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

УДК 004.415.53

Разработка алгоритма поиска утечек ресурсов в программах на языке С

***Исаев Д.С.**, студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Программное обеспечение ЭВМ и информационные технологии»

Научный руководитель: Ломовской И.В., старший преподаватель
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
irudakov@bmstu.ru

Введение

Языки программирования с ручным управлением памятью требуют от программиста вручную освобождать все запрашиваемые у операционной системы ресурсы: файлы, память, сокеты и др. Если программист забывает освободить ресурс, т.е. все ссылки на ресурс теряются без его освобождения, то происходит утечка этого ресурса. Пример утечки файлового потока изображен в листинге 1. В данном примере происходит утечка файлового потока, выделенного вызовом *fopen*, при возникновении ошибки.

```
static bool read_from_file_with_error_check(const char
*file_name)
{
     FILE *file = fopen(file_name, "r");
     if (!file)
          return false ;
     if (check_program_for_some_error())
          return NULL;
     read_from_file(file);
     fclose(file);
     return true;
}
```

Листинг 1. Пример утечки файлового потока

Технология сборки мусора освобождает программистов от необходимости ручного освобождения ресурсов: все ресурсы, ссылок на которых нет, будут освобождены сборщиком мусора. Но у данной технологии есть существенные недостатки: сложность реализации сборщика мусора и замедление работы программы в моменты сборки мусора. Более того, сборка мусора спасает не от всех утечек из-за особенностей подсчета количества ссылок (циклические ссылки и другие особенности), из-за чего даже в таких языках как Java и JavaScript возможна утечка ресурсов. Некоторые высокоуровневые

языки, например, *C++*, *Rust* и *D*, используют идиому *RAII*[1] для защиты от утечек ресурсов: работа с ресурсом происходит через специальный объект, в конструкторе которого ресурс запрашивается, а в деструкторе освобождается.

В данной работе рассматриваются утечки ресурсов в программах на языке C стандарта *C99*. В данном языке не предусмотрена сборка мусора и идиома *RAII* применена быть не может, т.к. язык не является объектно-ориентированным (расширение[2] компилятора *gcc* позволяет использовать деструкторы для произвольных объектов, однако данное расширение не является[3] частью стандарта языка C).

Таким образом, рассматриваемая в данной работе проблема поиска утечек ресурсов в программах на языке C является актуальной, т.к. язык не предоставляет возможностей для автоматического управления ресурсами и требует от программиста ручного освобождения каждого ресурса. Например, с начала 2014 года в браузере *Firefox* была обнаружена 91 утечка ресурсов[4], что подтверждает актуальность проблемы.

Статический и динамический анализ

Анализ кода программ делится на 2 вида: статический и динамический анализ. Динамический анализ – анализ программного обеспечения, выполняемый при помощи выполнения программ на реальном или виртуальном процессоре. Статический анализ – анализ программного обеспечения, производимый без реального выполнения исследуемой программы. Существенное различие между данными видами анализа состоит в том, что статический анализ исследует все возможные пути выполнения программы, а динамический – только пути, которые проходит программа во время выполнения.

Эффективность статического анализа зависит только от точности моделирования программы, тогда как точность динамического анализа зависит только от набора тестовых сценариев [5]. Если набор тестовых сценариев покрывает все возможные варианты использования программы, то динамический анализ будет эффективным. В данной работе рассматривается задача поиска утечек ресурсов в общем случае, поэтому полагаться на существование набора тестовых сценариев нельзя, и далее в данной работе рассматриваются только алгоритмы статического анализа.

Обзор существующих алгоритмов поиска утечек

На рисунке 1 изображена классификация алгоритмов анализа кода. Темным цветом выделены алгоритмы, использующиеся при разработке собственного алгоритма анализа кода для поиска утечек ресурсов.

