DOI: 10.15514/ISPRAS-2019-31(6)-1

Анализ корректности работы с памятью с использованием расширения теории символьных графов памяти предикатами над символьными значениями

А.А. Васильев, ORCID: 0000-0002-5738-9171 <vasilyev@ispras.ru>
В.С. Мутилин, ORCID: 0000-0003-3097-8512 <mutilin@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В работе мы рассмотрим подход статической верификации исходного кода программы на предмет корректной работы с памятью. Метод основывается на использовании символьных графов для представления памяти программы. В работе представлено расширение символьных графов памяти, позволяющее использовать предикаты над символьными значениями для повышения точности анализа. Предикаты позволяют отсекать недостижимые пути, уменьшая количество ложных сообщений об ошибках, а также находить новые ошибки за счет добавления новых проверок на символьных значениях. Метод реализован на основе инструмента CPAchecker. Практическая полезность продемонстрирована на драйверах ядра операционной системы Linux.

Ключевые слова: символьные графы памяти; верификация; модель памяти; предикатные абстракции; динамические структуры данных

Для цитирования: Васильев А.А, Мутилин В.С. Анализ корректности работы с памятью с использованием расширения теории символьных графов памяти предикатами над символьными значениями. Труды ИСП РАН, том 31, вып. 6, 2019 г., стр. 7–20. DOI: 10.15514/ISPRAS–2019–31(6)–1

Благодарности: Работа поддержана грантом РФФИ 18-01-00426

Predicate extension of symbolic memory graphs for analysis of memory safety correctness

A.A. Vasilyev, ORCID: 0000-0002-5738-9171 <vasilyev@ispras.ru>
V.S. Mutilin, ORCID: 0000-0003-3097-8512 <mutilin@ispras.ru>

Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Safety-critical systems require additional effort to comply with specifications. One of the required specification is correct memory usage. The article describes an efficient method for static verification against memory safety errors as a combination of Symbolic Memory Graphs and predicate abstraction on symbolic values used in graph. In this article, we introduce an extension of Symbolic Memory Graphs. In addition to symbolic values, the graph stores predicates over symbolic values, which allow to track the relationship between symbolic values in the graph. We also expand existing vertex types to support arbitrary abstract regions, which allow us to represent such dynamic data structures as lists and trees. One of the types of abstract regions is also the ODM region, which presents a special kind of on-demand memory that occurs when analyzing incomplete programs. For this memory, the size and structure of the contents are not known in advance, but it is believed that such memory can be operated safely. The method is implemented in CPAchecker tool. Practical usage is demonstrated on Linux kernel modules. The practical contribution of our work is to reduce false error messages by constructing more accurate abstractions using predicates over symbolic values.

Keywords: symbolic memory graphs; formal verification; predicate abstraction

For citation: Vasilyev A.A., Mutilin V.S. Predicate extension of symbolic memory graphs for analysis of memory safety correctness. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 6, 2019. pp. 7-20 (in Russian). DOI: 10.15514/ISPRAS-2019-31(6)-1

Acknowledgements. The research was supported by RFBR grant 18-01-00426.

1. Введение

Критические ошибки безопасности и непредвиденные прекращения работы программы часто происходят из-за неправильной работы с памятью. Подобные ошибки являются сложными для обнаружения и устранения, так как время и место появления видимых последствий ошибки может быть достаточно далеко от времени и места выполнения ошибочного действия.

Существуют подходы к проектированию и созданию критического программного обеспечения, предотвращающие возникновение определенных видов ошибок за счет активного использования формальных методов на всех этапах разработки. Примером успешно реализованных проектов с формальной верификацией требований являются микроядро seL4 [1] и оптимизирующий компилятор CompCert [2] языка CLight. Но зачастую подобные методы подразумевают создание программного обеспечения с нуля и тяжело адаптируются к анализу существующего кода.

Статическая верификация исходного кода является одним из подходов для доказательства отсутствия ошибок в программах. Для этого применяются такие методы как анализ потоков данных (data-flow analysis), методы ограничиваемой верификации моделей (Bounded Model Checking, BMC) [3], k-индукция [4], метод уточнения абстракций по контрпримерам (Counter-example guided abstraction refinement, CEGAR) [5]. Наша работа основывается на последнем методе и реализующем его инструменте CPAchecker [6].

Инструмент основывается на адаптивном статическом анализе CPA [7] (Configurable Program Analysis), который представляется единообразно в виде домена и набора операторов, таких как оператор перехода, останова и слияния.

На сегодняшний день в инструменте CPAchecker представлен широкий набор CPA, например, имеется предикатный анализ [8] и анализ явных значений [9]. Унифицированное представление анализов позволяет гибко комбинировать CPA.

Одним из ключевых факторов для эффективного процесса верификации является выбор модели для представления памяти программы.

В предикатных моделях памяти [10] используется предикатная логика и, соответственно, память программы описывается с помощью логических формул, в которых могут использоваться различные теории, например, теория массивов или теория неинтерпретируемых функций [11,12,13].

