Об одном подходе к анализу строк в языке Си для поиска переполнения буфера

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

Аннотация. Ошибки при работе с библиотечными функциями обработки строк в языке Си являются частой причиной переполнения буфера, что в свою очередь нередко приводит к отказу в обслуживании, некорректной работе программы или появлению эксплуатируемой уязвимости. Одним из способов устранения различных ошибок на стадии разработки программы является статический анализ. Существующие методы статического анализа, ориентированные на работу со строками, либо не обеспечивают должный уровень истинных срабатываний, либо пропускают большое количество ошибок, либо неприменимы к промышленным программам большого размера, либо реализованы в рамках закрытых инструментов. Для наиболее полного покрытия дефектов в реальных программах необходимо обнаруживать ошибки, происходящие лишь на некоторых путях выполнения и не определяемые единственной точкой программы, и, кроме того, находить ошибки, связанные с некорректным использованием не только библиотечных, но и пользовательских функций. Целью данного исследования является построение алгоритма поиска ошибок при работе со строками, удовлетворяющего этим свойствам, ограничению на количество ложных срабатываний (не более 40%), применимого к любым программам на языке Си и масштабирующегося на проекты из нескольких миллионов строк. Для решения этой задачи был использован ранее разработанный подход символьного исполнения с объединением состояний, который был адаптирован для поддержки строковых операций. На основе алгоритма отслеживания операций с целыми числами был предложен алгоритм отслеживания длин строк. Разработанный алгоритм реализован в качестве одного из детекторов семейства детекторов переполнения буфера в рамках инструмента статического анализа Svace. В результате на тестовом наборе Juliet test suite на тестах, связанных с переполнением правой границы буфера, покрытие срабатываниями увеличилось с 15,4% до 41,5%, при этом не было выдано ни одного ложного предупреждения. По сравнению с известным анализатором Infer на наборе Juliet инструмент Svace без поддержки строк показывает приблизительно те же результаты, за исключением случая сложных циклов, а связанные со строками переполнения Infer, как правило, не находит.

Ключевые слова: статический анализ; символьное исполнение; анализ строк

DOI: 10.15514/ISPRAS-2018-30(5)-3

Для цитирования: Дудина И. А., Малышев Н. Е. Об одном подходе к анализу строк в языке Си для поиска переполнения буфера. Труды ИСП РАН, том 30, вып. 5, 2018 г., стр. 55-74. DOI: 10.15514/ISPRAS-2018-30(5)-3

1. Введение

Важным случаем ошибки переполнения буфера является переполнение при работе со строками. Строка в языке Си представляет собой массив символов, концом строки считается позиция ближайшего к началу нулевого элемента. Особенностью таких дефектов переполнения является тот факт, что работа со строками в языке Си преимущественно осуществляется с помощью специальных библиотечных функций. При этом, как правило, происходит доступ к элементам массива по различным индексам, наибольший из которых может быть равен длине строки. Это поведение само по себе небезопасно тогда, когда невозможно гарантировать, что длина строки заведомо меньше размера отведённого под неё массива. В таких случаях используют «безопасные» версии функций, дополнительно принимающие в качестве параметра число, с помощью которого ограничивается максимальный индекс доступа к строке, но даже такой подход не может гарантировать отсутствия ошибок.

Одним из способов устранения различных дефектов на стадии разработки программы является статический анализ. Существующие методы статического анализа, ориентированные на работу со строками, либо не обеспечивают должный уровень истинных срабатываний, либо пропускают большое количество ошибок, либо неприменимы к промышленным программам большого размера, либо реализованы в рамках закрытых инструментов. Для наиболее полного покрытия дефектов в реальных программах необходимо обнаруживать ошибки, происходящие лишь на некоторых путях выполнения и не определяемые единственной точкой программы, и, кроме того, находить ошибки, связанные с некорректным использованием не только библиотечных, но и пользовательских функций. Для выполнения этих требований анализ должен быть межпроцедурным и чувствительным к путям.

В данной работе предлагается подход к анализу строк в языке Си на основе символьного исполнения с объединением состояний. Рассматриваемый метод был реализован в рамках инструмента статического анализа Svace [10]. Дальнейшее изложение организовано следующим образом. В разд. 2 приводится краткое описание базового алгоритма, включающее описание внутрипроцедурного анализа методом символьного исполнения, межпроцедурного анализа с помощью метода резюме и общего подхода к поиску переполнения буфера. Разд. 3 посвящён расширению алгоритма символьного исполнения и алгоритма поиска буфера для анализа строк

в языке Си. Разд. 4 содержит результаты тестирования реализации рассмотренного метода. Описание существующих методов статического анализа для поиска ошибок при работе со строками приводится в разд. 5. Разд. 6 завершает статью.

2. Общий подход к поиску дефектов переполнения

2.1. Символьное исполнение с объединением состояний

Алгоритм внутрипроцедурного анализа основан на подходе символьного исполнения с объединением состояний. Анализ производится над разверткой графа потока управления на несколько итераций. Абстрактное состояние в каждой точке программы включает в себя предикат достижимости этой точки π и абстрактные значения переменных и ячеек памяти.

Абстрактные значения программы представляются символьными выражениями, множество которых обозначим как SE. К таковым относятся:

1) константные битовые вектора фиксированного размера;
2) символьные переменные (их множество обозначим $S \subset SE$);
3) арифметические операции над символьными выражениями.

Если множество переменных обозначить как V, то соответствие переменных их абстрактным значениям задаётся отображением $\sigma : V \rightarrow SE$. Предикат достижимости, в свою очередь, записывается как свободная от кванторов формула в теории битовых векторов, где в качестве переменных выступают символьные переменных из множества S.