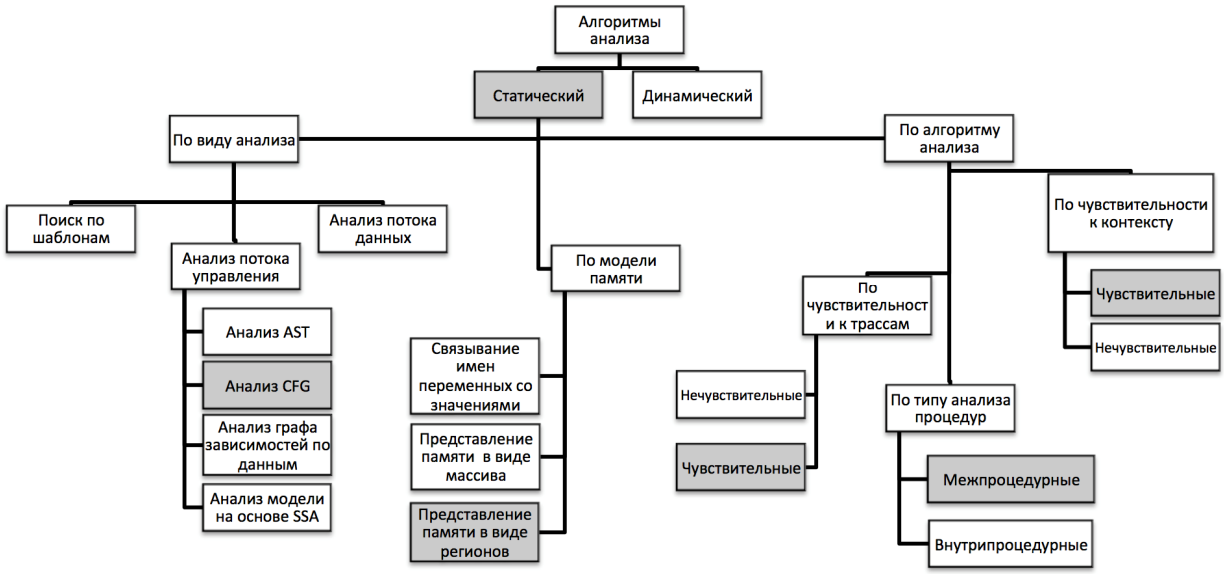

Рис. 1. Классификация алгоритмов анализа кода

В работе [6] вводится следующий алгоритм для поиска утечек в программах на языке *Java*: на вход алгоритму анализа поступает граф потока управления и алгоритм производит межпроцедурный[7], чувствительный к трассам и контексту[8] анализ. В ходе анализа в графе производится поиск путей, в которых присутствует выделение ресурса, но отсутствует освобождение этого ресурса. Анализ указателей производится с помощью выборочных предикатов равенства переменных. В то же время в работах [9] и [10] производится анализ совмещения указателей: для каждого указателя строится множество переменных, на которые он может указывать. В рассматриваемой работе производится анализ состояния лишь части данных – анализ состояния переменных ресурсов.

Статический анализатор *Clang SA* работает с *CFG*[11] и *AST*[12] на входе: *CFG* позволяет находить различные ошибки, например, утечки, а *AST* позволяет обнаруживать различные опечатки в коде. Как и предыдущая рассматриваемая работа, данный анализатор производит межпроцедурный, чувствительный к трассам и контексту анализ. Анализатор строит из *CFG* новый граф – граф состояний, в котором каждая вершина представляет собой совокупность состояния программы и места выполнения программы в коде, т.е. имя файла и номер строки. Данный анализатор строит модель памяти в виде регионов и полностью отслеживает состояние памяти программы. Выполняется оптимизация за счет отказа от анализа недостижимых путей в графе.

Обзор и тестирование существующих анализаторов

Был произведен поиск готовых решений для поиска утечек ресурсов с помощью статического анализа кода. Из всех найденных статических анализаторов были протестированы *Clang SA*[13], *Cppcheck*[14] и *Splint*[15].

