Обнаружение ошибок доступа к буферу в программах на языке C/C++ с помощью статического анализа

И.А. Дудина <eupharina@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. В данной работе рассматривается метод поиска межпроцедурных ошибок доступа к буферу с помощью статического анализа. В основе рассматриваемого подхода лежит разработанный ранее алгоритм внутрипроцедурного анализа на базе символьного исполнения с объединением состояний, который является чувствительным к путям и учитывает взаимосвязи между переменными, такие как сравнения, арифметические операции и инструкции приведения типа. В работе предложено формальное определение межпроцедурного дефекта и рассмотрены некоторые типы межпроцедурных ошибок доступа к буферу. Межпроцедурный анализ реализован с помощью метода резюме, что позволяет в некоторой степени добиться контекстной чувствительности. Показано, как можно расширить внутрипроцедурный алгоритм для отслеживания межпроцедурных связей между переменными. Кроме этого, приведен алгоритм построения двух типов достаточных условий наличия ошибки доступа к буферу в функции, которые сохраняются в резюме и проверяются при вызове этой функции. Описанный подход был реализован в инструменте статического анализа Svace. На проекте Android 5.0.2 было получено 351 предупреждение об ошибке доступа к буферу, среди которых 64% оказались истинными, при этом существенного замедления анализа не произошло.

Ключевые слова: статический анализ; поиск дефектов; переполнение буфера; чувствительность к путям; контекстная чувствительность; межпроцедурный анализ; символьное исполнение.

DOI: 10.15514/ISPRAS-2016-28(5)-7

Для цитирования: И.А. Дудина. Обнаружение ошибок доступа к буферу в программах на языке C/C++ с помощью статического анализа. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 119-134. DOI: 10.15514/ISPRAS-2016-28(5)-7

1. Введение

Поиск ошибок переполнения буфера в исходном коде программ остаётся актуальной задачей на протяжении уже нескольких десятилетий. Одним из традиционных подходов к решению этой задачи является статический (не предполагающий запуск программы) анализ исходного кода, к преимуществам которого можно отнести покрытие всех путей при анализе, отсутствие необходимости генерировать входные данные для анализируемой программы.

```
1  #define SIZE 10
2  int checkIdx(int, int);
3  int findIdx(int val) {
4    int x;
5    for (x=0; x<SIZE; x++)
6      if(checkIdx(x, val))
7        break;
8    return x;
9  }
10 void store(int *b,
11        int i, int val) {
12   b[i]=val;
13 }
14 void foo(int val) {
15   int buffer[SIZE];
16   int idx=findIdx(val);
17   store(buffer, idx, val);
18 }
```

Рис. 1. Пример межпроцедурного срабатывания

Fig. 1. An example of inter-procedural warning

Современные методы анализа позволяют находить всё более сложные типы ошибок, сохраняя при этом приемлемое время анализа и умеренное количество ложных срабатываний. В частности, анализ совместности условий переходов позволяет организовать чувствительный к путям поиск ошибок, т.е. находить такие последовательности из более чем одной точки программы, прохождение пути выполнения по которым обязательно приведет к возникновению ошибки. Построение детектора ошибок переполнения буфера такого типа в рамках одной функции подробно описано в статье [1].

Также в отдельных класс ошибок, обнаружение которых требует привлечения специализированных подходов, можно выделить межпроцедурные ошибки. Под этим термином мы будем понимать ситуацию, когда корректность работы некоторой функции зависит от выполнения некоторого условия над значениями, получаемыми из других функций (значения, возвращаемые или изменяемые вызываемыми функциями, либо передаваемые в качестве параметров из вызывающей функции), и это условие нарушается на некотором исполнении программы, что приводит к ошибке. Такое условие корректности выполнения функции далее будем называть контрактом.

В качестве иллюстрации межпроцедурной ошибки рассмотрим пример дефекта, аналогичный обнаруженному в реальном проекте (см. Рис. 1). В данном случае функция `findIdx` вычисляет некоторое значение, которое в том числе может быть равно `SIZE`. Функция `foo` передаёт результат вызова

`findIdx` и адрес своего локального буфера размера `SIZE` в функцию `store`, в которой к переданному буферу происходит обращение по переданному индексу. Сама по себе функция `store` не содержит ошибки, но её контракт подразумевает, что переданный индекс меньше размера переданного буфера. Как мы видим, в точке вызова в функции `foo` этот контракт может нарушаться, следовательно, в этом месте необходимо выдать предупреждение.

Данный пример иллюстрирует, что выделение буфера, вычисление индекса и инструкция доступа к буферу могут находиться в разных функциях, при этом можно говорить о наличии ошибки в функции, являющейся их ближайшим общим предком в ациклическом графе вызовов. Для обнаружения таких дефектов необходимо вычислять контракты функций, гарантирующие отсутствие ошибок доступа к буферу, анализировать результат и побочные эффекты функции.

