Поиск ошибок доступа к буферу в программах на языке C/C++

[1,2] *И.А. Дудина <eupharina@ispras.ru >*
[1] *В.К. Кошелев <vedun@ispras.ru >*
[1] *А.Е. Бородин <alexey.borodin@ispras.ru >*
[1] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. В статье рассматривается алгоритм статического анализа для поиска в исходном коде программы ошибок доступа к буферу. Алгоритм использует символьное исполнение с объединением состояний и является чувствительным к путям. Рассматриваются только обращения к буферам, имеющим известный в момент компиляции размер и размещённым в статической памяти либо на стеке. В работе приведено формальное определение ошибки доступа к буферу, возникающей при прохождении некоторой последовательности рёбер графа потока управления программы. Приведён алгоритм, позволяющий для переменных программы суммировать информацию о возможных значениях по всем путям с учётом совместности условий переходов, взаимосвязи переменных через арифметические операции, инструкции преобразования типов, бинарные отношения в условиях переходов. Для инструкций доступа к буферу с помощью вычисленной для переменной индекса информации о возможных значениях вычисляются достаточные условия выхода за границы. Выполнимость достаточных условий проверяется SMT-решателем и, в случае нахождения модели, с её помощью обнаруживается ошибочный путь и выдаётся предупреждение. На основе данного подхода в инструменте статического анализа Svace был реализован межпроцедурный чувствительный к путям детектор ошибок доступа к буферу, способный обнаруживать новые, не покрытые предыдущими реализациями детекторов типы ошибок.

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

DOI: 10.15514/ISPRAS-2016-28(4)-9

Для цитирования: Дудина И.А., Кошелев В.К., Бородин А.Е. 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

1. Введение

В последнее время автоматический поиск дефектов в программном коде всё чаще становится неотъемлемой частью процесса разработки современного программного обеспечения. Одним из подходов к решению этой задачи является статический анализ исходного кода, не предполагающий запуск анализируемой программы и позволяющий найти дефекты даже на не покрытых при тестировании путях.

Одним из наиболее распространенных типов дефектов в программах на языках C и C++ являются ошибки доступа к буферу [1]. Они возникают в том случае, когда обращение к буферу происходит по индексу, выходящему за его границы, т.е. происходит доступ (чтение или запись) к памяти вне данного буфера. Такое поведение может привести к падению программы, неправильной работе, в некоторых случаях к появлению эксплуатируемой уязвимости [2].

Задача поиска дефектов такого рода в общем случае является алгоритмически неразрешимой (т.к. к ней сводится задача останова [3]), т.е. идеальный (выдающий предупреждения обо всех реальных дефектах и только о них) анализатор построить нельзя. Поэтому сценарий использования анализатора определяет конкретные требования к нему: уровень полноты анализа, степень масштабируемости, ограничение на количество потребляемых ресурсов и время анализа, приемлемую долю ложных срабатываний. Совокупность этих требований в свою очередь определяет подходящие для данного сценария методы анализа.

Задачей данной работы является разработка алгоритма поиска ошибок доступа к буферу, удовлетворяющего следующим требованиям. Во-первых, он должен хорошо масштабироваться, т.к. необходимо анализировать большие программные системы (объем исходного кода исчисляется миллионами строк) за время ночной сборки (4-6 часов). Во-вторых, необходимо обеспечить высокий уровень истинных предупреждений (не менее 50-70%, в противном случае затраты на разбор ложных предупреждений нивелируют пользу автоматического поиска ошибок), при этом каждое выданное предупреждение должно сопровождаться достаточной аргументацией возникших подозрений о наличии ошибки для пользователя.

Разрабатываемый алгоритм предлагается реализовать в рамках инструмента статического анализа Svace [4], разрабатываемого в ИСП РАН. Уже существующий в Svace детектор ошибок доступа к буферу выдаёт предупреждения, если будет найдено некоторое ребро графа потока управления такое, что все проходящие через него пути содержат ошибку. В примере на рис. 1 этому свойству удовлетворяет пунктирное ребро – на любом проходящем через него пути произойдет доступ к буферу размера 7 по индексу 7.

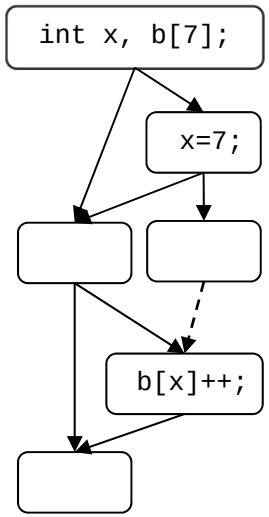

Рис. 1 Пример ошибки

Fig. 1. An example of error

К сожалению, существуют ошибочные ситуации, которые не удовлетворяют этому критерию. В качестве иллюстрации этого тезиса рассмотрим пример, изображенный на рис. 2.

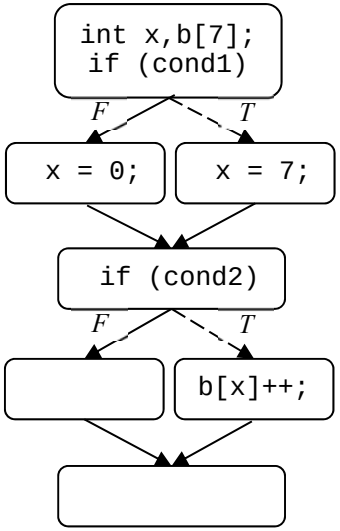

Рис. 2 Пример ошибки

Fig 2. An example of error

Если `cond1` и `cond2` могут одновременно принимать значение “истина”, то такая программа содержит ошибку доступа к буферу. В данном графе не существует единственного ребра, прохождение через которое гарантирует возникновение ошибки (для каждого ребра существует безошибочный путь через него). Тем не менее, можно предъявить такую последовательность рёбер (на рисунке выделены пунктиром), что любой путь, включающий эту последовательность, будет содержать ошибку. Ещё раз подчеркнём, что для выдачи предупреждения в такой ситуации необходимо прежде проанализировать совместность условий `cond1` и `cond2`.