Для тестирования существующих решений была создана тестовая программа с легко обнаружимой утечкой файлового потока. Исходный код программы приведен в листинге 2.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
     FILE *f = fopen("file.txt ", "r ");
     int a = 5, b = 5;
     if (a + b == 9)
          fclose(f);
     return 0;
}
```

Листинг 2. Тестовая программа с утечкой файлового потока

Ни один из продуктов не смог обнаружить утечку. При этом анализатор Splint не смог проанализировать программу до конца. Поэтому для эффективного решения проблемы поиска утечек ресурсов необходимо использовать комбинированный алгоритм поиска утечек, заключающий в себе сильные стороны каждого из рассмотренных анализаторов.

Разработка алгоритма анализа кода

Алгоритм анализа кода основывается на статическом анализе кода. Входные данные алгоритма это код на языке С, соответствующий стандарту С99. Выходные данные это список ошибок в коде и их описание. Алгоритм разрабатывается в общем виде с целью дальнейшего использования для обнаружения различных видов ошибок. В ходе работы алгоритм вызывает «конкретные анализаторы». Каждый из этих анализаторов представляет собой обособленный реентерабельный алгоритм обнаружения ошибок конкретного типа. В данной работе используется только один анализатор – анализатор ошибок утечки ресурсов.

В соответствии с проведенным анализом, разработанный алгоритм анализа кода использует анализ потока управления с помощью модели программы в виде графа потока

управления. Используется межпроцедурный контексточувствительный и чувствительный к трассам алгоритм. Модель памяти используется в виде регионов[16].

Разработанный алгоритм анализа изображен на рисунке 2 и состоит из совокупности алгоритмов.

Под *точкой входа* понимается такая функция, которая не вызывается нигде в программе. Разработанный алгоритм осуществляет поиск точек входа и начинает с них анализ программы.

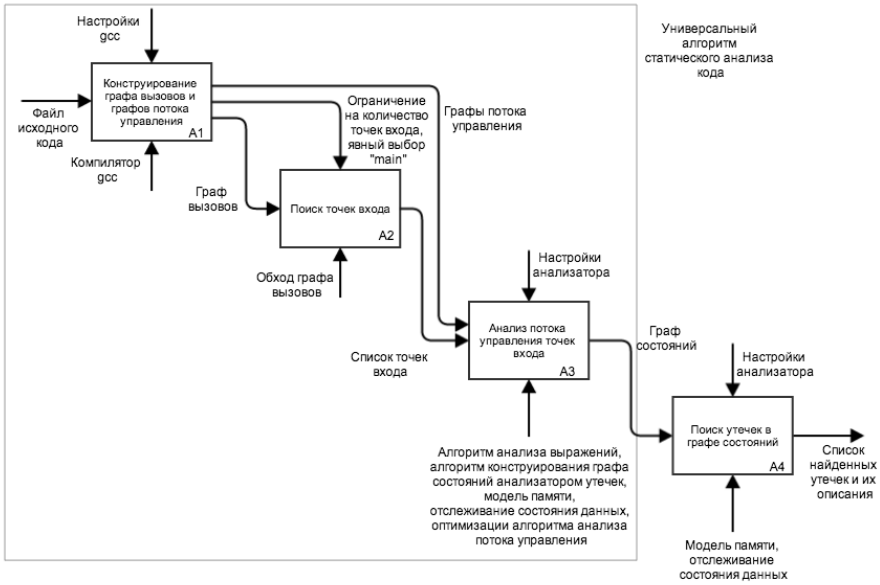

Рис. 2. Укрупненная и модифицированная IDEF-диаграмма алгоритма анализа кода

Состояние программы это совокупность всех данных о программе в данной точке выполнения. Под графом состояний понимается граф, вершины которого представляют состояния программы, и из вершины *A* идет дуга в вершину *B* тогда и только тогда, когда из состояния *A* программа может непосредственно перейти в состояние *B*. Переход из состояния в состояние возможен только при выполнении программой части кода, например, при присваивании переменной нового значения. Причем некоторые части кода могут порождать несколько состояний, например, при вызове неизвестной функции, возвращающей указатель, можно создать два новых состояния: состояние, в котором эта функция вернула ненулевой указатель и состояние, в котором она вернула нулевой указатель. На рисунке 3 изображен граф состояний для функции из листинга 3.