2. Постановка задачи

Целью данной работы является расширение разработанного ранее и описанного в статье [1] алгоритма поиска ошибок доступа к буферу для организации поиска межпроцедурных ошибок. Новый алгоритм предполагает те же ограничения и предположения об анализируемой программе, а именно:

- рассматриваются только обращения к буферам, имеющим константный (т.е. известный в момент компиляции) размер и размещённым в статической памяти либо на стеке;
- выполнено *предположение о контрактах*: «Контракт произвольной функции не влияет на выполнимость любого из путей на графе потока управления (ГПУ) этой функции (не существует выполнимого, но запрещенного контрактом пути)»;
- при проведении анализа каждая функция считается точкой входа в программу, что позволяет обнаруживать дефекты, проявляющиеся в потенциальных (возможных, но отсутствующих в доступном анализатору коде) контекстах вызова.

Пусть G – подграф межпроцедурного потока управления программы, содержащий только анализируемую функцию и всех её потомков в графе вызовов вплоть до листьев. Пусть G_k – граф G после развёртки каждого его цикла на k итераций [2]. Анализатор должен выдавать предупреждение об ошибке доступа к буферу, если в графе G_k существует путь, удовлетворяющий следующим условиям:

1. он содержит инструкцию обращения к буферу размера S по индексу i;
2. на любом соответствующем конкретном пути значение переменной i перед этой инструкцией не принадлежит интервалу $[0, S-1]$;
3. данный путь выполним.

В работе [1] было показано, что если контракт некоторой функция удовлетворяет предположению о контрактах то, с одной стороны, если эта функция удовлетворяет данному условию, то существует подходящий под контракт потенциальный контекст вызова этой функции, в котором её выполнение приведет к ошибке доступа к буферу; и наоборот – если существует такой контекст вызова, при котором выполнение пройдет не более k^n раз по каждому обратному ребру цикла вложенности n и приведет к ошибке доступа к буферу, то такая функция будет удовлетворять условию.

2.1 Определение межпроцедурной ошибки

Рассмотрим некоторый путь P в графе G_k, удовлетворяющий приведённому определению, т.е. содержащий ошибку доступа к буферу. Он представляет собой некоторую конечную последовательность рёбер $P = \{e_i\}$, включающую ребро, ведущее в инструкцию p_{access} доступа к буферу, в которой происходит ошибка. Выберем из $\{e_i\}$ произвольную подпоследовательность $\{e_{i_s}\}$ и на графе вызовов программы пометим все функции, содержащие рёбра из последовательности $\{e_{i_s}\}$. Далее рекурсивно отметим все функции, вызывающие отмеченные, вплоть до единого общего предка. В результате получится некоторый помеченный подграф графа вызовов, также являющийся деревом. Будем считать, что значения, изменяемые и возвращаемые непомеченными вызываемыми функциями, могут быть любыми, но обязаны удовлетворять предположению о контрактах; контекст вызова функции-корня помеченного поддерева может быть любым, удовлетворяющим предположению о контрактах. Если при данных условиях любой путь, проходящий через рёбра $\{e_{i_s}\}$, либо невыполним, либо ошибочен по данному выше определению с ошибкой доступа к буферу в точке p_{access}, то такой набор рёбер $\{e_{i_s}\}$ будем называть *критическим*. Если из некоторого критического набора нельзя выкинуть ни одного ребра с сохранением данного свойства, то такой набор будем называть *минимальным критическим*. Заметим, что для некоторых ошибочных путей можно построить более одного минимального критического набора. Если для некоторого пути, удовлетворяющего определению ошибки, не существует минимального критического набора, целиком состоящего из рёбер единственной функции, то такую ситуацию мы будем называть *межпроцедурной ошибкой*.

Заметим, что пример, изображенный на Рис. 1 является примером межпроцедурной ошибки, т.к. в любой критический набор ошибочного пути обязательно входит ребро, ведущее к инструкции доступа к буферу в функции `store`, но одного этого ребра недостаточно для определения ошибочного пути, т.к. для этой функции можно подобрать безопасный контекст вызова. Таким образом, в критический набор обязательно входят точки из других функций, т.е. ошибка межпроцедурная. В данном случае минимальный критический набор состоит из ребра в функции `store` и ребра,

соответствующего false-ветке условного оператора функции `findIdx` на k-ой итерации цикла.

3. Поиск межпроцедурных срабатываний

3.1 Описание внутрипроцедурного алгоритма

Внутрипроцедурный алгоритм поиска ошибок переполнения буфера реализован в виде модуля-детектора в статическом анализатора Svace и использует предоставляемую им инфраструктуру. Ядро Svace производит нумерацию значений, т.е. вычисляет классы эквивалентности значений переменных, называемые идентификаторами значений (VId) [7]. Детекторы ассоциируют с идентификаторами значений вычисленные свойства программы в виде атрибутов.

Ядро проводит символьное исполнение программы с объединением состояний [5]. При этом вычисляются необходимые условия достижимости каждой точки программы $q \in Instr$ в виде формул алгебры логики $ReachCond(q) = c,\ c \in Cond$, где роль переменных играют идентификаторы значений. Детекторы оповещаются о всех событиях, происходящих внутри функции. Реализация детектора заключается в описании обработчиков для этих событий.