Чтобы обнаруживать такие дефекты, при этом не превышая допустимое количество ложных срабатываний, необходимо реализовать чувствительный к путям анализ. Одним из методов решения этой задачи является символьное исполнение с объединением состояний [5]. Такой подход позволяет сводить задачу перебора путей к задаче определения выполнимости булевых формул, что позволяет сохранять масштабируемость анализа.

Дальнейшее изложение организовано следующим образом. Во второй части приведены априорные предположения об анализируемых программах, в рамках которых производятся построения алгоритма; дано формальное определение ошибки доступа к буферу; описана необходимая базовая инфраструктура ядра анализатора. В третьей части описан разработанный алгоритм, основанный на вычислении достаточных условий наличия ошибки, приведён алгоритм построения таких условий. Четвертая часть содержит описание реализации данного алгоритма в рамках анализатора Svace.

2. Постановка задачи

Необходимо разработать критерий ошибочной ситуации; разработать и реализовать алгоритм обнаружения таких ситуаций, удовлетворяющий ограничениям на количество ложных срабатываний.

2.1. Определение ошибки доступа к буферу

В данной работе рассматриваются только обращения к буферам, имеющим константный (т.е. известный в момент компиляции) размер и размещённым в статической памяти либо на стеке. Такой подход был выбран в качестве первого приближения к решению задачи т.к., с одной стороны, позволяет найти значительную часть ошибок доступа к буферу, а с другой – предложенные методы могут быть впоследствии доработаны для организации поиска ошибок более общего вида, в т.ч. анализа доступа к динамически выделенной памяти. Кроме этого, ошибки доступа к статическим массивам в некоторых ситуациях могут являться источником уязвимости [2].

При реализации конкретной функции разработчик как правило стремится к тому, чтобы её поведение было корректным во всех потенциальных контекстах вызова (а не только в реально существующих в проекте на данный момент). Это

требование в первую очередь важно для библиотечных функций, функций, которые ещё будут использоваться при разработке нового кода. Поэтому, чтобы обнаруживать дефекты, возникающие в потенциальных контекстах вызова, при проведении анализа каждая функция считается точкой входа в программу. Далее будем считать, что при анализе рассматривается некоторая функция вместе со всеми вызываемыми ею, вызываемыми вызываемыми ею и т.д.

Сценарий использования разрабатываемого анализатора предполагает анализ проектов при отсутствии части исходного кода (например, реализаций библиотечных функций, кода пользовательских расширений). Кроме этого, анализатор должен быть полностью автоматическим, а значит не предполагается предоставление никакой дополнительной информации от разработчиков. В данном случае при анализе функции подразумеваемое программистом предусловие неизвестно (с учетом того, что в рассмотрение включены в том числе отсутствующие в программе потенциальные контексты вызова). Вытекающая из этого сложность анализа заключается в том, что на значения, получаемые из неизвестных функций или передаваемые в функцию в качестве аргументов, могут быть наложены неизвестные анализу ограничения, например, программисты могут подразумевать существование некоторой взаимосвязи значений аргументов функции. Полное игнорирование возможности наличия таких взаимосвязей приведет к выдаче чрезмерного количества ложных срабатываний, т.к. анализатор будет сообщать об ошибках, возникающих в случае их нарушения, что нежелательно.

Для некоторой функции будем называть *неизвестными переменными* такие приходящие из других функций (вызывающих данную и вызываемых данной) значения, параметризующие выполнение анализируемой функции. Набор значений неизвестных переменных однозначно определяет *конкретный путь* исполнения функции, т.е. путь на графе потока управления (ГПУ) и значения всех переменных, состояние памяти на каждом его ребре. Такими неизвестными переменными будут являться входные параметры, начальное на входе в функцию состояние памяти, результаты вызова неизвестной функции. Совокупность ограничений на множество значений набора таких параметров будем называть *контрактом*. Таким образом, при анализе необходимо иметь в виду всё множество возможных контрактов и выдавать предупреждения только в тех случаях, когда ошибка происходит при каждом контракте из этого множества. Какие именно контракты считать возможными решается при построении конкретной стратегии анализа. Выше уже было отмечено что выбор пустого множества в качестве множества возможных контрактов приводит к чрезмерному количеству ложных срабатываний.

С другой стороны, если считать, что возможны абсолютно любые контракты, то придется пропустить слишком много реальных дефектов. Как правило, для ошибочной функции можно подобрать искусственное предусловие, исключающее возникновение ошибки, при этом оно будет куда более строгим, чем реальное, задуманное программистом. Т.е. существование такого

искусственного, полностью “безопасного” предусловия (как правило, лишающего функцию смысла) не является препятствием для выдачи предупреждения об ошибке.