Данные модели реализованы, например, в инструментах BLAST [14,15], SLAM [16], CPAchecker [8]. Границы применимости задаются решателями логических формул, которые сильно варьируются в зависимости от используемых теорий. Методы предикатных абстракций позволяют описывать структуры данных ограниченного размера, в том числе накладывать ограничения на хранящиеся в них значения, но не позволяют описывать динамические неограниченные структуры данных.

Существуют также методы, в которых динамические структуры данных представляются в структурах троичной логики (three-valued) [17]. Например, подход ленивого анализа связей (Lazy Shape Analysis) [18] реализован в инструменте BLAST.

Еще один класс моделей памяти основывается на логике разделения, являющейся расширением логики Хоара с возможностью локальных рассуждений за счет наличия в утверждениях пространственных связок [19]. Для целей верификации используются разрешимые подмножества логики разделения, но вводимые ограничения позволяют проверять только специальные классы программ (например, без использования массивов). Примеры инструментов, использующие логику разделения: SLAyer [20], VeriFast [21], SpaceInvader [22] и INFER [23]. Методы разделяемой логики позволяют описывать взаимосвязи для неограниченных данных, но для этого используются фрагменты логики, которые могут не иметь решения или иметь крайне неэффективное решение.

Наиболее перспективным [24] для верификации ошибок использования памяти является подход, основанный на символьных графах памяти (Symbolic Memory Graph, SMG) [25]. SMG – это ориентированный граф, представляющий состояние программы. Узлы этого графа хранят символьные значения и объекты. Объекты подразделяются на регионы памяти и абстракции структур данных. Дуги показывают взаимосвязи между узлами и делятся на ребра-указатели и ребра-значения. Каждая дуга и узел в SMG имеют атрибуты, представляющие размер, смещение, состояние выделения памяти.

SMG позволяет эффективно описывать ограниченные структуры данных, используя символьные имена и описание связей между ними в виде графов. Для борьбы с неограниченным ростом используются методы абстракции, с помощью которых похожие элементы структуры обобщаются, тем самым неограниченные данные представляются в виде конечного символьного графа.

В данной статье мы представим расширение SMG.

- В дополнение к символьным значениям граф будет хранить предикаты над символьными значениями, которые позволят отслеживать соотношения между символьными значениями в графе.
- Мы расширим существующие типы вершин для поддержки произвольных абстрактных регионов, что позволит представлять такие динамические структуры данных как списки и деревья. Одним из видов абстрактных регионов также является регион ODM, представляющий специального вида память по требованию (On-Demand Memory) [26], которая возникает при анализе неполных программ. Для этой памяти заранее неизвестен размер и структура содержимого, однако считается, что с такой памятью можно работать безопасно.

В следующем разделе мы дадим определение символьного графа памяти с нашими расширениями. В подразделе 2.1 определяются основные команды, позволяющие работать с SMG. В разд. 3 приводится пример интерпретации выполнения программы. Определение адаптивного статического анализа дается в разд. 4, а в разд. 5 определяется адаптивный статический анализ на основе символьных графов памяти (SMGCPA), работающий с расширенным SMG, что позволяет повысить точность анализа, используя предикаты для отсечения невыполнимых путей в операторе перехода, и поддержать неограниченные динамические структуры данных. Приводится пример работы анализа.

В разд. 6 мы представим экспериментальные данные на ядре ОС Linux. Практическим вкладом работы является сокращение ложных сообщений об ошибках за счет построения более точных абстракций, использующих предикаты над символьными значениями.

2. Символьные графы памяти

Наше определение *символьного графа памяти SMG* $\mathrm{G} = (\mathrm{O}, \mathrm{V}, \Lambda, \mathrm{H}, \mathrm{P}, \Phi, \Pi, \mathrm{E})$ опирается на определение из работы [25], со следующими отличиями:

- во множестве вершин O, вместо конкретного региона для двусвязного списка dls используется понятие абстрактного региона, которое в свою очередь может иметь тип

других структур данных, таких как деревья, а также специального вида памяти ODM;
- множества символьных значений V, функций аннотаций Λ, ребер-значений H, ребер-указателей P определяются аналогично [25];
- новый элемент Φ задает стек вызовов функций;
- Π задает новое множество предикатов над символьными значениями;
- E хранит известные явные значения для символьных значений V.

Перед формальным определением SMG обозначим множество предикатов над символьными значениями V в теории линейных неравенств над битовыми векторами как $\Xi(V)$ [10]. Для решения формул, построенных на основе данных предикатов, используются SMT решатели в соответствующих теориях.

Формально $G = (O, V, \Lambda, H, P, \Phi, \Pi, E)$ определяется следующим образом.