В начальном состоянии на входе в функцию предикат достижимости тождественно равен истине; значениям формальных параметров и ячейкам в памяти сопоставлены различные новые символьные переменные. Анализ осуществляется путём продвижения абстрактного состояния по рёбрам графа развёртки.

При прохождении инструкции, изменяющей значение некоторой переменной или ячейки памяти, её символьное выражение в отображении σ обновляется в соответствии с семантикой инструкции. При прохождении через инструкции ветвления обновляется предикат достижимости: происходит конъюнкция предиката π с условием выбранной ветки, которое вычисляется с помощью σ. Если в некоторой инструкцию входит больше одного ребра на графе, то происходит слияние соответствующих абстрактных состояний на входе. При этом предикат достижимости вычисляется как дизъюнкция условий с входных рёбер, каждое из которых представляет собой конъюнкцию соответствующего предиката достижимости и условий равенства значения после объединения значению на рассматриваемой ветке для всех переменных и ячеек памяти.

2.2. Поиск переполнения буфера в рамках одной функции

Задачей анализа является обнаружение таких путей на графе развёртки (назовём эти пути *ошибочными*), которые, во-первых, являются выполнимыми и, во-вторых, прохождение которых всегда (при любых возможных значениях входных переменных) приводит к ошибке переполнения буфера. Такой подход выбран с целью, с одной стороны, обнаруживать дефекты, для которых не существует единственной ошибочной точки программы, и в то же время не выдавать большое количество ложных предупреждений, связанных с неизвестными возможными контрактами анализируемых функций [7].

Для решения этой задачи в абстрактное состояние программы было добавлено частичное отображение $VS : SE \rightarrow Summary$, которое для некоторых целочисленных значений программы, представленных символьными выражениями, определяет соответствующий элемент из множества *Summary*.

Элементы множества *Summary* обобщают информацию о некотором значении в данной точке программы, которая может быть использована для обнаружения ошибки переполнения, если это значение используется в качестве индекса для доступа к буферу. В произвольной точке программы q, если для некоторого символьного выражения $x \in SE$ известно $VS(x) = s \in Summary$, то для значения s и произвольного символьного выражения $h \in SE$, не зависящего от входных параметров, можно построить формулы $NotLess(q, s, h)$ и $NotGreater(q, s, h)$ в теории битовых векторов, удовлетворяющие следующему условию: для любого конкретного пути выполнения функции от начала функции до точки q, если выполнена формула $NotLess(q, s, h)$ ($NotGreater(q, s, h)$), то для пройденного пути по ГПУ при любых возможных для этого пути значениях входных параметров всегда в точке q выполнено $x \geq h$ ($x \leq h$). Таким образом, если в точке *ac* доступа к буферу размером S символьное выражения для индекса в данной точке равно $i \in SE$, предикат пути равен π и выполнима формула $\pi \wedge NotLess(ac, i, S)$, то существует путь на ГПУ, проходящий через точку *ac* и такой, что:

1. он является выполнимым (так как для некоторого набора входных переменных истинна формула π);
2. всегда на этом пути в точке *ac* выполнено $i \geq S$ (так как выполнена формула $NotLess(ac, i, S)$), то есть происходит переполнение буфера.

Заметим, что найденный путь удовлетворяет определению ошибочного пути, а, значит, выполнимость формулы $\pi \wedge NotLess(ac, i, S)$ является достаточным условием ошибки. Данный факт позволяет свести задачу поиска ошибок к задаче построения как можно более слабых и удовлетворяющих указанным выше требованиям условий $NotLess(q, s, h)$ и $NotGreater(q, s, h)$.

Подробное рассмотрение элементов множества *Summary* и построения искомых условий для них выполнено в статье [8]. Здесь в качестве примера

отметим, что одним из типов элементов этого множества являются константы, которые отображением *VS* всегда переводятся в себя. Для произвольной константы $c \in SE$ искомым условием является формула $NotLess(q, c, h) = c \geq h$. Действительно, так как значения c и h не зависят от входных переменных, то для любого пути ГПУ условие $c \geq h$ выполнимо либо для всех наборов значений входных параметров, либо ни для одного.

2.3. Межпроцедурный анализ

Межпроцедурный анализ производится с помощью метода резюме. Граф вызовов программы приводится к ациклическому виду разрывом обратных рёбер, и затем все функции программы анализируются единожды в обратном топологическом порядке. В результате внутрипроцедурного анализа функции формируется и сохраняется для последующего использования так называемое «резюме» — краткое описание поведения функции в терминах самого анализа. Далее при анализе инструкций вызовов пользовательских функций в силу порядка анализа функций гарантируется (в отсутствие рекурсии), что вызываемая функция уже проанализирована, значит, для неё уже имеется резюме, которое применяется в точке вызова.

Для простоты будем считать, что в резюме функции записывается объединённое абстрактное состояние из состояний в точках возврата из функции. Рассмотрим в общих чертах, как при вызове функции происходит применение её резюме, т.е. с помощью сохранённого абстрактного состояния вызванной функции модифицируется текущее состояние в точке вызова. Это происходит в несколько этапов:

1. Символьным переменным, соответствующим в резюме формальным аргументам функции, ставятся в соответствие символьные выражения, соответствующие в точке вызова фактическим аргументам функции. Переменной, в которую в точке вызова сохраняется результат функции, ставится в соответствие символьное выражение, соответствующее в резюме возвращаемому значению.
2. Обновляются отображение σ и предикат π в точке вызова. Символьные выражения «мигрируют» в контекст вызываемой функции рекурсивно, базой рекурсии является соответствие символьных переменных и символьных выражений, полученное в п.1.
3. При сопоставлении символьных выражений мигрируют также значения *VS* для этих выражений [8].

