УДК 004.052.42

# ПОВЫШЕНИЕ ЭФФЕКТИВНОСТИ ВЫЯВЛЕНИЯ ОШИБОК ПАМЯТИ НА ОСНОВЕ СИМВОЛЬНОГО АНАЛИЗА ДАННЫХ

**Портнов Е.М., Федоров А.Ю.**

*Национальный исследовательский университет «МИЭТ», Москва, e-mail: scof.ru@gmail.com*

Одной из актуальных проблем, возникающих в процессе отладки и тестирования программ, написанных на языке C/C++, является эффективный поиск ошибок памяти. В статье проанализированы основные виды ошибок, возникающих при взаимодействии программ с памятью, а также основные сегменты памяти программ, в которых они проявляются. На основе проведенного анализа разработаны формализованные алгоритмы для повышения эффективности взаимодействия программного обеспечения с памятью. Алгоритмы эффективны с точки зрения полноты и точности, так как основаны на методе динамического символьного исполнения программы и способны анализировать не конкретные значения, а зависимости между данными. Проведенное экспериментальное исследование показало, что полученная программная реализация алгоритмов поиска ошибок виртуальной памяти имеет преимущество над аналогичными продуктами с точки зрения полноты анализа.

**Ключевые слова: анализ программного обеспечения, поиск ошибок памяти, алгоритмы символьного анализа, динамический анализ**

# IMPROVING THE EFFICIENCY OF MEMORY ERRORS DETECTION BASED ON SYMBOLIC DATA ANALYSIS

**Portnov E.M., Fedorov A.Y.**

*National Research University of Electronic Technology, Moscow, e-mail: scof.ru@gmail.com*

One of the urgent problems of testing and debugging software written in C/C++ languages is the effective memory errors detection. This paper analyses the typical memory errors, which are arise due to interaction between memory and software, and also it analyses typical memory segments of program where errors may be localized. Due to research formal algorithms to analyse interaction between software and memory are developed. The algorithms are effective by such criteria as completeness and accuracy of analysis, because they are based on symbolic execution technique, which allow to analyse not only concrete values, but also relations between data. Conducted experimental research has found that program implementation of virtual memory errors detection algorithms has the advantage over analog products by such criteria as completeness of analysis.

**Keywords: software analysis, memory errors detection, symbolic analysis algorithms, dynamic analysis**

В настоящий момент существуют два подхода к анализу программного обеспечения: статический и динамический. Преимуществом статического подхода является высокий процент покрытия кода, но недостаток заключается в невозможности анализа данных в процессе исполнения. Динамический подход лишен данного недостатка, но большинство таких алгоритмов нацелено на анализ одного из возможных вариантов исполнения программы, что дает низкий процент покрытия кода. В итоге, ни один из традиционно использующихся для анализа ошибок памяти алгоритмов не обладает ни полнотой анализа, ни точностью.

Данная работа направлена на создание эффективных программных средств, которые объединяют достоинства каждого из подходов и тем самым обеспечивают повышение эффективности выявления ошибок использования памяти, возникающих в процессе исполнения программы. Для этих целей рассматриваются алгоритмы анализа памяти в контексте символьного исполнения [5]. Символьное исполнение – вид абстрактного исполнения программы, где все входные данные являются набором булевых выражений (символьными), которые отражают ограничения, наложенные на эти данные.

**Таблица 1**

Типы ошибок при работе с памятью

| Тип ошибки | Возможные причины |
|---|---|
| чтение за пределами памяти, выделенной объекту | чтение за пределами массива;<br>разыменование невалидного указателя;<br>двойное освобождение выделенной памяти. |
| запись за пределами памяти, выделенной объекту | переполнение буфера;<br>выход за пределы массива;<br>запись значения по невалидному указателю. |
| утечки памяти | отсутствие вызова процедуры освобождения выделенной памяти. |

*Виды ошибок*

Жизненный цикл объектов памяти в исполняемых программах начинается с резервирования участка памяти необходимого размера. Затем с памятью можно взаимодействовать – записывать и читать данные до тех пор, пока она не будет освобождена.

При таких условиях работы с памятью все возможные ошибки можно свести к трем типам, представленным в табл. 1.

По месту локализации эти ошибки также делятся на несколько типов. В исполняемом процессе программы существуют следующие регионы памяти: стек (локальные переменные, адрес возврата и т.д.), куча (динамические данные), регион со статическими данными (глобальные данные). В каждом из них могут произойти ошибки.