- O – конечное множество объектов, которые включают в себя регионы памяти R и абстракции регионов памяти A, например для сегментов списков. В регионах выделен нулевой регион $R_{NIL} \in R$, являющийся эквивалентом памяти по нулевому адресу.
- V – конечное множество символьных значений с выделенным нулевым значением NIL и неопределенным значением $undef$.
- $\Lambda = \langle kind, size, valid \rangle$ – кортеж функций-аннотаций:
  - тип объекта $kind(o): O \to K = \{region, abstract\}$;
  - размер объектов или символьных значений $size(o): O \cup V \to N$;
  - валидность объекта $valid(o): O \to B$, для нулевого региона $valid(R_{NIL}) = false$.
- H – частичное отображение $O \times Z \times N \rightharpoonup V$, задающее ребра-значения $valueEdge(obj, offs, v, sz)$ из объекта $obj \in O$ по смещению $offs \in Z$ с размером значения $sz \in N$ в символьное значение $v \in V$.
- P – частичное отображение $V \rightharpoonup Z \times O$, задающее ребра-указатели $pointsToEdge(v, offs, obj)$ из значения $v \in V$ в объект $obj \in O$ по смещению $offs \in Z$.
- Стекфрейм Φ – кортеж идентификатора функции, ее аргументов, локальных переменных и памяти на стеке, регионов памяти. В стеке вызовов хранится специальный нулевой фрейм, в котором хранятся глобальные переменные.
- Предикаты над символьными значениями $\Pi \subseteq \Xi$.
- Множество точных значений для символьных значений $E: V \rightharpoonup Z$.

Для удобства записи определим вспомогательные функции над дугами H и P.

- Объект, из которого ведет ребро $h \in H$, обозначим $object(h): H \to O$.
- Символьное значение, в которое ведет ребро $h \in H$, обозначим $value(h): H \to V$.
- Смещение относительно объекта, из которого ведет ребро $h \in H$, обозначим $offset(h): H \to N$.
- Размер значения, записанного на ребре $h \in H$, обозначим $size(h): H \to N$.
- Символьное значение, из которого ведет ребро $p \in P$, обозначим $value(p): P \to V$.
- Объект, в которое ведет ребро $p \in P$, обозначим $object(p): P \to O$.
- Смещение относительно объекта, в который ведет ребро $p \in P$, обозначим $offset(p): P \to Z$.

2.1 Операции над SMG

В этом разделе для символьного графа мы определим операции, причем для каждой операции определим как действие этой операции, так и предусловие этой операции, в котором осуществляется проверка корректности обращения к памяти.

Мы будем рассматривать следующие классы ошибок (в круглых скобках указан соответствующий идентификатор уязвимости по базе MITRE):

- BUF_OVERRUN – чтение/запись за границами буфера (CWE-119, CWE-120, CWE-121, CWE-122, CWE-124, CWE-125, CWE-126, CWE-127, CWE-129, CWE-787);
- NIL_DEREF – обращение по нулевому указателю (CWE-476, CWE-690);
- USE_AFTER_FREE – использование памяти после освобождения (CWE-416);
- UNALLOC_FREE – освобождение ранее не выделенной памяти (CWE-590, CWE-761);
- DOUBLE_FREE – повторное освобождение памяти (CWE-415);
- MEM_LEAK – утечки памяти (CWE-401).

В процессе верификации в SMG могут появляться абстрактные регионы. В дальнейшем при определении SMGCPA мы рассмотрим их построение более детально, например, как результат абстрагирования над несколькими графами SMG или интерпретации специальной функции.

Сейчас для определения операций нам потребуется вспомогательная операция материализации $materialize: SMG \times A \to 2^{SMG \times R}$ обратная к абстрагированию.

Будем писать $materialize(G, a)$, где a – это абстрактный регион, возвращает множество пар (G', r), где G' – это граф, в котором абстрактный регион a заменен на реализацию конкретным регионом r.

Пример ее задания для двусвязного списка можно найти в работе [25].

Для символьного графа памяти $G = (O, V, \Lambda, H, P, \Phi, \Pi, E)$ определены следующие операции:

- Операция чтения $read(G, obj, offs, sz)$ значения размером $sz \in N$ из объекта $obj \in O$ по смещению $offs \in Z$, возвращающая значение $v \in V$ и потенциально новый граф G'.
  - Если $kind(obj) = region$, то происходит чтение без изменения графа $G = G'$. Формально, мы возвращаем: $v = value(h)$, если существует такое $h \in H$, что $object(h) = obj \wedge offset(h) = offs \wedge size(h) = sz$, иначе $v = undef$. Заметим, что может существовать только единственное ребро h для заданных $sz, offs, obj$, так как H это отображение.
  - Если $kind(obj) = abstract$, т.е. это абстрактный регион $obj \in A$, то перед чтением происходит его материализация $(G', r) = materialize(G, obj)$ и только затем чтение $read(G', r, offs, sz)$. При этом возвращается граф G'.

  До чтения проверяется валидность объекта $valid(obj)$, отвечающая за ошибки NIL_DEREF и USE_AFTER_FREE, а также проверяется выход за границы объекта BUF_OVERRUN: $offs \geq 0 \wedge offs + sz \leq size(obj)$. Если имеется явное значение для $offs$, т.е. определено $E(offs)$, то выражение вычисляется явно. Иначе, строится формула, являющаяся конкатенацией множества предикатов Π и отрицания условия выхода за границы. Тем самым, наличие решения построенной формулы, говорит о существовании состояния программы, в котором возможен выход за границы BUF_OVERRUN.