```
static bool example_function(const char *file_name) {
     FILE *file = fopen(file_name, "r");
     if (!file)
          return false;
     read_from_file(file);
     fclose(file);
     return true;
}
```

Листинг 3. Пример функции

Под графом потока управления понимается граф, вершины которого это базовые блоки, представляющие собой последовательность выражений (выражение, или *statement* – это неделимая инструкция языка С) и из вершины *A* идет дуга в вершину *B* тогда и только тогда, когда за выполнением последнего выражения базового блока *A* может непосредственно следовать выполнение первого выражения базового блока *B*. Последовательность выражений базового блока не может содержать ветвлений или переходов, она всегда линейна. На рисунке 4 изображен граф потока управления для функции из листинга 3. Конструирование графа потока управления не выполняется разработанным алгоритмом, данная часть общего алгоритма реализуется компилятором GCC[17].

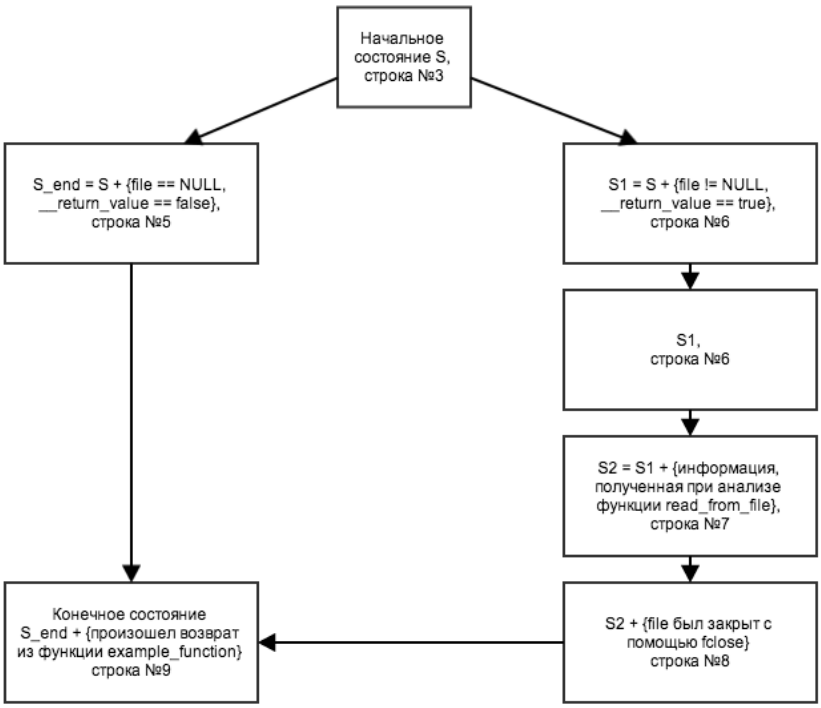

Рис. 3. Граф состояний программы, полученный в результате анализа функции *example_function*

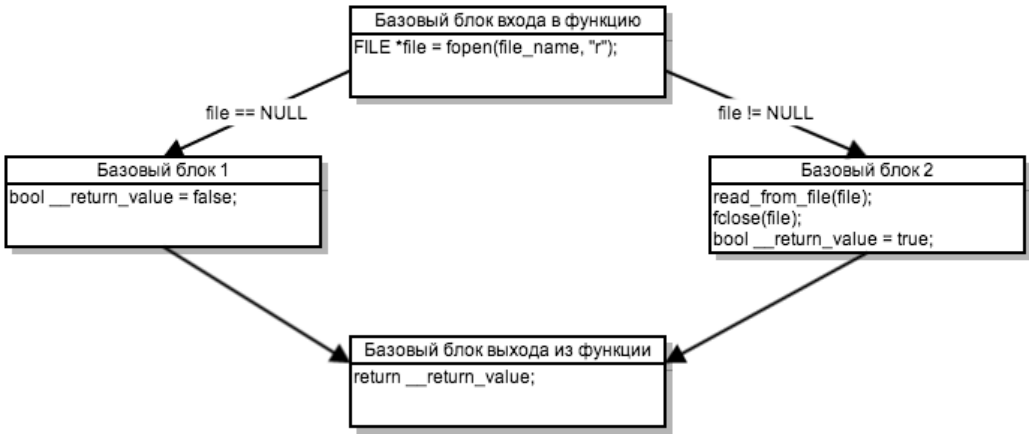

Рис. 4. Граф потока управления функции *example_function*

Алгоритм анализа потока управления получает на вход текущее состояние программы и граф потока управления определенной функции и возвращает сконструированный *граф состояний* программы на выход.

Разработанный алгоритм анализа потока управления производит рекурсивный обход базовых блоков входной функции. Для каждого из базовых блоков последовательно анализируются выражения этого блока. При анализе каждого из выражений может порождаться одно или более новых состояний. При порождении нового состояния в граф состояний добавляется новая вершина, представляющая порожденное состояние и дуга от предыдущего состояния к порожденному. При каждом значимом событии в анализаторе анализатор взаимодействует с «конкретными» анализаторами, которые тоже участвуют в конструировании графа состояний. При переходе из одного базового блока в другой производится рекурсивный анализ следующего базового блока и «слияние» полученного для него графа состояний с текущим графом состояний, при котором эти два графа объединяются в начальной вершине полученного графа состояний и конечной вершине текущего графа состояний. Схема алгоритма не приводится в данной работе, так как она является слишком объемной.

Алгоритм анализа кода, рассмотренный выше, анализирует граф потока управления и строит граф состояний. В конструировании графа состояний участвуют «конкретные» анализаторы, один из которых рассматривается в данном подразделе – анализатор утечек ресурсов.

В данной работе анализируются только ресурсы типа «файловый поток». Другие виды ресурсов могут быть тоже проанализированы подобным образом, для добавления поддержки нового типа ресурса необходимо разработать новый «конкретный» анализатор.

