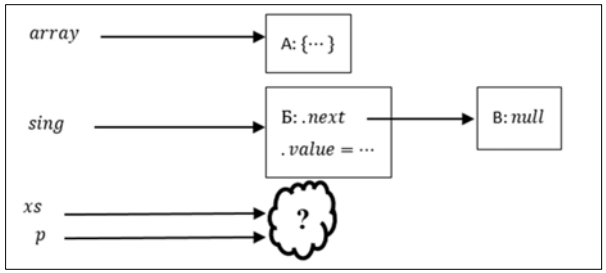

Рис. 1. Состояние символьной памяти после символьного исполнения строк 17-19 листинга 1
Fig. 1. Symbolic state after symbolic execution of 17-19 lines of code from listing 1.

Рассмотрим, как работает символьное исполнение с ленивой инициализацией на примере кода с листинга 1. Функция `main()` создаёт и инициализирует конкретный массив `array` при помощи функции `make_concrete_array`. Затем в строке 18 переменная `xs` инициализируется как символьный указатель на список, т.е. указатель на некоторую область в памяти, чьё значение нельзя определить статически – `xs` может указывать как на один из существующих выделенных фрагментов в памяти, так и некоторый новый. Далее создаётся и инициализируется список `sing` длины `SIZE`. Далее вычисляется ограниченная длина списка `xs` путём вызова функции `length`, которая проходит по ячейкам списка не более чем `BOUND` раз.

В процессе символьного исполнения строк 17-19 будет построено состояние символьной памяти, изображённое на рис. 1. Переменные `array` и `sing` указывают на конкретные фрагменты в памяти (выделенные прямоугольниками с метками «А», «Б» и «В»), а `xs` указывает на *лениво инициализируемую память* (представлена на рис. 1 в виде облака) – фрагмент памяти, который будет определён только при доступе.

Таким образом, в момент вызова функции `length` в строке 21 указатель `xs` оказывается недетерминированным. В строке 8 происходит разветвление процесса символьного исполнения по условию $p = 0$ $(null)$ для указателя p, на первой итерации равного xs (что также отражено на рисунке 1), как и при обычном символьном исполнении. Войдя в состояние с $p = xs = 0$, процесс символьного исполнения перейдёт к концу функции, вернёт значение 0 и тем самым отсечёт состояния, где `xs` указывает на блок памяти «В». Во втором состоянии ($xs \neq 0$) процесс символьного исполнения войдёт внутрь цикла, где будет выполнено разыменование лениво инициализируемой памяти.

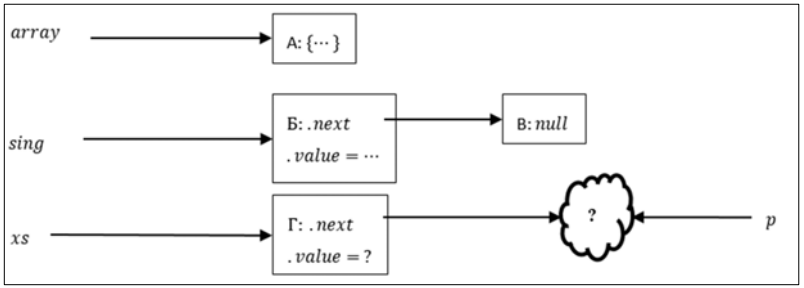

Рис. 2. Одно из состояний символьной памяти после символьного исполнения 17-21 и 7-9 строк кода с листинга
Fig. 2. One of symbolic states after symbolic execution of 17-19 and 7-9 lines of code from listing 1

Механизм ленивой инициализации работает следующим образом. При разыменовании указателя (`p->next`) в строке 9 будут созданы три символьных состояния, отличающиеся только тем, на что указывает `xs` – на массив (блок «А» на рис. 1), на существующий список (блок «Б» на рисунке 1) или на некоторый иной фрагмент памяти. Последнее символьное состояние программы представлено на рис. 2. В нём механизм ленивой инициализации создал новую структуру требуемого типа `List` и инициализировал все её поля символьными значениями примитивного типа (например, поле `value` в блоке «Г»), а все поля с типом указателя – новыми ленивыми значениями (поле `next`). Переменная `xs` по-прежнему указывает на начало списка, а переменная `p`, после выполнения инструкции в строке 9, будет указывать на `xs->next`. На следующей итерации цикла всё произойдёт аналогичным образом: указатель $p = xs \rightarrow next$ будет проверен на равенство нулю, а затем при