  Заметим, что ребро по условию чтения может не существовать, но при этом чтение завершаться без ошибок.
- Операция записи $write(G, obj, offs, v, sz)$ значения $v \in V$ с размером $sz \in N$ в объект $obj \in O$ по смещению $offs \in Z$, возвращающая новый граф $G' = (O', V', \Lambda', H', P', \Phi', \Pi', E')$.
  - Если $kind(obj) = region$, т.е. это регион $obj \in R$, то из множества ребер-значений удаляются все пересекающиеся с добавляемым ребром: $H_1 = H \setminus \{h \in H \mid object(h) = obj \wedge (offs \leq offset(h) \leq offs + sz \vee offs \leq size(h) + offset(h) \leq offs + sz)\}$; добавляется ребро-значение $h_v = (obj, offs, v, sz)$, в итоге $H' = H_1 \cup h_v$; символьное значение добавляется в множество $V' = V \cup \{v\}$; остальные компоненты не меняются $O' = O, \Lambda' = \Lambda, P' = P, \Phi' = \Phi, \Pi' = \Pi, E' = E$.
  - Если $kind(obj) = abstract$, т.е. это абстрактный регион $obj \in A$, то перед записью происходит его материализация $(G', r) = materialize(G, obj)$ и затем запись $write(G', r, offs, sz)$.

  Аналогично операции чтения, до записи проверяется валидность объекта $valid(obj)$, отвечающая за ошибки NIL_DEREF и USE_AFTER_FREE, а также проверяется выход за границы объекта BUF_OVERRUN: $offs \geq 0 \wedge offs + sz \leq size(obj)$.
- Операция создания объекта $alloc(G, sz)$, возвращающая новый регион r и SMG $G' = (O', V', \Lambda', H', P', \Phi', \Pi', E')$ с новым регионом $O' = O \cup \{r\}$, Λ' доопределяется на новом объекте так, что $kind(r) = region$, $valid(r) = true$, $size(r) = sz$. Остальные компоненты не изменяются: $V' = V, H' = H, P' = P, \Phi' = \Phi, \Pi' = \Pi, E' = E$.
- Операция освобождения объекта $free(G, obj, offs)$, возвращающая новый SMG $G' = (O', V', \Lambda', H', P', \Phi', \Pi', E')$, где в Λ' изменяется значение $valid(obj) = false$. Остальные компоненты не изменяются: $O' = O, V' = V, H' = H, P' = P, \Phi' = \Phi, \Pi' = \Pi, E' = E$.

  Перед этим проверяется валидность объекта $valid(o)$ для информирования об ошибке DOUBLE_FREE и $offs = 0$ для UNALLOC_FREE.
- Операция добавления переменной $addVariable(G, \phi, name)$ создает новый регион размером соответствующим типу переменной $r = addRegion(sizeof(type))$, далее в стекфрейм функции добавляется соответствие имени переменной и этого региона $\phi' = \phi \cup \{name, r\}$.
- Операция удаление переменной $delVariable(G, \phi, name)$, обратная к добавлению, удаляет соответствие имени региону из стекфрейма $\phi' = \phi \setminus \{name, r\}$.
- Добавление стекфрейма функции $addStackFrame(G, function, ret, arg)$ добавляет локальные переменные функции и выделяет память на стеке этой функции.
- Удаление стекфрейма функции $dropStackFrame$. Удаляет локальные переменные функции и память, выделенную на стеке этой функции. После этого проверяется достижимость всех валидных объектов по ребрам-значениям из значений у объектов на стеке для информирования об утечках памяти MEM_LEAK.
- $getExprAddress(G, lval)$ рекурсивно разбирает выражение $lval$ и возвращает $(obj, offs)$ – объект и смещение в объекте.
- $getExprValue(G, rval)$ рекурсивно разбирает выражение $rval$, по необходимости добавляет символьные значения в граф G и в результате возвращает (G', v, e, sz) – измененный граф G', символьное значение, явное значение и размер.

Аналогично определяются операции записи и чтения указателя $writePointer(G, v, obj, offs)$/$readPointer(G, v)$; добавление/взятие явного значения $addExplicit(G, v, e)$/$getExplicit(G, v)$; добавление/удаление предикатов $addPredicate(G, pr(v_i))$/$delPredicates(G, v)$; добавления региона $addRegion(G, sz)$; удаления объекта $delObject(G, label)$.

2.2 Конкретизация SMG

Для определения отображения графа в конкретные образы памяти нам потребуется граф без абстрактных регионов. Для SMG $G = (O, V, \Lambda, H, P, \Phi, \Pi, E)$ с множеством абстрактных регионов памяти A в O обозначим за порождаемое им множество *графов памяти* $MG(G)$ все графы с точностью до изоморфизма, которые можно получить из G посредством материализации всех абстрактных регионов памяти, т.е. $\{G' \mid a \in A: (G', r) = materialize(G, a)\}$.

Теперь определим *конкретные образы памяти* $MI(G')$ для графов $G' = MG(G)$, которые затем будут использоваться при конкретизации абстрактного домена CPA (см. разд. 5).

Конкретные образы памяти $MI(G)$ графа памяти $G = (O, V, \Lambda, H, P, \Phi, \Pi, E)$ – множество отображений графа памяти на модель физической памяти компьютера в виде массива ячеек памяти с натуральными адресами N – $\mu: O \to N$ (заметим, что $O = R$), и значениями $val: N \times N \to N$, т.е. $val(addr, size)$ – значение, записанное в памяти начиная с $addr$ по $addr + size$, удовлетворяющее следующим свойствам.

