Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя

В.К. Кошелев <vedun@ispras.ru >
И.А. Дудина <eupharina@ispras.ru >
В.Н. Игнатьев <valery.ignatyev@ispras.ru>
А.И. Борзилов <helendile@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. В данной работе рассматривается построение масштабируемого чувствительного к путям анализа дефектов в программах на языке C#. Предложенный метод анализа является адаптацией набора подходов, используемых в инструментах для статического анализа C/C++-программ Saturn, Calysto и Svace. Особое внимание уделяется связи рассматриваемой модели с реальным выполнением программы. Производится формализация дефекта разыменования нулевого указателя и сведение задачи поиска данного дефекта к задаче выполнимости формул логики предикатов. Для проведения формализации вводится модельный язык, на который может быть оттранслирована любая программа на языке C#. Язык представляет собой подмножество языка C#, избавленное от синтаксического сахара. Для упрощения формализации из числовых типов рассматривается только один целочисленный тип. Для решения проблемы анализа циклов используется метод развертки графа потока управления. Предлагается модель абстрактного состояния программы, описывающая множество возможных состояний в данной точке программы. Абстрактное состояние задается состоянием памяти программы, описанным с помощью набора неизвестных входных переменных, и предикатом пути над тем же набором переменных. Для каждой инструкции модельного языка определяются формальная семантика и передаточная функция, отражающая изменение абстрактного состояния в соответствии с семантикой. Каждая функция считается точкой входа в программу, а межпроцедурный анализ основан на методе резюме. Резюме строятся по результатам внутрипроцедурного анализа функций. Поиск дефектов происходит при помощи добавления дополнительной информации к абстрактному состоянию. Решение о выдаче предупреждения принимается на основе выполнимости формулы, описывающей условие возникновения ошибки. Предложенный метод реализован в анализаторе SharpChecker, разработанном в ИСП РАН. Приведены результаты тестирования, подтверждающие применимость метода для анализа промышленных программ.

Ключевые слова: Статический анализ; разыменование нулевого указателя; чувствительность к путям; резюме функции; поиск дефектов.

DOI: 10.15514/ISPRAS-2015-27(5)-5

Для цитирования: Кошелев В.К., Дудина И.А., Игнатьев В.Н., Борзилов А.И. Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 59-86. DOI: 10.15514/ISPRAS-2015-27(5)-5.

1. Введение

Автоматический поиск дефектов в программах является одной из важнейших задач программной инженерии. В рамках данной работы рассматривается задача автоматического поиска дефектов в программах на языке программирования C#. Для поиска дефектов используется метод статического анализа исходного кода программ. К анализируемой программе предъявляется единственное требование: она должна успешно компилироваться. Для промышленного применения анализатора он должен поддерживать как полный анализ проекта за время, ограниченное временем ночной сборки, так и инкрементальный анализ небольших изменений, например на лету при разработке в интегрированной среде. Такой анализ дефектов должен за отведенное ему время находить как можно больше срабатываний, сохраняя при этом процент истинных срабатываний на приемлемом уровне. Учитывая имеющиеся ограничения на время работы анализатора, фактически время анализа одной функции не должно превышать нескольких десятков секунд.

В отличие от языков C/C++, язык C# не подвержен некоторым характерным типам уязвимостей, как переполнение буфера и уязвимость форматной строки, с помощью которых осуществляется большинство атак, приводящих к исполнению произвольного кода. Однако, большее внимание требуется уделять специфичным типам эксплуатируемых ошибок, таким как утечка ресурсов, которые могут привести, например, к отказу в обслуживании.

При анализе, дефект формулируется либо с помощью конструкции абстрактного синтаксического дерева, либо с помощью ошибочного пути выполнения программы. Между данными формулировками существует принципиальная разница: поиск конструкции абстрактного синтаксического дерева алгоритмически разрешим, а точный и полный анализ путей выполнения в общем случае неразрешим.

Промышленные анализаторы ищут оба вида дефектов, поэтому они применяют два типа анализа: один используется для поиска потенциально опасных конструкций, другой – для анализа путей выполнения. Задачей анализа второго типа является поиск таких дефектов, как разыменование невалидного указателя, переполнение буфера, утечка памяти, неверное использование примитивов синхронизации, применение невалидных дескрипторов и т.д. Данная работа посвящена поиску дефектов второго типа.

Для их поиска предлагается использовать внутрипроцедурный чувствительный к путям анализ. Нарушения контракта использования функций при внутрипроцедурном анализе функций будем моделировать с помощью резюме. Использование резюме является неизбежным компромиссом между производительностью и качеством анализа. С одной стороны, суммарное время, необходимое на применение резюме намного меньше времени повторного анализа в каждой точке вызова. С другой стороны, резюме отражает только некоторые эффекты вызова функции, поэтому такое моделирование может быть причиной как ложных срабатываний, так и пропуска ошибок. Кроме того, с помощью резюме можно просто реализовать инкрементальный анализ.

Идея реализации чувствительного к путям анализа заключается в построение для рассматриваемых путей формулы логики предикатов над значениями переменных, отражающей истинность условий переходов, характеризующие данные пути. В том случае, если на рассматриваемых путях произошла ошибка и построенная формула пути выполнима, то данный путь может быть представлен пользователю в качестве примера пути, на котором происходит ошибка. Выполнимость формулы пути проверяется с помощью SAT-решателей или SMT-решателей. Анализ путей выполнения осуществляется с использованием точек слияния, задающих состояние одновременно нескольких путей выполнения. Использование точек слияния позволяет избежать комбинаторного взрыва, возникающего при переборе всех возможных путей даже в случае отсутствия циклов.

В предлагаемом методе анализ функций производится в обратном топологическом порядке графа вызовов и сопровождается построением резюме для каждой проанализированной функции. Таким образом при анализе очередной функции для всех вызываемых ею функции их резюме уже построены. Рекурсия при данном подходе игнорируется за счет удаления обратных ребер графа вызова. При внутрипроцедурном анализе каждая функция считается точкой входа в независимую подпрограмму, поэтому её входные параметры могут принимать произвольные значения.

Так как в данном методе параметры функции являются неизвестными значениями, то в формуле пути они участвуют как свободные переменные. Значения этих переменных в случае наличия модели являются значениями входных переменных, при которых произойдет ошибка. Так как параметры функции могут использоваться для доступа к куче, то для проведения анализа необходимо некоторым образом задать состояние кучи в точке входа анализируемой функции. Для этого в данной работе рассматривается формализация, позволяющая сопоставить начальному состоянию кучи и входным параметрам набор независимых переменных, которые в дальнейшем используются при построении формул пути.

Для поиска ошибок в проверяемой точке программы рассматривается формула над значениями переменных, выполнимая в случае, если ошибка

возможна. Например, для разыменования нулевого указателя можно рассмотреть формулу выполнимую, если в данной переменной может быть записан null. Тогда для поиска разыменования нулевого указателя достаточно убедиться, что в точке разыменования переменной условие того, что ее значения равно null, и формула пути до текущей точки совместны. Тогда модель, на которой они совместны, можно предъявить пользователю в качестве примера входных данных, приводящих к разыменованию нулевого указателя.

Особенную сложность при анализе программ на языке C# представляет собой полная поддержка следующих конструкций: исключения, виртуальные вызовы, асинхронные вызовы, лямбда-выражения и некоторые другие особенности языка. Особенности поддержки данных конструкций в рамках предлагаемого метода планируется рассмотреть в дальнейших работах.

Данный метод был реализован в анализаторе CSCC, разработанном в ИСП РАН, и получил практическое подтверждение.

Текст работы организован следующим образом. Во второй части производится формализация понятий входных данных и пути выполнения. При помощи введенных понятий формулируется ситуация разыменования нулевого указателя. В третьей части приводится описание внутрипроцедурного анализа, позволяющего находить разыменования нулевого указателя. В четвертой части рассматривается применение резюме функции для реализации межпроцедурного анализа. Пятая часть посвящена особенностям реализации предложенного метода. В шестой части описываются результаты применения данного метода для анализа промышленных проектов. Седьмая часть посвящена обзору схожих решений. В заключительной части подводятся итоги данной работы.

2. Постановка задачи

В данной работе предлагается рассматривать программы на модельном языке вместо программ на языке C#. Такое решение мотивировано тем, что реализованный анализатор не имеет собственного внутреннего представления, которое можно было бы использовать в качестве модельного языка, используя вместо инструкций в графе потока управления вершины абстрактного синтаксического дерева, соответствующие операторам языка C#. Введение модельного языка позволяет избежать избыточности, вызванной наличием в языке C# синтаксического сахара, и исключить не полностью поддерживаемые анализатором конструкции языка. Будем считать, что программа, написанная на подмножестве C#, поддерживаемым анализатором, может быть однозначно оттранслирована в программу на модельном языке с сохранением семантики.