разыменовании `p` механизм ленивой инициализации породит четыре символьных состояния, отличающиеся тем, на что указывает `p` (`xs->next`): на массив (блок «А» на рисунке 2), на существующий список (блок «Б» на рис. 2), на порождённую предыдущей итерацией структуру (блок «Г» на рис. 2) или на некоторый иной фрагмент памяти.

Далее, если в символьной памяти находятся n объектов, то при разыменовании символьного указателя p будет создано $n + 1$ символьное состояние: в n состояниях p будет указывать на существующие объекты, а в последнем – на новый объект в памяти, все поля которого лениво инициализированы. Таким образом, данный метод расширяет символьное исполнение возможностью *систематически* анализировать все возможные состояния памяти программы с недетерминированным указателем.

3. Ленивая инициализация с временными метками и типами

Одной из проблем ленивой инициализации является комбинаторный взрыв числа путей символьного исполнения. После каждого разыменования недетерминированного указателя число исследуемых сценариев поведения программы увеличивается на число уже выделенных в динамической памяти объектов подходящего размера.

Снова вернёмся к листингу 1. В результате вызова функции `make_concrete_list` (строка 19) в символьной памяти выделится $SIZE$ объектов. Во время символьного исполнения функции `length` происходит разыменование символьного указателя `p` и при этом создаётся, как минимум, $SIZE + 1$ новых символьных состояний: перебираются объекты, созданные в функциях `make_concrete_array` и `make_concrete_list`. Полученные значения разыменовываются в следующей итерации цикла. Для сценария, в котором произошла ленивая инициализация объекта (см. рис. 2), разыменование недетерминированного указателя приведёт к увеличению числа рассматриваемых сценариев, теперь уже на $SIZE + 2$. Аналогичные действия произойдут в каждой следующей итерации цикла. Число итераций цикла равно $BOUND$, поэтому после исполнения в функции `main` вызова функции `length` число исследуемых сценариев исполнения увеличится на $O(SIZE \times BOUND)$. Таким образом, каждый последующий вызов функции `length` увеличивает количество исследуемых сценариев программы экспоненциально. Чтобы уменьшить число возможных сценариев исполнения, мы предлагаем дополнить модель символьной памяти *временными метками* и *информацией о типах объектов* в памяти. Оба этих улучшения позволяют отсечь некорректные сценарии поведения программы, которые в итоге дают невоспроизводимые тесты.

3.1 Временные метки

Объекты в символьной памяти выделяются в порядке исполнения инструкций исследуемого кода. Этот порядок гарантирует, что объекты, которые выделены раньше, не могут хранить в своей памяти адреса объектов, выделенных позже. В частности, недетерминированный указатель `xs` не может указывать на объект, который создан функцией `make_concrete_list` (например, отмеченный меткой «Б» на рисунке 1). Более того, каждый указатель, полученный из лениво инициализированных объектов в ходе исполнения функции `length` (например, `xs->next`), также не может указывать на конкретный объект, выделенный позже (например, отмеченный меткой «Б» на рисунке 2). Чтобы запретить такие разыменования, предлагается связывать каждый объект в символьной памяти с *временной меткой*, хранящей время его выделения. Метки упорядочены по возрастанию – чем позже создан объект, тем больше его временная метка.

Недетерминированные указатели имеют временные метки, равные временным меткам объектов, из которых значения этих указателей были прочитаны. Временные метки лениво инициализированных объектов равны временным меткам указателей, при разыменовании

которых они были инициализированы. Чтобы запретить разыменования указателей на объекты, созданные позже, при разыменовании недетерминированного указателя с меткой n во время перебора объектов памяти нужно отбрасывать те объекты, временные метки которых больше, чем n. Такой запрет отсекает некорректные сценарии поведения программы, тем самым ускоряя процесс символьного исполнения программы.