- По нулевому адресу расположен только нулевой регион памяти, т.е. $\forall r \in R: \mu(r) = 0 \Leftrightarrow r = R_{NIL}$.
- Валидные регионы памяти не пересекаются, $\forall r_1, r_2 \in R: valid(r_1) \wedge valid(r_2) \Rightarrow \big(\mu(r_1), \mu(r_1) + size(r_1)\big) \cap \big(\mu(r_2), \mu(r_2) + size(r_2)\big) = \emptyset$.
- Указатели имеют значения адресов, по которым размещаются соответствующие регионы с учетом смещения, для каждой пары ребро-значение и ребро-указатель. Формально, для всех $h \in H, p \in P: val(h) = val(p)$ (связных друг с другом) выполнено $val\Big(\mu\big(object(h)\big) + offset(h), size(h)\Big) = \mu\big(object(p)\big) + offset(p)$.
- Поля, имеющие одинаковые значения, имеют одинаковые конкретные значения, т.е. для двух ребер-значений $h_1, h_2 \in H$ выполнено $val\Big(\mu\big(object(h_1)\big) + offset(h_1), size(h_1)\Big) = val\Big(\mu\big(object(h_2)\big) + offset(h_2), size(h_2)\Big)$.
- Поля, имеющие нулевые значения, заполнены нулями, т.е. для всех ребер-значений $h \in H: value(h) = NIL$ выполнено $val\Big(\mu\big(object(h)\big) + offset(h), size(h)\Big) = 0$.
- Значения, хранящиеся в E совпадают со значениями памяти.
- Значения, хранящиеся в памяти, удовлетворяют предикатам П.

3. Пример интерпретации выполнения программы

Подробно рассмотрим операции над графом SMG на примере простой программы (Листинг 1).

```
void f(void) {
    int * ar;
    uint ind;
    ar = malloc(SIZE);
    ind = random();
    assume (ind < SIZE)
    ar[ind] = 1;
    free(ar);
}
```

Листинг 1. Пример программы
Listing 1. Sample program

Подобная программа разворачивается в последовательность операций, показанных в табл. 1.

Табл. 1. Развертывание программы в последовательность операций SMG
Table 1. Transforming a program into a sequence of operations of SMG

Вход в функцию f	
`f(void)`	$G_1 = addStackFrame(G, "f", void)$
Декларация переменных	
`int * ar;`	$G_2 = addVariable\big(G_1, "ar", sizeof(int*)\big)$
`uint ind;`	$G_3 = addVariable\big(G_2, "ind", sizeof(uint)\big)$
Вызов функции аллокации и запись значения в переменную	
`ar = malloc(SIZE);`	$\{G_4, r_{alloc}\} = alloc(G_3, SIZE)$ $G_5 = writePointer(G_4, getObject("ar"), 0, r_{alloc}, 0)$
Вызов функции random и запись значения в переменную	
`ind = random();`	$v_{ind} = read\big(G_5, getObject("random"), 0, sizeof(uint)\big)$ $G_6 = write\big(G_5, getObject("ind"), 0, v_{ind}, sizeof(uint)\big)$
`assume (ind < len);`	$v_{ind} = read\big(G_6, getObject("ind"), 0, sizeof(uint)\big)$ $v_{len} = read\big(G_6, getObject("len"), 0, sizeof(uint)\big)$ $G_7 = addPredicate\big(G_6, E(v_{ind}) < E(v_{len})\big)$
`ar[ind] = 1;`	$v_{ind} = read\big(G_7, getObject("ind"), 0, sizeof(uint)\big)$ $v_{ar} = read\big(G_7, getObject("ar"), 0, sizeof(int*)\big)$ $r_{alloc} = readPointer(G_7, v_{ar})$ $G_8 = write\big(G_7, E(1), sizeof(int), r_{alloc}, v_{ind} * sizeof(int)\big)$
`free(ar);`	$v_{ar} = read\big(G_8, getObject("ar"), 0, sizeof(int*)\big)$ $r_{alloc} = readPointer(G_8, v_{ar})$ $G_9 = free(G_8, r_{alloc})$
Выход из функции f	
	$G_{10} = dropStackFrame(G_9, "f")$

4. Адаптивный статический анализ

Адаптивный статический анализ реализован в инструменте CPAchecker и подробно описан в статье [7], где приводится следующее формальное описание:

Адаптивный статический анализ $D = (D, \Pi, merge, stop, prec, \rightsquigarrow)$ состоит из абстрактного домена D, множества точностей П, оператора объединения абстрактных состояний $merge$, оператора проверки остановки анализа $stop$, функции изменения точности анализа $prec$ и оператора перехода $\rightsquigarrow$.

1. *Абстрактный домен* $D = (\mathcal{C}, \mathcal{E}, [\![\cdot]\!])$, где $\mathcal{C}$ – множество конкретных состояний; $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ – решетка, состоящая из множества абстрактных состояний E, частичного порядка $\sqsubseteq \subseteq E \times E$, верхнего элемента $\top \in E$, нижнего элемента $\bot \in E$ и оператора объединения состояний $\sqcup: E \times E \to E$; функция конкретизации $[\![\cdot]\!]: E \to 2^{\mathcal{C}}$, задающая значение абстрактного состояния как множество конкретных состояний.
2. *Множество точностей* П определяет возможные точности абстрактного домена.
3. *Оператор перехода* $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ строит для абстрактного состояния e и ребра графа потока управления g множество новых абстрактных состояний e' с точностями π.