Для организации поиска внутрипроцедурных срабатываний был введён атрибут $ValueSummary$, представляющий собой отображение:

$$VS\colon Instr \times VId \longrightarrow Summary.$$

Это означает, что в произвольной точке программы q для некоторых идентификаторов значений $v \in VId$ определено значение $s \in Summary$, суммирующее необходимую детектором информацию о значениях v по всем путям, заканчивающихся в q (подробно элементы множества $Summary$ и построение атрибута VS в ходе символьного выполнения с объединением состояний рассмотрено в статье [1]). Для каждого $s \in Summary$ определены функции:

$$HB, LB\colon Summary \times VId \ \rightarrow Cond.$$

Для любых $x \in VId, q \in Instr$, если $VS(q,v) = s$, то $HB(s,x)$ является достаточным условием того, что существует путь на ГПУ, заканчивающийся в q, такой что для каждого соответствующего конкретного пути выполнено $v \geq x$ (соответственно $v \leq x$ для формулы $LB(s,x)$). С помощью этих формул для произвольных идентификаторов значения $v, x \in VId$ в произвольной точке программы $q \in Instr$ можно вычислить условия $NotLess$ и $NotGreater$:

$$NotLess, NotGreater\colon Instr \times VId \times VId \rightarrow Cond,$$
$$NotLess(q,v,x) = HB(VS(q,v),x),$$
$$NotGreater(q,v,x) = LB(VS(q,v),x).$$

Формула $NotLess(q,v,x)$ представляет собой достаточное условие того, что, если управление пришло в точку q по некоторому пути графа потока управления, что для него в точке q всегда выполнено $v \geq x$.

Было показано, что для инструкции $ac \in Instr$ доступа к буферу с известным размером $s \in VId$ по индексу $i \in VId$ достаточным условием наличия ошибки в точке ac будет являться выполнимость формулы

$$ReachCond(ac) \wedge \big(NotLess(ac,i,s) \vee NotGreater(ac,i,-1)\big). \qquad (1)$$

С помощью описанных выше построений поиск внутрипроцедурных ошибок доступа к буферу осуществляется в три этапа:

1. В ходе символьного исполнения для идентификаторов значений $v \in VId$ в каждой точке программы $q \in Instr$ строится частичное отображение
$$VS\colon Instr \times VId \ \longrightarrow Summary.$$
2. При обработке инструкции ac доступа к буферу `b` по индексу `i` на основе значения $VS(ac,i)$ составляется формула (1) и проверяется на выполнимость.
3. В случае, если формула выполнима, т.е. подобраны значения переменных, приводящие к переполнению, из $VS(ac,i)$ путём подстановки конкретных значений переменных извлекается конкретный путь, приводящий к ошибке, и выдается предупреждение, указывающее на этот путь.

Значения атрибута VS принадлежат одному из пяти классов:

$$Summary = Const \cup Assume \cup Arithm \cup Cast \cup Join,$$

и сопоставляются идентификаторам значений при обработке соответствующих событий (см. Рис. 2): объявление константы ($newConst$), сравнение с идентификатором, имеющим непустое значение атрибута ($assume$); арифметические операции ($binaryO$) инструкции приведения типов ($castZext$, $castTrunc$), слияния значений по путям с условиями ($join$), если идентификаторы значений их аргументов имеют непустые значения атрибута VS.

Каждое из значений множества $Summary$ представляет собой ациклический ориентированный граф, все узлы которого также являются элементами множества $Summary$, а листья являются значениями типа $Const$. Так, для переменной **`res`** значение $VS(p_7, res) = s_4$ изображено на **Error! Reference source not found.**.