2.1 Модельный язык

В модельном языке присутствует только два типа: integer — 32–битное знаковое число и object — ссылка на объект, находящийся в динамической памяти. К переменным типа integer могут применяться арифметические и битовые операции. К переменным типа object применяются операции сравнения и взятия поля. Определение поля для конкретной ссылки на объект в памяти, как и в скриптовых языках, происходит в момент первого присваивания в него значения. В реальных программах на языке C# используются и другие целочисленные типы, однако для просты повествования ограничимся только типов integer.

Рассмотрим грамматику модельного языка, записанную в БНФ. Выражение < текст > обозначает строковый литерал, а символ <> означает отсутствие каких-либо символов. Сразу оговоримся, что, так как рекурсия исходным анализатором не поддерживается, в модельном языке она также запрещена.

- *identifier* ::= строка
- *const* ::= число |*null*
- *type* ::= *integer* | *object*
- *var* ::= *identifer*;
- *field* ::= *identifer*;
- *param* ::= type identifer;
- *assignConst* ::= *var* = *const*;
- *assign* ::= *var* = *var*;
- *setfield* ::= *var.field* = *var*;
- *getfield* ::= *var* = *var.field*;
- *binop* ::= *var* = *var* ◇ *var*;
- *unop* ::= *var* = □*var*;
- *if* ::=< *if* > (*var*){*statements*}|
  < *if* > (*var*){*statements*} < *else* > {*statements*}
- *loop* ::=< *do* > {*statements*} < *while* > (*var*);
- *control* ::=< *break* > | <*continue* > | < *return* > *var*;
- *statement* ::= *assign* | *assignConst* | *setfield* |
  getfield | *binop* | *unop* | *if* | *loop* | *control* | *functioncall*

- *statements* ::= *statement*|*statementstatements*| <>
- *paramlist* ::= *param, paramlist* | *param* | <>
- *arglist* ::= *var* | *var, arglist* | <>
- *functioncall* ::= *var* = *identifer*(*arglist*);
- *function* ::= *type identifier*(*paramlist*){*statements*}
- *program* ::= *function* | *function program*

Определение переменных в данном языке также происходит «лениво», в момент первого присваивания значения в данную переменную. Тип переменной выводится из типа правой части присваивания. Аналогично определяются поля объектов. В данном языке все функции возвращают значение. Если в исходной программе функция имела тип void, то будем считать, что она возвращает null. В модельном языке отсутствуют глобальные переменные, т.к. в программах на языке C# они, как правило, не используются. В операторах if и loop, в качестве операндов допускаются только переменная типа integer, переход по истинной ветке осуществляется в том случае, если значение переменной отлично от нуля. Из циклов в данном языке представлен только цикл *do...while*. Данное решение принято для того, чтобы при построении графа потока управления выход из цикла мог осуществляться без прохождения по обратному ребру.

Построим для каждой функции на модельном языке граф потока управления. Вершинами графа являются операторы, определяемые правилами *assign*, *assignConst*, *setfield*, *getfield*, *binop*, *unop* и *functioncall*. Далее будем называть данные операторы инструкциями. Для учета условий переходов добавим специальную операцию «*assume* ::=< *assume* > (*var*);», продолжающую выполнение только в случае, если значение переменной *var* отличается от нуля. Для всех вершин графа потока управления, имеющих двух потомков, потребуем, чтобы ребра вели в инструкции *assume* с взаимоисключающими условиями, соответствующими исходному условию перехода.

Символом ◇ обозначается некоторый бинарный оператор, а символом □ - некоторый унарный оператор.

2.2 Конкретные состояния

При анализе будем считать, что каждая функция на модельном языке может являться точкой входа в программу. На возможные значения её параметров не накладывается никаких ограничений кроме того, что все ссылочные значения указывают на разные объекты. Для описания входных значений функции определим следующие множества:

- *W* — множество переменных, определяемых в функции;

- $P, P \subset W$ — множество параметров функции;
- N — множество значений типа integer;
- R — множество значений типа object, включая *null*;
- $V = N \cup R$ — множество значений;
- $F = F_N \cup F_R$ — множество полей, по которым может производиться взятие поля;
- F_N, F_R – множества полей типа integer и типа object соответственно.

Тогда состояние программы в момент входа в анализируемую функцию можно задать следующей парой отображений:

- $Params : P \to V$ — значения параметров в момент вызова функции;
- $HEAP : R \times F \rightharpoonup V$ — задание значений кучи.

Данную пару отображений будем называть начальным состоянием, множество всех возможных начальных состояний определим как *EState*. Состояние программы в конкретной точки анализируемой функции данной парой отображений:

- $Vars : W \to V \cup \perp$ — значения переменных; символ $\perp$ означает, что переменная не была инициализирована;
- $HEAP : R \times F \rightharpoonup V$ — задание значений кучи.

Вторую из заданных пар отображений будем называть конкретным состоянием, множество всех возможных конкретных состояний обозначим как *CState*. Нетрудно заметить, что состояние программы в момент входа в функцию является частным случаем конкретного состояния, в котором $Vars(v) \mapsto \perp, \forall v \in W \setminus P$.

Если выполнение очередной инструкции возвращает управление, будем считать, что инструкция определяет отображение *CState* → *CState* в соответствии с семантикой данной инструкции. Особенно важно, что все функции в данном модельном языке являются чистыми, т.е. их выполнение определяется конкретным состоянием в момент вызова, поэтому, если они вернут управление, их действие также будет задаваться некоторым отображением *CState* → *CState*.

Данное отображение для инструкции *instr* будем обозначать как $\xrightarrow{instr}$.

Рассмотрим некоторое начальное состояние $estate \in EState$. Тогда последовательно применяя отображения $\xrightarrow{instr}$ для очередной инструкции к текущему конкретному состоянию, получим некоторый путь выполнения.

Если у вершины есть два исходящих ребра, то, из-за взаимоисключающих условий последующих инструкций *assume*, ребро, по которому продолжится выполнение, всегда выбирается однозначно.

Тогда каждому $estate \in EState$ можно поставить в соответствие некоторый, возможно бесконечный, путь выполнения, задающийся последовательностью пар $\langle cstate_i, e_i \rangle$, для которых верно, что:

- последовательность e_i образует путь в графе потока управления – $cstate_0 = estate$, e_0 – ребро, выходящее из точки входа.
- $cstate_i \xrightarrow{instr} .cstate_{i+1}$, где $instr(e)$, инструкция в вершине, в которую входит ребро e.

Данную последовательность пар будем называть конкретным путем. Как видно из построения, каждое начальное состояние определяет возможно бесконечный набор вложенных друг в друга конкретных путей выполнения.

Тогда сформулируем задачу поиска дефекта как задачу поиска конкретного пути выполнения, на котором произошла ошибка. Для эффективного решения данной задачи необходимо ограничить длину рассматриваемых путей, причём достаточно ограничить количество вхождений обратных ребер в рассматриваемые конкретные пути выполнения некоторой константой k. Поэтому вместо графа потока управления предлагается рассматривать ациклический граф развертки, полученный из графа потока управления.

2.3 Развертка графа потока управления

Построим бесконечную развертку $UnRoll = \langle V_u, E_u \rangle$ графа потока управления $G = \langle V, E \rangle$. Множество обратных рёбер обозначим как $BE, BE \subset E$. Введем операцию *Reachable*, которая для графа потока управления G и вершины $v \in V$ строит новый граф $G' = \langle V', E' \rangle$, изоморфный подграфу G, состоящему из вершин и ребер, достижимых из v только по прямым ребрам. Изоморфизм между вершинами G' и G задается отображением $T : V' \to V$.

Первую компоненту данной развертки построим как $Comp_1 = Reachable(entry)$, где *entry* – точка входа в C*FG*, соответственно, T_{Comp1} обозначает соответствие между вершинами $Comp_1$ и *CFG*. Рассмотрим алгоритм построения из данной компоненты графа развертки *Comp*, достижимого из неё компонент. Для этого рассмотрим все вершины $Comp = \langle V_{comp}, E_{comp} \rangle$, такие, что $v \in V_{comp} \langle T_{Comp}(v), u \rangle \in BE$. Таким образом, каждой из рассматриваемых вершин было поставлено в соответствие некоторое ребро $be \in BE$, обозначим множество данных ребер как BE_{comp}. Пронумеруем рассматриваемые обратные ребра в порядке топологической сортировки, i-тое ребро данного множества обозначим как $be_i = \langle T(v_i), u_i \rangle$. Тогда компоненту $Comp_i$ из компоненты *Comp* получим применением операции R к вершине u_i. Соответственно, добавим к графу *UnRoll* граф $Reachable(u_i)$ и соединим его с компонентой *Comp* при помощи

ребра $\langle v_i, entry \rangle$, где *entry* – точка входа в граф $R(u_i)$. Ребро $\langle v_i, entry \rangle$ соответствует ребру be_i.