Ресурсы файловых потоков идентифицируются с помощью указателей на структуру *FILE*. Выделение ресурса происходит с помощью вызова *fopen*, освобождение ресурса происходит с помощью вызова *fclose*.

Алгоритм поиска утечек ресурсов состоит из двух частей: конструирование графа состояний при обходе графа потока управления и поиск утечек в сформированном графе состояний.

Алгоритм анализа кода взаимодействует с алгоритмом поиска утечек ресурсов обращаясь к данному алгоритму в определенные моменты, например, при обнаружении вызова функции в коде. Для данного алгоритма необходимо анализировать вызовы функций работы с ресурсами *fopen* и *fclose*.

При вызове «конкретного» анализатора при обнаружении вызова функции на вход «конкретному» анализатору поступает вызываемая функция и аргументы для ее вызова. «Конкретный» анализатор может создать при этом новые вершины графа состояний, а может ничего не создать. Для каждой из добавленных вершин анализатор может указать возвращаемое функцией значение, если оно известно.

На рисунке 5 изображен алгоритм конструирования графа состояний анализатором утечек ресурсов. На вход алгоритму поступает вызываемая функция, ее аргументы и текущее состояние программы. Первым шагом алгоритма является проверка того, что текущая функция это *fopen* или *fclose*. Если это другая функция, то алгоритм на этом завершается.

Далее проверяются аргументы функции. Если это функция *fopen*, то проверяется, что она принимает на вход 2 аргумента типа *char* *. Если это функция *fclose*, то проверяется, что она принимает на вход один аргумент типа *FILE* *.

Следующим шагом, если это функция *fopen*, является создание двух новых состояний: состояний при успешном и неуспешном открытии файла. В первом случае *fopen* возвращает *NULL*, это значение записывается как возвращаемое значение функции и создается дуга от текущего состояния к данному новому состоянию. Дуга также может быть помечена как «открытие файла не удалось, *fopen* вернул *NULL*».

Во втором случае *fopen* возвращает указатель, не равный *NULL*. Создается новое состояние, в котором этот указатель записывается как возвращаемое значение функции. На значение этого указателя накладывается ограничение не равенства *NULL*. Аналогичным образом создается дуга от текущего состояния программы к данному новому состоянию. Состояние программы имеет специальное поле, хранящее список

открытых файлов. В данное поле записывается возвращаемое при успешном открытии файла значение.

Если входная функция это функция *fclose*, то в список закрытых файлов добавляется значение аргумента функции *fclose*.