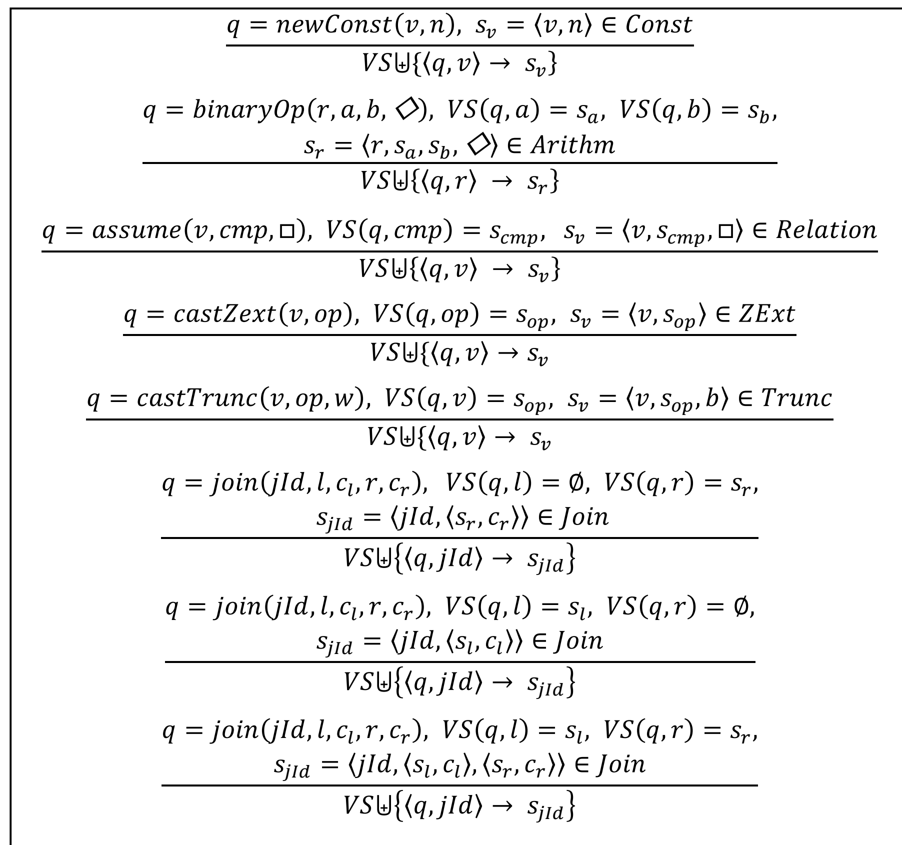

Рис. 2. Правила вывода

Fig. 2. Inference rules

3.2 Поиск межпроцедурных срабатываний с помощью резюме

Межпроцедурный анализ в инструменте Svace реализуется с помощью резюме. Данный подход заключается в том, что все функции анализируются единожды в порядке от листьев к корню в графе вызовов программы, приведенном к ациклическому виду разрывом некоторых рёбер. На основе результата внутрипроцедурного анализа функции формируется и сохраняется её резюме, т.е. краткое описание эффекта от её исполнения, включающее значения некоторых атрибутов для выбранных идентификаторов значений (стратегия формирования резюме для атрибута определяется соответствующим детектором). При обработке инструкции вызова известной

функции происходит применение её резюме, которое обязательно уже сформировано в силу порядка обхода функций. При этом идентификаторам значений вызываемой функции сопоставляются соответствующие идентификаторы значений в контексте вызывающей функции. Значения атрибутов последних вычисляются на основе значений соответствующих идентификаторов в резюме (например, просто копируются из резюме). К преимуществам данного подхода можно отнести однократный анализ каждой функции, естественную контекстную чувствительность.

```
1  int plusOne(int x){
2     if (x1 >= 10){
3         x2 = 10;
4     }
5     x3 = phi(x1, x2);
6     int res = x3 + 1;
7     return res;
8  }
9  int buf11[11];
```

```
10 int innerAccess1(int a){
11    int idx = plusOne(a);
12    return buf11[idx];
13 }
14
15 int buf5[5];
16 int innerAccess2(){
17    int idx = plusOne(4);
18    return buf5[idx];
19 }
```

Рис. 3. Пример ошибки с межпроцедурным вычислением индекса

Fig. 3. An example of defect with inter-procedural index calculation

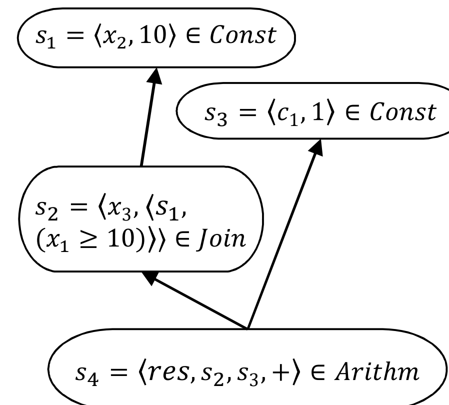

Рис. 4. Значение атрибута Summary для `res`
Fig. 5. Value of Summary attribute for `res`

3.3 Ошибки с межпроцедурным вычислением индекса

Рассмотрим произвольную ошибку доступа к буферу, возникающую в инструкции p_{access}. В данной инструкции используются две переменные – адрес буфера и индекс, значение каждой из которых может определяться в рамках текущей функции, либо вычисляться с помощью значений, вычисляемых в вызываемых или вызывающей функциях. В рамках текущей работы для переменной, содержащей адрес буфера, будем рассматривать два варианта: адрес известен в самой функции (явно используется определенный в данной функции или глобальный массив), адрес передан в качестве параметра-указателя. Для начала рассмотрим первый случай, проиллюстрированный на рис. 3. Функция `innerAccess1` содержит межпроцедурную ошибку доступа к буферу, т.к. можно привести ошибочный путь (11)-**(2)-(3)-(5)-(6)-(7)**-(12). Точку доступа к буферу на строке 12 обозначим p_{12}. Обнаружить такую ошибку можно, используя резюме. Из формулы (1), а также из того что $ReachCond(p_{12}) = tr$ следует, что достаточным условием ошибки будет являться формула $NotLess(p_{12}, idx, 11)$.