На Рисунке 1 изображен пример применения развертки к графу потока управления. Обратные ребра обозначены пунктирными линиями. Имена компонент содержат полную информацию о том, как данная компонента была построена. Например, имя «Компонента 1.2» означает, что данная компонента была построена из компоненты «Компонента 1» при помощи обратного ребра номер 2 в топологическом порядке.

Введем понятие глубины для графа развертки. Первая компонента графа развертки имеет глубину ноль. Если компонента графа развертки построена из компоненты глубины n, то она имеет глубину $n + 1$. Таким образом, для того чтобы ограничиться конкретными путями, проходящими по обратным ребрам не более k раз, достаточно рассмотреть развертку, содержащую все компоненты глубиной не более k.

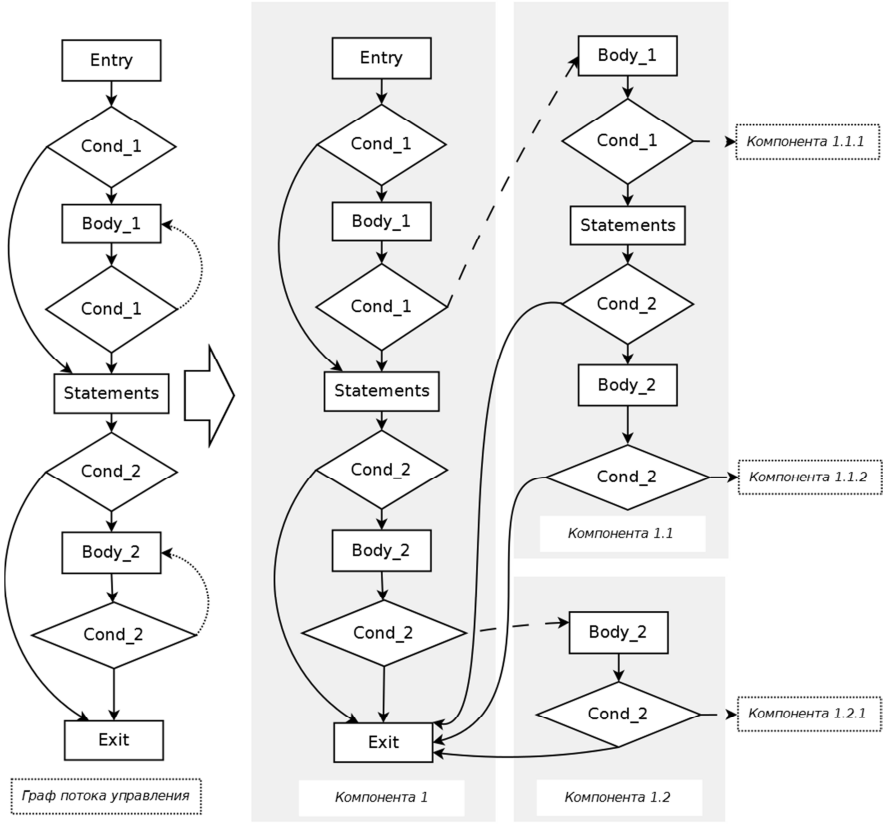

Рис. 1 — Развертка графа

Переход от путей на графе потока управления является аппроксимацией снизу, т.к. происходит ограничение рассматриваемых путей. Данная аппроксимация уменьшает множество ошибок, которые можно обнаружить с помощью анализа.

Ограничение на рассматриваемые пути действует также для вызовов функций. Теперь отображение $CState \rightarrow CState$ задается только для таких *cstate*, что вызываемая функция возвращает управление, пройдя не более, чем через k обратных ребер в вызываемой функции.

2.4 Определения понятия ошибки

При поиске ошибок нулевого указателя наибольший интерес будут представлять следующие две ситуации: во-первых, разыменование значения, полученного из присваивания переменной константы *null*, и во-вторых, взятие поля у входного параметра функции после успешного сравнения его на равенство с константой *null*. Выразим данные ситуации на языке конкретных путей.

Для этого введем два значения для обозначения нулевого указателя: *null* и *nil*. Данные значения являются эквивалентными в том смысле, что *null* == *nil* вычисляется в истину. Тем не менее, в начальном состоянии могут быть использованы только значения *nil*, а все константы, используемые в программе, имеют значение *null*. Задачу поиска ошибки вида разыменования нулевого указателя разделим на две подзадачи. Во-первых, будем искать ситуации, в которых на некотором конкретном пути значение *null* используется для взятия поля. С разыменованием значения *nil* ситуация обстоит сложнее. Если во всех случаях разыменования *nil* сообщать о разыменовании нулевого указателя, то количество ложных срабатываний будет чрезмерно велико, т.к. скорее всего анализируемая функция обладает неявным неизвестным контрактом, который запрещает данному параметру принимать значение *nil*. Поэтому предлагается выдавать предупреждение только в том случае, когда находится такой конкретный путь, что значение *nil* становится разыменованным после сравнения именно этого значения с *null*. Рассмотрим следующий пример:

```
string foo ( object a , object b) {
if (a == null && b == null ) return "";
return b.ToString( ) ;
}
```

С точки зрения данных выше определений в данном примере отсутствует ошибка разыменования нулевого указателя, т.к. поле входного параметра *b* берется только в том случае, когда *b* не участвует в сравнении с *null*.

3. Алгоритм поиска ошибок

Идея алгоритма поиска ошибок достаточно проста. Рассмотрим некоторую инструкцию взятия поля в графе развертки. Данная инструкция достижима на некотором наборе входных состояний. Если в данном наборе найдется такое входное состояние, что, когда соответствующий ему конкретный путь доберется до данной инструкции, в разыменованной переменной будет находиться значение *null*, и произойдет ошибка.

Для поиска ошибок необходимо ввести описание начальных состояний, из которых может быть достигнута конкретная точка развертки. В данном разделе рассматривается только внутрипроцедурный анализ. Межпроцедурному анализу посвящен следующий раздел.

Рассмотрим входные параметры функции. Путем доступа назовем последовательность вида $p.f_1.f_2....f_n$, где $p \in P, f_i \in F_R, f_n \in F, 0 \le i < n, n \ge 0$. Данная последовательность описывает доступ к куче через параметр функции p. Тип пути доступа будем определять по типу последнего поля. Так как развертка является конечной, то количество путей доступа, начинающихся с параметров функции, к которым может осуществляться доступ, также является конечным. Обозначим описанное множество путей доступа как AP. Пусть $|AP| = m$, тогда пронумеруем пути доступа из AP от 1 до m. Каждому пути доступа ap_i поставим в соответствие некоторую символьную переменную x_i, тип которой совпадает с типом пути доступа. Если символьная переменная x_i имеет тип integer, то она может принимать любое значение данного типа. Если же символьная переменная x_i имеет тип object, то она может принимать только два значения: *null* или r_i, где $r_i \in R$. Все r_i не равны один другому.

По аналогии с конкретным начальным состоянием, введем символьное начальное состояния. Пусть X — множество символьных переменных, поставленных в соответствие путям доступа AP. Тогда символьным начальным состоянием назовем пару отображений:

- $RV_S : P \to X$ — отображение параметров функции в соответствующие символьные переменные;
- $\text{HEAP}_S : R \times F \rightharpoonup X$ — состояние памяти в момент входа в функцию.

Отображение RV_S строится по путям доступа, не содержащих обращения к полям. Рассмотрим все $ap_i = p, p \in P$, тогда $RV_S(p) \mapsto x_i$.

Отображение HEAP_S строится, исходя из связей между путями доступа. Если пути доступа ap_i и ap_j такие, что $ap_i.f = ap_j$, то $\text{HEAP}_S(\langle r_i , f \rangle) \mapsto x_j$. Тогда, подставляя вместо символьных переменных конкретные значения, получим множество начальных состояний, задаваемое данным символьным состоянием.

Дальнейший поиск ошибок будем проводить только среди конкретных начальных состояний, задаваемых символьным состоянием. Потери истинных

срабатываний не произойдет, поскольку построенные таким образом состояния содержат все варианты входных данных для рассматриваемой развертки.