Алгоритм поиска утечек в сформированном графе состояний приведен на рисунке 6. Алгоритм поиска утечек ресурсов заключается в последовательном обходе конечных вершин графа состояний. Под конечной вершиной понимается вершина, из которой не выходит ни одной дуги. Конечные вершины обозначают состояние программы перед выходом из программы. Также «конкретные» анализаторы могут создавать конечные вершины при обнаружении ошибок, т.к. при обнаружении ошибки продолжать анализ текущей ветви выполнения кода не имеет смысла.

При последовательном обходе каждой вершины графа состояний происходит просмотр списка открытых файлов рассматриваемого состояния. Если в этом списке есть открытый файл, которого нет в списке закрытых файлов этого состояния, то обнаружена утечка этого файла.

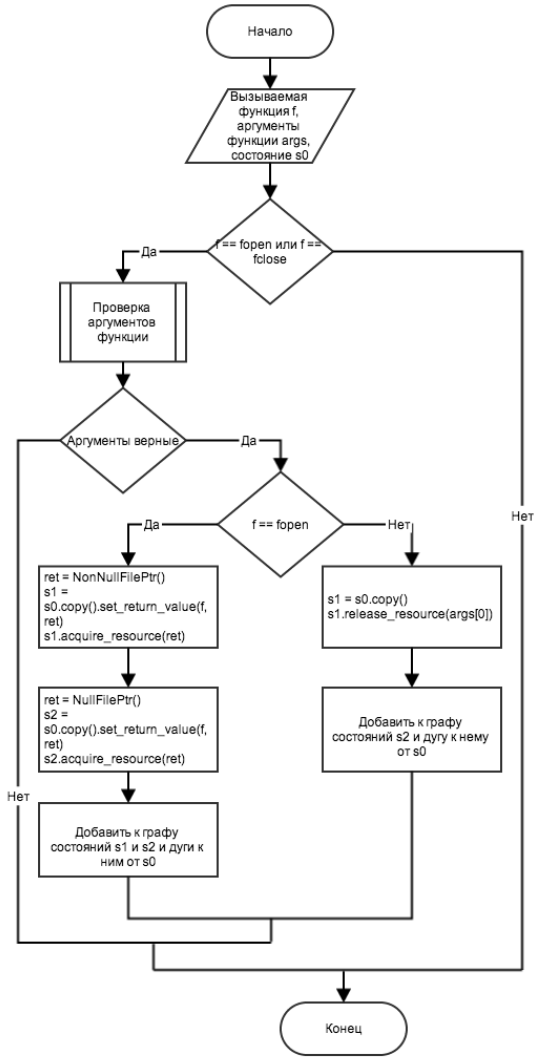

Рис. 5. Алгоритм конструирования графа состояний анализатором утечек

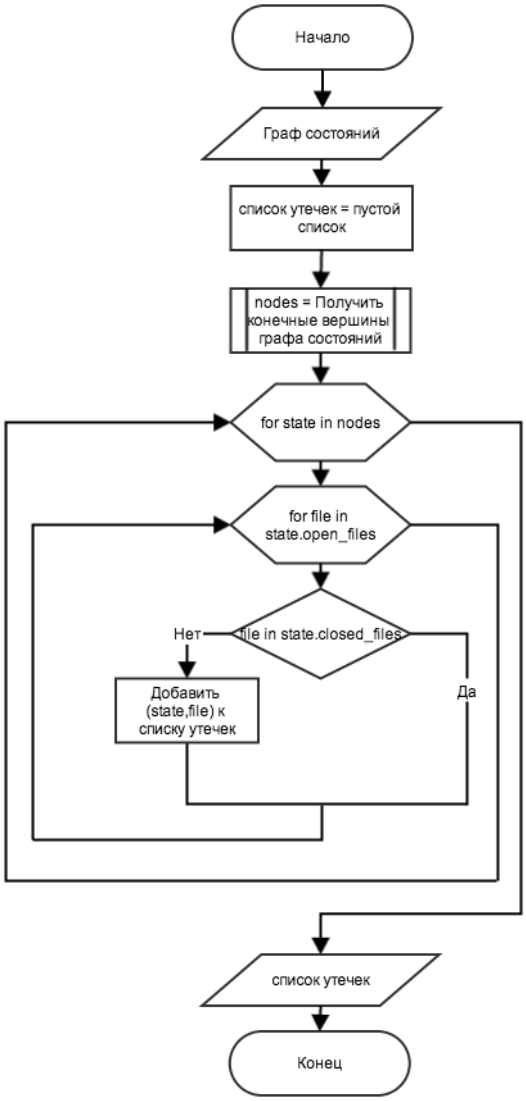

Рис. 6. Алгоритм поиска утечек в графе состояний

Исследование разработанного алгоритма на работах студентов

Для исследования разработанного анализатора на работах студентов были автоматизированно загружены лабораторные работы 85 студентов по предмету «Программирование на Си». Загруженные работы содержали 2287 файлов исходного кода на языке С. Из этих файлов исходного кода для анализа были выбраны только файлы, содержащие работу с файловыми потоками. Было обнаружено 977 таких файлов.

Все исследования проводятся в операционной системе *Ubuntu* 14.04 с компилятором *GCC* 4.8.2. Операционная система запущена в виртуальной машине, которой выделено 3 Гбайт оперативной памяти и одно ядро процессора *Intel Core i5* с частотой 2.4 ГГц.

Для каждого из выбранных файлов исходного кода был автоматизированно выполнен поиск утечек файловых потоков с помощью разработанного анализатора. При успешном выполнении анализа файла исходного кода результат анализа записывался в файл журнала. В данном файле журнала в конце исследования отмечены все обнаруженные утечки файловых потоков.

При завершении анализа с ошибкой проверяется, является ли ошибка следствием того, что файл исходного кода не может быть скомпилирован, или это следствие ошибочной работы анализатора. Для проверки того, что файл может быть скомпилирован, запускается компиляция данного файла с помощью компилятора *GCC*. При обнаружении некомпилируемого файла информация о нем заносится в журнал ошибок. При обнаружении ошибочной работы анализатора процедура анализа работ студентов приостанавливался до исправления ошибки в анализаторе. В ходе выполнения данного исследования было выявлено и исправлено несколько ошибок в анализаторе.

Общее время выполнения анализа всех 977 файлов исходного кода составило 2 часа 43 минуты. Было успешно проанализировано 812 файлов (83.1 %), 165 файлов (16.9 %) не удалось проанализировать из-за ошибок компиляции. Ошибки компиляции вызваны отсутствием необходимых заголовочных файлов для данной операционной системы. В результате анализатор выявил 23 утечки, из которых 100 % являются правильно обнаруженными утечками, т.е. не было ни одного ложноположительного срабатывания. Тот факт, что анализатор обнаружил в работах студентов утечки, которые, возможно, не были обнаружены преподавателями, показывает, что анализатор работоспособен.