4. *Оператор объединения* $merge: E \times E \times \Pi \to E$ ослабляет второе состояние на основе первого состояния и возвращает новое абстрактное состояние заданной точности.
5. *Оператор останова* $stop: E \times 2^E \times \Pi \to B$ определяет, покрывается ли абстрактное состояние с заданным уровнем точности множеством абстрактных состояний.
6. *Функция изменения точности анализа* $prec: E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ вычисляет новое абстрактное состояние и уточнение по заданному абстрактному состоянию с уточнением и множества абстрактных состояний с уточнениями.

4.1 Комбинация адаптивных анализов

Комбинация адаптивных анализов $D_1 = (D_1, \Pi_1, merge_1, stop_1, prec_1, \rightsquigarrow_1)$ и $D_2 = (D_2, \Pi_2, merge_2, stop_2, prec_2, \rightsquigarrow_2)$ также является адаптивным анализом $D = (D_1 \times D_2, \Pi_1 \times \Pi_2, merge_1 \times merge_2, stop_1 \times stop_2, prec_1 \times prec_2, \rightsquigarrow_1 \times \rightsquigarrow_2)$.

В статье [6] показано, что метод верификации, основанный на комбинации адаптивных анализов D, является полным, если отдельные методы, соответствующие анализам D_1 и D_2, являются полными. Подобный подход позволяет использовать существующие статические анализы из проекта CPAchecker.

5. Адаптивный статический анализ на основе символьных графов памяти (SMGCPA)

В концепции адаптивного статического анализа он задается следующим образом.

1. *Абстрактный домен* $D = (\mathcal{C}, \mathcal{E}, [\![\cdot]\!])$, где $\mathcal{C}$ – множество конкретных образов памяти, $\mathcal{E}$ – множество символьных графов памяти SMG и $[\![\cdot]\!] = MI(MG(G))$ – соответствие символьного графа памяти G его множеству конкретных образов памяти.
2. *Множество точностей* $\Pi = \emptyset$.
3. *Оператор перехода* $\rightsquigarrow (G, op)$$, на графе $G = (O, V, \Lambda, H, P, \Phi, \Pi, E)$ и ребре графа потока управления op. Результирующие графы будем обозначать как $G' = (O', V', \Lambda', H', P', \Phi', \Pi', E')$.
   a. Присваивание (assignment) $op = assign(lvalexpr, rvalexpr)$.
      i. Если $lvalexpr$ не является указателем, то:
         Пусть $(obj, offs) = getExprAddress(G, lvalexpr)$,
         $(\check{G}, v, e, sz) = getExprValue(G, rvalexpr)$,
         $\hat{G} = write(\check{G}, obj, offs, v, sz)$.
         Тогда результирующее E' совпадает с E за исключением значения на v, где $E'(v) = e$, а остальные компоненты G' берутся из $\hat{G}$.
      ii. Если $lvalexpr$ указатель, то:
         Пусть $(obj, offs) = getExprAddress(G, lvalexpr)$,
         $(\check{G}, v, e, sz) = getExprValue(G, rvalexpr)$,
         $(obj_1, offs_1) = getExprAddress(\check{G}, rvalexpr)$,
         $\hat{G} = write(\check{G}, obj, offs, v, sz)$.
         Тогда $\tilde{G} = writePointer(\hat{G}, v, obj_1, offs_1)$ и результирующее E' совпадает с E за исключением значения на v, где $E'(v) = e$, а остальные компоненты G' берутся из $\tilde{G}$.
   b. Условный переход $op = assume(expr)$ либо $op = if(expr)$ добавляет предикат перехода в Π. Проверка существования перехода осуществляется следующим образом.
      i. Если выражение можно вычислить с помощью явных значений и оно ложно, то данный переход отсекается и возвращается состояние $\bot$.
      ii. При наличии предикатов над символьными значениями Π осуществляются дополнительные проверки. По выражению $expr$ условия вычисляется сильнейшее постусловие. В результате получается формула, которая конкатенируется с предикатами из Π. Если формула оказывается неразрешимой, то переход отсекается и возвращается состояние $\bot$.
   c. Выделение памяти на стеке $op = alloca(expr)$.
      Пусть $(\check{G}, v, e, sz) = getExprValue(G, expr)$.
      Тогда на SMG выполняется $(\tilde{G}, r) = addRegion(\check{G}, e)$,
      а получившийся регион добавляется в стекфрейм текущей функции $\Phi' = \tilde{\Phi} \cup \{r, funcName\}$.
   d. Выделение памяти в куче $op = malloc(expr)$.
      Пусть $(\check{G}, v, e, sz) = getExprValue(G, expr)$.
      Тогда $G' = alloc(\check{G}, e)$.
   e. Освобождение памяти (free) $op = free(lvalexpr)$.
      Пусть $(obj, offs) = getExprAddress(G, lvalexpr)$.
      Тогда $G' = free(G, obj, offs)$.
   f. Выход из функции $funcName$ (return).
      $G' = dropStackFrame(G, funcName)$.
   g. Остальные операторы программы op описываются аналогично.
4. *Оператор объединения* $merge$ соответствует $join$ символьных графов SMG.
5. *Оператор останова* $stop$ – сравнение вложенности символьных графов памяти.
6. *Функция изменения точности анализа* $prec$ не используется.