Индексом, по которому происходит доступ на строке 12 является возвращаемое значение функции `plusOne`. Таким образом, $NotLess(p_{12}, idx, 11) = NotLess(p_7, res, 11)$. В ходе анализа функции `plusOne` было установлено, что $VS(p_7, res) = s_4$. Исходя из этого можно построить достаточное условие $NotLess(p_7, res, 11)$:

$$NotLess(p_7, res, 11) = H\quad(s_4, 11)$$
$$= (res = x_3 + c_1) \wedge \exists\widetilde{x_3}\exists\widetilde{c_1}\left(HB(s_2, \widetilde{x_3}) \wedge HB(s_3, \widetilde{c_1}) \wedge (\widetilde{x_3} + \widetilde{c_1} \geq 11)\right)$$
$$= (res = x_3 + c_1)$$
$$\wedge \exists\widetilde{x_3}\exists\widetilde{c_1}\ ((x_3 = x_2) \wedge (x_1 \geq 10) \wedge HB(s_1, \widetilde{x_3}) \wedge (c_1 = 1)$$
$$\wedge (1 \geq \widetilde{c_1}) \wedge (\widetilde{x_3} + \widetilde{c_1} \geq 11))$$
$$= (res = x_3 + c_1).$$
$$\wedge \exists\widetilde{x_3}\exists\widetilde{c_1}\ ((x_3 = x_2) \wedge (x_1 \geq 10) \wedge (x_2 = 10) \wedge (10 \geq \widetilde{x_3}) \wedge (c_1 = 1)$$
$$\wedge (1 \geq \widetilde{c_1})\ \wedge (\widetilde{x_3} + \widetilde{c_1} \geq 11)).$$

Таким образом, для вычисления достаточного условия ошибки в вызывающей функции необходимо поместить в резюме значение атрибут $VS(p_7, res)$. При применении резюме следует сопоставить формальные и фактические аргументы, результат вызова и возвращаемое значение и т.п., например, идентификатору x_1 будет сопоставлен идетификатор a в контексте вызывающей функции. По этому правилу будет вычислено значение $VS(p_{12}, idx)$ путём последовательной миграции узлов дерева $VS(p_7, res) = s_4$ (см. **Error! Reference source not found.**) от листьев к корню. Таким образом, достаточное условие ошибки:

$$NotLess(p_{12}, idx, 11) = HB(VS(p_{12}, idx), 11) = (res = x_3 + c_1)$$
$$\wedge \exists\widetilde{x_3}\exists\widetilde{c_1}\left((x_3 = x_2) \wedge (a \geq 10) \wedge (x_2 = 10) \wedge (10 \geq \widetilde{x_3}) \wedge (c_1 = 1)\right.$$
$$\left.\wedge (1 \geq \widetilde{c_1}) \wedge (\widetilde{x_3} + \widetilde{c_1} \geq 11)\right).$$

Данная формула выполнима при следующих значениях: $c_1 = 1, \widetilde{c_1} = 1, x_2 = 10, x_3 = 10, \widetilde{x_3} = 10, res = 11, a = 20$, следовательно, необходимо выдать предупреждение об ошибке.

К сожалению, данный подход сам по себе не позволит обнаружить ошибку, происходящую в функции `innerAccess2` на . 3. Т.к. анализ функций производится «снизу-вверх», то при анализе функции `plusOne` не было ничего известно о возможных значениях параметра `x`. Поэтому информация о возвращаемом значении, вычисляемом из параметра по пути **(2)-(4)-(5)-(6)-(7)** отсутствует в значении $VS(p_7, res)$. Для того, чтобы отслеживать такие межпроцедурные зависимости между значениями, был введено ещё два класса значений атрибута $FParam \cup AParam \subset Summary$. Значение атрибута типа $FParam = \{v | v \in VId\}$ сопоставляется каждому формальному аргументу функции и содержит его идентификатор значения. Далее производится обычный анализ и его результаты сохраняются в резюме. В результате значение $VS(p_7, res)$ будет иметь вид, изображенный на Рис. 5. *Значение* $\boldsymbol{VS(p_7, res)}$.

При применении резюме, если в контексте вызывающей функции идентификатор значения v, передаваемый в качестве фактического аргумента, имел некоторое непустое значение атрибута в точке вызова $VS(p_{call}, v) = s_{actual}$, то для всех мигрирующих из резюме значений атрибутов происходит подстановка на место листа-формального параметра нового значения $s_v = \langle v, s_{actual} | \ v \in VId,\ s_{actual} \in Summary\rangle \in AParam$. Т.к. в точке вызова $VS(p_{17}, \text{c}_4) = s_{11} = \langle c_4, 4\rangle \in Const$, то вместо $s_5 = \langle x_1\rangle \in FParam$ будет подставлено значение $s_7 = \langle c_4, s_7\rangle \in APa$. В результате значение $VS(p_{18}, idx) = s_{10}$ будет иметь вид, изображенный на Рис. .