3.2 Информация о типах объектов

Спецификации некоторых языков могут гарантировать корректное поведение программ, в которых происходит разыменование указателей, только в том случае, если тип указателя и тип объекта в памяти совместимы друг с другом. Понятие совместимости в этом случае определяется в спецификации языка. Например, для языка C совместимость типов определяется правилами разыменования указателей, которые описаны в стандарте **Ошибка! Источник ссылки не найден.**. Таким образом, нарушение правил разыменования указателей в коде может привести к некорректному поведению программы. Если символьное исполнение будет учитывать эти правила, то это позволит отсекать некорректные сценарии исполнения программы и тем самым ослабить эффект экспоненциального взрыва числа исследуемых сценариев исполнения.

Поэтому кроме временных меток предлагается хранить типы объектов в памяти и указателей. При разыменовании недетерминированный указатель, во-первых, должен быть сопоставлен только с объектами подходящего типа, и, во-вторых, при ленивой инициализации нового объекта по этому указателю такому объекту должен быть присвоен тот же тип.

Вернемся к листингу 1. В строке 9 процесс символьного исполнения выполняет попытку разыменования недетерминированного указателя `xs` с типом `List`. При этом механизм ленивой инициализации должен рассмотреть все объекты с типом `List` и только их. Таким образом, во-первых, будет отсечен сценарий исполнения, в котором указатель сопоставляется объекту, созданному функцией `make_concrete_array` (например, отмеченный меткой «А» на рис. 1). Во-вторых, при ленивой инициализации нового объекта по указателю `xs`, полученному объекту (отмечен меткой «Г» на рисунке 2) будет присвоен тип `List`. Таким образом, на следующей итерации цикла при переборе объектов с этим типом полученный новый объект будет также учтен, поскольку он имеет подходящий тип. Заметим, что при одновременном использовании временных меток и типов проверка в строке 23 будет выполняться всегда.

Механизм ленивой инициализации и описанные в данном разделе улучшения могут быть реализованы в любой символьной виртуальной машине, основанной на *сегментированной символьной модели памяти* [11]. Так как в этой модели вся память разбита на непересекающиеся сегменты, соответствующие выделенным в памяти объектам, каждый сегмент памяти можно связать с временной меткой и типом, которые соответствуют этому сегменту. В следующем разделе будет описана реализация механизма ленивой инициализации с улучшениями в символьной виртуальной машине KLEE (основана на сегментированной символьной модели памяти).

4. Поддержка ленивой инициализации в KLEE

Мы использовали виртуальную символьную машину KLEE версии 2.3 [14], реализовав в ней механизм ленивой инициализации и оптимизации процесса работы с символьной памятью – временные метки и типы. Кроме того, были внесены модификации в процедуры генерации и воспроизведения тестов. Эти модификации описаны ниже.

Воспроизведение сгенерированных тестов в исходной версии KLEE реализовано следующим образом. Во время генерации теста KLEE находит подходящие значения для переменных, которые были объявлены символьными с помощью исполнения функции

`klee_make_symbolic`. Результатом генерации теста является файл *.`ktest`. В него записываются значения символьных переменных в том порядке, в котором произошли вызовы функции `klee_make_symbolic`. При воспроизведении теста вызовы функции `klee_make_symbolic` в исходной программе используются для записи подобранных значений в соответствующие переменные.

Однако тесты с символьными указателями не могут быть воспроизведены в оригинальной версии KLEE, поскольку во время генерации теста в символьные указатели записываются конкретные адреса, выделенные для указываемых объектов во время работы символьного исполнения. Эти адреса не соответствуют адресам объектов, выделяемых во время воспроизведения теста, так как эти адреса неизвестны на этапе генерации теста. В связи с этим мы изменили в KLEE процедуры генерации и воспроизведения тестов.

4.1 Модификация процедуры генерации тестов

В символьное состояние добавлено множество указателей: выражений, через которые происходит доступ к объектам в памяти. Эта информация используется во время генерации теста для того, чтобы определить, какие части символьных объектов представляют собой указатели, и на какие объекты эти указатели указывают. Информация об указателях впоследствии записывается в сгенерированный тест.