Для организации межпроцедурного поиска переполнения буфера в первую очередь необходимо поддержать случай межпроцедурного вычисления индекса. Для этого реализовано отслеживание зависимостей между целочисленными параметрами функции и возвращаемого и изменяемых внутри функции целочисленных значений (с учётом условий путей).

Кроме этого, также важно учесть возможность межпроцедурного доступа к буферу. Для инструкций доступа, корректность которых возможно проверить только во внешнем контексте (буфер и/или индекс вычисляются из параметров), в резюме функции добавляется условие ошибки. Условие при применении резюме мигрирует в контекст вызова, где либо проверяется (если информации уже достаточно), либо снова записывается в резюме, если проверить его по-прежнему можно только в вызываемой функции.

3. Анализ строк

Поддержка строковых операций будет заключаться, во-первых, в расширении абстрактного состояния отображением, задающим длину каждой строки, и учётом его значений при построении предиката π. Это позволит обнаруживать несовместные из-за ограничений на длины строк пути, что сократит количество ложных срабатываний не только для детектора переполнения буфера при работе со строками (для которого анализ выполнимости таких путей особенно критичен), но и для остальных чувствительных к путям детекторов.

Во-вторых, предлагается расширить отображение *VS* для обнаружения ошибок при работе со строками. Это позволит использовать уже имеющийся механизм анализа целочисленных значений для анализа длин строк.

3.1. Расширение абстрактного состояния для работы со строками

Содержимое каждой строки в абстрактном состоянии представлено одним символьным выражением, означающем длину данной строки. Выбор такой абстракции, с одной стороны, поможет найти больше ошибок и не выдавать ложных предупреждений на невыполнимых путях, предикаты которых содержат условия на длины строк. С другой стороны, добавление всего лишь одного символьного выражения для каждой строки не приведёт к значительному увеличению размера абстрактного состояния.

Таким образом, к абстрактному состоянию программы добавляется новое отображение $Slen : P \rightarrow SE$, где P — множество переменных, которые могут указывать на строку с точки зрения языка Си (например, можно выбрать множество всех переменных указательного типа).

В начальном состоянии программы каждой переменной из P сопоставляется новая символьная переменная. Строковым литералам сопоставляются константы, равные значению длин этих литералов в беззнаковом представлении.

Расширяются передаточные функции для инструкций присваивания переменных из множества P, арифметических операций над ними, инструкций доступа к массиву.

Также среди инструкций вызова функции отдельно рассматриваются инструкции вызова библиотечных функций работы со строками. Рассмотрим соответствующую передаточную функцию на примере функции `strncpy`.

$$\text{STRNCPY}\frac{\begin{array}{c} Slen_{in} \vdash \texttt{src} \to ls \quad Slen_{in} \vdash \texttt{dst} \to ld \quad \sigma \vdash n \to vn \\ \pi' = \bigvee \begin{array}{l} vn >_u ls \ \wedge \ lr = ls \\ vn \leqslant_u ls \ \wedge \ vn \leqslant_u ld \ \wedge \ lr = ld \\ vn \leqslant_u ls \ \wedge \ vn >_u ld \ \wedge \ lr \geqslant_u vn \end{array} \end{array}}{Slen_{out} = Slen_{in}\{\texttt{dst} \mapsto lr\} \quad \pi_{out} = \pi_{in} \wedge \pi'}$$

Данная функция принимает три параметра: адрес `dst`, куда копируются данные, адрес `src`, откуда копируется строка, и целочисленное значение `n`, определяющее наибольшее количество скопированных байт. Пусть на ребре, входящем в инструкцию вызова данной функции, значения отображения *Slen* для строк `dst` и `src` равны *ld* и *ls* соответственно, символьным выражением для переменной `n` является *vn*. Тогда на выходном ребре из этой инструкции для значения длины `dst` выберем новую символьную переменную *lr*, а условия на её значения добавим к предикату точки.

Для определения значения *lr* следует рассмотреть три случая. Если значение переменной `n` больше длины `src` (*vn* > *ls*), то длина `dst` после вызова будет равняться длине `src` (*lr* = *ls*). Если значение `n` не больше длины `src` (*vn* ≤ *ls*), то среди первых `n` байт строки `src` заведомо нет нулевого, и здесь возможны два случая. Если значение `n` также не превосходит длину `dst` (*vn* ≤ *ld*), то длина `dst` останется прежней (*lr* = *ld*), так как положение ближайшего к началу строки нулевого байта не изменится. В противном же случае про длину `dst` можно сказать лишь, что она заведомо не меньше `n`. Аналогичные построения можно провести и для других функций работы со строками.

Связь между значениями целочисленных переменных и длинами строк возникает при вызове функций, вычисляющих длину строки, и работе с массивами посимвольно. Так, например, при обработке инструкции `x = strlen(str)` символьное выражение для длины строки `str` копируется для переменной `x`.

$$\text{STRLEN}\frac{Slen_{in} \vdash \texttt{str} \to ls}{\sigma_{out} = \sigma_{in}\{x \mapsto ls\}}$$

При присваивании нового значения в элемент массива `str[i] = x` возможны два случая: если присваиваемое значение равно нулю и индекс меньше текущей длины, то длина строки `str` станет равна `i` либо не изменится в противном случае.

$$\text{BUFASSIGN}\frac{\begin{array}{c} Slen_{in} \vdash \texttt{str} \to ls \quad \sigma \vdash \texttt{x} = vx \quad \sigma \vdash \texttt{i} = vi \\ \pi' = \bigvee \begin{array}{l} (vx \neq 0 \ \vee \ ls <_u vi) \ \wedge \ lr = ls \\ (vx = 0 \ \wedge \ ls \geqslant_u vi) \ \wedge \ lr = vi \end{array} \end{array}}{Slen_{out} = Slen_{in}\{\texttt{str} \mapsto lr\} \quad \pi_{out} = \pi_{in} \wedge \pi'}$$

Поддержка нового отображения требуется и при объединении состояний в точках слияния: при объединении значений переменных из множества P объединяются не только их символьные выражения, но и длины их строк объединяются аналогичным способом с учётом условий на объединяемых ветках.