Во время анализа процедуры никакой априорной информации о значениях её параметров нет, поэтому:

$$s_v = \langle v\rangle \in FParam \Rightarrow HB(s_v, x) = LB(s_v, x) = false.$$

Для значений, обозначающих фактические параметры выполнено:

$$s_v = \langle v, s_{actual}\rangle \in AParam \Rightarrow \begin{matrix} HB(s_v, x) = HB(s_{actual}, x) \\ LB(s_v, x) = LB(s_{actual}, x) \end{matrix}.$$

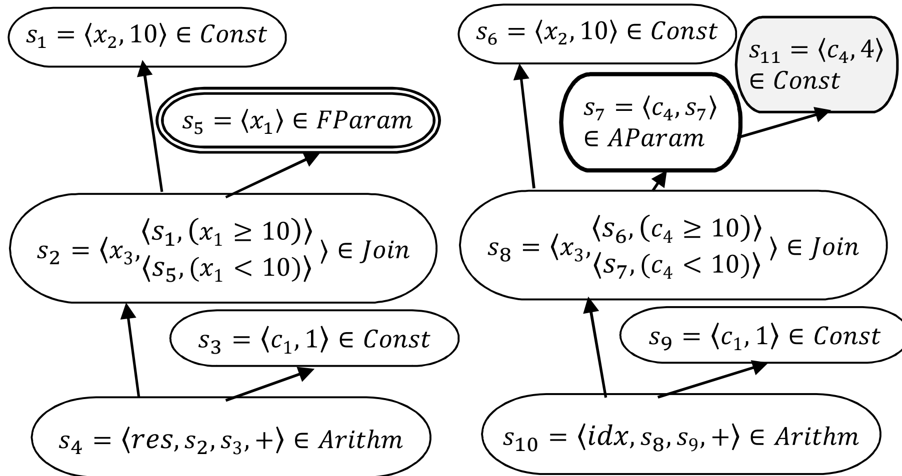

Рис. 5. Значение $\boldsymbol{VS(p_7, res)}$

Fig. 5. Value of $\boldsymbol{VS(p_7, res)}$

Рис. 6. Значение $\boldsymbol{VS(p_{18}, idx)}$

Fig. 6. Value of $\boldsymbol{VS(p_{18}, idx)}$

3.4. Построение достаточных условий ошибки для функции

Для поиска ошибок доступа к буферу в ситуациях, когда размер буфера или значение индекса определяется в одной из функций-предков (с точки зрения графа вызовов) по отношению к инструкции доступа к буферу, было введено два типа факта доступа к буферу внутри функции, сохраняемых в резюме для последующего для анализа в вызывающей функции:

$$KnowBufferAccess = \{\langle s_{idx}, accessCond, bufferSize\rangle \mid s_{idx} \in Summary, \quad accessCond \in Cond, \quad bufferSize \in \mathbb{N}\}$$

$$UnknowBufferAccess = \{\langle s_{idx}, accessCond, bufferVId\rangle \mid s_{idx} \in Summary, \quad accessCond \in Cond, \quad bufferVId \in VId\}$$

В данном разделе для краткости изложения будем рассматривать только ошибку выхода за правую границу буфера.

Рассмотрим произвольную инструкцию доступа к буферу p_{access}. Если для идентификатора значения индекса в данной точке значение атрибута VS не определено s_v, то информация об его возможных значениях отсутствует как в данной функции, так и во всех вызывающих, поэтому проверить такой доступ невозможно. Предположим, что значение атрибута VS определено для индекса и равно s_{idx}. Пометим в соответствующем ему графе все листовые вершины типа $FParam$, далее рекурсивно пометим все вершины, у которых хотя бы один из потомков помечен (кроме вершин типа $Double \subset Join$ (слияние двух значений с условиями), которые помечаются если оба потомка помечены). Непомеченные вершины соответствуют значениям, полностью определенным

в данной функции. Помеченные вершины соответствуют значениям, которые могут полностью определены только в вызывающей функции.

Если размер буфера известен в точке доступа и равен S, и в s_{idx} есть непомеченные вершины, то необходимо проверить переполнение в данной точке в ходе анализа текущей функции с помощью формулы (1). Если ошибка была найдена, то проверка этой инструкции доступа заканчивается.

В противном случае, если s_{idx} содержит помеченные вершины, то в резюме записывается факт доступа к буферу известного размера:

$$ac = \langle s_{idx}, ReachCond(p_{access}), S\rangle \in KnownBufferAccess.$$

Если размер буфера известен только в вызывающей функции (адрес буфера передан в качестве параметра v_{buf}), то в резюме записывается факт доступа к буферу неизвестного размера

$$ac = \langle s_{idx}, ReachCond(p_{access}), v_{buf}\rangle \in UnknownBufferAccess.$$