В качестве компромисса предлагается ввести априорное предположение о свойствах контрактов функций, которое определит множество возможных контрактов, и, как следствие, позволит отделить потенциально возможные ошибочные сценарии исполнения функции от предположительно запрещенных контрактом. В рамках данной работы будем считать подозрительными такие ситуации, в которых наличие ошибки доступа к буферу следует из свойств ГПУ программы, но не зависит напрямую от множества допустимых значений неизвестных переменных.

Проиллюстрируем это различие на примере на рис. 3. В функции `foo` может произойти переполнение буфера `buf`, если будет выполнено `idx` $\geq$ `S`. Исходя из вышесказанного, данную ситуацию не будем считать ошибочной, т.к. считаем, что такие значения `idx` запрещены контрактом функции. Заметим, что, наличие ошибки напрямую зависит от множества возможных значений `idx`. Теперь рассмотрим функцию `bar`, здесь переполнение произойдет, если будет выполнено:

$$(a \geq S - 1) \wedge (b \neq 0) \qquad (1)$$

```
1  #define S 10
2
3  int buf[S];
4
5  void foo(int idx) {
6      buf[idx]++;
7  }
8
```

```
9  int bar(int a, int b) {
10     if (a >= S-1) {
11         // …
12     }
13     if (b)
14         a++;
15     return buf[a];
16 }
```

Рис. 3 Функции с неизвестными контрактами
Рис. 3. Functions with unknown contracts

Мы будем считать такую ситуацию дефектом, т.к. ошибка следует из свойств ГПУ, а именно из наличия возможно выполнимого пути, на котором гарантированно произойдёт переполнение. При этом также контракт функции может запрещать (1), – тогда ошибки нет, т.е. наличие дефекта зависит не напрямую от множества значений неизвестных переменных, а косвенно, т.к. контракт влияет на выполнимость некоторых путей на ГПУ. Чтобы строго различить две рассмотренные в функциях `foo` и `bar` ситуации, будем считать, что контракты функций не могут влиять на выполнимость путей, т.е. контракта, запрещающего (1) не существует, тогда очевидно, что функция `bar` содержит

ошибку. Заметим, что рассмотренный для функции `foo` контракт удовлетворяет этому предположению, значит предупреждение об ошибке не будет выдано. Сформулируем описанное предположение о контрактах строго.

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 [6]. Рассмотрим множество P всех путей графа G_k. Будем говорить, что некоторый путь $p \in P$ выполним, если существует хотя бы один соответствующий ему конкретный путь. Пусть $P' \subseteq P$ — подмножество путей графа G_k, состоящее только из тех путей, которые выполнимы при условии, что набор неизвестных переменных может принимать абсолютно любые сочетания значений. Обозначим как P'' — подмножество путей графа потока управления программы, состоящее только из тех путей, которые выполнимы, если набор неизвестных переменных удовлетворяет подразумеваемому программистом контракту. Очевидно, что $P'' \subseteq P' \subseteq P$. Наше предположение о контрактах будет заключаться в том, что эти контракты не сужают множество выполнимых путей, т.е. $P'' = P'$.

Введение такого предположения приводит нас к следующему определению ошибки:

Будем говорить, что функция содержит ошибку 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. *данному пути соответствует хотя бы один конкретный путь исполнения (в предположении, что набор неизвестных переменных может принимать любые комбинации значений).*

Покажем, что если программа удовлетворяет описанному определению, то существует выполнимый путь, содержащий некорректный доступ к буферу. Необходимо убедиться, что если найденный путь выполним в случае, когда набор неизвестных переменных принимает некоторое произвольное значение, то он выполним и при условии выполнения контракта. Это напрямую следует из предположения о контрактах.

Покажем, что если в программе при любых удовлетворяющих предположению контрактах существует выполнимый путь, проходящий не более k^n раз по каждому обратному ребру цикла вложенности n и содержащий некорректный доступ к буферу, то она соответствует определению. Допустим в анализируемой программе происходит ошибка доступа к буферу на некотором конкретном пути c, не проходящем более k^n раз по каждому обратному ребру цикла вложенности n. Тогда в графе G_k существует путь $p : c \in concrete(p)$, и он, очевидно, удовлетворяет первому и третьему пункту определения.

Предположим, что для этого пути существует другой конкретный путь $c' : c' \in concrete(p),\ c' \neq c$, на котором не происходит ошибки. Значит, условие возникновения ошибки зависит от значения неизвестных переменных. Следовательно, существует контракт, который запрещает значения неизвестных переменных, задающих конкретный путь c, (запрещает только c). Такой контракт удовлетворяет предположению, т.к. p по-прежнему выполним, раз существует c'. Существование такого контракта противоречит свойствам рассматриваемой ошибки, поэтому такого пути c' не может существовать, а значит, верен и второй пункт.

Наличие ошибки, удовлетворяющей этому определению, следует из свойств графа потока управления и не зависит от множества допустимых значений неизвестных параметров.

Рассмотрение графа G_k вместо графа G приводит к тому, что игнорируются ошибки, происходящие на итерации цикла большей чем k-ая. В реализации алгоритма используется эвристика, позволяющая частично обойти эти ограничения. Она заключается в изменении семантики арифметических операций, позволяющем моделировать некоторую обобщенную итерацию цикла.

2.2. Инфраструктура анализатора

В данной работе предполагается, что рассматриваемый подход будет реализован в качестве модуля-детектора в рамках общей инфраструктуры статического анализа Svace. На уровне ядра анализатора в Svace решаются задачи построения ГПУ, поиска недостижимого кода, анализа алиасов, анализа функций, завершающих выполнение программы. Всем детекторам доступна информация о результатах этих анализов [7].

