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

УДК 004.05

Проблемы поиска ошибок в пользовательских функциях, использующих системные вызовы ввода-вывода

***Ромащенко Н.С.**, магистр*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Программное обеспечение ЭВМ и информационные технологии

Научный руководитель: Филиппов М.В., доцент
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
irudakov@bmstu.ru

Введение

Ввод-вывод традиционно считается операцией, более всего подверженной влиянию различного рода ошибок. Во-первых, это связано с тем, что операции ввода-вывода взаимодействуют с некоторым интерфейсом, который представляет физический объект: участок физической памяти, внешнего носителя или сетевой адаптер. Во-вторых, операции ввода неизбежно взаимодействуют с пользовательскими данными и вынуждены взаимодействовать с сырым и еще необработанным вводом пользовательской информации.

В связи с этим осуществляющие ввод/вывод функции занимают особенное положение в тестировании кода. Однако еще одной особенностью функций ввода/вывода является зависимость мощности пространства состояний автомата работы функции от размера обрабатываемых данных. Для определения возможностей применения традиционных способов анализа кода к функциям рассматриваемого типа необходимо провести исследование.

1. Особенности функций ввода-вывода

В данном разделе более подробно описываются тип исследуемых функций ввода/вывода. Также раскрываются особенности функций этого типа, затрудняющих применение традиционных методов анализа, дается оценка мощности пространства состояний автомата функции рассматриваемого типа.

1.1 Исследуемые функции

Для определенности будем считать, что функция ввода/вывода:

- написана на языке программирования C;
- содержит в себе только код, осуществляющий ввод/вывод, а также код обработки ошибок и непредвиденных ситуаций;
- осуществляет ввод/вывод посредством определенных в стандарте POSIX системных вызовов write и read.

POSIX определяет сигнатуры указанных системных вызовов следующим образом:

```
ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
```

где:

- fd - файловый дескриптор для чтения/записи,
- buf - буфер для обрабатываемых данных,
- count - размер буфера.

В случае успешного завершения возвращается количество байтов, которые были записаны (ноль означает, что не было записано ни одного байта). В случае ошибки возвращается -1, а переменной errno присваивается соответствующее значение. Если count равен нулю, а файловый описатель ссылается на обычный файл, то будет возвращен ноль и больше не будет произведено никаких действий.

Предполагается, что функция ввода-вывода производит системный вызов до тех пор, пока он возвращает положительное значение.

1.2 Мощности пространств состояний функций ввода/вывода на примере функции чтения, использующей read

Отличительной особенностью функций ввода/вывода, использующих системные вызовы write и read, является непредсказуемость возвращаемого системным вызовом результата. В связи с тем, что заранее неизвестно, сколько байт будет передано/принято в результате системного вызова, необходимо организовывать обработку данных в цикле, пока не будет обработан массив целиком либо не будут предприняты особые действия в случае неудачи. Ниже приводится пример простой функции, осуществляющей системный вызов read:

```
ssize_t read_example(int fd, const void *buffer, size_t count) {
    int read_count = 0;
    while (true) {
        read_count = read(fd, buffer, count);
```

```
                if (( read_count < 0) && (errno == EAGAIN || errno == EINTR)){
                        continue;
                }
                else if (read_count < 0) {
                        /* ... */
                        /* л о г и к а  о б р а б о т к и  о ш и б о к */
                        /* ... */
                }
                else {
                        /* ... */
                        /* л о г и к а  о б р а б о т к и  п р о ч и т а н н ы х
д а н н ы х */
                        /* ... */
                }
        }
        return read_count;
}
```

При успешном завершении функции будет прочитано n ($n = count$) байт в буфер *buffer* в результате некоторых k системных вызовов read. Таким образом, количеством способов, которыми функция read_example может корректно отработать, является количество способов выбрать для каждого байта n_i последовательности длиной n k_j системный вызов из k попыток. Но при этом необходимо учесть, что вызовы совершаются последовательно, то есть учесть порядок следования байтов в сообщении.

Другими словами, необходимо выяснить, сколькими способами возможно разбить множество байт buffer на непересекающиеся подмножества, передаваемые единичным системным вызовом read.

Эта формулировка эквивалентна задаче разбиения натурального числа на сумму натуральных слагаемых, и ее решение называется *композицией* или *разложением* натурального числа. Композиция натурального числа n длины k равно количеству $(k-1)$-элементных подмножеств $(n-1)$-элементного множества, то есть биномиальному коэффициенту C_{n-1}^{k-1}. Для подсчета общего числа композиций числа n достаточно просуммировать эти биномиальные коэффициенты. В результате суммирования получим

2^{n-1} комбинаций. Подробнее доказательство последнего утверждения приводится в [1].

Таким образом, функция read_example может успешно отработать 2^{n-1}способами, и это означает, что мощность множества вершин графа состояний ассоциированного функции автомата также ограничивается снизу функцией 2^{n-1}, в зависимости от длины nвходного буфера.