Теперь рассмотрим алгоритм применения резюме в точке p_{call}. Предположим, в нём содержится факт доступа к буферу внутри вызываемой функции. Значение s_{idx} при применении резюме трансформируется в значение s_{actual} по обычным правилам, описанным в предыдущем разделе. Рассмотрим случай, когда размер буфера либо был известен в вызываемой функции ($KnownBufferAccess$), либо стал известен при сопоставлении идентификатора значения буфера v_{buf} из факта доступа ($UnknownBufferAccess$), обозначим его за S'. Тогда, если в s_{actual} есть непомеченные вершины, то необходимо проверить наличие ошибки в данной точке, установив выполнимость формулы ($migratedCond$ – условие из факта доступа $accessCond$, транслированное в контекст вызывающей функции):

$$ReachCond(p_{call}) \wedge HB(s_{actual}, S') \wedge migratedCond.$$

Если ошибка была обнаружена, то обработка этой инструкции доступа заканчивается. В противном случае, если s_{actual} содержит помеченные вершины, то в резюме записывается факт доступа к буферу известного размера:

$$\langle s_{actual}, ReachCond(p_{call}) \wedge migratedCond, S'\rangle \in KnownBufferAccess.$$

Если адрес буфера в вызываемой функции был неизвестен, а после сопоставления его идентификатора значения с фактическим аргументом он оказался параметром текущей функции v'_{buf}, то в резюме текущей функции записывается новый факт доступа к буферу неизвестного размера:

$$\langle s_{actual}, ReachCond(p_{call}) \wedge migratedCond, v'_{buf}\rangle \in UnknownBufferAccess.$$

5. Реализация и результаты

Рассмотренный подход был реализован в рамках статического анализатора Svace. В рассмотренный в статье подход был внесён ряд технических изменений. Во-первых, для улучшения производительности были введены ограничения на размер значения атрибута VS как в рамках

внутрипроцедурного анализа, так и (более строгие) для сохранения в резюме. Кроме того, с той же целью был реализован алгоритм упрощения помещаемых в резюме формул (условий в узлах типа $Join$ и в фактах доступа к буферу $KnownBufferAccess$ и $UnknownBufferAccess$). Кроме этого, был выделен ряд типичных ситуаций, в которых нарушается предположение о контрактах, для которых были разработаны подавляющие ложные срабатывания эвристики. Так, например, зачастую сравнения некоторой переменной с параметром функции во многих контекстах всегда имеет одинаковый результат, поэтому нельзя полагаться на то, что обе ветки сравнения достижимы, поэтому такие сравнения игнорируются.

Табл. 1. Результаты работы детекторов на проекте Android 5.0.2

Table 2. Checker results on Android 5.0.2

Тип срабатывания	Кол-во	TP, %
BUFFER_OVERFLOW.EX	221	62
BUFFER_OVERFLOW.LIB.EX	64	64
OVERFLOW_AFTER_CHECK.EX	66	67

Результаты работы детектора на проекте Android 5.0.2 приведены в Табл. 1. В качестве инструкций доступа к буферу рассматривались обычные инструкции индексации и вызовы библиотечных функций, осуществляющих доступ к переданному в качестве аргумента буферу (например, `memcpy`). Исходя из этого детектор выдает предупреждения двух типов: BUFFER_OVERFLOW.EX и BUFFER_OVERFLOW.LIB.EX.

Кроме того, разработан эвристический алгоритм, который, используя информацию об индуктивных переменных и граничных условиях цикла, строит значения VS для переменных цикла и ищет ошибочные ситуации на основе этих значений. Детектор, разработанный на его основе, выдает предупреждения типа OVERFLOW_AFTER_CHECK.EX

5. Обзор существующих подходов

В работе [8] был проведён подробный сравнительный анализ работ, посвященных поиску ошибок доступа к буферу с помощью статического анализа. Существующие походы можно сравнивать с точки зрения следующих критериев:

- анализ исходного, либо бинарного кода;
- полностью автоматический, либо автоматизированный анализ;
- чувствительность к потоку, путям, контексту;
- внутрипроцедурный, либо межпроцедурный анализ;
- масштабируемость.

С точки зрения данной классификации наиболее близкой к данной работе можно считать инструмент ARCHER [9], реализованный на OCaml. Данный инструмент не требует от пользователя никаких дополнительных данных о программе, способен анализировать большие программы (анализ Linux 2.5.53, представляющий собой 2158 файлов и 1,6 млн. строк кода, занял 4 часа). При этом сохраняется высокий процент истинных срабатываний (65% из 139 срабатываний на Linux 2.5.53). В основе его подхода к анализу лежит традиционное символьное выполнение с ограничением на количество рассмотренных путей и время анализа одной функции (авторы утверждают, что при анализе Linux в среднем в функции было покрыто 96% путей). Поиск межпроцедурных срабатываний организован с помощью метода резюме. К минусам, присущим как инструменту ARCHER, так и рассматриваемому в данной статье, можно отнести отсутствие полной поддержки библиотечных функций работы со строками в языке C.