4.2 Модификация процедуры воспроизведения тестов

Во время воспроизведения теста информация об указателях используется для того, чтобы проинициализировать все необходимые объекты и их поля. Это происходит следующим образом. Все символьные объекты памяти инициализируются последовательно. Во время инициализации каждого объекта обрабатывается информация о его указателях. Все объекты, которые указывают на текущий объект и еще не инициализированы, инициализируются рекурсивно. У каждого инициализированного объекта запоминается его реальный адрес. После инициализации объекта его адрес записывается в соответствующее поле указывающего на него объекта. Таким образом все поля с типом указателя действительно указывают на нужные объекты в памяти.

Описанная реализация доступна по ссылке[1].

5. Эксперименты

Цель экспериментального исследования заключалась в проверке эффективности предложенного в статье метода. С этой целью сравнивались следующие инструменты: базовая версия KLEE 2.3, в которой ленивая инициализация отсутствует (KLEE-BASIC); версия KLEE 2.3 с реализованной нами классической ленивой инициализацией (CLI); версия KLEE 2.3 с классической ленивой инициализацией (KLEE-LI), а также предложенными в данной статье временными метками и типами (KLEE-LI-OPT). В качестве SMT-решателя был использован Z3 версии 4.9 [15].

Для экспериментов был разработан набор из 42-х тестовых C-программ. Эти программы инструментированы вызовом функции `klee_make_symbolic`, которая делает символьным объект, адрес которого передаётся в качестве аргумента. 27 программ содержат операции с указателями на следующие рекурсивные структуры данных: связный список, биномиальная куча, деревья (красно-чёрное, AVL, двоичное) и бор. В данных программах указатели на эти структуры делаются символьными. Остальные 15 программ содержат объекты примитивных структур данных и программы с явным выделением данных на куче посредством функции `malloc`. В них символьными делаются не указатели, а сами объекты в памяти. Это

[1] https://github.com/misonijnik/klee/tree/klee-2.3-li-opt

программы, на исполнение которых рассчитана обычная версия KLEE и которые требуются, чтобы показать, что наши оптимизации не ухудшают работу базового инструмента.

Для сравнения были выбраны следующие метрики: время работы инструмента на тестовой программе (в секундах), количество используемой при этом оперативной памяти (в мегабайтах) и процент покрытых инструментом инструкций кода тестовой программы. Процент покрытых инструкций измерялся с помощью инструмента gcov [16].

Эксперименты проводились на рабочей станции с процессором Intel Core I5-8265U с 16 ГБ оперативной памяти под управлением операционной системы Linux. Процесс символьного исполнения может не завершаться при исследовании рекурсивных и циклических программ, поэтому временные ограничения необходимы, чтобы отсечь зависания. Каждому запуску было задано ограничение 60 секунд. Для программ из набора выбранное время подобрано эмпирически: как правило, его хватает, чтобы исследовать большинство интересных сценариев исполнения. После истечения этого времени символьное исполнение следующей инструкции тестовой программы не может начаться, однако исполнение текущей инструкции обязано завершиться. Также часть времени после завершения исполнения тратится непосредственно на генерацию тестов. Из-за этого реальное время исполнения может быть больше заданного ограничения в 60 секунд. Во время символьного исполнения SMT-решатель может зависнуть, пытаясь выполнить запрос, поэтому было необходимо также ограничить его время работы. Время исполнения одного запроса SMT-решателем было ограничено 5 секундами. Это ограничение также было подобрано эмпирически для программ из предложенного набора: в нашем случае большинство завершающихся запросов завершают работу за указанное время.

Сводные результаты экспериментов представлены в табл. 1, детальное описание поведения каждого инструмента на каждой тестовой программе в соответствии с выбранными метриками представлено табл. 2 в Приложении.

Табл. 1. Сводные результаты экспериментов
Table 1. Summary results of experiment

Инструмент	Время работы, с.			Оперативная память, МБ			Среднее покрытие кода, %
	Мин.	Макс.	Средн.	Мин.	Макс.	Средн.	
KLEE-BASIC	0.02	98.8	42.3	17.3	106.5	41	48.3
KLEE-LI	0.02	130.7	55.7	17.3	124.3	35.1	81.9
KLEE-LI-OPT	0.02	128.1	43.2	17.3	120.1	28.4	86.9

Опишем результаты экспериментов.

