Метод выявления некоторых типов ошибок работы с памятью в бинарном коде программ

В.В. Каушан <korpse@ispras.ru>
А.Ю. Мамонтов <mamontov@ispras.ru>
В.А. Падарян <vartan@ispras.ru>
А.Н. Федотов <fedotoff@ispras.ru>
ИСП РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В статье рассматривается метод выявления ошибок работы с памятью в бинарном коде программ, таких как выход за границы буфера при чтении и записи. Предлагаемый метод основывается на использовании динамического анализа и символьного выполнения. Метод применяется к бинарным файлам программ без дополнительной отладочной информации. Описанный метод был реализован в виде программного инструмента. Возможности инструмента продемонстрированы на примере поиска ошибок в 11 программах, которые работают под управлением ОС Windows и Linux, в 7 из них ошибки не были исправлены на момент написания статьи.

Ключевые слова: выявление уязвимостей; бинарный код; динамический анализ; символьное выполнение.

DOI: 10.15514/ISPRAS-2015-27(2)-7

Для цитирования: Каушан В.В., Мамонтов А.Ю., Падарян В.А., Федотов А.Н. Метод выявления некоторых типов ошибок работы с памятью в бинарном коде программ. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 105-126. DOI: 10.15514/ISPRAS-2015-27(2)-7.

1. Введение

На сегодняшний день, важными практическими задачами в компьютерной безопасности является поиск ошибок в программном обеспечении (ПО). Люди всё чаще используют различное ПО для собственных нужд: передача и получение разной (в том числе и конфиденциальной) информации, использование программ для работы и т.д. Таким образом, обеспечение надёжности, конфиденциальности и доступности работающего программного обеспечения является актуальной задачей.

Нарушать работу программного обеспечения могут различного рода ошибки. Среди всевозможных типов ошибок присутствует большой класс – ошибки в реализации (дефекты). В свою очередь данные ошибки разделяются на множество различных подклассов [1]. Мы будем рассматривать дефекты, приводящие к нарушению доступа к памяти. Под нарушением доступа к памяти, будем понимать выход за границы буфера в памяти при операции чтения или записи в этот буфер. В статье предлагается метод поиска таких дефектов. Зачастую нарушение доступа к памяти происходит из-за неправильного копирования данных, выделения и освобождения памяти. Поиск дефектов такого рода осуществляется в бинарном коде, где отсутствует какая-либо информация о границах буферов памяти. Следовательно, эту информацию необходимо восстановить. Предложенный метод основан на анализе трасс выполнения программ, полученных при помощи полносистемного симулятора [2-7]. Для заданного набора входных данных фиксированного размера (далее эти входные данные обозначаются как *префикс*), подбираются значения префикса и значение длины этого префикса, при которых происходит нарушение работы с памятью. Для подбора этих данных используется технология символьной интерпретации [8]. Применение символьной интерпретации для анализа трасс выполнения описано в работе [9]. Одним из дополнений в предлагаемом подходе по сравнению с работой [9] является наличие абстрактного значения длины у начального набора данных. Помимо того, предлагается особая обработка некоторых функций: ввод пользовательских данных, строковые функции, функции выделения и освобождения памяти.

Статья организована следующим образом. Во втором разделе описываются теоретические аспекты предлагаемого метода. В третьем разделе описана общая схема работы программного инструмента, реализующего данный метод. В четвертом разделе рассказывается о деталях реализации этого инструмента. В пятом разделе представлены результаты применения данного метода. В шестом разделе представлен обзор близких работ. В последнем, седьмом, разделе анализируются результаты и обсуждаются дальнейшие направления исследований.

2. Моделирование работы с буферами памяти в бинарном коде

Для решения поставленной задачи необходимо формально описать правила доступа к памяти, при нарушении которых возникает ошибочная ситуация. Чаще всего, нарушение работы с памятью происходит во время выполнения машинных команд, которые обращаются к памяти с использованием косвенной адресации. При косвенном обращении к памяти, адрес ячейки памяти, в который происходит загрузка или выгрузка данных, может зависеть от входных данных, что потенциально позволяет обращаться за границы допустимой области памяти. Эти машинные команды могут входить в состав

библиотечных функций работы с памятью. Современные версии библиотечных функций используют расширения процессора, такие как SSE2, для ускорения работы. Анализ машинных команд SSE2 значительно усложняет задачу поиска ошибок. В то же время, семантика многих библиотечных функций работы с памятью известна, что позволяет обрабатывать их как единое целое вместо обработки отдельных инструкций, входящих в эти функции. Можно значительно упростить задачу поиска ошибок доступа к памяти используя следующий подход. Для машинных команд, входящих в состав функций с известной семантикой, выполняется обработка функций целиком с помощью правил, соответствующих семантике этих функций. Для всех остальных машинных команд используется свой набор правил. Таким образом, вытекает необходимость явного выделения в коде программы функций, отвечающих за работу с памятью и формального описания их свойств, что позволит вести учет доступных буферов памяти и обнаруживать выход за их пределы. Помимо того, анализ помеченных данных требует задания функций, осуществляющих ввод пользовательских данных.