Ядром производится нумерация значений, т.е. вычисляются классы эквивалентности значений переменных, называемые идентификаторами значений [8]. Детекторы ассоциируют с идентификаторами значений вычисленные свойства программы.

Ядро проводит символьное исполнение программы с объединением состояний. При этом вычисляются необходимые условия достижимости каждой точки программы в виде формул алгебры логики, где роль переменных играют идентификаторы значений. Детекторы оповещаются о всех событиях, происходящих внутри функции. Реализация детектора заключается в описании обработчиков для этих событий. Описание интересных с точки зрения данной работы событий приводится в разделе 3.2.

Далее множество точек программы будем обозначать как $Instr$, множество идентификаторов значений как VId, необходимые условия достижимости точки $q \in Instr$ как $ReachCond(q) = c,\ c \in Cond$.

3. Поиск внутрипроцедурных срабатываний

В рамках данной работы остановимся на рассмотрении методов поиска внутрипроцедурных ошибок доступа к буферу, при этом буфер расположен на стеке анализируемой функции, либо в доступной ей статической памяти. Для обнаружения таких ошибок для каждой подходящей инструкции доступа к буферу необходимо проверить, существует ли путь на ГПУ, проходящий через данную инструкцию и удовлетворяющий пунктам 2-3 определения ошибки.

Предположим, что для любых идентификаторов значения $v, x \in VId$ в каждой точке программы $q \in Instr$ известно условие в виде формулы алгебры логики $NotLess(q, v, x)$, из которой следует, что управление пришло в точку q по некоторому пути графа потока управления, при этом на каждом соответствующем ему конкретном пути в точке q выполнено $v \geq x$ (идентификаторы значений играют роль переменных в логической формуле). Аналогичная формула $NotGreater(q, v, x)$ известна для условия $v \leq x$. Тогда для инструкции $ac \in Instr$ доступа к буферу размером $s \in VId$ по индексу $i \in VId$ достаточным условием наличия ошибки в точке ac будет являться выполнимость формулы

$$ReachCond(ac) \wedge \big(NotLess(ac, i, s) \vee NotGreater(ac, i, -1)\big) \qquad (2)$$

Если выполнено $NotLess(ac, i, s) \vee NotGreater(ac, i, -1)$, то существует путь на ГПУ, удовлетворяющий второму пункту определения. Выбор инструкции ac обеспечивает выполнение первого пункта, а т.к. одновременно выполнено $ReachCond(ac)$, то найденный путь выполним, и, следовательно, верен третий пункт определения.

Таким образом, задача поиска ошибок описанного типа сводится к вычислению как можно более слабых условий $NotLess(q, v, x)$ и $NotGreater(q, v, x)$. Для её решения для идентификатора значения v в точке q определяется значение $s \in Summary$, суммирующее информацию о значениях v по всем путям, заканчивающихся в q. Искомые условия $NotLess(q, v, x)$ и $NotGreater(q, v, x)$ будут вычисляться с помощью s.

Предлагается организовать поиск ошибок доступа к буферу в три этапа:

1. В ходе символьного исполнения для идентификаторов значений $v \in VId$ в каждой точке программы $q \in Instr$ построить 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 (2) 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.

3.1. Отображение ValueSummary

Рассмотрим подробнее что представляет из себя отображение VS и как вычислять условия $NotLess$ и $NotGreater$ с его помощью.

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

Каждое значение $s \in Summary$ содержит в себе собственный идентификатор значения, информацию о котором оно суммирует по разным путям ГПУ, т.е. если $VS(q, v) = s$, то v является собственным идентификатором s.

Для вычисления из s условий $NotLess$ и $NotGreater$ 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 будем вычислять условия $NotLess$ и $NotGreater$:

$$NotLess(q, v, x) = HB(VS(q, v), x),$$
$$NotGreater(q, v, x) = LB(VS(q, v), x).$$

Таким образом, задача свелась к построению отображения VS и вычислению условий $HB(s, x)$ и $LB(s, x)$ (опять, чем слабее будут эти условия, тем лучше). Рассмотрим подробнее что представляют из себя элементы множества $Summary$ и как для них строить искомые условия. Значения множества принадлежат одному из следующих типов:

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

1. $Const = \{\langle v, n\rangle \mid v \in VId,\ \ n \in \mathbb{Z}\}$ – определение константы.

Значения констант всегда одни и те же на всех путях, поэтому если

$$s_v = \langle v, n\rangle \in Const,$$

то из этого следует:

$$HB(s_v, x) = (v = n) \wedge (n \geq x),$$
$$LB(s_v, x) = (v = n) \wedge (n \leq x).$$

2. $Relation = \{\langle v, s_{comparand}, \square\rangle \mid id \in VId,$ $s_{comparand} \in Summary,\ \square \in \{>, \geq, <, \leq, =\}\}$ – факт истинности отношения $\square$ для пары идентификаторов значений v и $comparand$, вытекающий из перехода по условию $v \square comparand$. Элемент данного типа является результатом отображения VS идентификатора значения v в точке q в том случае, когда $VS(q, comparand) = s_{comparand}$, и у предшествующего условного оператора ветка с условием $v \square comparand$ доминирует над текущей точкой. Очевидно, что условие перехода по данной ветке гарантированно выполнено в текущей точке. Отсюда можно вывести искомые достаточные условия, рассмотрим их на примере отношения строгого сравнения. Пусть:

$$s_v = \langle v, s_{comp}, >\rangle \in Relation \mid\ s_{comparand} \in Summary$$

Тогда:

$HB(s_v, x) = (v > comp) \wedge HB\big(s_{comp,} x - 1\big)$,

$LB(s_v, x) = false$ (ничего не известно о верхней границе v).