1.3 Использование системных вызовов, отличных от read

Следует заметить, что рассмотрение предложенной фукнции read_example не ограничивает общность рассуждений, так как в расчетах нами использовались только возможные результаты работы системного вызова read. Следовательно, оценка мощности пространства состояний справедлива для всех функций подобного типа (во всяком случае, приведенная оценка является точной нижней гранью множества возможных значений мощностей).

Кроме того, аналогичные рассуждения можно применить к функциям, осуществляющим системные вызовы write, так как они принимают и возвращают одинаковые параметры, и системный вызов write так же может возвращать любые значения в диапазоне$[-1, n]$.

Стандартом POSIX также определены следующие системные вызовы ввода-вывода:

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

При значении параметра флагов flags, равному нулю, recv и send становятся аналогами read и write соответственно, а функции, использующие эти вызовы для работы с буфером также могут быть включены в рассмотрение на аналогичных условиях.

2. Проблемы классических подходов к анализу кода

В данном разделе описываются классические подходы к анализу кода и проблемы, с которыми они сталкиваются при попытке их применить к анализу функций описанного в предыдущем разделе типа.

2.1 Статический анализ

Статический анализ кода - широкое направление анализа программного обеспечения, особенностью которого является непосредственная обработка кода программа без какого бы то ни было его выполнения [2]. Детальный обзор трех из таких

анализаторов – PolySpace Verifier [3], Coverity Prevent [4] и Klocwork K7 [5], реализующих наиболее сложные виды анализа, проведен в [6]. Проверенные на практике правила корректности кода или шаблоны типичных ошибок переносятся в интегрированные среды разработки и постепенно становятся семантическими правилами языков программирования. Впоследствии их проверка возлагается уже на компиляторы, что привело к массовому внедрению статического анализа в промышленную разработку ПО.

Одним из основных применений статического анализа является поиск ошибок в программах на основе правил, составляемых разработчиками анализатора, а также формирование статистик – различных показателей, характеризующий код программы с той или иной стороны [7]. Статический анализатор хорошо справляется с поиском "человеческих" ошибок, то есть тех, которые случились по недосмотру и невнимательности программиста. К таким ошибкам относится и неправильная обработка исключительных ситуаций, которая является одной из основных в функциях ввода/вывода.

Результатом работы статического анализатора может являться детерминированный конечный автомат работы рассмотренной процедуры, или, например, список обнаруженных недостатков в коде, которые нарушают внедренные в анализатор правила. Основной проблемой применения статического анализа к рассматриваемому типу функций является показательная зависимость количества состояний автомата от длины обрабатываемых данных.

2.2 Динамический анализ

Динамический анализ, в противоположность статическому анализу, проводит исследование программы во время её исполнения. Исследование осуществляется путем выполнения набора тестов для выявления интересующего поведения функции. Обычно выполнение наборов тестов для функции осуществляется на виртуальном процессоре. Традиционным применением динамического анализа является обнаружение утечек памяти, неправильного выделения и освобождения памяти, обнаружение гонок, однако подход можно использовать и в целях установления корректности работы функции.

Как и статический анализ, динамический анализ страдает от зависимости количества состояний автомата работы функции от длины обрабатываемых данных, так как все состояния автомата должны быть покрыты тестами. Кроме этого, классическая генерация тестов на основе кода строится на идее создания пакета тестов объема не менее цикломатической сложности анализируемой подпрограммы [7]. Однако в случае с

подпрограммами, использующими системные вызовы ввода-вывода, существует дополнительная проблема. Если программист не учел все возможные типы результатов работы системного вызова и не отразил этого в коде, реальное цикломатическое число подпрограммы будет меньше количества концептуально разных возможностей работы подпрограммы.

2.3 Символьное выполнение

В связи с перечисленными недостатками возникает необходимость каким-то образом определить классы эквивалентности состояний для покрытия вместо покрытия каждого из состояний автомата функции.

Одним из таких способов может являться символьное выполнение (англ. *symbolic execution*) [8], которое широко используется в настоящее время для тестирования и анализа программ [9]. Символьное выполнение представляет из себя моделирование выполнения программы, при котором часть входных данных представляется в символьном виде.

Символ переменной обозначает множество значений входной переменной программы из области ее определения. Таким образом, каждое символьное выполнение эквивалентно выполнению программы на наборе конкретных тестовых значений входных переменных, что сокращает мощность множества создаваемых тестов, то есть фактически реализует идею разбиения пространства состояний подпрограммы на подмножества. Также символьным выполнением определяется альтернативная семантика выполнения программы – семантика символического выполнения для языка программирования, в котором объекты данных представлены в виде символов. Семантика задается путем расширения базовых конструкций языков программирования для работы с символьными значениями.

