электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

 # 10, октябрь 2015

УДК 004.432.2

Поиск неопределённого поведения в исходном коде на языке C с помощью символьного выполнения с использованием SMT решателя

Красиков А.С., *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Программное обеспечение ЭВМ и информационные технологии»

Научный руководитель: Ломовской И.В., старший преподаватель
«Программное обеспечение ЭВМ и информационные технологии»,
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
irudakov@bmstu.ru

Неопределённое поведение (англ. undefined behaviour) — свойство некоторых языков программирования в определённых ситуациях выдавать результат, зависящий от реализации компилятора и случайных параметров времени исполнения. Такое поведение может возникнуть при некорректном использовании некоторых программных инструкций, на которое в стандарте Programming languages — C не накладывает никаких ограничений [1].

Неопределённое поведение является следствием того, что в конструкции языка преднамеренно закладывают большую мощность — в целях оптимизации, например, для ускорения работы с памятью. Обеспечивается это в том числе и тем, что в стандарт языка вносятся некоторые исключения, ограничивающие каким-то образом применимость конструкций. Таким образом, задача поиска неопределённого поведения представляет собой поиск ограниченного числа исключений. Вся дальнейшая логика будет строиться с точки зрения допущения ошибки первого рода при доказательстве присутствия неопределённого поведения.

Как пример можно рассмотреть неопределённое поведение, возникающее в результате деления на 0. Если выделить в программном коде все операции деления и рассматривать их на предмет деления на 0, то с точностью можно сказать о делении на 0 только когда делитель – константа 0. С ошибкой первого рода деление на 0 может произойти при любой операции деления на не-константу, например в `c = a / d`. Такой критерий получается слишком

агрессивным и пользы от него не так много: большое количество лишних срабатываний, например в if (d != 0) c = a / d, будет лишь мешать нахождению реальных ошибок. Для сокращения количества подобных ложных срабатываний можно применить метод символьного выполнения: накапливать полезную для определения каждого неопределённого поведения информацию о переменной в ходе анализа исходного кода по графу потока управление. Например, для поиска использования указателя на невыделенную память будет полезна информация о том, передавался ли он в realloc первым аргументом, а для поиска переполнения в алгебраических целочисленных операциях будет полезна информация о равенстве переменной 1.

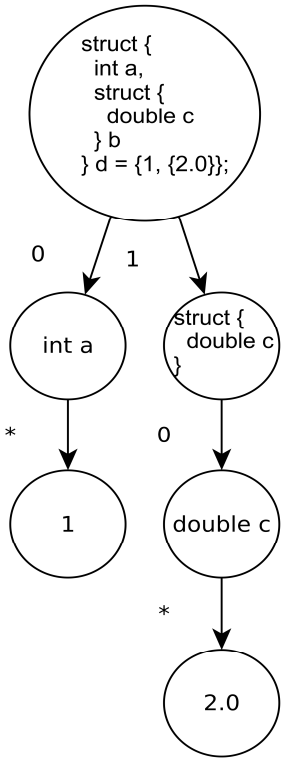
Граф зависимостей данных

Язык C даёт возможность ссылаться на один и тот же участок памяти несколькими переменными. В связи с этим задача отслеживания значений и ограничений на значения переменных превращается в задачу отслеживания значений и ограничений на участки памяти, на которые отображаются все операции с переменными. В более частом случае, для поддержания SSA [2] и из-за автоматизации генерации кода, содержимое одной и той же переменной может быть получено и установлено с помощью нескольких независимых, но взаимозаменяемых временных переменных. Например, для struct { int a; } c; c.a = 2; int b = c.a; компилятор clang сгенерирует инструкцию сохранения числа 2 в одну переменную, а для записи значения в b будет использовать уже значение другой временной переменной.

Для решения обозначенной проблемы введём граф зависимостей данных, на котором выделим 3 класса эквивалентности отношений:

- обращение к значению переменной;
- получение ссылки на под-элемент по статическому смещению;
- получение ссылки на под-элемент по переменному смещению.

Для структурных типов можно сделать отображение имён полей структур в номера, начиная с 0 для первого поля. Пример такого отношения с вложенными структурами изображён на рисунке.



Каждая вершина в графе соответствует множеству переменных во временном представлении. При введении новой переменной, если для неё уже существует вершина в графе зависимостей данных, появляется возможность использовать уже накопленную информацию о её значении. Таким образом, граф зависимостей данных можно использовать как оракул:

1. в него подаётся следующая инструкция внутреннего представления;
2. он для всех переменных инструкции создаёт новую вершину или обновляет старую в графе.

В итоге он может отвечать на вопросы о входящих в инструкцию переменных: является ли переменная новым узлом, какие у неё есть родительские узлы.

Использование SMT решателя

Для поиска различных неопределённых поведений могут требоваться различные виды знаний о значениях переменных. SMT решатель может использоваться для решения универсальных задач о значениях переменных с целью определения возможности какого-либо условия, например, равенства делителя 0, или переполнения в вычислениях.

Факты для SMT решателя наращиваются в ходе символьного выполнения, при этом SMT задача может решаться множество раз в ходе символьного выполнения. Для того, чтобы не решать для каждой инструкции программного кода большую SMT задачу, содержащую в себе весь анализируемый код, можно ввести простое отношение зависимости на значениях, имеющее структуру схожую с графом потока данных. Например для конструкции int b = q * f; int d = 5 / b; переменная d будет зависеть от значения b, которая в свою очередь будет зависеть от q и f, соответственно для проверки деления на 0 в int d = 5 / b; будут вычислены ещё раз все ограничения на q и f. Очевидно, что в такой структуре зависимостей могут быть циклы, например в int b = c; c = b. Для этого при объявлении SMT задачи можно выделить этап объявления всех переменных и последующий этап наложения ограничений, который можно реализовать как проход по структуре зависимостей с запоминанием, в каких вершинах уже были, и наложения имеющихся ограничений.