Как следует из табл. 1, среднее время работы KLEE-LI-OPT (43.2 с) сопоставимо с KLEE-BASIC (42.3 с), при этом среднее время работы KLEE-LI оказывается больше, чем KLEE-LI-OPT — следовательно, предложенные оптимизации ленивой инициализации приводят к ускорению последней. Однако верхняя граница времени работы у KLEE с ленивой инициализацией больше, что связано с появлением новых исследуемых сценариев исполнения. Потребление оперативной памяти с применением ленивой оптимизации уменьшается, что связано с тем, что версия KLEE-BASIC тратит существенные ресурсы на исследование сценариев исполнения, в которых указатели разыменовываются в глобальные объекты памяти. Такие объекты имеют большой размер и занимают много места. При каждом разыменовании указателя для записи в память KLEE-BASIC копирует объект, на который указатель был разыменован. В дальнейшем KLEE-BASIC тратит значительное время на исследование таких сценариев исполнения. Из-за этого копирование глобальных объектов происходит чаще, чем при ленивой инициализации. Вместо этого KLEE с ленивой инициализацией большую часть времени исследуют сценарии исполнения с лениво инициализированными объектами, размер которых часто меньше, чем размер глобальных объектов. Более того, наши оптимизации ещё больше уменьшают используемую оперативную память. Однако на ряде тестов KLEE-LI-OPT показывает потребление памяти,

превосходящее KLEE-BASIC, поскольку для символьного исполнения с оптимизациями необходимо хранить в памяти дополнительную информацию о временных метках и типах. Из-за этого для простых программ, на которых KLEE-LI-OPT не даёт выигрыша во времени исполнения и покрытии кода, эта версия использует больше памяти, чем версия KLEE-BASIC. Наконец, KLEE-LI-OPT почти в два раза увеличивает покрытие кода в тестовых С-программах по сравнению с KLEE-BASIC: ленивая инициализация позволяет исследовать недостижимые для KLEE-BASIC сценарии исполнения программы, а оптимизации позволяют отсекать некорректные сценарии исполнения, снижая влияние проблемы экспоненциального взрыва числа исследуемых сценариев.

Таким образом, KLEE-BASIC не может эффективно анализировать программы, в которых происходит разыменование недетерминированных указателей на рекурсивные структуры данных, что выражается в низком проценте кодового покрытия на тестовых программах со связными списками, биномиальными кучами, деревьями и борами. Представленные оптимизации в версии KLEE-LI-OPT, в свою очередь, позволяют отсеять заведомо невоспроизводимые варианты исполнения программы, тем самым уменьшить время работы инструмента и увеличить процент тестового покрытия.

6. Близкие работы

6.1 Символьное исполнение с ленивой инициализацией

Идея ленивой инициализации в символьном исполнении была предложена в 2003 году [10] и с тех пор была реализована в различных символьных виртуальных машинах. К примеру, работа [17] описывает метод ограниченной ленивой инициализации, реализованный в инструменте SPF [18] для языка Java. В отличие от классической версии, количество лениво инициализированных объектов одного типа здесь зафиксировано заранее. Во время разыменования символьного указателя выполняется перебор заранее фиксированных вариантов, т.н. «плотных границ» (tight bounds), полученных с помощью «плотного анализа полей» (tight field analysis). «Плотные границы» хранятся как некоторое отношение на объектах, которые возможно лениво инициализировать, и результат разыменования указателя на объект должен удовлетворять этому отношению. В статье также представлен механизм, позволяющий избежать порождения изоморфных структур. Метод с ограниченной ленивой инициализацией улучшен в [19]. Однако в этих работах не представлены методы порождения исполняемых тестов на базе инстанцированных структур данных.

Идея ленивой инициализации использовалась при создании логики HEX [20] – языка спецификации пред- и постусловий для Java, удобного для работы с динамической памятью. Пользовательские предусловия позволяют уменьшить пространство перебора объектов при ленивой инициализации. Однако необходимость вручную описывать инварианты структур данных является существенным ограничением этого метода.