Итак, основным понятием символьного выполнения является символьное состояние (англ. *symbolic state*), которое включает в себя:

- Множество символьных значений переменных программы A = $\{a_1, a_2, a_3, \ldots\}$
- Условие реализуемости пути, *pc* (англ. *path condition*)
- Счетчик операторов программы

Каждое символьное выполнение программы начинается с инициализации условия реализуемости пути *pc* значением «истина». В каждой точке ветвления программы в *pc* добавляются предположения о входных переменных в порядке выбора между путями выполнения. Другими словами, по ходу выполнения *pc* аккумулирует те условия,

выполнению которых соответствует определенное подмножество состояний подпрограммы и соответствующий путь выполнения подпрограммы.

```
if(условие 1){
        /* блок 1*/
}
else {
        /* блок 2*/
}
```

Рассмотрим пример. Для приведенного куска кода вычисляется символьное условное выражение *q(B)*, где $B \subseteq A$, соответствующее условию в форме, определенной семантикой символьного выполнения. Затем, если *q(B)* определено, с учетом текущего *pc* формируется пара формул:

$$(1)\ pc \rightarrow q(B)$$

$$(2)\ pc \rightarrow q(B)$$

Если *pc* – истинна, то по свойству непротиворечивости только одна из формул (1) или (2) может быть истинной. Если истинна формула (1), то управление передается на блок1, если (2) – на блок2.

Если ни формула (1), ни формула (2) не являются истинной, существует набор значений входных переменных, удовлетворяющий *pc* и приводящий к выполнению ветви с блоком 1 ($pc \rightarrow q(B)$), а также существует набор, удовлетворяющий pc и приводящий к выполнению ветви else с блоком 2 ($pc \rightarrow q(B)$).

В подобном случае символьное выполнение программы должно быть разделено на два независимых маршрута, выполняемых параллельно. При выполнении ветви с блоком 1 в условие реализуемости пути *pc* добавляется следующее условие путем присваивания:

$$pc \leftarrow pc \wedge q(B)$$

При выполнении ветви else с блоком 2, соответственно, к *pc* добавляется условие

$$pc \leftarrow pc \wedge q(B)$$

Для пути выполнения программы *pc* никогда не принимает значение “ложь”, так как оно инициализируется значением “истина” и единственная операция над *pc* – присваивание $pc \leftarrow pc \wedge r$, где *r* это либо *q(B)*, либо *q(B)*.

Присваивание осуществляется только в случае дальнейшего следования по выбранному пути, т.е. на этом пути условие *r* истинно. Если условие *r* ложно, то и *pc* принимает значение “ложь” и не существует множества входных значений,

удовлетворяющих *pc*, в этом случае символьное выполнение пути программы завершается. Это позволяет отбросить недостижимые состояния исследуемой подпрограммы.

Таким образом *pc* аккумулирует ограничения на входные переменные, которые определяют уникальный путь выполнения в программе. В результате этого получается символическое описание путей выполнения программы в виде логического выражения над входными символьными переменными.

Подобное описание путей выполнения программ позволяет выбирать новые тестовые ситуации так, чтобы они покрывали другие пути и строить тесты с помощью техник разрешения ограничений (англ. *constraint solving*) [10].

С другой стороны, эта техника так же, как и предыдущие, подвержена проблеме неучтенного поведения системных вызовов ввода-вывода в коде подпрограммы. Тем не менее, она может рассматриваться как способ ликвидации проблемы комбинаторного взрыва в пространстве состояний исследуемой подпрограммы.

Заключение

В данной статье рассмотрены традиционные метода анализа кода и верификации программ и показано, что они сопряжены с определенными трудностями в применении к функциям, использующим системные вызовы ввода-вывода. Такие функции имеют показательную зависимость роста количества вершин графа состояний ассоциированного автомата от величины обрабатываемого массива, вследствие чего непосредственное применение традиционных способов анализа кода затруднительно. Показано, что вместо этого целесообразно рассмотреть возможность построения некоторых классов эквивалентности состояний автомата.

Список литературы

1. Сачков В. Н. Комбинаторные методы дискретной математики. М.: Наука, 1977. С. 241. 319 с.
2. Erik van Veenendaal. Standard glossary of terms used in Software Testing. International Software Testing Qualifications Board, 2007.
3. Static Analysis with Polyspace Products. Available at: http://www.mathworks.com/products/polyspace, accessed 15.05.2014.
4. Coverity. Available at: http://www.coverity.com/html/prod_prevent.html, accessed 15.05.2014.

5. Klocwork: Source Code Analysis Tools for Software Security & Reliability. Available at: http://www.iplbath.com/pdf/klocwork/KlocworkEmbeddedSystems.pdf, accessed 15.05.2014.
6. Emanuelsson and Ulf Nilsson. A Comparative Study of Industrial Static Analysis Tools. Electron. Notes Theory Computer Science. 217. 2008. P. 5-21.
7. Horst Zuse. Software Complexity. Walter de Gruyter, 1991.
8. Dr. A.J. Sobey. Basis Path Testing. Available at: http://users.csc.calpoly.edu/~jdalbey/206/Lectures/BasisPathTutorial/index.html, accessed 15.05.2014.
9. King J.C. Symbolic Execution and Program Testing // Journal of the ACM. 1976. P. 385–394.
10. Corina S. Pasareanu, Willem Visser. A survey of new trends in symbolic execution for software testing and analysis // Int J Softw Tools Technol Transfer, 2009. P. 339–353.