Для описания всех начальных состояний, соответствующих данной вершине графа развертки, можно воспользоваться формулами, построенными над символьными переменными. Данные формулы будем называть символьными выражениями. Так же как и символьные переменные, символьные выражения могут иметь либо тип integer, либо тип object. Построение символьных выражений задается следующими правилами:

- все числовые константы и *null* являются символьными выражениями;
- все символьные переменные являются символьными выражениями;
- если x и y — символьные выражения типа integer, то $x \diamondsuit y$ символьное выражение типа integer; если x — символьное выражение типа integer, то $\square x$ также является символьным выражением типа integer;
- если x, y – символьные выражения одного типа, а *cond* – символьное выражение типа integer, то $ite(c,x,y)$ – символьное выражение того же типа. Функция *ite* является условным оператором и возвращает свой второй аргумент, если первый не равен нулю, или третий аргумент, если первый аргумент равен нулю.

Множество всех символьных выражения обозначим как SE. По аналогии с конкретным состоянием, введем понятие символьного состояния, определяемого следующим образом:

- $RV_S : W \to SE$ — отображение переменных функции в соответствующие символьные выражения;
- $HEAP_S : R \times F \rightharpoonup SE$ — текущее состояние памяти функции.
- $\mathcal{G}, \quad \in SE$ — символьное выражения, задающее ограничения на символьные переменные.

Как и в случае с конкретными состояниями, начальное символьное состояние является частным случаем символьного состояния, в котором $\mathcal{G} = \top$.

Определим правила вывода, позволяющие из начального символьного состояния построить символьные состояния для всех вершин графа развертки с учетом семантики инструкций. Символьное состояние будем обозначать тройкой $\langle RV,HEAP,\mathcal{G} \rangle$, начальное символьное состояние обозначим как Г. Обозначим e_1,e_2 два последовательных ребра развертки, такие что $e_1 = \langle v,u \rangle, e_2 =$

$\langle u,w\rangle$. Инструкцию, содержащуюся в вершине, на которую указывает ребро e, обозначим $\mathcal{I}(e)$.

$$\frac{\langle \Gamma,\ e_1\rangle \vdash \langle RV, HEAP, \mathcal{G}\rangle\ \ \mathcal{I}(e_1) = (\text{"}a\ =\ b;\text{"})}{\langle \Gamma, e_2\rangle\ \vdash\ \langle RV\ \uplus\ \{a\ \mapsto\ RV\ (b)\}, HEAP, \mathcal{G}\rangle}$$

$$\frac{\langle \Gamma, e_1\rangle \vdash\ \langle RV, HEAP, \mathcal{G}\rangle\ \ \mathcal{I}(e_1)\ =\ \text{"}a\ =\ const;\text{"}}{\langle \Gamma, e_2\rangle\ \vdash\ \langle RV\ \uplus\ \{a\ \mapsto\ const\}, HEAP, \mathcal{G}\rangle}$$

$$\frac{\langle \Gamma, e_1\rangle \vdash\ \langle RV, HEAP, \mathcal{G}\rangle\ \ \mathcal{I}(e_1)\ =\ \text{"}a\ =\ b \diamondsuit c;\text{"}}{\langle \Gamma, e_2\rangle\ \vdash\ \langle RV\ \uplus\ \{a\ \mapsto\ RV\ (b) \diamondsuit RV\ (c)\}, HEAP, \mathcal{G}\rangle}$$

$$\frac{\langle \Gamma, e_1\rangle \vdash\ \langle RV, HEAP, \mathcal{G}\rangle\ \ \mathcal{I}(e_1)\ =\ \text{"}a\ =\ \square b;\text{"}}{\langle \Gamma, e_2\rangle\ \vdash\ \langle RV\ \uplus\ \{a\ \mapsto\ \square b\}, HEAP, \mathcal{G}\rangle}$$

$$\frac{\langle \Gamma, e_1\rangle \vdash\ \langle RV, HEAP, \mathcal{G}\rangle\ \ \mathcal{I}(e_1)\ =\ \text{"}assume(c);\text{"}}{\langle \Gamma, e_2\rangle\ \vdash\ \langle RV, HEAP, \mathcal{G}\ \wedge\ RV\ (c)\rangle)}$$

Здесь и далее операция $\uplus$ обозначает переопределение отображения для заданных значений:

$$(A\ \uplus\ B)(x)\ = \begin{cases} B(x), \text{если } x\ \in\ dom(B); \\ A(X), \text{иначе} \end{cases}$$

Введём операцию $PT : SE \rightarrow 2^{R\times \mathrm{SE}}$, которая для символьного выражения типа object строит пары из ссылки и условия, при котором данное выражение равно данной ссылке. Все условия являются попарно несовместными. Также введём операцию разыменования DEREF : $2^{R\times \mathrm{SE}} \times f \rightarrow SE$, которая определяется следующим образом:

$$pt = \{ri,ci\},\ |pt| = n,\ pt \in 2R\times SE\ vi = HEAP(ri,f)$$

$$DEREF(pt,f) = \begin{cases} ite\left(c_1, v_1, ite\left(c_2, v_2, \ldots\ ite(c_{n-1}, v_{n-1}, v_n)\right)\right), \text{для } n > 1 \\ v_1, \text{для } n = 1 \end{cases}$$

$$\frac{\langle \Gamma, e_1\rangle \vdash\ \langle RV, HEAP, \mathcal{G}\rangle\ \ \mathcal{I}(e_1)\ =\ \text{"}a\ =\ b.f;\text{"}}{\langle \Gamma, e_2\rangle\ \vdash\ \langle RV\ \uplus\ \{a\ \mapsto\ DEREF(PT(RV\ (b)), f)\}, HEAP, \mathcal{G}\ \wedge\ RV\ (b)\ \neq\ null\ \rangle}$$

$$\frac{\langle \Gamma, e_1\rangle \vdash\ \langle RV, HEAP, \mathcal{G}\rangle\ \ \mathcal{I}(e_1)\ =\ \text{"}a.f\ =\ b;\text{"}}{\langle \Gamma, e_2\rangle\ \vdash\ \langle RV, HEAP\ \uplus_{i=1}^{n} \left\{\{s_i, f\}\ \mapsto \mathrm{ite}\left(c_i, RV(b), HEAP(si, f)\right)\right\}, \mathcal{G}\ \wedge\ RV(a) \neq null\ \rangle}$$

Определим правило вывода для точки слияния. Будем считать, что в графе потока управления точки слияния выделены в отдельные вершины, которые

имеют входящую степень два. Будем считать, что все остальные вершины имеют степень один. Для точек слияния определим условия того, по какому из ребер пришло управление. Для этого воспользуемся подходом, аналогичным построению Gated SSA [1]. Таким образом, получим символьное выражение *cond*, которое точно определяет ребро, по которому управление пришло в точку слияния.

Будем считать, что, если условие *cond* верно, переход произошел по ребру e_l, иначе – по ребру e_r. Ребро, ведущее из точки слияния, обозначим как e_j. Тогда его символьное состояние задается следующим образом:

$$\frac{\langle \Gamma, e_\mathrm{l}\rangle \vdash\ \langle\ RV_\mathrm{l}, HEAP_\mathrm{l}, \mathcal{G}_\mathrm{l}\rangle\ \ \langle \Gamma, e_\mathrm{r}\rangle \vdash\ \langle RV_\mathrm{r}, HEAP_\mathrm{r}, \mathcal{G}_r\rangle\ \ \mathcal{I}(e_\mathrm{l}) = \mathcal{I}(e_\mathrm{r}),\ \ e_\mathrm{l} \neq\ e_\mathrm{r}}{\begin{gathered}\langle \Gamma, e_\mathrm{j}\rangle\ \vdash\ \langle \cup\, v\{v\ \mapsto\ ite(cond, RV_\mathrm{l}(v), RV_\mathrm{r}(v))\}, \\ \cup\, \langle sr, f\rangle\{\langle sr, f\rangle \mapsto\ ite\big(cond, HEAP_\mathrm{l}(sr, f), HEAP_\mathrm{r}(sr, f)\big)\}, \\ \mathcal{G}_l \vee\ \mathcal{G}_r\rangle\end{gathered}}$$

Для реализации различных проверок будем использовать следующий механизм пометок. К каждому символьному выражению, использующемуся в программе, алгоритм проверки может прибавить или убрать некоторую пометку, установленную для некоторого множества начальных состояний. Данное действие можно формализовать, считая для каждого ребра графа развертки отображение T : SE×TAG → SE, причём множество TAG обозначает множество пометок, используемых анализами. Данное отображение ставит соответствие между парой «символьное выражение, тип пометки» и условием, при котором данное символьное выражение будет помечено. Отображение является частичным, отсутствие соответствующего элемента для пары $\langle se,tag\rangle$ означает отсутствие пометки *tag* для выражения *se*.