3. $Arithm = \{\langle v, s_a, s_b, \diamondsuit\rangle \mid v \in VId,\ s_a, s_b \in Summary, \diamondsuit \in \{+, -, \times, /\}\}$ – результат арифметической операции $v = a \diamondsuit b$, для каждого из операндов которой в данной точке определено значение отображения:

$$VS(q, a) = s_a, \qquad VS(q, b) = s_b.$$

Рассмотрим вычисление достаточных условий $HB(s_v, x)$ на примере вычитания. Пусть

$$s_v = \langle v, s_a, s_b, -\rangle \in Arithm \mid s_a, s_b \in Summary.$$

Предположим, что необходимо для некоторого x доказать, что поток управления достиг данной точки по некоторому пути ГПУ, такому что на любом соответствующем ему конкретном пути выполнено $v \geq x$, т.е. $a - b \geq x$. Заметим, что для любых $a, \tilde{a}, b, \tilde{b} \in \mathbb{Z}$ верно:

$$a \geq \tilde{a} \ \wedge\ b \leq \tilde{b} \ \wedge\ \tilde{a} - \tilde{b} \geq x \quad \Rightarrow \quad a - b \geq x.$$

Следовательно, достаточно предъявить для пути ГПУ, проходящего через данную точку, два целых числа $\tilde{a}$ и $\tilde{b}$, таких что на любом соответствующем ему конкретном пути выполнена посылка импликации: $a \geq \tilde{a} \ \wedge\ b \leq \tilde{b} \ \wedge\ \tilde{a} - \tilde{b} \geq x$. Отсюда получаем формулу

$$HB(s_v, x) = (v = a - b) \wedge \Big(\exists \tilde{a} \exists \tilde{b}\ LB(s_a, \tilde{a}) \wedge HB\big(s_b, \tilde{b}\big) \wedge \big(\tilde{a} - \tilde{b} \geq x\big)\Big).$$

Аналогично выводится формула $LB(s_v, x)$ и такие же формулы для сложения.

Введение дополнительных переменных $\tilde{a}$ и $\tilde{b}$ здесь необходимо, т.к. заранее (до анализа совместности условий переходов) определить нижние и верхние границы значений переменных a и b невозможно, поэтому значения $\tilde{a}$ и $\tilde{b}$ определит решатель.

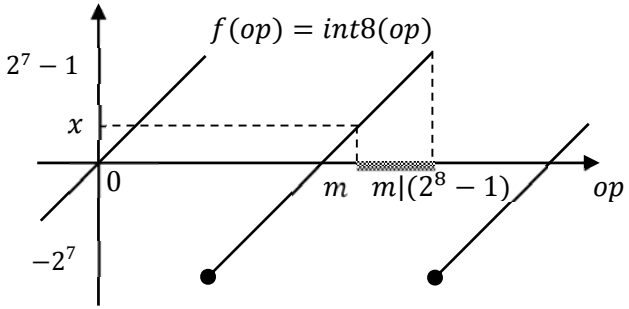

Рис. 4. Вычисление достаточного условия для нижней границы для результата инструкции приведения типа

Fig. 4. Calculating the sufficient condition for the lower BOUND for the result of the CAST instruction

4. $Cast = Trunc \cup ZExt$ – результат инструкции преобразования типов, для аргумента которой в данной точке имеет определено значение отображения $VS(q, op) = s_{op}$.

a. $ZExt = \{\langle v, s_{op}\rangle \mid v \in VId,\ s_{op} \in Summary\}$ – беззнаковое расширение значения op. Пусть

$$s_v = \langle v, s_{op}\rangle \in ZExt \mid s_{op} \in Summary.$$

Тогда:

$$HB(s_v, x) = HB\big(s_{op}, x\big),$$
$$LB(s_v, x) = LB\big(s_{op}, x\big) \wedge (x \geq 0).$$

b. $Trunc = \{\langle v, s_{op}, w\rangle \mid v \in VId,\ s_{op} \in Summary,\ w \in \mathbb{N}\}$ – приведение op к типу меньшего размера, равного w. Тогда если:

$$s_v = \langle v, s_{op}, w\rangle \in Trunc \mid s_{op} \in Summary, \qquad w \in \mathbb{N},$$

то

$$HB(s_v, x) = \exists m\ (m\&(2^b - 1) = x) \wedge HB\big(s_{op}, m\big) \wedge LB\left(s_{op}, m\middle|(2^b - 1)\right).$$

Вывод этой формулы для случая $w = 8$ поясняется на рис. 4, где показана зависимость результата инструкции приведения целочисленного значения op к однобайтному целому типу. У числа m старшие (обрезаемые) биты совпадают с соответствующими в числе op, а младшие 8 бит совпадают с младшими 8-ю битами числа x. Таким образом, $op \geq x$ тогда и только тогда, когда op принадлежит интервалу $[m, m|(2^8 - 1)]$ (на рисунке этот интервал выделен штриховкой).

Условие $\mathrm{LB}(s_v, x)$ вычисляется аналогично.