Разобьём правила доступа к памяти на две группы:

- правила, описывающие нарушения при использовании библиотечных функций;
- правила, описывающие нарушения при косвенной адресации в машинных инструкциях.

Для задания правил первой группы важным классом функций являются строковые функции, такие как: копирование, конкатенация и определение длины нуль-терминированных строк. В предположении, что семантика таких функций известна, их можно моделировать целиком, не погружаясь в код реализации.

При описании правил доступа к памяти на уровне машинных инструкций достаточно рассматривать адрес памяти, по которому происходит доступ.

Приведенные рассуждения приводят к следующим определениям и правилам интерпретации трассы машинных команд.

Под трассой будем понимать упорядоченную последовательность пар

$$(\mathrm{instr}_0, \mathrm{M}_0)$$
$$(\mathrm{instr}_1, \mathrm{M}_1)$$
$$\ldots$$
$$(\mathrm{instr}_{N-1}, \mathrm{M}_{N-1})$$

где $\mathrm{instr}_i, 0 \le i < N$ – выполнявшаяся на шаге i машинная команда, а M_i – состояние памяти компьютера перед выполнением этой команды. Память представляет набор адресуемых машинных слов $\mathrm{M} = (\mathrm{m}_0 \ldots \mathrm{m}_{\mathrm{TOM}})$ в котором единообразно объединены все ячейки, обладающие состоянием: как, собственно, оперативная память компьютера, так и регистры.

Машинная команда – тройка, описывающая операцию над данными и ее фактические операнды: $\mathrm{instr} = \langle \text{КОП}, \{\mathrm{use}_i\}, \{\mathrm{def}_j\}\rangle$, где КОП – код операции,

$\mathrm{use}, \mathrm{def} \in \mathrm{M}$ – множества ячеек, считываемых и записываемых данной машинной командой. В наборах операндов явно указываются ячейки памяти, которые считываются и записываются при выполнении команды. Непосредственно адресуемые операнды не указываются. В случае косвенной адресации явно указываются обе ячейки: фактический операнд и ячейка, задающая адрес.

В работах, описывающих динамический анализ помеченных данных, перечень кодов операций реализует минимальный набор RISC команд [10], расширив его псевдокомандой ввода пользовательских данных. В данном случае расширение предполагает следующие 7 псевдокоманд:

- выделение памяти $\langle \mathrm{malloc}, \{\mathrm{size}\}, \{\mathrm{addr}\}\rangle$,
- освобождение памяти $\langle \mathrm{free}, \{\mathrm{addr}\}, \{\}\rangle$,
- определение длины строки $\langle \mathrm{strlen}, \{\mathrm{addr}\}, \{\mathrm{len}\}\rangle$,
- копирование строк $\langle \mathrm{strcpy}, \{\mathrm{dest.addr}, \mathrm{src.addr}\ , \}, \{\}\rangle$,
- конкатенация строк $\langle \mathrm{strcat}, \{\mathrm{dest.addr}, \mathrm{src.addr}\ , \}, \{\}\rangle$,
- копирование фиксированной длины $\langle \mathrm{memcpy}, \{\mathrm{dest.addr}, \mathrm{src.addr}\ , \mathrm{n}\}, \{\}\rangle$,
- ввод пользовательских данных $\langle \mathrm{read}, \{\mathrm{addr}, \mathrm{size}\}, \{\mathrm{readed}\}\rangle$

Остальные команды:

- унарная операция $\langle \Diamond_u, \{\mathrm{use\_cell}\}, \{\mathrm{def\_cell}\}\rangle$,
- бинарная операция $\langle \Diamond_b, \{\mathrm{use\_cell}_1, \mathrm{use\_cell}_2\}, \{\mathrm{def\_cell}\}\rangle$,
- загрузка данных из памяти $\langle \mathrm{load}, \{\mathrm{src\_cell}, \mathrm{src\_addr\_cell}\}, \{\mathrm{dest\_cell}\}\rangle$,
- выгрузка в память $\langle \mathrm{store}, \{\mathrm{src\_cell}, \mathrm{dest\_addr\_cell}\}, \{\mathrm{dest\_cell}\}\rangle$,
- безусловная передача управления $\langle \mathrm{jmp}, \{[\mathrm{addr\_cell}]\}, \{\}\rangle$
- для удобства интерпретации трассы явно разделены сработавший и не сработавший условные переходы: $\langle \mathrm{jct}, \{\mathrm{cond\_cell}[, \mathrm{addr\_cell}]\}, \{\}\rangle$ $\langle \mathrm{jcf}, \{\mathrm{cond\_cell}, \mathrm{addr\_cell}\}, \{\}\rangle$, соответственно.

Стоит отметить, что приведённый список псевдокоманд, обеспечивающих обработку функций с известной семантикой, не является конечным и при необходимости может быть расширен.

В процессе интерпретации поддерживается множество символьных переменных $\mathfrak{S}$, над которыми допустимы унарные и бинарные операции, а также операции загрузки и выгрузки данных.

Для описания работы с памятью используем понятие буфера, имеющего символьную длину [11]. Далее в тексте будем обозначать его как L-буфер. Задается L-буфер l в виде тройки $l = [\![\mathrm{base}, \mathrm{clen}, \mathrm{slen}]\!]$, содержащей адрес базы, реальную и символьную длину, отвечающую за абстрактный размер буфера. На рис. 1. схематично представлен L- буфер.

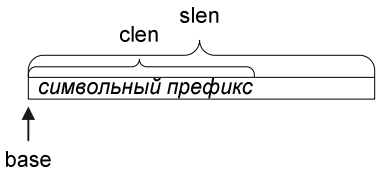

Рис. 1. L-буфер.

Интерпретация шагов трассы происходит в контексте, состоящем из четырёх компонент: текущего символьного состояния, предиката пути Pp, множества символьных переменных $\mathfrak{S}$ и двух множеств *L*-буферов памяти *A* и *I*.

Отображение Δ связывает ячейки (машинные слова) памяти и выражения над символьными переменными. Получение символьного и конкретного численного значения ячейки памяти будем описывать $\Delta[m]$ и $M[m]$ соответственно.

В ходе интерпретации трассы на основе предиката пути и дополнительных ограничений проверяются условия выхода за границы буферов. Для обозначения проверок вводится функция *Assert(условие)*, которая проверяет истинность заданного условия.

Также существует отдельный вид *L*-буферов, которые соответствуют областям выделенной памяти. У таких буферов абстрактная длина может быть константным выражением. В отличие от работы [11], в предлагаемом подходе каждый *L*-буфер помещен в одно из множеств: *A* или *I*, в зависимости от того, соответствует буфер выделению памяти или вводу данных. Входные данные могут поступать из различных источников: сеть, файлы, аргументы командной строки. Поддержка двух множеств *A* и *I* позволит в дальнейшем выполнять дополнительные проверки.

Введем отображение, позволяющие получить для заданного адреса буфер в заданном множестве $T(X, addr) = x, x \in X$, адрес *addr* указывает на одну из ячеек буфера *x, X* одно из множеств *A* и *I*. Для краткости будем в дальнейшем обозначать отображения, связанные с конкретным множеством буферов как *T(A)* и *T(I)*.

Для отображений *T* и Δ введём операцию обновления $\leftarrow$. Например, задание связи между ячейкой памяти m и символьным выражением s будем обозначать как $\Delta[m \leftarrow s]$. Правила интерпретации команд представлены ниже в виде продукций. Для каждой команды приводится ее запись, под которой размещена продукция, описывающая преобразование контекста. В верхней части продукции приведены выполняющиеся действия, в нижней – состояния контекста до и после интерпретации команды. Исходя из рассмотренного выше, контекст представляется кортежем $\Delta, Pp, A, I, \mathfrak{S}$. Для краткости, в нижней части продукции показываются только изменившиеся элементы контекста.

$$\langle malloc, \{size\}, \{addr\}\rangle$$

$$\frac{l = [\![M(addr), M(size), \Delta[size]]\!]}{A \vdash A \cup l}$$

Malloc. Во множестве *A* происходит создание *L*-буфера. В зависимости от операнда *size*, абстрактная длина *L*-буфера может иметь как символьное, так и константное (конкретное) значение.