6.2 Тестирование структур данных в KLEE

Символьная виртуальная машина KLEE [12] является очень популярной в академии и индустрии, и уже были сделаны попытки поддержать в ней обработку сложных структуры данных. Самым близким к нам является инструмент UC-KLEE [21], который реализует фрагмент механизма ленивой инициализации, позволяющий символьное исполнение кода с входными структурами данных без алиасинга (т.е. не содержащим двух ссылок на один и тот же участок памяти). К сожалению, нам не удалось найти исходный код или публично доступную рабочую версию UC-KLEE.

Другая недавняя работа [22] фокусируется на множестве стратегий смягчения комбинаторного взрыва путей исполнения и также реализует ограниченный вариант ленивой инициализации, напоминая UC-KLEE. Однако наш метод, в отличие от UC-KLEE, может

быть применён для тестирования циклических структур данных и структур с алиасингом (например, графов).

7. Заключение

Мы представили нашу реализацию инструмента автоматического символьного исполнения LLVM-программ со сложными входными структурами данных и продемонстрировали его эффективность для реальных структур данных на языке C. Также был разработан новый формат представления KLEE-тестов с инициализацией структур данных, при этом была переработана вся инфраструктура KLEE для поддержки этого нового формата. Это позволило значительно улучшить качество кодового покрытия программ с указателями на рекурсивные структуры данных по сравнению с оригинальной версией KLEE.

В дальнейшем наша реализация может использоваться как по прямому назначению (генерация тестовых данных), так и для других академических экспериментов (вычисление слабейших предусловий, обратное символьное исполнение, вывод инвариантов циклов и т.д.).

Список литературы / References

[1] Korel B., Al-Yami A.M. Automated Regression Test Generation. ACM SIGSOFT Software Engineering Notes, vol. 23, issue 2, 1998, pp. 143-152.
[2] Cadar C., Sen K. Symbolic execution for software testing: three decades later. Communications of the ACM, vol. 56, issue 2, 2013, pp. 82-90.
[3] Baldoni R., Coppa E. et al. A survey of symbolic execution techniques. ACM Computing Surveys (CSUR), vol. 51, issue 3, 2018, pp. 1-39.
[4] Barrett C., Tinelli C. Satisfiability Modulo Theories. In Handbook of Model Checking, Springer, 2018, pp. 305-343.
[5] Volanschi N., Lawall J. The impact of generic data structures: decoding the role of lists in the linux kernel. In Proc. of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2020, pp. 103-114.
[6] Corbet J. Trees II: red-black trees. URL: https://lwn.net/Articles/184495/, 2006.
[7] Bacthu N., Banerjee A. et al. Dynamic and compressed trie for use in route lookup. United States Patent Application 20180212876, URL: https://www.freepatentsonline.com/20180212876.pdf, 2018.
[8] Braione P., Denaro G. et al. Combining symbolic execution and search-based testing for programs with complex heap inputs. In Proc. of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017), 2017, pp. 90-101.
[9] Galeotti J.P., Rosner N. et al. Analysis of invariants for efficient bounded verification. In Proc. of the 19th International Symposium on Software Testing and Analysis (ISSTA '10), 2010, pp. 25-36.
[10] Khurshid S., Puasuareanu C.S., Visser W. Generalized symbolic execution for model checking and testing. In Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2003, pp. 553-568.
[11] Kapus T., Cadar C.A segmented memory model for symbolic execution. In Proc. of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 774-784.
[12] Cadar C., Dunbar D., Engler D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation, 2008, pp. 209-224.
[13] ISO/IEC 9899:201x - N1570 - Programming languages – C. URL: https://web.cs.dal.ca/~vlado/pl/C_Standard_2011-n1570.pdf.
[14] Cadar C., Dunbar D. KLEE. URL: https://github.com/klee/klee/tree/v2.3, 2022.
[15] de Moura L., Bjorner N. Z3: An Efficient SMT Solver. Lecture Notes in Computer Science, vol. 4963, 2008, pp. 337-340.
[16] Gough B., Stallman R. M. An Introduction to GCC for the GNU Compilers gcc and g++. Network Theory Ltd, 2004, 144 p.