Таким образом, для эффективного обнаружения всех ошибок памяти необходимо разработать алгоритмы, которые бы могли выявлять чтение и запись за пределами памяти, выделенной объекту и утечки памяти. При этом нужно учитывать места локализации данных ошибок и время жизни объектов памяти.

*Разработка алгоритмов поиска ошибок памяти*

Разрабатываемые алгоритмы предполагают перехват процедур выделения и освобождения памяти. Также необходимо хранить информацию о выделенном участке памяти. Для этого предлагается хранить данные в виде набора объектов памяти (ОП). Эти объекты можно представить в виде кортежей:

$$MemoryObject = \langle Address,\ Size,\ MemoryObject,\ Status \rangle, \quad (1)$$

где *Address* – виртуальный адрес, по которому расположены данные,
*Size* – *раз*мер этого участка,
*MemoryObject* – владелец данного участка (в случае если объект представляет собой ссылку на другой объект),
*Status* – объект, отображающий текущее состояние участка памяти.

Объект «статус» может иметь значения: 0, static, dynamic, free. Если тип выделенной памяти является статическим, то Status принимает значение «static»; если тип памяти является динамическим – значение «dynamic»; если память была освобождена – значение «free». Выразим правила в формальном виде по аналогии с работой [1]. Таким образом, память, используемая в процессе, может быть представлена в виде формулы (2):

$$Memory = \bigcup_i MemoryObject_i\ . \quad (2)$$

Введем в терминологию следующие функции.

Получение объекта-родителя:

$$GetMemObject(pointer): \langle pointer,\ Size, MemObj, Status \rangle \to MemObj\ . \quad (3)$$

Получение размера региона памяти:

$$GetSize(\mathrm{MemoryObj}):\ \mathrm{MemoryObj}\langle pointer,\ Size, MemObj, Status \rangle \to Size\ . \quad (4)$$

Получение адреса региона памяти:

$$GetAddress(\mathrm{MemoryObj}):\ \mathrm{MemoryObj}\langle pointer,\ Size, MemObj, Status \rangle \to pointer\ . \quad (5)$$

Рассмотрим основные правила для выявления ошибок памяти. Для начала возьмем правило, которое будет обрабатывать объявление объектов памяти.

*Объявление объектов:*

$$\left| intarray[size] \right|:\ Memory = Memory \cup \langle array,\ size, 0, static \rangle\ . \quad (6)$$

Каждый массив ассоциируется с объектом памяти (memory object). ОП хранит в себе адрес и размер занимаемого региона.

*Выделение динамических объектов*

Для получения свободного региона в куче в языках C/C++ используются специальные функции, предоставляемые стандартной библиотекой. В языке C такой функцией является malloc, в C++ операторы new и new[]. Для получения информации о выделяемых регионах нам необходимо перехватывать данные функции и создавать ОП для каждого выделяемого региона. Таким образом, правило будет выглядеть следующим образом:

$$\left|int^{*}\ array = malloc(Size)\right| : Memory = Memory \cup \langle array,\ Size, 0, dynamic\rangle. \quad (7)$$

*Операция присвоения адреса*

Операция получения адреса предполагает создание новой ссылки на объект, то есть связывание нового указателя с родителем:

$$\left|int^{*}pointer = \&array[0]\right| : Memory = Memory \cup$$

$$\langle \&array[0],\ sizeof(^{*}pointer), GetMemObject(array), 0\rangle; \quad (8)$$

$$\left|int^{*}newpointer = \text{pointer}\right| : Memory = Memory \cup$$

$$\langle pointer,\ sizeof(^{*}pointer), GetMemObject(pointer), 0\rangle. \quad (9)$$

*Освобождение объектов*

Освобождение статических массивов в языкахC/C++ для программиста происходит прозрачно. Эту операцию производит компилятор и библиотеки времени исполнения (CRT0). Если массив выделен в стеке, то он освободится по выходу из функции, в которой был объявлен. В случае объявления массива в сегменте данных, массив освобождается по выходу из программы. Так как для данных операций не предусмотрены конструкции языка, то обозначим их как undeclare:

$$\left|undeclare(array)\right| : \text{Memory} = Memory \setminus \langle array,\ Size, 0, static\rangle. \quad (10)$$