$$\langle free, \{addr\}, \{\}\rangle$$

$$\frac{l = T(A, M(addr)), Assert(M(addr) == l.base)}{A \vdash A \backslash l}$$

Free. Происходит удаление *L*-буфера из множества *A*, конкретное значение операнда *addr* должно соответствовать полю base у *L*-буфера l.

$$\langle strlen, \{addr\}, \{len\}\rangle$$

$$\frac{l = T(I, M(addr))}{\Delta \vdash \Delta[len \leftarrow l.slen - (M(addr) - l.base)]}$$

Strlen. Происходит присваивание операнду *len* значения абстрактной длины *L*-буфера, который найден, исходя из значения операнда *addr*.

$$\langle strcpy, \{dest.addr, src.addr, \ldots\}, \{\ldots\}\rangle$$

$$\frac{i = T(I, M(src.addr)), i' = \left[\!\!\left[\begin{matrix} M(dst.addr), i.clen - (M(src.addr) - i.base), \\ i.slen - (M(src.addr) - i.base) \end{matrix}\right]\!\!\right], \quad a = T(A, M(dst.addr)), Assert(i' \sqsubset a)}{\Delta, I \vdash \Delta[m_{i'.base}, \ldots, m_{i'.base+i'.clen-1} \leftarrow m_{M(src.addr)}, \ldots, m_{M(src.addr)+i'.clen-1}], I \cup i'}$$

Strcpy (Strcat). Происходит проверка на переполнение буфера при копировании (конкатенации), также создаётся новый буфер из множества *I*, затем обновляется отображения ячеек памяти на символьные переменные.

$$\langle \text{strcat}, \{\text{dest.addr}, \text{src.addr}\ , \}, \{\}\rangle$$

$$\frac{\begin{array}{c} i_{src} = T\big(I, M(\text{src.addr})\big), i_{dst} = T\big(I, M(\text{dst.addr})\big), \\ \text{offset}_{dst} = (M(\text{dst.addr}) - i_{dst}.\text{base}), \\ \text{offset}_{src} = (M(\text{src.addr}) - i_{src}.\text{base}), \\ i' = \left[\!\!\left[\begin{array}{c} M(\text{dst.addr})\ , i_{src}.\text{clen} - \text{offset}_{src} + i_{dst}.\text{clen}, \\ i_{src}.\text{slen} - \text{offset}_{src} + i_{dst}.\text{slen} \end{array} \right]\!\!\right], \\ a = T\big(A, M(\text{dst.addr})\big), \text{Assert}(i' \sqsubset a) \end{array}}{\Delta, I \vdash \Delta[m_{i_{dst}.\text{base}+i_{dst}.\text{clen}}, .., m_{i_{dst}.\text{base}+i'.\text{clen}-1} \leftarrow m_{M(\text{src.addr})}, .., m_{M(\text{src.addr})+i_{src}.\text{clen}-\text{offset}_{src}-1}], I \cup i' \backslash i_{ds}}$$

$$\langle \text{memcpy}, \{\text{dest.addr}, \text{src.addr}\ , \text{n}\}, \{\}\rangle$$

$$\frac{\begin{array}{c} i_{src} = T\big(I, M(\text{src.addr})\big), i_{dst} = T\big(I, M(\text{dst.addr})\big), \\ i' = [\![M(\text{dst.addr})\ , M(\text{n}), i_{src}\Delta(\text{n})]\!], \\ \text{Assert}\big(i_{src}.\text{slen} - (\text{src.addr} - i_{src}.\text{base}) < \Delta(\text{n})\big) \\ a_{src} = T\big(A, M(\text{src.addr})\big),\ \text{Assert}(\ i_{src} \sqsubset a_{src}) \\ a = T\big(A, M(\text{dst.addr})\big), \text{Assert}(i' \sqsubset a) \end{array}}{\Delta, I \vdash \Delta\big[m_{i'.\text{base}}, .., m_{i'.\text{base}+i'.\text{clen}-1} \leftarrow m_{M(\text{src.addr})}, .., m_{M(\text{src.addr})+i'.\text{clen}-1}\big], I \cup i'}$$

Memcpy. Происходит проверка на выходы за границы при чтении буфера источника и на переполнение буфера назначения при копировании, далее создаётся новый буфер из множества *I*, затем обновляется отображения ячеек памяти на символьные переменные.