[17] Geldenhuys J., Aguirre N. et al. Bounded Lazy Initialization. Lecture Notes in Computer Science, vol. 7871, 2013, pp. 229-243.
[18] Puasuareanu C. S., Rungta N. Symbolic PathFinder: symbolic execution of Java bytecode. In Proc. of the IEEE/ACM International Conference on Automated Software Engineering, 2010, pp. 179-180.
[19] Rosner N., Geldenhuys J. et al. BLISS: improved symbolic execution by bounded lazy initialization with SAT support. IEEE Transactions on Software Engineering, vol. 41, issue 7, 2015, pp. 639-660.
[20] Braione P., Denaro G., Pezze M. Symbolic Execution of Programs with Heap Inputs. In Proc. of the 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 602-613.
[21] Ramos D. A., Engler D. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In Proc. of the 24th USENIX Security Symposium (USENIX Security 15), 2015, pp. 49-64.
[22] Rutledge R., Orso A. PG-KLEE: Trading Soundness for Coverage. In Proc. of the IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2020, pp. 65-68.

Информация об авторах / Information about authors

Александр Владимирович МИСОНИЖНИК. Получил степень магистра в области информационных технологий в Санкт-Петербургском государственном университете в 2021 году. Его исследовательские интересы включают методы эффективного поиска недостижимых состояний в символьном анализе программ.

Alexander Vladimirovich MISONIZHNIK. Received his master's degree in computer science at St. Petersburg State University in 2021. His research interests include efficient pruning of unreachable states in symbolic program analysis.

Алексей Александрович БАБУШКИН. Получил степень бакалавра в области математики и компьютерных наук в Санкт-Петербургском государственном университете в 2022 году. Его исследовательские интересы включают методы анализа программ и автоматической генерации тестов.

Alexey Alexandrovich BABUSHKIN. Received his bachelor's degree in mathematics and computer science at St. Petersburg State University in 2022. His research interests include program analysis and automatic test generation.

Сергей Антонович МОРОЗОВ. Студент 3 курса бакалавриата “Прикладная математика и информатика” в Национальном исследовательском университете “Высшая школа экономики” в Санкт-Петербурге. Его исследовательские интересы включают методы анализа программ и оптимизации символьного исполнения.

Sergey Antonovich MOROZOV. A 3rd-year undergraduate student in Applied Mathematics and Computer Science at the National Research University Higher School of Economics in St. Petersburg. His research interests include methods of program analysis and symbolic execution optimization.

Юрий Олегович КОСТЮКОВ, аспирант кафедры системного программирования Санкт-Петербургского государственного университета, получил степень магистра в области информационных технологий в Санкт-Петербургском государственном университете в 2021 г. Его исследовательские интересы включают проблему автоматического вывода индуктивных инвариантов и автоматическое порождение тестовых покрытий.

Yurii Olegovich KOSTYUKOV, PhD student of the Department of System Programming at St. Petersburg State University, received his master's degree in computer science at St. Petersburg State University in 2021. His research interests include automatic inductive invariant inference and automatic test coverage generation.

Дмитрий Александрович МОРДВИНОВ, кандидат физ.-мат. наук, доцент кафедры системного программирования Санкт-Петербургского государственного университета. Область его научных интересов включает формальную верификацию, синтез программ и решение систем дизъюнктов Хорна.

Dmitry Alexandrovich MORDVINOV, PhD in Physics and Mathematics, Associate Professor at the Department of System Programming of St. Petersburg State University. His research interests include formal verification, program synthesis, and constraint Horn clause solving.

Дмитрий Владимирович КОЗНОВ, доктор технических наук, профессор кафедры системного программирования. Научные интересы: программная инженерия, модельно-ориентированная разработка программного обеспечения, программные данные, машинное обучение.

Dmitry Vladimirovich KOZNOV, Doctor of Technical Sciences, Professor of the System Programming Department. Research interests: software engineering, model-driven software development, program data, machine learning.