При помощи данного отображения произвольный анализ дефектов может доопределить правила вывода для передачи пометок через инструкции, таким образом, возможна реализация поиска произвольных дефектов. Зачастую для объединения пометок может быть использовано следующее правило вывода:

$$\frac{\langle \Gamma, e_\mathrm{l}\rangle \vdash\ \langle se, tag\rangle \mapsto c_l\ \ \langle \Gamma, e_\mathrm{r}\rangle \vdash\ \langle se, tag\rangle \mapsto c_r\ \ \mathcal{I}(e_\mathrm{l}) = \mathcal{I}(e_\mathrm{r}),\ \ e_\mathrm{l} \neq\ e_\mathrm{r}}{\langle \Gamma, e_\mathrm{j}\rangle \vdash\ \langle se, tag\rangle \mapsto c_l \vee\ c_r}$$

Так же как и ранее, *cond* обозначает критерий выбора ребра. Если отображение отсутствует по одной из веток объединения, но присутствует по другой, будем считать, что его условием, в случае отсутствия, является ложь.

Дополнительно потребуем согласованности пометок для операции *ite*, иными словами:

$$\forall a,b,c \in \mathrm{SE}, \forall t \in TAG, T(ite(c,a,b),tag) = T(a,t) \wedge c \vee T(b,c)$$

$$\wedge \neg c$$

Далее будем считать, что условия пометок для *ite* выражений определяются автоматически по приведённому выше правилу.

Для поиска ошибок вида разыменования нулевого указателя достаточно ввести три вида пометок. Пометку *nullable* будем использовать для обозначения того, что выражение равно *null*. Данную пометку достаточно разместить только на значение *null*. Для всех производных *ite* выражений значение пометки определится автоматически.

Тогда для инструкции, берущей поле у символьного выражения *se*, верно, что все решения $T(se,nullable) \wedge \mathcal{G}$ задают начальные конкретные состояния, на которых произойдет разыменования нулевого указателя. Для поиска ошибок разыменования нулевого указателя после сравнения с *null* необходимо ввести два дополнительных типа пометок: *compared* и *deref*. Соответственно, пометкой *compared* будем помечать такие выражения, которые были сравнены с *null*, *deref* — те, что были разыменованы.

Ошибка для пометки *compared* определяется точно так же, как и для пометки *nullable*. Пометка *deref* используется при межпроцедурном анализе.

4. Межпроцедурный анализ

Для проведения межпроцедурного анализа при построении начального символьного состояния необходимо рассматривать пути доступа, по которым происходят обращения в вызванных функциях. Ввиду отсутствия рекурсии и циклов, данное множество путей доступа также будет конечным. Поэтому, учитывая его при построении начального состояния, анализ вызываемых функций можно проводить при помощи встраивания. Однако подход встраивания функций не масштабируется на промышленные приложения, поэтому вместо встраивания функций предлагается использовать резюме.

4.1 Резюме функции

Для построения резюме функции воспользуемся результатами, полученными при её анализе. Будем предполагать, что каждая функция имеет одну точку выхода. Тогда в качестве резюме функции возьмем символьное состояние, полученное в точке выхода. Таким образом, полученное резюме будет описывать состояние после выполнения функции, при условии, что все входные ссылки не являются псевдонимами друг друга.

Для межпроцедурного поиска дефектов необходимо также сохранить в резюме условия пометок *nullable* и *deref*, при этом важно, что значение пометок *compared* сохранять не требуется, т.к. наличие сравнения с *null* символьного значения внутри вызываемой функции не говорит ничего о возможности равенства нулю данного значения в контракте вызывающей функции.

Однако ввиду того, что результат применения резюме при вызове некоторой функции входит в резюме вызывающей функции, существует проблема

разрастания размера резюме при анализе промышленных проектов. Для решения данной проблемы предлагается ограничить максимальный размер сохраняемого резюме. При реальной реализации больше всего памяти требуется для сохранения символьных выражений, поэтому предлагается ограничить количество символьных выражений, которые войдут в резюме. Будем считать, что размер сохраняемых символьных выражений ограничен константой *M*. Тогда вместо символьных выражений, не вошедших в резюме, будем использовать новые символьные переменные. Если в резюме имеются символьные переменные, то его применение к конкретному состоянию становится недетерминированным. При наличии недетерминированных резюме начальное конкретное состояние перестает однозначно задавать конкретный путь выполнения. Однако для проводимого анализа данное свойство начального конкретного состояния критично, поэтому для каждого вызова функции в развертке введем свой набор неявных параметров p_i, таких, что результат применения резюме однозначно определяется по конкретному состоянию и значениям параметров p_i. Так как количество вызовов функции в развертки составляет конечное число, включим все их неявные параметры функции в входные параметры анализируемой функции. Дополненный таким образом набор входных параметров анализируемой функции снова однозначно определяет дальнейшее выполнение.

Для вычисления всех возможных путей доступа при построении начального символьного состояния потребуем от резюме, чтобы они явно задавали использующиеся в них пути доступа. Чтобы определить набор неявных параметров для текущего вызова, также, как и в случае с символьными переменными, необходимо произвести анализ возможных путей доступа для результата применения резюме.

При помощи недетерминизма в резюме можно моделировать результат выполнения неизвестных функций. На практике такое моделирование необходимо при анализе промышленных программ.

4.2 Построение резюме

Как уже говорилось ранее, в качестве резюме можно использовать символьное состояние и пометки, полученные после выполнения функции. Отдельно необходимо сохранить символьное выражение, соответствующее возвращаемому функцией значению. Также необходимо определиться с тем, какие символьные выражения должны входить в итоговое резюме.

Прежде всего постараемся сохранить те символьные выражения, которые необходимы для описания условий пометок. Если очередное условие пометки нельзя добавить к резюме, не превысив *M*, такая пометка не включается в резюме. В случае, когда все символьные выражения, требуемые для формул пометок, не могут быть сохранены, можно рассмотреть задачу аппроксимации данных снизу более строгими формулами. Аппроксимация осуществляется именно снизу для того, чтобы не допустить появления ложных срабатываний.

С точки зрения такой аппроксимации корректно заменять формулы на ложь, т.е. игнорировать формулы, не подходящие под ограничение по размеру.

Все вхождения символьных выражений, не вошедших в резюме, в символьное состояние, предлагается заменять на новые символьные переменные. Символьное состояние, полученное в результате таких замен, является аппроксимацией сверху исходного символьного состояния.

4.3 Применение резюме

Рассмотрим инструкцию вызова функции. Будем считать, что инструкция вызова функции имеет следующий вид:

$$a = foo(x_1,x_2,...,x_n);$$

где x_i – некоторая переменная, определённая в программе. Будем считать, что резюме содержит начальное состояние $entry = \langle RV_{entry}, \mathrm{HEAP}_{entry}, \top\rangle$ и конечное состояние $exit = \langle RV_{exit}, \mathrm{HEAP}_{exit}, \mathcal{G}_{exit}\rangle$. Точки применения резюме соответствует символьное состояние $\langle RV, \mathrm{HEAP}, \mathcal{G}\rangle$. Символьные выражения в entry и exit сформулированы над символьными переменными вызванной функции. Для применения резюме считается, что задано отображение $W2\underline{P}$: $W \to \underline{P}$, которое задает отображение переменных, используемых при вызове во входные параметры вызываемой функции $\underline{P}$, в том числе неявные переменные резюме. С помощью отображения $W2\underline{P}$ построим отображение $\underline{S}2S : \underline{SE} \to SE$, где $\underline{SE}$ — символьные выражения вызываемой функции. Данное отображение ставит в соответствие входным символьным значениям вызываемой функции символьные выражения вызывающей функции. Для построения $\underline{S}2S$ воспользуемся следующим алгоритмом. Во-первых, выполним следующее сопоставление:

$$\underline{S}2S : RV_{entry}(W2\underline{P}(x_i)) \mapsto RV(x_i), \forall i \in [1,n]$$

Обозначим за R множество ссылок вызываемой функции. Рассмотрим $\langle \underline{r}, f\rangle, r \in \underline{R}, f \in F$, таких, что для $\langle \underline{r}, f\rangle$ определено отображение HEAP_{entry}. Пусть x – символьная переменная вызванной функции, которая может принимать значение $\underline{r}$. По построению символьных переменных, такая переменная только одна. Тогда, если $\underline{S}2S$ определено для x, то доопределим $\underline{S}2S$ следующим образом:

$$\underline{S}2S: \mathrm{HEAP}_{entry}(\langle r, f\rangle) \mapsto \mathrm{HEAP}(\langle \underline{S}2S(x), f\rangle)$$

Также доопределим в $S2S$ отображение для символьных выражений вызываемой функции следующим образом:

$$\underline{S}2S\,(a \diamondsuit b) = \underline{S}2S\,(a) \diamondsuit \underline{S}2S\,(b)$$

$$\underline{S}2S\,(\square a) = \square \underline{S}2S\,(a)$$

$$\underline{S}2S\,(ite(cond, a, b)) = ite(\underline{S}2S\,(cond), \underline{S}2S\,(a), \underline{S}2S\,(b))$$

Рассмотрим отображения HEAP_{entry} и HEAP_{exit}. Найдем для них такое множество пар $\langle r,f\rangle$, что $\mathrm{HEAP}_{entry}(\langle r,f\rangle) /= \mathrm{HEAP}_{post}(\langle r,f\rangle)$. Данное множество пар обозначим за Diff. Тогда действие резюме определим как набор присваиваний. Для этого рассмотрим множество всех пар $\langle r,f\rangle \in$ Diff и выполним присваивание значения $\underline{S}2S(\mathrm{HEAP}_{exit}(\langle r,f\rangle))$ в поле $\underline{S}2S(x).f$ по аналогии с инструкцией $a.f = b$. Для выполнения присваивания возвращаемого функцией значения необходимо просто произвести отображение сохраненного для него символьного выражения.