*Освобождение динамических объектов*

Для освобождения ранее полученного динамического региона памяти программисты C/C++ используют специальные функции, предоставляемые стандартной библиотекой. В языке C такой функцией является free, в C++ операторы delete и delete[]. Для получения информации об освобождаемых регионах необходимо перехватывать данные функции, проверять корректность данной операции и, в случае успеха, отмечать в соответствующем ОП освобождение данной операции. Таким образом, правило будет выглядеть следующим образом:

$$\left|free(array)\right| : \text{MemObj} = GetMemObject(array):$$

$$MemObj = \langle GetAddress(MemObj), 0, 0, free\rangle. \quad (11)$$

Правило для нахождения ошибки освобождения памяти:

$$\left|free(array)\right| : \exists\langle array,\ Size,\ MemObj, Status\rangle \in Memory$$

$$Status \neq dynamic \wedge MemObj = 0 \vee$$

$$\vee GetStatus(MemObj) \neq dynamic \wedge MemObj \neq 0. \quad (12)$$

Если освобождаемый указатель – это ссылка на объект, то проверяется статус первоначально выделенного объекта. Если же освобождается сам объект – проверяется его статус. Если статус не равен dynamic, то это означает, что освобождаемый объект не валиден.

*Операция чтения и записи по адресу*

Данные операции могут представляться в различных формах. Например, разыменование указателя или обращение по индексу в массиве. Такие операции являются наиболее опасными. Перед ними необходимо вставить нижеописанную проверку.

В случае, если производится попытка получить доступ за пределами ОП, необходимо указать на наличие ошибки. Правило для обнаружения ошибок чтения и записи по неверному адресу:

$$|*pointer| : MemObj = GetMemObject(pointer):$$

$$pointer + sizeof(*pointer) \geq GetAddress(MemObj) + GetSize(MemObj) \vee$$

$$\vee\ pointer < GetAddress(MemObject) . \quad (13)$$

*Утечки памяти*

Утечки памяти обнаруживаются следующим образом. Если к концу выполнения программы остается хоть один ОП со статусом dynamic, значит, произошла утечка памяти. Соответствующее правило для этого случая будет выглядеть следующим образом:

$$|programterminate| : \exists\langle pointer, Size, MemObj, dynamic\rangle \in Memory . \quad (14)$$

*Символьная интерпретация*

Если применить вышеописанные алгоритмы в динамическом анализаторе, то в некоторых случаях алгоритмы не смогут найти ошибки, например, когда в операциях с памятью участвуют входные параметры. Если индекс, по которому происходит обращение в массиве, является входным параметром, то ошибка будет обнаружена лишь тогда, когда этот индекс выйдет за пределы допускаемых значений. Такие ошибки можно обнаружить, если подавать на вход тестовые наборы данных. Более перспективным в этом плане является подход символьного исполнения.

Символьные анализаторы отличаются от динамических анализом не конкретных значений, а символьных. Символьные переменные представляют собой набор булевских ограничений, наложенных на эти переменные. Может ли быть решено то или иное булевское ограничение, символьные анализаторы узнают при помощи решателей ограничений (constraint solver [3]). На каждом условном переходе решается вопрос о возможности выполнения каждой из ветвей. Если обе ветви возможны, то анализ продолжается одновременно в обеих ветвях с наложением ограничений на переменные, участвующие в условии. Таким образом, в теории, достигается большой процент покрытия кода (символьный анализ способен обойти все возможные ветви программы при условии достаточных ресурсов). Рассмотрим, как можно применить вышеописанные алгоритмы для работы с символьными данными.

Модифицируем кортежи ОП таким образом, что теперь в них можно будет хранить не только конкретный адрес и размер выделяемого региона, но и символьные данные. Обозначим новый кортеж следующим образом:

$$MemoryObject = \langle SymbAddress, SymbSize, MemoryObject, Status\rangle , \quad (15)$$

где SymbAddress – символьная переменная, указывающая на регион памяти;
SymbSize – символьная переменная, хранящая размер выделенного региона в памяти;
MemoryObject – указатель на родительский ОП;
Status – переменная, отображающая текущее состояние ОП.

Таким образом, правила (12), (13), (14) будут анализировать не только конкретные данные, но и символьные, что позволяет, в теории, за один запуск анализатора обойти всевозможные ветви исполнения программы.