Для организации межпроцедурного анализа объединение отображений *Slen* для точек выхода из функции помещается в резюме функции. При применении резюме в точках вызова значения *Slen* из резюме транслируются в контекст вызывающей функции и сопоставляются соответствующим переменным из множества *P* по тому же принципу, как происходит трансляция отображения σ.

3.2. Расширение отображения *VS* для обнаружения ошибок при работе со строками

Идея рассматриваемого алгоритма заключается в том, чтобы анализировать длины строк аналогично тому, как анализируются целочисленные переменные, при этом обеспечивая совместимость этих двух анализов, то есть чтобы информация о длинах строк могла быть использована для анализа целочисленных значений и обратно. С этой целью отображение *VS* расширяется для символьных выражений, сопоставляемых строкам отображением *Slen.*

Как уже было сказано, все константы отображением *VS* переводятся в себя, поэтому для строковых литералов, значения *Slen* для которых являются константами, расширение *VS* происходит естественным образом.

Стандартные операции над строками моделируются по аналогии с операциями с целыми числами: так, вызов функции `strcpy` моделируется как присваивание длин, `strcat` — как сложение и т. п. Для проверки вызовов библиотечных функций на возможное переполнение строятся формулы *NotLess* и *NotGreater*, аналогичные тем, что используются при анализе инструкций доступа к буферу, но в качестве индекса используется максимальное смещение, по которому в соответствии с семантикой инструкции будет производиться доступ к строке.

Рассмотрим описанный в общих чертах подход на примере. В функции, приведённой на листинге 1, путь 2-3-4-5-6-7 является ошибочным. Чтобы обнаружить эту ошибку, необходимо доказать, что для некоторого пути в точке на строке 7 всегда `strlen(s)` $\geq$ `10` и `n` $>$ `10`.

```
1  void foo (int cond, int n) {
2      char s[100], dst[10]="";
3      if (cond)
4          strcpy(s, "very very long string");
5      if (n > 15)
6          //...
7      strncat(dst, s, n);
8  }
```

Листинг 1. Пример ошибки
Listing 1. Defect example

Первое условие заведомо выполнено, если в `s` была скопирована строка, длина которой всегда не меньше 10, т.е. если искомый путь проходит через точку 4 и `strlen("very very long string")>10`. Для точки 7 это можно записать как `cond ∧ 21>10`.

Второе условие выполнено, если было выполнено сравнение на строке 5 и `15≥10`. В точке 7 достаточным условием этого будет формула `n>15 ∧ 15≥10`.

С учётом того, что предикат достижимости точки 7 тождественно равен истине, итоговое достаточное условие ошибки будет иметь вид:

`cond ∧ 21>10 ∧ n>15 ∧ 15≥10.`

Данное условие построено таким образом, что наличие хотя бы одного удовлетворяющего ему набора значений входных переменных `n` и `cond` автоматически означает наличие ошибочного пути. Легко подобрать такие значения: `n = 16, cond = 42`. Подставив эти конкретные значения в условия переходов функции, можно получить искомый ошибочный путь.

4. Реализация и результаты

4.1. Реализация детектора

Задачей настоящего исследования было улучшение статического анализатора Svace [10] путём расширения чувствительного к путям анализа поддержкой строк языка Си. В предыдущей версии был реализован нечувствительный к путям анализ длин строк, основанный на интервальном анализе. Как следствие, информации, полученной в результате данного анализа, не было достаточно для организации чувствительного к путям поиска переполнения буфера при обработке строк. Кроме того, длины строк не учитывались для построения достаточных условий точки при символьном исполнении. Также имелся нечувствительный к путям детектор переполнения буфера при обработке строк, но он выдавал большое количество ложных срабатываний и не обеспечивал хорошее покрытие ошибочных ситуаций.

Для решения этих проблем была реализована поддержка строк при символьном исполнении с помощью алгоритма, описанного в разделе 3.1, и расширен имеющийся детектор переполнения буфера в соответствии с подходом, описанным в разделе 3.2. Рассмотренные алгоритмы были реализованы как для обычных строк, так и для строк с широкими символами (wide characters — символы, хранящиеся в типе `wchar_t` языка Си). Также был реализован нечувствительный к путям детектор ошибки, связанной с использованием обычной строки в качестве «широкой» строки, например при вызове функции `wcscpy`.

Для оценки масштабируемости новой версии Svace она была протестирована на проекте Android 5.0.2 без заметного ухудшения производительности по сравнению с базовой версией. Было выдано шесть истинных предупреждений, связанных с переполнением буфера при вызове обёртки над функцией `strcpy`.

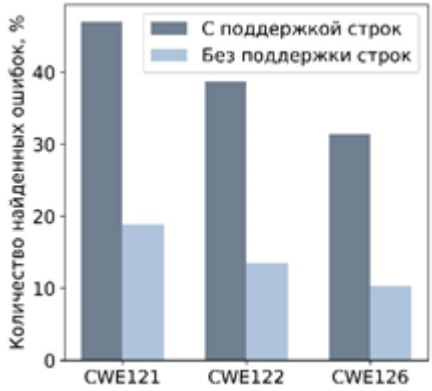

Рис. 1 Срабатывания анализатора на различных CWE
Fig. 1. Reported warnings by CWE

4.2. Тестирование с помощью Juliet Test Suite

Для тестирования использовался набор синтетических тестов Juliet Test Suite C/C++, разработанный в NSA's Center for Assured Software [11]. Для языков Си/Си++ в этом наборе представлен ряд тестов, размеченных по классификации CWE. К ошибке переполнения буфера среди всех групп набора относятся группы CWE 121 — «Stack-based Buffer Overflow» (переполнение буфера на стеке при записи), CWE 122 — «Heap-based Buffer Overflow» (переполнение буфера на куче при записи), CWE 124 — «Buffer Underwrite» (запись за левой границей буфера), CWE 126 — «Buffer Over-read» (чтение за правой границей буфера), CWE 127 — «Buffer Under-read» (чтение за левой границей буфера). Для задач настоящего исследования интерес представляли группы CWE 121, CWE 122, CWE 126, т.к. при работе со строками переполняется, как правило, правая граница массива.