5. $Join = Single \cup Double$,

$Single = \{\langle joinedId, \langle s_{br}, c\rangle\rangle \mid joinedId \in VId, s_{br} \in Summary, c \in Cond\}$,

$Double = \{\langle joinedId, \langle s_l, c_l\rangle, \langle s_r, c_r\rangle\rangle \mid joinedId \in VId,\ s_l, s_r \in Summary,\ c_l, c_r \in Cond\}$

– значение в точке слияния двух веток с условиями c_l и c_r, таких что, значение отображения определено на обоих ветках: $VS(q, l) = s_l,\ VS(q, r) = s_r$, либо только на одной $VS(q, br) = s_{br}$ (в этом случае условие этой ветки обозначается просто c). Для случая двух веток, если:

$$s_{jId} = \langle jId, \langle s_l, c_l\rangle, \langle s_r, c_r\rangle\rangle \in Join \mid s_l, s_r \in Summary,\ c_l, c_r \in Cond,$$

то

$$HB\big(s_{jId}, x\big) = \bigvee \begin{matrix} (joinedId = l) \wedge HB(s_l, x) \wedge c_l \\ (joinedId = r) \wedge HB(s_r, x) \wedge c_r \end{matrix}.$$

Достаточные условия $LB(s_{jId}, x)$ и оба вида достаточных условий для случая, когда отображение определено только на одной ветке, записываются аналогично.

В случае, если для некоторого идентификатора значения v в данной точке q значение отображения не определено $VS(q, v) = \emptyset$, то информации о возможных его значениях нет, поэтому функции HB и LB можно доопределить:

$$HB(\emptyset, x) = false, \qquad LB(\emptyset, x) = false.$$

3.2. Построение отображения ValueSummary

Перед началом символьного исполнения $VS = \emptyset$. Обновление отображения производится для следующих событий:

- $newConst(v, n)$, $v \in VId$, $n \in \mathbb{Z}$ – объявление константы.
- $binaryOp(r, a, b, \diamondsuit)$, $r, a, b \in VId$, $\diamondsuit \in \{+, -, \times, /\}$ – арифметическая операция $r = a \diamondsuit b$
- $assume(v, cmp, \square)$, $v, cmp \in VId, \square \in \{>, \geq, <, \leq, =\}$ – условие на ребре инструкции ветвления.
- $castZext(v, op)$, $v, op \in VIds$ – беззнаковое расширение.
- $castTrunc(v, op, w)$, $v, op \in VIds$, $w \in \mathbb{N}$ – приведение к типу меньшего размера, равного w.
- $join(jId, l, c_l, r, c_r)$, $jId, l, r \in VId$, $c_l, c_r \in Cond$ – слияние двух идентификаторов l и r в jId по веткам с условиями c_l и c_r.

Обновление отображения для этих событий происходит в соответствии с правилами вывода (см. рис. 5). Для прочих инструкций значения отображения для всех идентификаторов копируются с предыдущего состояния.

$$\frac{q = newConst(v, n),\quad s_v = \langle v, n\rangle \in Const}{VS \uplus \{\langle q, v\rangle \rightarrow s_v\}}$$

$$\frac{q = binaryOp(r, a, b, \diamondsuit),\quad VS(q, a) = s_a, VS(q, b) = s_b,\quad s_r = \langle r, s_a, s_b, \diamondsuit\rangle \in Arithm}{VS \uplus \{\langle q, r\rangle \rightarrow s_r\}}$$

$$\frac{q = assume(v, cmp, \square),\quad VS(q, cmp) = s_{cmp},\quad s_v = \langle v, s_{cmp}, \square\rangle \in Relation}{VS \uplus \{\langle q, v\rangle \rightarrow s_v\}}$$

$$\frac{q = castZext(v, op),\quad VS(q, op) = s_{op},\quad s_v = \langle v, s_{op}\rangle \in ZExt}{VS \uplus \{\langle q, v\rangle \rightarrow s_v}$$

$$\frac{q = castTrunc(v, op, w),\quad VS(q, v) = s_{op},\quad s_v = \langle v, s_{op}, b\rangle \in Trunc}{VS \uplus \{\langle q, v\rangle \rightarrow s_v}$$

$$\frac{q = join(jId, l, c_l, r, c_r),\quad VS(q, l) = \emptyset,\ VS(q, r) = s_r,\quad s_{jId} = \langle jId, \langle s_r, c_r\rangle\rangle \in Join}{VS \uplus \{\langle q, jId\rangle \rightarrow s_{jId}\}}$$

$$\frac{q = join(jId, l, c_l, r, c_r),\quad VS(q, l) = s_l,\quad VS(q, r) = \emptyset,\quad s_{jId} = \langle jId, \langle s_l, c_l\rangle\rangle \in Join}{VS \uplus \{\langle q, jId\rangle \rightarrow s_{jId}\}}$$

$$\frac{q = join(jId, l, c_l, r, c_r),\quad VS(q, l) = s_l, VS(q, r) = s_r,\quad s_{jId} = \langle jId, \langle s_l, c_l\rangle, \langle s_r, c_r\rangle\rangle \in Join}{VS \uplus \{\langle q, jId\rangle \rightarrow s_{jId}\}}$$

Рис. 5 Правила вывода
Fig. 5. Inference rules

По построению отображения и достаточных условий наличия ошибки, в случае если программа удовлетворяет введенным ранее предположениям и результаты проведенных ядром анализов корректны, то выполнимость формулы (2) всегда будет означать наличие дефекта в анализируемой программе.