$$\langle \text{read}, \{\text{addr}, \text{size}\}, \{\text{readed}\}\rangle$$

$$\frac{\begin{array}{c} a_{src} = T\big(A, M(\text{src.addr})\big),\ s_0, .. s_{M(\text{readed})} \in \mathfrak{S} \\ i_{src} = [\![M(\text{addr})\ , M(\text{readed}), s_{\text{len}}]\!], s_{\text{len}} \in \mathfrak{S} \\ \text{Assert}\big(\ i_{src} \sqsubset a_{src}\ \&\wedge i_{src}.\text{slen} \le M(\text{size})\big) \end{array}}{\Delta, I \vdash \Delta\big[m_{i_{src}.\text{base}}, .., m_{i_{src}.\text{clen}-1}, \leftarrow s_0, .. s_{M(\text{readed})}\ \big], I\ \cup i_{src}}$$

Read. Происходит проверка на выход за границы буфера при чтении данных из внешнего источника, учитывая максимальный размер считанных данных, также создаётся новый буфер из множества *I*, затем обновляется отображения ячеек памяти на символьные переменные.

$$\langle \Diamond_u, \{\text{use\_cell}\}, \{\text{def\_cell}\}\rangle$$

$$\frac{}{\Delta \vdash \Delta[\text{def\_cell} \leftarrow \Diamond_u\ \Delta[\text{use\_cell}]]}$$

Унарная операция (Бинарная операция). Происходит обновление отображения ячеек памяти на символьные переменные с учётом выполнения операции.

$$\langle \Diamond_b, \{\text{use\_cell}_1, \text{use\_cell}_2\}, \{\text{def\_cell}\}\rangle$$

$$\frac{}{\Delta \vdash \Delta[\text{def\_cell} \leftarrow \Delta[\text{use\_cell}_1]\ \Diamond_b\ \Delta[\text{use\_cell}_2]]}$$

$$\langle \text{load}, \{\text{src\_cell}, \text{src\_addr\_cell}\}, \{\text{dest\_cell}\}\rangle$$

$$\frac{\begin{array}{c} a_{src} = T\Big(A, M\big(\text{src}_{\text{addr}_{\text{cell}}}\big)\Big), \\ \text{Assert}\big(\Delta\big(\text{src}_{\text{addr}_{\text{cell}}}\big) < a_{src}.\text{base} + a_{src}.\text{slen}\big) \end{array}}{\Delta \vdash \Delta[\text{dest}_{\text{cell}} \leftarrow \text{src}_{\text{cell}}]}$$

Load. Происходит проверка на выход за границы *L*-буфера при чтении данных из него. Описывает операции загрузки данных из памяти.

$$\langle \text{store}, \{\text{src\_cell}, \text{dest\_addr\_cell}\}, \{\text{dest\_cell}\}\rangle$$

$$\frac{\begin{array}{c} a_{dst} = T\Big(A, M\big(\text{dest}_{\text{addr}_{\text{cell}}}\big)\Big) \\ \text{Assert}\big(\Delta\big(\text{dest}_{\text{addr}_{\text{cell}}}\big) < a_{dst}.\text{base} + a_{dst}.\text{slen}\big)' \end{array}}{\Delta \vdash \Delta[\text{dest}_{\text{cell}} \leftarrow \text{src}_{\text{cell}}]}$$

Store. Происходит проверка на выход за границы *L*-буфера при записи данных в него. Описывает операцию выгрузки данных в память.

$$\langle \text{jct}, \{\text{cond\_cell}, \text{addr\_cell}\}, \{\}\rangle$$

$$\frac{}{\text{Pp} \vdash \text{Pp}\ \cup (\Delta[\text{cond\_cell}] = \text{true})}$$

Jct (Jcf). Описывает операцию выполнения (не выполнения) условного перехода, добавляя уравнение в предикат пути.

$\langle \text{jcf}, \{\text{cond\_cell}, \text{addr\_cell}\}, \{\} \rangle$
$\dfrac{}{\text{Pp} \vdash \text{Pp} \cup (\Delta[\text{cond\_cell}] = \text{false})}$

3. Схема работы

В данном разделе описана схема работы алгоритма, реализующего представленный метод.

3.1 Используемые методы анализа

Алгоритм реализован в рамках среды динамического анализа бинарного кода [12]. Эта среда позволяет работать с трассами выполнения программ, полученными в результате работы полносистемного эмулятора. Трасса содержит последовательность выполненных инструкций процессора, а также значения регистров перед выполнением каждой инструкции. Анализ трасс выполнения позволяет анализировать поведение программы после того, как она была выполнена. Благодаря этому, алгоритмы анализа не замедляют работу анализируемых программ, что позволяет анализировать программы, работающие с сетью, и программы, противодействующие отладке.