В данном разделе производится сравнение разработанного анализатора (далее он обозначается как *gsa*) с существующими статическими анализаторами кода *clang sa* и *cppcheck*.

Для сравнения данных параметров производился запуск анализаторов на всех 977 файлах. В таблице отражено количество правильно и неправильно (ложноположительные срабатывания) найденных утечек. Разработанный анализатор обнаружил 23 утечки, анализатор *cppcheck* обнаружил 25 утечек, анализатор *clang sa* обнаружил 15 утечек. Анализатор *cppcheck* обнаружил больше всех утечек, но 6 из них обнаружены неверно, они являются ложноположительными срабатываниями. При этом разработанный анализатор не совершил ни одного ложноположительного срабатывания, в отличие от анализатора *cppcheck*.

В таблице 1 также отражены максимальный объем резидентной памяти процесса анализатора и время выполнения анализа. Разработанный анализатор потребляет памяти больше, чем остальные анализаторы. Это вызвано большим потреблением памяти самим компилятором *GCC* и использованием языка *Python* для реализации алгоритма, использующего сборщик мусора, который освобождает память лишь в определенные моменты времени. В то время как *cppcheck* и *clang sa* написаны на языке *C++* и не используют сборщик мусора.

Параметр		Анализаторы		
		GSA	Cppcheck	Clang SA
Кол-во утечек	Правильно найденные	23	19	15
	Неправильно найденные	0	6	0
Максимальный объем резидентной памяти, Мбайт		25	3	17
Время выполнения анализа, с		0,8	0,1	0,2

Разработанный анализатор *gsa* выполняет анализ дольше остальных анализаторов. Это вызвано особенностями работы *GCC*, использованием языка *Python* и присутствием большого количества оптимизаций у анализаторов *clang sa* и *cppcheck*.

По результатам исследования разработанный анализатор правильно нашел больше всех утечек, не совершив ложноположительных срабатываний. Таким образом, разработанный анализатор работает медленнее и потребляет больше памяти, чем анализаторы *clang sa* и *cppcheck*, но обладает приемлемой точностью и полнотой поиска утечек.

Заключение

В данной работе была проанализирована проблема поиска утечек ресурсов в программах на языке С. Были рассмотрены существующие решения, их достоинства и недостатки. Была обоснована необходимость разработки собственного алгоритма анализа кода для поиска утечек ресурсов. Был разработан алгоритм анализа кода для поиска утечек файловых потоков в программах на языке С. Алгоритм был реализован и