*Результаты*

В процессе экспериментального исследования были использованы 6 анализаторов: GCC, PVS-Studio, CppCheck, Clang Address Sanitizer (ASAN) [2], Intel Inspector XE [4], KLEE [6]. Часть анализаторов являются статическими, например, такие как компилятор GCC с флагом -WAll, PVS-Studio, CppCheck. Анализаторы Clang Address Sanitizer и Intel Inspector XE являются динамическими. Предложенные выше подходы были реализованы в символьном анализаторе KLEE.

Исследование заключалось в запуске на каждом из анализаторов ряда тестов, содержащих ошибки памяти. Список ошибок, содержащихся в тестах:

1. Чтение за пределами стека по константному смещению.
2. Чтение за пределами стека по динамическому смещению.
3. Запись за пределами стека по константному смещению.
4. Запись за пределами стека по динамическому смещению.

**Таблица 2**

Результаты сравнения анализаторов ошибок взаимодействия с памятью

| Анализатор | GCC | PVS–Studio | CppCheck | Clang (ASAN) | IntelInspector XE | модифицированный KLEE |
|---|---|---|---|---|---|---|
| 1 | – | + | – | + | + | + |
| 2 | – | – | – | + | – | + |
| 3 | – | + | – | – | + | + |
| 4 | – | – | – | + | + | + |
| 5 | – | – | – | + | – | + |
| 6 | – | + | – | + | + | + |
| 7 | – | – | – | + | + | + |
| 8 | – | + | – | – | + | + |
| 9 | – | – | – | – | + | + |
| 10 | – | – | – | + | + | + |
| 11 | – | – | – | + | + | + |
| 12 | – | + | + | + | + | + |
| 13 | – | – | + | + | + | + |
| 14 | – | – | + | – | + | + |
| 15 | + | + | + | – | + | + |
| 16 | – | + | + | – | + | + |
| Итог (%) | 6,25 | 62,5 | 31,25 | 43,75 | 87,5 | 100 |

5. Запись за пределами стека (передача неверного размера буфера функциям ввода/вывода).

6. Чтение за пределами кучи по константному смещению.

7. Чтение за пределами кучи по динамическому смещению.

8. Запись за пределами кучи по константному смещению.

9. Запись за пределами кучи по динамическому смещению.

10. Запись за пределами кучи (передача неверного размера буфера функциям ввода/вывода).

11. Запись за пределами массива (с динамическим размером).

12. Двойное освобождение указателя.

13. Двойное освобождение указателя (в различных функциях).

14. Утечка памяти.

15. Разыменование неинициализированного указателя.

16. Разыменование нулевого указателя.

Результаты исследования представлены в табл. 2. Знак «+» означает, что анализатор обнаружил ошибку, в противном случае ставился знак «–».

По результатам экспериментального исследования можно сделать несколько выводов:

1. Динамические анализаторы обнаруживают в среднем большее количество ошибок, чем статические.

2. Лучшим анализатором среди динамических является модифицированный в рамках этой работы анализатор KLEE.

Реализованное программное средство KLEE не менее чем на 12,5 % эффективнее аналогов по количеству обнаруженных ошибок. Благодаря использованию символьного анализатора все ошибки были обнаружены за один запуск анализатора, в отличие от других динамических анализаторов.

Предложенные выше подходы могут найти применение в сфере анализа программного кода. Помимо использования в качестве сертификационного средства, предложенная программная реализация может быть использована в образовании (в качестве наглядной демонстрации работы символьного анализатора).

## Список литературы

1. Ицыксон В.М., Моисеев М.Ю., Ахин М.Х., Захаров А.В. Алгоритмы анализа указателей для обнаружения дефектов в исходном коде программ // Системное программирование. – 2009. – С. 5–30.

2. Address Sanitizer // Clang 3.8 documentation: сайт. – URL: http://clang.llvm.org/docs/AddressSanitizer.html (дата обращения: 14.12.15).

3. Cadar C., Dunbar D., Engler D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs // OSDI`08 Proceedings of the 8th USENIX conference on Operating systems design and implementation. – 2008. – P. 209–224.

4. Intel Inspector // Intel Developer Zone: сайт. – URL: https://software.intel.com/en-us/intel-inspector-xe (дата обращения: 14.12.15).

5. KingJ.C. Symbolic Execution and Program Testing // Communications of the ACM. – 1976. – Vol. 19, № 7. – P. 385–394.

6. KLEE LLVM Execution Engine // Github: сайт. – URL: https://klee.github.io(дата обращения: 14.12.15).