Операторы ветвления при символьном выполнении

Ветвление интересно как с точки зрения появления нескольких независимых путей анализа, так и с точки зрения появления дополнительных фактов. Очевидным решением является полное копирование текущего состояния и продолжение анализа по независимым путям.

Необходимо также отслеживать, по какому пути прошел анализатор. Обусловлено это тем, что в SSA-форме записи существует такой вид инструкций как phi-инструкция: результат такой инструкции зависит от потока управления, а точнее от того, какой блок при ветвлении исполнялся прямо перед этой инструкцией [3].

При учёте дополнительных фактов встаёт задача выбора из них однозначно верной информации. Если на текущем этапе имеем только один элементарный факт — учитываем его. Если же присутствует некоторое множество фактов, соединённых через логическую конъюнкцию, необходимо применить рекурсивный алгоритм учёта этих фактов в зависимости от выбранного пути выполнения. Данный алгоритм описан в листинге 1. Например, в выражении switch для пути default фактом является объединение фактов-неравенств по каждому из case-пути и все эти факты должны быть учтены в SMT задаче.

Листинг 1. Анализ фактов с учётом их верности

```
1   fun addFact(fact, truth, smt)
2    if fact is comparasion
3     smt.addFact(fact, truth)
4    elsif
5           fact is binaryOperation
6     and (
7              (fact is conjunction and truth is true)
8            or
9              (fact is disjunction and truth is false)
10    )
11    addFact(fact.first, truth, smt)
12    addFact(fact.second, truth, smt)
```

Анализ на деление на 0

Уменьшение количества ложных срабатываний при поиске деления на 0 крайне важно, так как операция деления достаточно часто встречается в программном коде, а критерий доказательства возможного деления на 0 достаточно грубый.

При поиске деления на 0 целевыми инструкциями для анализа являются бинарные операции, а именно деление. Для каждой бинарной операции деления решается SMT задача с добавлением факта равенства делителя 0, и, если у SMT задачи есть решение, значит в рассматриваемой операции возможно и деление на 0.

Анализ на использование неинициализированного значения

Целевыми инструкциями для анализа при поиске использования неинициализированных значений является обращение к содержимому переменной. Если к одной и той же переменной ведут несколько путей в графе зависимости данных, и хотя бы один из них не определён, то и сама переменная тоже не определена.

Отдельное внимание стоит уделить специальным функциям, как realloc и free. Тут стоит понимать, что никто не защищён от вызова free над передаваемым аргументов в другой функции. Для полного покрытия этого случая необходимо строить полный граф потока управления и проверять, может ли в ходе выполнения вызываться функция free для передаваемых указателей.

При разыменовании указателей и получении элементов структуры результат можно

считать неопределённым, если базовый указатель или структура не инициализированы. Если разыменование уже производилось ранее, тогда определённость текущего разыменования зависит от определённости предыдущего. Иначе же разыменование не определено.

При сохранении значений в переменные логика схожая:

- если хоть один родительский узел целевой переменной не определён, то целевая переменная всё так же не определена;
- иначе определённость переменной считается по сохраняемому значению.

Анализ на переполнение при целочисленных операциях

При поиске переполнения при целочисленных операциях основными для анализа являются инструкции бинарных операций: для каждой операции ищется решение SMT задачи с добавлением условия, что результат бинарной операции может не поместиться в соответствующий тип.

При допущении ошибки первого рода можно решать только прямую SMT задачу, т.е. для выражения `d = x + 100`, где `x` – ничем не ограниченный входной аргумент в функцию, SMT решатель скажет, что существует такое `x`, при котором возможно переполнение.

Использование результатов поиска неопределённого поведения

Не смотря на то, что при описанном подходе к поиску неопределённого поведение количество ложных срабатываний может быть слишком велико для непосредственного использования в качестве прямых указаний программисту при разработке, данный подход может быть использован для построения оптимизирующих компиляторов с учётом неопределённого поведения [4] или для поиска оптимизационно-нестабильного поведения [5], что, по сути, является частью автоматического тестирования.

Список литературы

1. Internation Standard Organization C Standard 1999 / ISO, ISO/IEC 9899:1999 draft. WG14 ed., 1999. Available at: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf, accessed 24.05.2015.
2. Cytron, Ferrante, Rosen, Wegman, Zadeck Efficiently Computing Static Single Assignment Form and the Control Dependence Graph // ACM Transactions on Programming Languages and Systems. 1991. № 13(4). P. 451-490.

3. Sreedhar V. C., Gao G. R. Computing φ-nodes in linear time using DJ graphs // Journal of Programming Languages. 1996. № 13(4). P. 191-213.
4. Wang X., Zeldovich N., Kaashoek M. F., Solar-Lezama A. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. MIT CSAIL, 2013. Available at: http://dl.acm.org/citation.cfm?doid=2517349.2522728, accessed 24.05.2015.
5. Красиков А.С. Обобщённый метод обнаружения оптимизационно-нестабильного поведения в коде на языке С // Молодежный научно-технический вестник. МГТУ им. Н.Э. Баумана. Электрон. журн. 2014. № 12. Режим доступа: http://sntbul.bmstu.ru/doc/746939.html (дата обращения 24.05.2015).