3.3. Пример обнаружения ошибки доступа к буферу

```
1  int bar(int a, int b){
2    if (a₁ >= c₉){
3        // …
4    }
5    if (b)
6        a₂ = a₁ + c₁;
7    a₃ = phi(a₁, a₂);
8    return buf[a₃];
9  }
```

Рис. 6. Пример ошибки
Fig. 6. An example of error

Рассмотрим предложенный подход на примере поиска ошибки в функции `bar` из разд. 2 (см. рис. 6). Здесь для удобства вместо исходных переменных приведены идентификаторы значений.

В ходе символьного исполнения обрабатываются следующие события, перечисленные в табл. 1.

Табл. 1. События
Table 1. Events

Стр.	Событие
2	$const(c_9, 9)$
2	$assume(a_1, c_9, \geq)$
4	$join(a_1, \begin{matrix} a_1, (a_1 \geq 9), \\ a_1, (a_1 < 9) \end{matrix})$
6	$const(c_1, 1)$
6	$binaryOp(a_2, a_1, c_1, +)$
7	$join(a_3, \begin{matrix} a_1, (b = 0), \\ a_2, (b \neq 0) \end{matrix})$
$a_1, a_2, a_3, c_1, c_9, b \in VId$	

В результате для идентификатора a_3 перед инструкцией доступа ac: `buf`$[a_3]$ значение $s_6 = VS(ac, a_3)$ можно представить в виде графа на рис. 7.

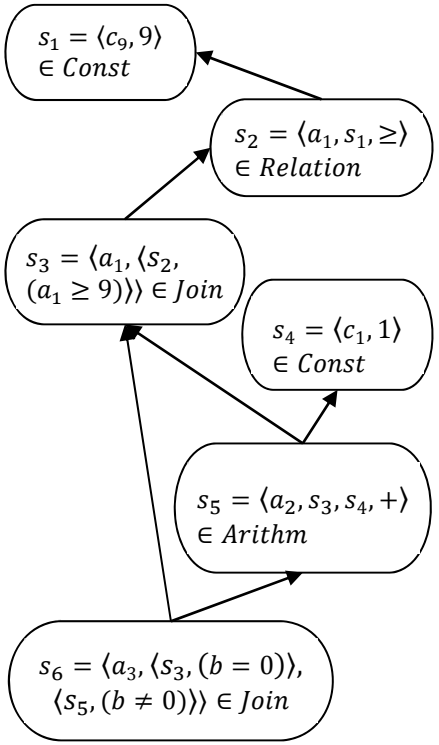

Рис.7. Граф значения $s_6 = VS(ac, a_3)$
Fig. 7. Graph of value $s_6 = VS(ac, a_3)$

Обрабатывается инструкция ac доступа к буферу на строке 8. Т.к. размер буфера равен 10, то необходимо проверить условие $a_3 \geq 10$.

$$NotLess(ac, a_3, 10) = HB(VS(ac, a_3), 10) = HB(s_6, 10).$$

Значение a_3 получилось из слияния двух значений ($s_6 = VS(ac, a_3))\rangle\rangle \in Join$), поэтому нужно проверить каждое из них с учётом условий слияния:

$$HB(s_6, 10) = \bigvee \begin{matrix} (a_3 = a_1) \wedge HB(s_3, 10) \wedge (b = 0) \\ (a_3 = a_2) \wedge HB(s_5, 10) \wedge (b \neq 0) \end{matrix}$$

Рассмотрим вторую ветвь. Значение в этой ветви является суммой двух значений ($s_5 = \langle a_2, s_3, s_4, +\rangle$). В общем случае для произвольной суммы нижние границы каждого из слагаемых неизвестны (т.к. в графе их значений могут быть узлы $Join$, а совместность условий путей еще не анализировалась), поэтому обозначим эти границы некоторыми вспомогательными переменными. Тогда искомое условие будет иметь вид:

$$HB(s_5, 10) = (a_2 = a_1 + c_1) \wedge \left(\exists \widetilde{a_1} \exists \widetilde{c_1}\, HB(s_3, \widetilde{a_1}) \wedge HB(s_4, \widetilde{c_1}) \wedge (\widetilde{a_1} + \widetilde{c_1} \geq 10)\right).$$

Т.к. c_1 это просто константа 1, то условие для неё выглядит тривиально:

$$s_4 = \langle c_1, 1\rangle \in Const \Rightarrow HB(s_4, \widetilde{c_1}) = (c_1 = 1) \wedge (1 \geq \widetilde{c_1}).$$

Для значения a_1 информация о значении есть только на одном из путей, сливающихся перед инструкцией на строке 13, поэтому:

$$HB(s_3, \widetilde{a_1}) = HB(ac, s_2, \widetilde{a_1}) \wedge (a_1 \geq 9).$$

Аналогично для первой ветви $HB(s_6, 10)$ можно записать:

$$HB(s_3, 10) = HB(s_2, 10) \wedge (a_1 \geq 9).$$

Про значение a_1, пришедшее с ветки истинности условия точно известно, что для него это условие выполнено, т.е. $(a_1 \geq c_9)$. Очевидно, что, если $c_9 \geq \widetilde{a_1}$, то $a_1 \geq \widetilde{a_1}$. Отсюда получаем:

$$HB(s_2, l') = (a_1 \geq c_9) \wedge HB(ac, c_9, \widetilde{a_1}) = (a_1 \geq c_9) \wedge (c_9 = 9) \wedge (9 \geq \widetilde{a_1}).$$

Аналогично можно развернуть оставшиеся выражения и вычислить итоговое условие. Получившаяся формула (2) будет выполнима, т.к. она верна при следующих значениях переменных:

Табл. 2. Модель для условия переполнения
Table 2. A model for condition of overflow

Переменная	c_1	c_9	$\widetilde{a_1}$	$\widetilde{c_1}$	a_3	a_2	a_1	b
Значение	1	9	9	1	10	10	3	1

Чтобы получить ошибочный путь необходимо подставить полученные значения в условия в узлы типа $Join$. В результате подстановки получим, что любой путь, для которого выполнено $(a_1 \geq 9)$ и $(b \neq 0)$ будет ошибочным. В данном случае этому условию удовлетворяет единственный путь (2)-(3)-(4)-(5)-(6)-(7)-(8), и он действительно содержит ошибку доступа к буферу. В соответствующем подграфе значения s_6 на рис. 7 ребра выделены жирным.