Для каждого теста из набора также указан номер т. н. *варианта потока* (flow variant), который соответствует определенному виду потока управления и потока данных для данного теста. Среди вариантов потока управления рассматриваются различные случаи условий перехода, в том числе проверка глобальной переменной, условие из глобальной функции, различные виды оператора языка (`switch`, `while`, …). Варианты потока данных описывают различные случаи межпроцедурной и внутрипроцедурной передачи данных, такие как: пересылка через аргументы функции (в т. ч. по указателю, ссылке, как элемент массива или коллекции и пр.), через возвращаемое значение функции, через глобальную переменную. Некоторые из вариантов специфичны для языка Си++ и не применимы к тестам на Си.

Существует также классификация тестовых функций по *функциональным вариантам* (functional variants) — с учётом зависящих от конкретного типа дефекта особенностей тестового примера. В случае переполнения буфера к таким критериям можно отнести: имя библиотечной функции, вызов которой привёл к переполнению, тип элементов массива, способ выделения памяти под массив и т.п. Данная классификация, как правило, отражается в имени файла с тестом, например, `char_alloca_ncpy`.

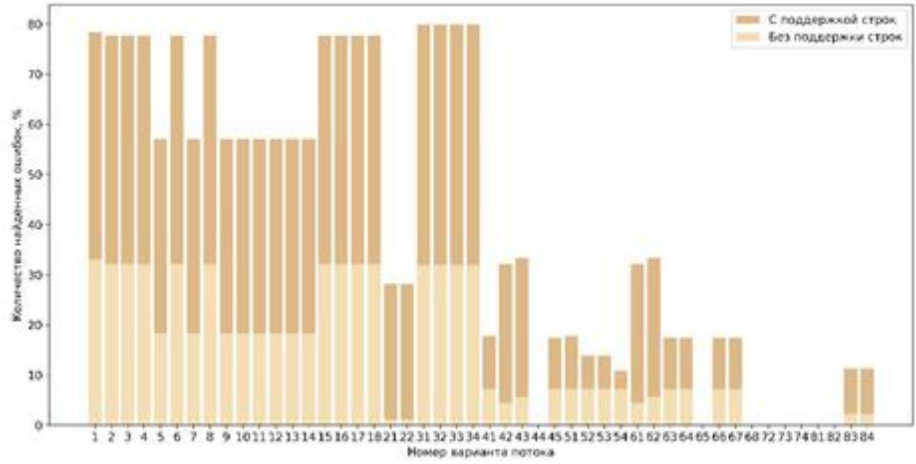

Рис. 2. Срабатывания анализатора на различных вариантах потока
Fig. 2. Reported warnings by flow variants

Изучение распределения тестов в интересных с точки зрения данного исследования группах показывает, что тесты распределены примерно равномерно между всеми вариантами потока. Почти треть тестов использует строки с широкими символами. Также значительное количество тестов проверяют использование библиотечных функций, таких как `memcpy`–подобные функции, функции обработки строк, использующие форматную

строку, и т. п. На рассмотренных группах тестов был дважды запущен анализатор Svace: с включенной поддержкой строк и без. Далее приведен анализ результатов тестирования.

Во-первых, ни на одном из запусков не было отмечено ложных срабатываний детекторов ошибки переполнения буфера на выбранных группах тестов.

Во-вторых, анализ результатов показал, что число срабатываний в интересующих группах увеличилось примерно в 2,5-3 раза (см. рис. 1). Общее число обнаруживаемых ошибок увеличилось с 15,4 % до 41,5 %. Также отдельно были рассмотрены функции, использующие тип `wchar_t`; число срабатываний детекторов на них увеличилось примерно в пять раз.

В-третьих, было произведено сравнение результатов внутри групп, соответствующих вариантам потока. Число срабатываний в 40 из 48 группах выросло в 2-10 раз (см. рис. 2). Оставшимся вариантам потока в обоих запусках соответствует нулевое количество срабатываний — эти варианты потока не поддерживаются (к таковым относятся, например, вызов функции по указателю, вызов виртуальной функции, передача данных через коллекции языка Си++).

Аналогичным образом рассматривались группы тестов, объединённые по функциональным вариантам. Результаты для 49 % от общего числа групп не изменились после реализации алгоритма — это в первую очередь тесты, в которых переполнение буфера не связано с работой со строками. При этом для 21 % групп было выдано нулевое количество срабатываний; в основном, это тесты, где используются функции с форматными строками (например, `snprintf`). Разбор форматной строки, необходимый для нахождения ошибок в этих тестах, реализован в инструменте Svace в отдельном детекторе, который не входит в рамки данной работы. Число срабатываний для остальных 51 % групп было нулевым до включения поддержки строк и стало равно 30-80 % от числа функций в группе. При этом лучшие результаты были получены на функциях, где ошибка возникает при использовании библиотечных функций (`strcat`, `strcpy`, `memcpy`, `memmove`), а худшие там, где ошибка возникает в цикле или при использовании функций с форматной строкой.