6. Эксперименты

Описанный в разделе 5 анализ SMGCPA на основе расширенных символьных графов памяти был реализован в инструменте CPAchecker, ревизия 32467[1].

Для экспериментов использовались 3484 задания, подготовленных на основе драйверов ядра ОС Linux версии 4.18-rc5 (см. набор linux-4.18-rc5-memsafety[2]) с помощью системы Klever 2719].

Запуски производились с ограничением процессорного времени исполнения в 60 секунд на процессоре Intel Core i5-4590 в следующих конфигурациях:

- SMGCPA с выключенными предикатами – (стандартный SMG);
- SMGCPA с включенными предикатами – (расширенный SMG).

Результаты приведены в табл. 2. Вердикт *true* означает, что нарушений корректного использования памяти не выявлено. Вердикт *false* означает, что выявлено потенциальное нарушение корректного использования памяти. В скобках указано конкретное нарушение:

- *valid-deref* – BUF_OVERRUN или NIL_DEREF;
- *valid-memtrack* – MEM_LEAK;
- *valid-free* – USE_AFTER_FREE, DOUBLE_FREE или UNALLOC_FREE.

[1] https://github.com/mutilin/cpachecker/commit/0f486a996
[2] https://gitlab.com/sosy-lab/software/ldv-benchmarks

Табл.2. Изменение вердиктов стандартный SMG → расширенный SMG
Table 2. Change verdicts standard SMG → extended SMG

Стандартный SMG	Расширенный SMG	Количество
True	false(valid-deref)	4
false(valid-deref)	True	1
True	TIMEOUT	1
True	false(valid-free)	1
TIMEOUT	false(valid-deref)	19
TIMEOUT	false(valid-free)	1
TIMEOUT	True	5
false(valid-free)	false(valid-deref)	2
false(valid-memtrack)	false(valid-deref)	2
false(valid-memtrack)	TIMEOUT	1
false(valid-deref)	TIMEOUT	6

TIMEOUT означает, что не удалось получить вердикт в заданное время (60 сек.).

Переходы вердиктов в табл. 2 объясняются двумя эффектами:

i. отсечение недопустимых путей (см. описание *оператора перехода* в разд. 5).
ii. уточненная проверка выхода за границы памяти (см. описание операций `read` и `write` в подразделе 2.1).

В табл. 2 мы видим, что один переход из *false* в *true* был ложным срабатыванием (i).

Еще 4 перехода из *true* в *false(valid-deref)* были упущенными ошибками (ii). Остальные переходы связаны с тем, что (i) влияет на возможность получения вердикта в заданное время TIMEOUT. Суммарное количество позитивных переходов из TIMEOUT – 25, превышает количество негативных – 7.

Сокращение вердиктов TIMEOUT связано с сокращением количества состояний, построенных в процессе анализа за счет отсечения по условию (i). Увеличение состояний также возможно, так как для состояний с несовместными наборами предикатов не производится слияние в операторе `merge`.

Всего количество состояний уменьшилось в 836 заданиях, увеличилось в 240 и не поменялось в 2354.

При верификации предикаты использовались в 3225 модулях из 3484. С их помощью было отсечено 14163 недостижимых путей, что составляет в среднем 4.39 на задание, а максимально в одном задании отсекается 1271 путь.

Эффект (ii) позволяет обнаружить 25 новых ошибок.

7. Заключение

В работе представлено расширение символьных графов памяти SMG, позволяющее использовать предикаты над символьными значениями для повышения точности анализа.

Во-первых, продемонстрирована возможность отсечения недостижимых путей, уменьшая количество ложных сообщений об ошибках.

Во-вторых, использование новых проверок на основе предикатов над символьными значениями позволяет находить новые ошибки. Так стало возможным проверять выход за границу объекта в операциях чтения и записи SMG не только на явных значениях, но и на символьных.

Практическая реализация выполнена на основе инструмента CPAchecker и проведены эксперименты на драйверах ядра ОС Linux версии 4.18-rc5, по которым получено 3484 заданий.

При верификации предикаты использовались в 3225 заданиях из 3484 с помощью них было отсечено 14163 недостижимых путей и обнаружено 25 новых ошибок.