протестирован на 977 файлах исходного кода из лабораторных работ студентов МГТУ им. Н. Э. Баумана, и по результатам тестирования доказал свою работоспособность, обнаружив утечки, которые, возможно, не были найдены преподавателями. Было проведено сравнение разработанного алгоритма и существующих решений на работах студентов. По итогам сравнения разработанный алгоритм показал наилучшую точность и полноту поиска утечек ресурсов, но наихудшее время анализа и потребление памяти. Таким образом, точность и полнота поиска утечек разработанного алгоритма являются приемлемыми.

Список литературы

1. The RAII Programming Idiom. Available at: http://www.hackcraft.net/raii/, accessed 02.12.2014.
2. Specifying Attributes of Variables. Available at: http://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html, accessed 02.12.2014.
3. Extensions to the C Language Family. Available at: http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html, accessed 02.12.2014.
4. Mozilla Firefox Bugs Tracker. Available at: https://bugzilla.mozilla.org/buglist.cgi?short_desc=leak&resolution=---&query_format=advanced&short_desc_type=allwordssubstr&product=Firefox, accessed 02.12.2014.
5. Ernst M. Static and dynamic analysis: synergy and duality. MIT Lab for Computer Science, 2003. Available at: https://homes.cs.washington.edu/~mernst/pubs/staticdynamic-woda2003.ps, accessed 02.12.2014.
6. Torlak E., Chandra S. Effective Interprocedural Resource Leak Detection. Available at: http://homes.cs.washington.edu/~emina/talks/tracker.icse10.slides.pdf, accessed 02.12.2014.
7. Compilers and Translators: Software Verification Tools. Available at: http://dragonbook.stanford.edu/lecture-notes/Columbia-COMS-W4117/07-10-16.html, accessed 02.12.2014.
8. Data-flow analysis. Available at: http://en.wikipedia.org/wiki/Data-flow_analysis, accessed 02.12.2014.
9. Das M., Lerner S., Seigle M. ESP: Path-Sensitive Program Verification in Polynomial Time. Available at: http://www.cs.cornell.edu/courses/cs711/2005fa/papers/dls-pldi02.pdf, accessed 02.12.2014.

10. Precise and Scalable Software Analysis. Available at: http://saturn.stanford.edu/, accessed 02.12.2014.
11. Control flow graph. Available at: http://en.wikipedia.org/wiki/Control_flow_graph, accessed 02.12.2014.
12. Abstract Syntax Tree. Available at: http://en.wikipedia.org/wiki/Abstract_syntax_tree, accessed 02.12.2014.
13. Clang Static Analyzer. Available at: http://clang-analyzer.llvm.org/, accessed 02.12.2014.
14. Cppcheck. Available at: http://cppcheck.sourceforge.net/, accessed 02.12.2014.
15. Splint Home Page. Available at: http://www.splint.org/, accessed 02.12.2014.
16. Xu Z., Kremenek T., Zhang J. A Memory Model for Static Analysis of C Programs. Available at: http://cpl0.net/~argp/papers/44f93d2a09e3d91eca160b387db6cd8c.pdf, accessed 02.12.2014.
17. The GNU Compiler Collection. Available at: https://gcc.gnu.org, accessed 02.12.2014.