4.3. Сравнение с инструментом Infer

Для оценки эффективности метода было произведено сравнение со статическим анализатором Infer static analyzer [14, 15]. Этот инструмент разрабатывается компанией Facebook, имеет открытый исходный код и интенсивно развивается в настоящее время. Он активно используется в индустрии, например, в таких крупных IT-компаниях, как Amazon, Spotify, Uber, Mozilla Corporation [12].

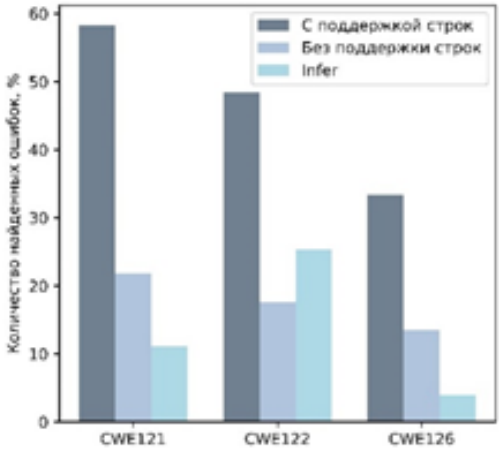

Рис. 3 Срабатывания анализаторов Svace и Infer на различных CWE
Fig. 3. Svace and Infer warnings by CWE

Для поиска ошибок переполнения буфера в Infer имеется экспериментальный детектор InferBO [13]. На данный момент для этого детектора заявлена только поддержка языка Си, поэтому из тестового набора, описанного выше, были исключены функции на языке Си++. На оставшихся функциях были протестированы версии Svace с и без поддержки строк в сравнении с анализатором Infer (см. рис. 3).

Так же, как и Svace, Infer не имеет ложных срабатываний на рассмотренном наборе. Процент обнаруживаемых ошибок составляет 15,2 % для Infer, 19,5 % для Svace без поддержки строк и 48,7 % при использовании описываемого метода. Говоря об отдельных группах CWE, можно отметить, что Infer лучше базовой версии Svace справляется с ошибками переполнения буфера на куче и хуже с ошибками переполнения буфера на стеке и чтения за границей буфера, значительно уступая версии с анализом строк во всех группах.

Только в трех вариантах потока из 37 ошибки не были обнаружены ни одним инструментом. Infer с различными вариантами потока справляется примерно одинаково, находя 15-20 % ошибок в 31 варианте и 0 % в оставшихся 7. В 15 вариантах базовая версия Svace превосходит Infer на 10-15 процентных пункта, на 1-2 пункта уступая в остальных вариантах с ненулевым числом найденных ошибок. В 23 вариантах версия Svace с поддержкой строк находит значительно больше ошибок чем Infer, на 30-60 процентных пунктов, выигрывая 0-10 процентных пунктов в остальных вариантах.

Табл. 1. Сравнение результатов Svace и Infer
Table 1. Warnings types detected by Svace and Infer

	Число функций в группе	**Svace с поддержкой строк, %**	**Svace без поддержки строк, %**	**Infer, %**
`Memcpy`	1 294	63,1	33,9	23,8
`Memmove`	1 258	64,5	34,9	24,5
`Loop`	1 222	22,9	16,5	23,7
`Cpy/Cat/Ncat`	1 532	65,7	0,0	0,0
`Ncpy`	652	56,7	15,0	17,8
`Other`	1 350	20,1	18,4	6,7

При сравнении результатов по вариантам ошибки в целом наблюдается следующая картина. Практически все функциональные варианты, покрываемые Infer, обнаруживает и базовая версия Svace. При этом на большинстве этих вариантов Infer показывает на 10-15 процентных пунктов лучший результат, в основном для тех функций, которые имеют дополнительную пометку CWE 805 – «Buffer Access with Incorrect Length Value» (доступ к буферу с некорректным значением длины). В то же время Svace обнаруживает несколько функциональных вариантов, которые Infer не обнаруживает вовсе, за счет чего имеет немного более высокий общий процент обнаруживаемых ошибок. Версия с поддержкой строк обнаруживает более чем в 2 раза больше различных функциональных вариантов по сравнению с Infer.

В табл. 1 приведены результаты инструментов в укрупнённых группах функциональных вариантов. Данные группы были получены путем объединения функциональных вариантов по последнему слову их имени. Например, функциональный вариант `char_type_overrun_memcpy` попадет в группу `memcpy`. Это было сделано для краткости в связи с очень большим количеством функциональных вариантов. В группе `loop` ошибка проявляется во время обращений к буферу по индексу, вычисляемому в цикле и превышающему его размер. В группах `memcpy`, `memmove` ошибка возникает при вызове соответствующих библиотечных функций с некорректным параметром размера буфера, в группе `Cpy/Cat/Ncat` — при вызове функций `strcpy`, `strcat`, `strncat`. Отдельно была рассмотрена группа `ncpy` с библиотечной функцией `strncpy`, для обнаружения переполнения первого аргумента которой не нужно знать длины строк-аргументов. Функциональные варианты, не попавшие ни в одну из перечисленных групп, были объединены в группу `other`. Они содержат небольшое количество функций и характеризуются относительно низким количеством срабатываний каждого анализатора.

Из таблицы видно, что Infer лучше обрабатывает циклы и чуть лучше, чем Svace без поддержки строк, находит ошибки, связанные с `strncpy`, при этом показывая более слабый результат на функциях `memmove` и `memcpy`. Также можно заключить, что поддержка строк в Infer отсутствует.

Следует отметить, что тестовый набор Juliet не является репрезентативной выборкой типов дефектов, встречающихся в коде реальных программ, поэтому на основании только лишь результатов тестирования нельзя сделать исчерпывающих выводов о качестве и сравнительной практической пользе рассматриваемых инструментов.