Список литературы / References

[1]. Klein G., Elphinstone K. et al. sel4: Formal verification of an os kernel. In Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, 2009, pp. 207–220.
[2]. Stewart G., Beringer L., Cuellar S., Appel A.W. Compositional compcert. In Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2005, pp. 275–287.
[3]. Beyer D., Keremoglu M.E.: CPAchecker: A tool for configurable software verification. Lecture Notes in Computer Science, vol. 6806, 2011, pp. 184-190.
[4]. Donaldson A.F., Haller L., Kroening D., R¨ummer P. Software verification using k-induction. Lecture Notes in Computer Science, vol. 6887, 2011, pp. 351-368.
[5]. Clarke E., Grumberg O., Jha S., Lu Y., Veith H. Counterexample-guided abstraction refinement. Lecture Notes in Computer Science, vol. 1855, 2000, pp. 154–169.
[6]. Beyer D., Keremoglu, M.E., Wendler, P. Predicate abstraction with adjustable-block encoding. In Proc. of the 10th International Conference on Formal Methods in Computer-Aided Design, 2010, pp. 189–197.
[7]. Beyer D., Henzinger T., Theoduloz G. Program analysis with dynamic precision adjustment. In Proc. of the 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. pp. 29–38.
[8]. Beyer D., Löwe S. Explicit-state software model checking based on CEGAR and interpolation. Lecture Notes in Computer Science, vol. 7793, 2013, pp. 146–162.
[9]. Biere A., Cimatti A., Clarke E.M., Zhu Y. Symbolic model checking without bdds. Lecture Notes in Computer Science, vol. 1579, 1999, pp. 193–207.
[10]. Graf S., Saidi H. Construction of abstract state graphs with PVS. Lecture Notes in Computer Science, vol. 1254, 1997, pp. 72–83.
[11]. Andrianov P., Friedberger K., Mandrykin M., Mutilin V., Volkov A. CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions. Lecture Notes in Computer Science, vol. 10206, 2017, pp. 355–359.
[12]. Yang H., Lee O., Berdine J., Calcagno C., Cook B., Distefano D., O'Hearn P. Scalable shape analysis for systems code. Lecture Notes in Computer Science, vol. 5123, 2008, pp. 385–398.
[13]. Volkov A., Mandrykin M. Predicate Abstractions Memory Modeling Method with Separation into Disjoint Regions. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 4, 2017, pp. 203-216. DOI: 10.15514/ISPRAS-2017-29(4)-13.
[14]. Beyer D., Henzinger T., Jhala R., Majumdar R. The software model checker BLAST. International Journal on Software Tools for Technology Transfer, vol. 9, issue 5-6, 2007, 505–525.
[15]. Shved P., Mandrykin M., Mutilin V. Predicate analysis with BLAST 2.7. Lecture Notes in Computer Science, vol. 7214, 2012, pp. 525–527.
[16]. Ball T., Bounimova E., Kumar R., Levin V. SLAM2: Static driver verification with under 4% false alarms. In Proc. Of the 10th International Conference on Formal Methods in Computer-Aided Design, 2010. pp. 35–42.
[17]. Sagiv M., Reps T.W., Wilhelm R. Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems, vol. 24, issue 3, 2002, pp. 217–298.
[18]. Beyer D., Henzinger T.A., Théoduloz G. Lazy shape analysis. Lecture Notes in Computer Science, vol. 4144, 2006, pp. 532–546.
[19]. Reynolds J.C. Separation logic: A logic for shared mutable data structures. In Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science, 2002, pp. 55–74.
[20]. Berdine J., Cook B., Ishtiaq S. Slayer Memory safety for systems-level code. Lecture Notes in Computer Science, vol. 6806, 2011, pp. 178-183.
[21]. Jacobs B., Smans J., Piessens F. A quick tour of the verifast program verifier. Lecture Notes in Computer Science, vol. 6461, 2010, pp. 304–311.
[22]. Volkov A., Mandrykin M. Predicate Abstractions Memory Modeling Method with Separation into Disjoint Regions. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 4, 2017, pp. 203-216. DOI: 10.15514/ISPRAS-2017-29(4)-13.

[23]. Calcagno C., Distefano D. et al. Moving fast with software verification. Lecture Notes in Computer Science, vol. 9058, 2015, pp. 3-11.
[24]. Beyer D. Automatic verification of C and Java Programs: SV-COMP 2019. Lecture Notes in Computer Science, vol. 11429, 2019, pp. 133–155.
[25]. Dudka K., Peringer P., Vojnar T.: Byte-precise verification of low-level list manipulation. Lecture Notes in Computer Science, vol. 7935, 2013, pp. 215–237.
[26]. Vasilyev A.A. Static verification for memory safety of Linux kernel drivers. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 6, 2018. pp. 143-160. DOI: 10.15514/ISPRAS-2018-30(6)-8.
[27]. Novikov E., Zakharov I. Towards automated static verification of GNU C programs. Lecture Notes in Computer Science, vol. 10742, 2018, pp. 402–416.

Информация об авторах / Information about authors

Антон Александрович ВАСИЛЬЕВ – стажер-исследователь, аспирант ИСП РАН. Сфера научных интересов: верификация программ, теория графов, математическая логика.

Anton Aleksandrovich VASILIEV – intern researcher, PhD student of ISP RAS. Research interests: program verification, graph theory, mathematical logic.

Вадим Сергеевич МУТИЛИН – кандидат физико-математических наук, старший научный сотрудник ИСП РАН. Сфера научных интересов: верификация программ, статический анализ, операционные системы.

Vadim Sergeevich MUTILIN – PhD in physical and mathematical sciences, Senior Researcher at ISP RAS. Research interests: program verification, static analysis, operating systems.