Сразу оговоримся, что в случае, если параметры вызывающей функции являются псевдонимами один для другого, то результат такого применения резюме будет частично некорректным, т.к. при построении функции предполагалось, что все входные значения не являются псевдонимами один для другого. Однако на практике практически не встречаются ложные срабатывания, вызванные подобной ситуацией. Для переноса условий из резюме достаточно воспользоваться построенным отображением $\underline{S}2S$.

5. Особенности реализации

Обратим внимание на некоторые особенности реализации данного метода.

Построение символьных переменных для начального состояния можно проводить «лениво», по мере их использования во время основной фазы анализа. В данном случае, начальное символьное состояние будет доступно только по завершении анализа функции. Разворот цикла предлагается разделить на две итерации, чтобы ошибка, причина которой возникает на первой итерации, могла быть обнаружена на второй.

Для поиска допустимых начальных состояний предлагается использовать SMT-решатель. Решение формул предлагается производить в логике QF_BV согласно классификации SMT-LIB [2]. Данная логика описывает пропозициональные формулы над битовыми векторами, что позволяет достичь битовой точности при анализе. Тем не менее, запросы к решателям могут занимать достаточно долгое время, поэтому важной задачей является минимизация количества запросов. Для минимизации количества запросов вместе с точным анализом, основанным на формулах, можно также производить неточный анализ, который в простых случаях способен доказывать наличие либо отсутствие ошибки. Тогда, если неточный анализ смог доказать наличие, либо же отсутствие ошибки, запрос к SMT-решателю не является обязательным.

Например, для задачи разыменования нулевого указателя можно рассмотреть следующий анализ потока данных над символьными выражениями. Каждому символьному выражению поставим в соответствие одно из трех абстрактных значений: *Nil*, *NotNil*, *Unknown*. Данные абстрактные значения образуют

нижнюю полурешетку: *Unknown* ≺ *Nil,Unknown* ≺ *NotNil*. Данными абстрактными значениями помечены символьные выражения при прохождении соответствующих инструкций *assume*(*x*).

Операция объединения происходит в соответствии с порядком на полурешетке. Важным отличием данных пометок от пометок, используемых раннее, является то, что из равенства *null ite* выражения не может быть сделано никаких выводов о его операндах.

Соответственно, в случае разыменования символьного выражения, помеченного меткой *nullable* или *compared*, вызов SMT-решателя нужен только в том случае, когда данное символьное выражение помечено как *Unknown*.

6. Практические результаты

Данный подход реализован в рамках инструмента Svace [3] в стороннем анализаторе CSCC. Анализатор CSCC разработан на базе компиляторной инфраструктуры Roslyn. Детали реализации данного метода в CSCC планируется описать в последующих работах. Тестирование CSCC проводилось на программном обеспечении с открытым исходным кодом. В ходе тестирования была установлена хорошая масштабируемость предложенного метода на промышленных приложениях с открытым исходным кодом. Результаты тестирование приведены в таблице 1. В первом столбце таблицы указано название анализируемого проекта, второй столбец содержит количество строк кода. Последующие четыре столбца содержат число срабатываний соответствующего типа на данной проекте. Последний столбец указывает время работы анализатора. Данные результаты показывают, что число срабатываний и время работы анализатора коррелируют с объемом исходного кода. Расхождение между объемом кода и временем работы для проекта OpenBVE связано с наличием в проекте большого числа сложных для анализа функций. Большое число срабатываний на проекте Jil связано с дублированием ошибочного паттерна. Процент истинных срабатываний для реализованных проверок составил более 50%. Дальнейший анализ результатов показал, что основной причиной ложных срабатываний является неточность используемых резюме при анализе.

Табл. 1 Результаты работы анализатора на проектах с открытым исходным кодом

Проект	LOC	Разыменование константы null	Разыменование константы null в вызванной функции	Разыменование после сравнения с null	Разыменование после сравнения с null в вызванной функции	Время
Sake	1093	0	0	0	0	00:08
Polly	4828	0	0	0	0	00:13

BobBuilder	6426	1	1	0	0	00:21
Shadowsocks	17862	0	1	0	0	00:44
Perspective	20590	0	0	0	0	00:39
CSParser	21087	2	0	5	0	01:16
NetMQ	30258	0	1	1	1	01:06
Jil	49333	0	0	32	0	02:10
LibGit	51032	0	5	0	0	01:15
OpenBVE	56859	0	2	0	2	10:26
Cassandra	62757	1	1	1	1	02:12
OpenRA	104505	9	5	10	1	06:01
FSpot	110765	8	5	3	2	03:17
ShareX	144641	2	5	5	2	04:49
Banshee	167828	3	5	13	2	06:52
Lucene.Net	528120	13	12	21	2	22:01
SharpDevelop	1224434	32	186	57	99	2:11:42
Roslyn	1356367	114	222	70	25	1:46:45
NetOffice	2560115	2	56	10	245	3:31:52

7. Обзор существующих подходов

Точный и полный алгоритм поиска дефектов на путях выполнения невозможен в силу неразрешимости задач останова и анализа псевдонимов [4]. На практике используется некоторая комбинация аппроксимаций множества рассматриваемых путей сверху и снизу, что приводит к возникновению ложных срабатываний (при аппроксимации сверху) и пропуску дефектов (при аппроксимации снизу). Исходя из анализа работ по статическим анализаторам, предлагается следующий набор характеристик, по которым можно классифицировать статические анализаторы:

- покрытие — множество рассматриваемых путей выполнения;
- точность — процент истинных срабатываний, выдаваемых анализатором;
- масштабируемость — возможность анализа промышленных проектов с миллионами строк кода;

- автономность — возможность осуществления анализа без какой-либо дополнительной информации от пользователя.

Далее будем рассматривать только автономные методы поиска дефектов. В задачах автономной верификации на моделях зачастую требуется покрытие, как можно более близкое к реальному поведению программы. Самыми точными подходами в данной области являются подходы, основанные на bounded model checking(CBMC [5], LLBMC [6]), т.к. они моделируют каждый возможный путь выполнения с точностью до бита. Однако применение данных инструментов всегда связанно с поиском компромисса между временем работы и покрытием путей, т.к. данные инструменты заведомо неспособны перебрать все возможные пути выполнения за конечное время. Стоит отметить, что попытка использовать инструмент CBMC для автоматической верификации, предпринятая авторами Calysto, показала полную неприменимость данного анализатора для промышленных приложений [7]. Аналогичная попытка применить LLBMC, произведённая в рамках подготовки данной работы, также показала его неприменимость.

Одним из способов автономного доказательства корректности программ являются методы на основе подхода CEGAR [8] (SLAM [9], Blast [10], CPAchecker [11]). Идея данных методов заключается в итеративном поиске ошибочного пути выполнения с помощью уточнения абстракции. Однако для промышленных приложений в некоторых случаях такой итеративный подход принципиально не будет сходиться. Зацикливание, как правило, происходит из-за невозможности представления инварианта цикла в виде конечной формулы логики высказываний. Однако, в отличие от bounded model checking, данный подход может доказывать отсутствие ошибки сразу для всех путей выполнения. По сравнению с bounded model checking, алгоритмы, основанные на CEGAR, имеют лучшее покрытие и немного более низкую точность (например, CPAchecker обычно не поддерживает битовые операции).