5. Обзор существующих решений

Одним из первых статических анализаторов, ориентированных на анализ строк для поиска переполнения, стал созданный в 2000 г. инструмент BOON [1]. Анализатор моделирует каждую строку парой переменных, определяющих размер выделенной памяти и длину строки. В процессе обхода абстрактного синтаксического дерева исходной программы строится система целочисленных интервальных неравенств. После решения полученной системы для каждой строки анализируется условие безопасности: если не доказано, что длина строки меньше размера буфера, то выдается предупреждение. Анализ является межпроцедурным, нечувствительным к потоку и контексту вызова. Исследования независимых авторов [2, 3] отмечают хорошую производительность, но при этом большое количество ложных срабатываний. Количество обнаруживаемых ошибок также существенно ниже по сравнению с другими детекторами переполнения буфера, что отчасти может быть объяснено узкой специализацией инструмента как анализатора строковых операций.

В 2003 г. был создан инструмент CSSV [4], целью которого является обнаружение всех ошибок переполнения буфера в программе на языке CoreC (подмножество Си) с небольшим количеством ложных срабатываний. Анализ осуществляется отдельно для каждой функции, межпроцедурный анализ организован с помощью аннотаций, предоставленных пользователем. CSSV преобразует исходную программу в программу над целочисленными значениями и консервативно проверяет её на наличие ошибок. Необходимость написания аннотаций пользователем существенно ограничивает применимость данного инструмента. К недостаткам также можно отнести отсутствие масштабируемости на большие программы.

Подход, предложенный в работе [5], как и CSSV, гарантирует обнаружение всех ошибок переполнения при работе со строками. Он основан на абстрактной интерпретации с использованием полиэдров в качестве абстракции для моделирования возможных значений размеров массивов и длин строк. В работе [6] предлагается в качестве более точной абстракции использовать *тропические* полиэдры — аналог выпуклых полиэдров в тропической алгебре. Такой подход позволяет вычислять более точные

инварианты над длинами строк в результате строковых преобразований. Данные методы позволяют гарантировать обнаружение всех ошибок рассматриваемого типа, однако плохо масштабируются на программы большого размера.

Ряд современных промышленных анализаторов, таких как Coverity Prevent, Klocwork, HP Fortify, включают в себя детекторы переполнения буфера при работе со строками, однако используемые ими алгоритмы закрыты, что затрудняет суждения о качестве этих детекторов.

Перспективными в контексте анализа строк для переполнения буфера представляются решатели с поддержкой строк [9]. В настоящее время они получили широкое применение для анализа значений строк в веб-приложениях, где они позволяют обнаруживать и автоматически анализировать процедуры проверки входных данных («санитайзеры») для устранения потенциальных уязвимостей. Способность данных инструментов анализировать не только размер, но и содержимое строк может быть также полезна и при анализе переполнения буфера, однако число реально встретившихся нам на практике примеров, для которых требуется выносить суждения о содержимом строк, относительно невелико.

6. Заключение

В статье представлен метод поиска переполнений буфера, происходящих при работе со строками языка Си. Метод основан на символьном исполнении с объединением состояний, является межпроцедурным, обладает чувствительностью к путям выполнения и контекстам вызовов. Идея метода заключается в расширении для операций со строками предложенного в предыдущих работах подхода [7, 8], который заключается в отслеживании операций с целочисленными значениями и построению на этой основе достаточных условий возникновения ошибки.

Разработанный метод реализован в статическом анализаторе Svace и обладает теми же показателями масштабируемости и точности анализа, что и основные детекторы ошибки переполнения буфера. В частности, для тестов из пакета Juliet поиск переполнений строк позволил резко увеличить покрытие (число находимых ошибок) в части, относящейся к переполнению буфера,— рост составил около трёх раз. Оставшаяся не найденной часть ошибок в основном связана с недостатками не предложенного метода как такового, а основной части (ядра) анализатора Svace — это анализ сложных циклов и коллекций языка Си++. Работы над улучшением анализатора в этих направлениях ведутся в настоящее время.

Сравнение результатов тестирования на пакете Juliet с результатами статического анализатора Infer позволил сделать вывод, что в Infer поддержка строк при поиске переполнений буфера не реализована, а обычные переполнения ищутся примерно с тем же качеством, что и в базовой версии Svace. Infer лучше обрабатывает тестовый исходный код со сложными

циклами, что подтверждает наш вывод о необходимости доработки Svace в этом направлении.

С точки зрения доработки предложенного метода перспективным направлением авторам видится интеграция решателей с поддержкой строк (Z3str2, Z3str3, CVC4 и т.п.), что позволит находить ошибки переполнения, для которых нужно выносить суждения о содержимом строк.

Список литературы

[1]. Wagner D., Foster J., Brewer E., Aiken A. A first step towards automated detection of buffer overrun vulnerabilities. In Proc. of the Network and Distributed System Security Symposium, 2000, pp. 3-17.

[2]. Zitser M., Lippmann R., Leek T. Testing static analysis tools using exploitable buffer overflows from open source code. ACM SIGSOFT Software Engineering Notes, vol. 29, issue 6, 2004, pp. 97-106.

[3]. Kratkiewicz K. Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code. Master's Thesis, Harvard University, 2005.

[4]. Dor N., Rodeh M., Sagiv M. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. ACM SIGPLAN Notes, vol. 38, 2003, pp. 155-167.

[5]. Simon A., King A. Analyzing String Buffers in C. Algebraic Methodology and Software Technology, vol. 2422, 2002, pp. 365-379.

[6]. Allamigeon X. Static analysis of memory manipulations by abstract interpretation. Algorithmics of tropical polyhedra, and application to abstract interpretation. PhD Thesis, École Polytechnique, 2009.