6. Заключение

В данной работе был метод поиска межпроцедурных ошибок доступа к буферу основанный на символьном исполнении с объединением состояний. Данный алгоритм является чувствительным к путям и учитывает взаимосвязи между переменными, такие как сравнения, арифметические операции и инструкции приведения типа, и кроме этого, взаимосвязь значений переменных между различными функциями, что позволяет обнаруживать межпроцедурные дефекты. Описанный подход был реализован в инструменте статического анализа Svace. На проекте Android 5.0.2 было получено 351 предупреждение об ошибке доступа к буферу, среди которых 64% оказались истинными, при этом существенного замедления анализа не произошло.

Список литературы

[1]. И.А. Дудина, В.К. Кошелев, А.Е. Бородин, Поиск ошибок доступа к буферу в программах на языке C/C++. Труды ИСП РАН, том 28, вып. 4, 2016, стр. 149-168. DOI: 10.15514/ISPRAS-2016-28(4)-9

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

[3]. D. Larochelle, D. Evans. Statically detecting likely buffer overflow vulnerabilities. 10th USENIX Security Symposium, Washington, D.C., August 2001.

[4]. 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 ИСП РАН, том 26, 2014 г., стр. 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7.

[5]. V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. 2012. Efficient state merging in symbolic execution. SIGPLAN Not. 47, 6 (June 2012), 193-204. DOI: 10.1145/2345156.2254088

[6]. А.Е. Бородин, А.А. Белеванцев. Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 111-134. DOI: 10.15514/ISPRAS-2015-27(6)-8.

[7]. А.Е. Бородин, Межпроцедурный контекстно-чувствительный статический анализ для поиска ошибок в исходном коде программ на языках Си и Си++: дис. канд. ф.-м. наук. Москва, 2016 г.

[8]. Shahriar, H., and Zulkernine, M. Classification of static analysis-based buffer overflow detectors. SSIRI-C 2010 - 4th IEEE International Conference on Secure Software Integration and Reliability Improvement Companion, 2010, pp. 94-101.

[9]. Y. Xie, A. Chou, and D. Engler, "ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors," Proceedings of the 9th European Software Engineering Conference, Helsinki, Finland, 2003, pp. 327-336.

Inter-procedural buffer overflows detection in C/C++ source code via static analysis

I. Dudina <eupharina@ispras.ru>
ISP RAS,
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation
CMC MSU, CMC faculty, 2 educational building,
MSU, Leninskie gory str., Moscow 119991, Russian Federation

Abstract. We propose inter-procedural static analysis tool for buffer overflow detection. It is based on previously developed intra-procedural algorithm which uses symbolic execution with state merging. This algorithm is path-sensitive and supports tracking several kinds of value relations such as arithmetic operations, cast instructions, binary relations from constraints. In this paper we provide a formal definition for inter-procedural buffer overflow errors and discuss different kinds of such errors. We use function summaries for inter-procedural analysis, so it provides natural path-sensitivity in some degree. This approach allowed us to improve intra-procedural algorithm by tracking inter-procedural value dependencies. Furthermore, we introduce a technique to extract the sufficient condition of buffer overflow for a function, which is supposed to be stored in the summary of this function and checked at every call site. This approach was implemented for Svace static analyzer as the new buffer overflow detector, and it has shown 64% true-positive ratio on Android 5.0.2.

Keywords: static analysis, software error detection, buffer overflow, path-sensitivity, symbolic execution, context-sensitivity, inter-procedural analysis.

DOI: 10.15514/ISPRAS-2016-28(5)-7

For citation: I. Dudina. Inter-procedural buffer overflows detection in C/C++ source code via static analysis. Trudy ISP RAN/Proc. ISP RAS, 2016, vol. 28, issue 5, 2016, pp. 119-134 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-7

References

[1]. I. Dudina, V. Koshelev, A. Borodin. [Statically detecting buffer overflows in C/C++]. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 4, 2016, pp. 149-168 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-9

[2]. V. Koshelev, I. Dudina, V. Ignatyev, A. Borzilov. [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

[3]. D. Larochelle, D. Evans. Statically detecting likely buffer overflow vulnerabilities. 10th USENIX Security Symposium, Washington, D.C., August 2001.

[4]. V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignatiev, D.M. Zhurikhin, A.I. Avetisyan, M.I. Leonov. [Static analyzer Svace for finding of defects in program source code]. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 231-250 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-7

[5]. V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. 2012. Efficient state merging in symbolic execution. SIGPLAN Not. 47, 6 (June 2012), 193-204. DOI: 10.1145/2345156.2254088

[6]. A. Borodin, A. Belevancev. [A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels]. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, pp. 111-134 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-8.

[7]. A. Borodin. PhD thesis. Interprocedural contex-sensitive static analysis for error detection in C/C++ source code. ISP RAN, Moscow, 2016

[8]. Shahriar, H., and Zulkernine, M. Classification of static analysis-based buffer overflow detectors. SSIRI-C 2010 - 4th IEEE International Conference on Secure Software Integration and Reliability Improvement Companion, 2010, pp. 94-101.

[9]. Y. Xie, A. Chou, and D. Engler, "ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors," Proceedings of the 9th European Software Engineering Conference, Helsinki, Finland, 2003, pp. 327-336.