Типичным применением инструментов, основанных на методе CEGAR, является верификация драйверов операционных систем, размер которых ограничен несколькими тысячами строк кода. Для моделирования поведения ядра операционной системы используются модельные реализации интерфейсов. Таким образом, несмотря на то, что сами методы являются автономными, для их эффективного применения требуется описание инфраструктуры, в рамках которой происходит анализ. Однако, что важно для автономности, генерация такой инфраструктуры также может производиться автоматически. В качестве примера можно привести разработанный в ИСП РАН проект LDV [12], который автоматически генерирует модельное окружение драйвера для последующего анализа при помощи инструмента CPAchecker.

Для достижения масштабируемости, ввиду алгоритмической неразрешимости решаемых задач, необходимо вводить различные упрощающие

предположения. Данные предположения влияют прежде всего на покрытие и точность проводимого анализа. Однако предположения, значительно ухудшающие точность проводимого анализа, делают его результаты бесполезными, т.к. анализ срабатываний пользователем анализатора является крайне трудоемким процессом, поэтому большинство упрощающих предположений направлено именно на уменьшение области покрытия. В отношении снижения точности важно понимать, что упрощающее предположение можно считать допустимым только в случае, если оно не приводит к резкому увеличению срабатываний определённого типа на интересующих программах. Упрощающие предположения направлены прежде всего на решение следующих проблем:

- поддержка циклов;
- поддержка рекурсии;
- контекстная чувствительность.

В данном случае считается, что решение таких проблем, как неограниченность кучи и анализ псевдонимов, следует из того, как анализ решает проблему циклов и рекурсии. При решении задачи циклов и рекурсии можно выделить два основных подхода: развертку циклов и итеративный алгоритм.

Итеративный алгоритм используется в таких подходах, как абстрактная интерпретация [13] и анализ потока данных. При использовании данных подходов, как правило, уменьшается точность анализа при сохранении покрытия. Основным применением для анализаторов, основанных на абстрактной интерпретации, таких как Astree [14], является анализ программ для встраиваемых систем, в которых отсутствует динамическое выделение памяти, рекурсия, а структура графа потока управления более простая, нежели у обычных программам. К сожалению, для программ общего назначения данный применение подхода затруднено из-за недостаточной точности анализа, применения аппроксимации снизу, и проблем, связанных с временем сходимости итеративного алгоритма. Более перспективным для программ общего назначения видится применение развертки циклов.

Набор подходов, посимвольно интерпретирующих выполнение программы, называется символьным выполнением [15]. К нему относятся такие анализаторы, как Saturn [16], Calysto [7], Varvel [17], CSA [18]. Данная группа примечательна тем, что, сохраняя точность близкую к CBMC, она может производить анализ промышленных приложений. Все перечисленные анализаторы производят развертку циклов и не учитывают, либо разворачивают рекурсию. Разница между данными анализаторами касается прежде всего использующихся компромиссов, которые заключаются в применении различных подходов для реализации контекстной чувствительности.

Рассматривая CSA в сравнении с CBMC, можно заметить, что данные анализаторы объединяет то, что они используют встраивание для реализации межпроцедурного анализа. Однако CSA, в отличие от CBMC анализатора, анализирует каждую функцию вне контекста, считая её точкой входа. Встраивание CSA также применяется только в том случае, когда размер встраиваемой функции невелик, иначе он считает данный вызов неизвестным, что зачастую эквивалентно его игнорированию. Кроме того, CSA использует упрощенный решатель вместо полноценных SAT/SMT решателей, что негативно сказывается на точности анализа. Однако, благодаря данным ограничениям, CSA может быть легко использован для анализа промышленных проектов. Отдельную проблему для CSA представляет межмодульный анализ, т.к. CSA работает на внутреннем представлении компилятора clang, который за одну итерацию обрабатывает один модуль трансляции.

Подход инструмента Saturn идейно близок к CBMC, с той разницей, что, во-первых, вместо встраивания функций используются резюме, во-вторых, при анализе точно объединяет символьные состояния путей в точках слияния. Для представления условий достижимости также используется предикатная абстракция. При межпроцедурном анализе Saturn также использует все функции в обратном порядке топологической сортировки графа вызовов. Каждая функция анализируется как точка входа, а также используется контекстно-нечувствительное упрощающее предположение о том, что входные данные функции могут быть произвольными. Благодаря данным решениям, разработчикам Saturn удалось добиться масштабируемости их подхода для программ, состоящих из нескольких сотен тысяч строк.

Инструмент Calysto является идейным продолжателем анализатора Saturn, однако по сравнению с Saturn, Calysto имеет следующие отличия. Во-первых, Calysto использует специализированный SMT-решатель SPEAR [19], в то время как Saturn использует SAT-решатели. Во-вторых, в качестве резюме Calysto сохраняет не наборы предусловий и постусловий, а граф символьный значений, схожий по структуре с классическим графом значений [20]. Благодаря представлению резюме функции в виде графа значений, становится возможным последовательное уточнение формул при межпроцедурном анализе по аналогии с методом CEGAR. Главной задачей инструмента Calysto было достижение большей точности из-за улучшения контекстной чувствительности. Исходя из данных, приведённых в статье, анализатор Calysto работает в среднем быстрее Saturn, имеет намного более низкий процент ложных срабатываний и находит куда больше истинных срабатываний. К сожалению, авторы Calysto не распространяют данный инструмент, что делает невозможным проверку предоставленных результатов. Кроме того, ни Calysto, ни Saturn являются исследовательскими проектами, а не промышленными анализаторами.

В отличие от двух предыдущих анализаторов, Varvel является промышленным анализатором, использующимся в NEC. Анализатор Varvel состоит из трех основных компонент: абстрактного интерпретатора, BMC-анализатора, подобного CBMC с возможностью объединения состояний, и инфраструктуры создания резюме функций. Идея метода заключается в применении абстрактной интерпретации для доказательства отсутствия ошибки. При помощи слайсинга из программы исключаются инструкции, не влияющие на недоказанные потенциально уязвимые инструкции. По словам авторов, при помощи такой комбинации удается сократить объем анализируемого кода на 60%. Далее, подобно Saturn, в Varvel все функции рассматриваются как возможные точки входа с произвольными входными параметрами. Для каждой такой функции запускается BMC-анализ с ограничением на глубину стека вызова. Все функции, имеющие глубину вызова относительно данной функции больше заданной величины, заменяются на свои резюме, содержащие предусловия и постусловия их выполнения. Авторы утверждают, что данный подход может быть эффективно применён для анализа проектов, состоящих из нескольких десятков миллионов строк кода. Опыт Varvel доказывает, что анализаторы, основанные на BMC, могут масштабироваться на промышленные проекты практически любых размеров.

Рассматриваемый в данной работе анализатор имеет общий набор упрощающих предположений с такими анализаторами, как Saturn и Varvel, а именно:

- обратные ребра в графе вызовов игнорируются;
- развертка циклов происходит на определенное число итераций;
- при анализе каждая функция считается точкой входа, а её аргументы могут принимать произвольные значения;
- все неизвестные указатели не являются псевдонимами один другого.

Предложенный в данной работе метод имеет следующие отличия от рассмотренных инструментов:

1. В отличии от Saturn, для решения формул используется SMT, а не SAT решатель, что существенно упрощает построение формул.
2. Запрос к SMT решателю происходит только в том случае, когда более простой анализ не смог доказать отсутствие ошибки. Данный подход существенно уменьшает количество запросов для проверки разыменования нулевого указателя.

3. В отличие от инструмента Valvel, для анализа и для построения резюме используется один и тот же анализ.
4. Ввиду ограничения на время работы и размер потребляемой памяти, в отличие от инструмента Calysto, размер сохраняемого резюме дополнительно ограничивается. Данное ограничение необходимо для анализа проектов, состоящих из миллионов строк кода.

8. Заключение

В данной работе был предложен метод межпроцедурного анализа программ на языке C#, позволяющий производить поиск различных дефектов в программе. В частности, были разобраны вопросы построения внутрипроцедурного чувствительного к путям анализа и использование резюме функции для реализации межпроцедурного анализа. Предложенный метод является масштабируемым и может быть использован для анализа промышленных проектов. На основе данного метода был предложен способ поиска ошибок разыменования нулевого указателя. Данный метод был реализован в анализаторе CSCC. Реализованный метод показал хорошую производительность и приемлемое количество ложных срабатываний. В последующих работах планируется более подробно описать особенности реализации данного метода в анализаторе CSCC.

Список литературы

[1]. P. Tu, D. Padua. Efficient Building and Placing of Gating Functions, SIGPLAN Not. 1995. Vol. 30, no. 6. pp. 47–55. doi: 10.1145/223428.207115

[2]. B. Clark, F. Pascal, T. Cesare. The SMT-LIB Standard: Version 2.5. Department of Computer Science, The University of Iowa, 2015. www.SMT-LIB.org.