Приложение

Табл. 2. Расширенное сравнение стандартной версии KLEE (KLEE-BASIC), KLEE с ленивой инициализацией (KLEE-LI) и KLEE с ленивой инициализацией и оптимизациями (KLEE-LI-OPT)
Table 2. Extended comparison of the standard version of KLEE (KLEE-BASIC), KLEE with lazy initialization (KLEE-LI), and KLEE with lazy initialization and optimizations (KLEE-LI-OPT)

Тестовые С-программы	**Время работы, с**			**Оперативная память, МБ**			**Процент покрытого кода,%**		
	KLEE-BASIC	KLEE-LI	KLEE-LI-OPT	KLEE-BASIC	KLEE-LI	KLEE-LI-OPT	KLEE-BASIC	KLEE-LI	KLEE-LI-OPT
abs.c	0.03	0.04	0.04	20	20	20	100	100	100
aggregate.c	0.02	0.02	0.02	17.3	17.3	17.4	100	100	100
array_equality.c	0.02	0.02	0.02	17.3	17.3	17.3	100	100	100
array_sum.c	0.02	0.03	0.02	17.4	17.4	17.4	100	100	100
arraylist.c	3	0	0.05	20.5	22.3	17.5	0	88	88
avl_balance.c	62.4	196	77.7	42	31	32.31	0	29	29
avl_find.c	72	222	90	44.6	36.5	33.4	66.7	83	83
avl_height.c	68.8	182	78.5	47.9	29.4	26.3	0	76.9	76
avl_insert.c	61	258	98	32	40	40	60.9	69.6	69
binomial_heap.c	69.7	79.3	61.5	43.4	38	30	0	91.2	95
boolean.c	0.04	0.06	0.04	17.3	17.4	17.4	100	100	100
get_sign.c	0.03	0.04	0.03	17.3	17.4	17.4	100	100	100
integer_series.c	7.8	0.00	56.1	21.3	26.8	23.6	0	0	100
list_nonempty.c	60.3	63.9	124	60	45.9	30.3	0	100	100
list_pointer_access.c	65.7	98.3	0.8	61.5	45.04	18.4	0	100	100
list_size.c	61	127	60.3	60.3	43.5	28.9	77.8	88.9	100
list.c	69.4	61.5	70.2	53.4	41.3	30	100	100	100

malloc_big.c	0.02	0.02	0.02	17.3	17.3	17.3	100	100	100
malloc.c	0.02	0.02	0.02	17.3	17.3	17.3	100	100	100
manyvar.c	0.05	0.06	0.05	17.4	17.4	17.4	100	100	100
narrow.c	0.02	0.02	0.02	20	20	20	57.1	57.1	57
queue_head_peek_pop.c	61.2	108	0.2	56	46	18	0	90.5	95
queue_peek_head_tail.c	59.9	101	0.3	59.54	47	21	0	78.6	100
queue_push_pop.c	73.2	67.1	0.02	57.6	40.3	17.5	0	94	93
rb_grandparent.c	61	157	78.5	30.8	32.7	33.5	0	87.5	87
rb_insert_find.c	68	222	92.3	42.47	35.6	34	0	39.8	39
rb_remove.c	98.8	210	86	57.6	38.3	33.5	0	9.7	9
recursive.c	61	60	61	106	124	120	100	100	100
regexp.c	11.2	60	13.7	42	66.3	66	100	100	100
simple.c	3	0	2.2	20.6	21.8	17.8	0	100	100
sort.c	0.02	0.03	0.02	17.5	17.6	17.6	87.5	87.5	87
structs.c	0.02	0.03	0.02	17.3	17.4	17.4	100	100	100
tree_bfs.c	62.1	77.5	123.3	60.4	43.3	29.3	80	100	100
tree_count_nodes.c	62.1	209	87.2	30.3	38	35.61	80	90	90
tree_delete_find.c	68.1	66.3	68.8	70	45	47	28.9	52.6	89
tree_find_height.c	71	76.4	128	58.4	32	30	63.6	91	90
tree_insert_find.c	63	60.7	61	59.7	40.5	30.3	60	92	100
tree_makeempty.c	63	134	98.5	54.8	40.8	29.3	66.7	100	100
trie_insert_remove.c	61	61	60.5	41.7	44.5	29.8	0	22.4	25
trie_lookup.c	78.5	61	64.5	57.7	42.3	29.2	0	81.2	93
trie_num_entries.c	67.2	69	0.07	58.5	39.4	17.5	0	100	100
trie_remove.c	81.5	68.3	68.6	58.1	45.4	31	0	38.2	44