4. Реализация детектора

На основе рассмотренного подхода в инструменте статического анализа Svace был реализован межпроцедурный путе- и контекстно-чувствительный детектор ошибок доступа к буферу. 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.

В Svace и ранее имелось несколько межпроцедурных, но не чувствительных к путям детекторов выхода за границы массива. Преимуществами новой реализации, благодаря которым удается обнаружить новые типы ошибок, являются:

- чувствительность к путям, позволяющая обнаруживать ошибки, характеризующиеся последовательностью рёбер (более, чем одного);
- отслеживание взаимосвязей переменных, (включая арифметические операции, бинарные отношения между переменными в условиях перехода, значения с условиями в точках слияний), позволяющие строить цепочки из таких взаимосвязей для доказательства переполнения;
- поддержка инструкций преобразования типов (практика показала, что округление информации о результате преобразования в любую из сторон вместо тщательной обработки приводит к появлению существенного количества либо ложных срабатываний, либо пропущенных ошибок).

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

Для достижения хороших показателей кроме описанных в данной статье подходов необходима также поддержка межпроцедурного анализа, рассмотрение механизмов которого не вошло в данную работу. Поэтому, подробные результаты работы детекторов, а также сравнение с другими работами в этой области будет приведено в следующей статье. Здесь рассмотрим пример внутрипроцедурной ошибки, обнаруженной при анализе проекта Android-5.0.2. Слева на рис.8 приведён фрагмент исходного кода, а справа – трасса срабатывания детектора (последовательность событий записывается снизу-вверх). Здесь анализатор сообщает пользователю, что на некоторой итерации `ci` может равняться 2 исходя из сравнения на строке 8, и тогда, если цикл не завершится, на следующей итерации произойдет переполнение буфера `indices`.

1	`for (ci = 0; ci < folder->NumCoders;`	Array 'indices' of size 3 is accessed by 3 at line 6. This may lead to buffer overflow. • Buffer overflow at line 6. • Add: ci + 1 >= 3 at line 1. • Variable ci may be equal to 2 at line 8.
2	`ci++) {`	
3	`// …`	
4	`if (folder->NumCoders == 4) {`	
5	`UInt32 indices[] = { 3, 2, 0 };`	
6	`si = indices[ci];`	
7	`//…`	
8	`if (ci == 2) {`	
9	`//…`	
10	`}`	
11	`}`	
12	`}`	

Рис.8. Пример реального срабатывания
Fig. 8. An example of real operation

Список литературы

[1]. CVE and CCE Statistics Query Page. https://web.nvd.nist.gov/view/vuln/statistics

[2]. A. One, "Smashing the Stack for Fun and Profit", Phrack Magazine, Volume 7, Issue 49, November 1996.

[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=http://dx.doi.org/10.1145/2345156.2254088

[6]. 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 г., стр. 59-86. DOI: 10.15514/ISPRAS-2015-27(5)-5.

[7]. 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.

[8]. 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, 2016г.

Statically detecting buffer overflows in C/C++

[1,2]I. Dudina <eupharina@ispras.ru>

[1]V. Koshelev <vedun@ispras.ru>

[1]A. Borodin < alexey.borodin@ispras.ru >

[1]ISP RAS, 25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation;

[2]CMC MSU, CMC faculty, 2 educational building,

MSU, Leninskie gory str., Moscow 119991, Russian Federation

Abstract. The paper describes a static analysis approach for buffer overflow detection in C/C++ source code. This algorithm is designed to be path-sensitive as it is based on symbolic execution with state merging. For now, it works only with buffers on stack or on static memory with compile-time known size. We propose a formal definition for buffer overflow errors that are caused by executing a particular sequence of program control-flow edges. To detect such errors, we present an algorithm for computing a summary for each program value at any program point along multiple paths. This summary includes all joined values at join points with path conditions. It also tracks value relations such as arithmetic operations, cast instructions, binary relations from constraints. For any buffer access we compute a sufficient condition for overflow using this summary for index variable and the reachability condition for the current function point. If this condition is proved to be satisfiable by an SMT-solver, we use its model given by the solver to detect error path and report the warning with this path. This approach

was implemented for Svace static analyzer as the new buffer overflow detector, and it has found a significant amount of unique true warnings that are not covered by the old buffer overflow detector implementations.

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

DOI: 10.15514/ISPRAS-2016-28(4)-9

For citation: Dudina I., Koshelev V., Borodin A. 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

References

[1]. CVE and CCE Statistics Query Page. https://web.nvd.nist.gov/view/vuln/statistics

[2]. A. One, "Smashing the Stack for Fun and Profit", Phrack Magazine, Volume 7, Issue 49, November 1996.

[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=http://dx.doi.org/10.1145/2345156.2254088

[6]. 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, 2015, vol. 27, issue 5, pp. 59-86 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)- 5.

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

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