[3]. В.П. Иванников А.А. Белеванцев А.Е. Бородин В.Н. Игнатьев Д.М. Журихин А.И. Аветисян М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды Института системного программирования РАН. 2014. Том. 26, выпуск 1. pp. 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7

[4]. G. Ramalingam. The Undecidability of Aliasing. ACM Trans. Program. Lang. Syst. 1994. Vol. 16, no. 5. pp. 1467–1471. doi: 10.1145/186025.186041

[5]. E. Clarke, D. Kroening. Hardware Verification using ANSI-C Programs as a Reference. Proceedings of ASP-DAC. 2003, pp. 308–311.

[6]. Falke Stephan, Merz Florian, Sinz Carsten. LLBMC: Improved Bounded Model Checking of C Programs Using LLVM. Tools and Algorithms for the Construction and Analysis of Systems. pp. 623–626. doi: 10.1007/978-3-642-36742-7_48

[7]. Babic D., Hu A.J. Calysto. Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on. 2008. May. pp. 211–220.

[8]. E. Clarke, O. Grumberg, S. Jha at al. Counterexample-Guided Abstraction Refinement. Computer Aided Verification. 2000. pp. 154–169. doi: 10.1007/10722167_15

[9]. SLAM2: Static Driver Verification with Under 4% False Alarms, T. Ball, E. Bounimova, R. Kumar, V. Levin. Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design. 2010. pp. 35–42.

[10]. D. Beyer, T. Henzinger, R. Jhala, R. Majumdar. The software model checker Blast. International Journal on Software Tools for Technology Transfer. 2007. Vol. 9, no. 5-6. Pp. 505–525.

[11]. D. Beyer, M.E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. Computer Aided Verification. 2011. pp. 184–190.

[12]. И.С. Захаров М.У. Мандрыкин В.С. Мутилин Е.М. Новиков А.К. Петренко А.В. Хорошилов. Конфигурируемая система статической верификации модулей ядра операционных систем. Программирование. 2015. Том. 41, номер. 1. сс. 44–67.

[13]. P. Cousot, R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. 1977. pp. 238–252. doi: 10.1145/512950.512973

[14]. P. Cousot, R. Cousot, J. Feret at al. The ASTREÉ Analyzer. Programming Languages´ and Systems Vol. 3444 of Lecture Notes in Computer Science. Pp. 21–30.

[15]. J. King. Symbolic Execution and Program Testing. Commun. ACM. 1976. Vol. 19, no. 7. pp. 385–394.

[16]. Y. Xie, A. Aiken. Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability ACM Trans. Program. Lang. Syst. 2007. Vol. 29, no. 3.

[17]. F. Ivančić, G. Balakrishnan, A. Gupta at al. Scalable and scope-bounded software verification in Varvel. Automated Software Engineering. 2015. Vol. 22, no. 4. pp. 517–559.

[18]. Z. Xu, and T. Kremenek and J. Zhang. A memory model for static analysis of C programs. Leveraging Applications of Formal Methods, Verification, and Validation. 2010. Pp 535–548.

[19]. D. Babic, F. Hutter. Spear theorem prover. Proc. of the SAT. 2008. pp. 187–201.

[20]. J. Reif, H. Lewis. Symbolic evaluation and the global value graph. Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. 1977. pp. 104–118.

Path-Sensitive Bug Detection Analysis of C# Program Illustrated by Null Pointer Dereference

V. Koshelev <vedun@ispras.ru>
I. Dudina <eupharina@ispras.ru>
V. Ignatyev <valery.ignatyev@ispras.ru>
A. Borzilov <helendile@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Annotation. This paper proposes an approach for detecting bugs in C# programs and uses null pointer deference as the main example. The approach employs a scalable path-sensitive analysis, which involves symbolic execution with state merging and function summary methods. C/C++ program analyzers like Saturn Software Analysis Project, Calysto or Svace use similar approaches. We analyze functions in backward topological order with account for previously calculated summaries. For summary construction, we use the same analysis engine as for bug detection. The paper contains a formal description of the proposed approach applied to reduced "sugar-free" subset of C# language. For each instruction of the considered language, we define a formal semantics and transfer function according to the symbolic state. During the path-sensitive analysis, we store additional information related to possible bugs in the symbolic state, and the decision whether the warning should be reported is made upon the satisfiability of the corresponding formula. Therefore, we reduce the problem of bug detection to satisfiability of a first-order logical formula defined on atoms, which are arithmetic expressions on function input values. It can be efficiently solved with modern SMT solvers. We have implemented the approach in our Roslyn-based analyzer, called SharpChecker. Evaluation of SharpChecker on open-source commodity applications has shown acceptable scalability and reasonable amount of warnings.

Keywords: static analysis; null pointer dereference; path-sensitive analysis; function summary; bug detection.

DOI: 10.15514/ISPRAS-2015-27(5)-5

For citation: Koshelev V., Dudina I., Ignatyev V., Borzilov A. Path-Sensitive Bug Detection Analysis of C# Program Illustrated by Null Pointer Dereference. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 59-86 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-5.

References

[1]. P. Tu, D. Padua. Efficient Building and Placing of Gating Functions, SIGPLAN Not. 1995. Vol. 30, no. 6. pp. 47–55. doi: 10.1145/223428.207115

[2]. B. Clark, F. Pascal, T. Cesare. The SMT-LIB Standard: Version 2.5. Department of Computer Science, The University of Iowa, 2015. www.SMT-LIB.org.

[3]. V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignatiev, D.M. Zhurikhin, A.I. Avetisyan, M.I. Leonov. Staticheskij analizator Svace dlja poiska defektov v ishodnom kode programm. [Static analyzer Svace for finding of defects in program source code] // Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26, issue 1, 2014, pp. 231-250 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-7

[4]. G. Ramalingam. The Undecidability of Aliasing. ACM Trans. Program. Lang. Syst. 1994. Vol. 16, no. 5. pp. 1467–1471. doi: 10.1145/186025.186041

[5]. E. Clarke, D. Kroening. Hardware Verification using ANSI-C Programs as a Reference. Proceedings of ASP-DAC. 2003, pp. 308–311.

[6]. Falke Stephan, Merz Florian, Sinz Carsten. LLBMC: Improved Bounded Model Checking of C Programs Using LLVM. Tools and Algorithms for the Construction and Analysis of Systems. pp. 623–626. doi: 10.1007/978-3-642-36742-7_48

[7]. Babic D., Hu A.J. Calysto. Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on. 2008. May. pp. 211–220.

[8]. E. Clarke, O. Grumberg, S. Jha at al. Counterexample-Guided Abstraction Refinement. Computer Aided Verification. 2000. pp. 154–169. doi: 10.1007/10722167_15

[9]. T. Ball, E. Bounimova, R. Kumar, V. Levin. SLAM2: Static Driver Verification with Under 4% False Alarms, Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design. 2010. pp. 35–42.

[10]. D. Beyer, T. Henzinger, R. Jhala, R. Majumdar. The software model checker Blast. International Journal on Software Tools for Technology Transfer. 2007. Vol. 9, no. 5-6. Pp. 505–525.

[11]. D. Beyer, M.E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. Computer Aided Verification. 2011. pp. 184–190.

[12]. I.S. Zakharov, M.U. Mandrykin, V.S. Mutilin, E.M. Novikov, A.K. Petrenko, A.V. Khoroshilov. Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software. January 2015, Volume 41, Issue 1, pp 49-64.

[13]. P. Cousot, R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. 1977. pp. 238–252. doi: 10.1145/512950.512973

[14]. P. Cousot, R. Cousot, J. Feret at al. The ASTREÉ Analyzer. Programming Languages´ and Systems Vol. 3444 of Lecture Notes in Computer Science. Pp. 21–30.

[15]. J. King. Symbolic Execution and Program Testing. Commun. ACM. 1976. Vol. 19, no. 7. pp. 385–394.

[16]. Y. Xie, A. Aiken. Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability ACM Trans. Program. Lang. Syst. 2007. Vol. 29, no. 3.

[17]. F. Ivančić, G. Balakrishnan, A. Gupta at al. Scalable and scope-bounded software verification in Varvel. Automated Software Engineering. 2015. Vol. 22, no. 4. pp. 517–559.

[18]. Z. Xu, and T. Kremenek and J. Zhang. A memory model for static analysis of C programs. Leveraging Applications of Formal Methods, Verification, and Validation. 2010. Pp 535–548.

[19]. D. Babic, F. Hutter. Spear theorem prover. Proc. of the SAT. 2008. pp. 187–201.

[20]. J. Reif, H. Lewis. Symbolic evaluation and the global value graph. Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. 1977. pp. 104–118.