[7]. 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 г., стр. 149–168, 2016. DOI: 10.15514/ISPRAS-2016-28(4)-9

[8]. Dudina I. A., Belevantsev A. A. Using static symbolic execution to detect buffer overflows. Programming and Computer Software, vol. 43, issue 5, pp. 277–288, 2017. DOI: 10.1134/S0361768817050024

[9]. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Berzish, M., Dolby, J., Zhang, X. Z3str2: an efficient solver for strings, regular expressions, and length constraints. Formal Methods in System Design, vol. 50, 2017, pp.249-288.

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

[11]. Juliet Test Suite v1.2 for C/C++. User Guide. Center for Assured Software National Security Agency, December 2012.

[12]. Infer static analyzer Infer. URL: https://fbinfer.com/ (Дата обращения: 21.09.2018)

[13]. Inferbo: Infer-based buffer overrun analyzer. URL: https://research.fb.com/inferbo-infer-based-buffer-overrun-analyzer/ (Дата обращения: 21.09.2018)

[14]. Calcagno C., Distefano D. et al. Moving Fast with Software Verification. Lecture Notes in Computer Science, vol. 9058, 2015, pp. 3-11.

[15]. Calcagno C., Distefano D., O'Hearn P., Hongseok Y. Compositional Shape Analysis by means of Bi-Abduction. In Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '09), 2009, pp. 289-300.

An approach to the C string analysis for buffer overflow detection

I. A. Dudina <eupharina@ispras.ru>
N. E. Malyshev <neket@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia

Abstract. Many buffer overrun errors in C programs are caused by erroneous string manipulations. These can lead to denial of service, incorrect computations or even exploitable vulnerabilities. One approach to eliminate such defects in the course of program development is static analysis. Existing static analysis methods for analyzing strings either produce many false positives, miss too many errors, scale poorly, or are implemented as a part of a proprietary software. To cover a significant amount of the real program defects it is necessary to detect errors that could happen only on a particular program path and cannot be defined by a single erroneous point. Also, it is essential to find misusage of library functions and user functions. The aim of this study is to develop a detection algorithm that will cover such cases, will produce at most 40% false warnings, will be applicable to any C programs without any additional restrictions, and will scale up to millions of lines of code. We have extended our approach of symbolic execution with state merging to support string manipulations. Also we have developed a string overflow detector based on our buffer overflow approach with integer indices. The new algorithm was implemented in the Svace static analyzer. As a result, the coverage of the buffer-overflow related testcases from the Juliet test suite has increased from 15.4% to 41.5% with zero false positives. Also we have compared our Juliet results to those of the Infer static analyzer. The basic Svace version (without string support) is on par with Infer except for the flow variant of complex loops, whereas string-related buffer overflows are not detected by Infer.

Keywords: static analysis; static symbolic execution; string analysis

DOI: 10.15514/ISPRAS-2018-30(5)-3

For citation: Dudina I. A., Malyshev N. E. An approach to the C string analysis for buffer overflow detection. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 5, 2018, pp. 55-74 (in Russian). DOI: 10.15514/ISPRAS-2018-30(5)-3

References

[1]. Wagner D., Foster J., Brewer E., Aiken A. A first step towards automated detection of buffer overrun vulnerabilities. In Proc. of the Network and Distributed System Security Symposium, 2000, pp. 3-17.

[2]. Zitser M., Lippmann R., Leek T. Testing static analysis tools using exploitable buffer overflows from open source code. ACM SIGSOFT Software Engineering Notes, vol. 29, issue 6, 2004, pp. 97-106.

[3]. Kratkiewicz K. Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code. Master's Thesis, Harvard University, 2005.

[4]. Dor N., Rodeh M., Sagiv M. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. ACM SIGPLAN Notes, vol. 38, 2003, pp. 155-167.
[5]. Simon A., King A. Analyzing String Buffers in C. Algebraic Methodology and Software Technology, vol. 2422, 2002, pp. 365-379.
[6]. Allamigeon X. Static analysis of memory manipulations by abstract interpretation. Algorithmics of tropical polyhedra, and application to abstract interpretation. PhD Thesis, École Polytechnique, 2009.
[7]. Dudina I. A., Koshelev V. K., Borodin A. E. Statically detecting buffer overflows in C/C++. Trudy ISP RAN/Proc. ISP RAS, vol 28, issue 4, pp. 149–168, 2016. DOI: 10.15514/ISPRAS-2016-28(4)-9
[8]. Dudina I. A., Belevantsev A. A. Using static symbolic execution to detect buffer overflows. Programming and Computer Software, vol. 43, issue 5, pp. 277–288, 2017. DOI: 10.1134/S0361768817050024
[9]. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Berzish, M., Dolby, J., Zhang, X. Z3str2: an efficient solver for strings, regular expressions, and length constraints. Formal Methods in System Design, vol. 50, 2017, pp.249-288.
[10]. Borodin A., Belevantcev A. A static analysis tool Svace as a collection of analyzers with various complexity levels. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 111—134.
[11]. Juliet Test Suite v1.2 for C/C++. User Guide. Center for Assured Software National Security Agency, December 2012.
[12]. Infer static analyzer Infer. URL: https://fbinfer.com/ (Дата обращения: 21.09.2018)
[13]. Inferbo: Infer-based buffer overrun analyzer. URL: https://research.fb.com/inferbo-infer-based-buffer-overrun-analyzer/ (Дата обращения: 21.09.2018)
[14]. Calcagno C., Distefano D. et al. Moving Fast with Software Verification. Lecture Notes in Computer Science, vol. 9058, 2015, pp. 3-11.
[15]. Calcagno C., Distefano D., O'Hearn P., Hongseok Y. Compositional Shape Analysis by means of Bi-Abduction. In Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '09), 2009, pp. 289-300.