Среда анализа бинарного кода предоставляет различные инструменты и структуры данных для высокоуровневого анализа: разметка трассы на процессы и потоки ОС, выделение вызовов функций, построение графа потока управлений, графа зависимостей по данным и многие другие. Для сокращения числа анализируемых инструкций используется алгоритм слайсинга трассы, основанный на анализе графа зависимостей по данным. Алгоритм слайсинга трассы отбирает те инструкции, которые имеют отношение к обработке или формированию заданного буфера данных в памяти. В современных процессорных архитектурах содержится множество инструкций со сложной семантикой и нетривиальными побочными эффектами. Для унификации обработки инструкций традиционно применяется метод, основанный на трансляции инструкций в промежуточное представление. В используемой среде анализа бинарного кода используется промежуточное представление Pivot [13], позволяющее единообразно описывать операционную семантику инструкций различных процессорных архитектур.

В процессе работы алгоритма выполняются проверки нарушений доступа к памяти с помощью SMT-решателя. В качестве SMT-решателя в системе анализа бинарного кода используется решатель Z3 [14].

3.2 Параметры алгоритма

Работа алгоритма начинается с некоторого шага трассы, заданного аналитиком. Также, для этого шага трассы задаётся буфер с входными данными. Для этого буфера создаётся абстрактный буфер с символьной длиной, при этом данные из входного буфера выступают в качестве префикса. Помимо этого, иногда конкретная длина входного буфера хранится отдельно от самих данных (например, в случае чтения данных с помощью функции recv). В этом случае, конкретная длина входного буфера связывается с символьной длиной соответствующего ему абстрактного буфера. Аналитик может указать расположение ячейки памяти, в которой хранится конкретная длина входного буфера. Если же длина буфера с входными данными задана неявно (например, в случае нуль-терминированной строки), эту ячейку памяти указывать не обязательно.

3.3 Трансляция инструкций

Начиная с заданного аналитиком шага трассы, начинается обработка инструкций процессора с учётом зависимостей по данным. Каждая инструкция сначала транслируется в промежуточное представление, а затем на основе этого промежуточного представления создаются формулы и уравнения для SMT-решателя. Уравнения, которые создаются во время трансляции инструкций, добавляются в предикат пути – множество уравнений, описывающих прохождение программы по некоторому пути выполнения. Перед началом трансляции множество выделенных буферов памяти пополняется набором буферов, которые уже выделены, но ещё не освобождены. Для отбора инструкций используется алгоритм, аналогичный алгоритму слайсинга трассы с некоторыми дополнениями. Главной особенностью работы алгоритма является пропуск трансляции отдельных функций. Многие библиотечные функции имеют известные побочные эффекты, которые можно описать явно с помощью уравнений над входными и выходными параметрами. Такой подход позволяет не транслировать инструкции, принадлежащие известной библиотечной функции, а вместо этого обновлять состояние контекста интерпретации в соответствии с описанием побочных эффектов для этой библиотечной функции. Это позволяет значительно сократить сложность уравнений, а также реализовать возможность работы с буферами символьной длины. Кроме того, использование слайсинга позволяет значительно сократить количество обрабатываемых инструкций процессора за счёт отбора только тех инструкций, которые связаны с обработкой помеченных данных. В табл. 1 приведено сравнение количества обрабатываемых инструкций. Во втором столбце приведены значения количества инструкций, которые были бы обработаны без использования слайсинга. В третьем столбце приведено количество инструкций, отобранных в результате работы слайсинга, а в

четвёртом столбце – количество инструкций, отобранных в результате работы слайсинга с использованием интерпретации функций с известной семантикой.

Табл. 1. Сравнение количества обрабатываемых инструкций.

Программа	Количество инструкций	Размер слайса	Количество оттранслированных инструкций
httpdx	712029	12576	12367
GoldMP4Player	22009330	9353	9347
mysql_plugin	220710	8268	105

На примере программы mysql_plugin видно, что с помощью интерпретации функций можно значительно сократить число обрабатываемых инструкций.

Во время обработки вызова библиотечной функции выполняются проверки различных условий нарушения доступа к памяти. Для этого составляются уравнения предиката безопасности. После этого предикат пути и предикат безопасности объединяются и передаются SMT-решателю. Если система уравнений оказалась совместной, решение системы уравнений будет содержать длину входного буфера, при которой происходит нарушение доступа к памяти. Семантика обработки некоторых библиотечных функций описана в разделе 2. Все остальные инструкции, отобранные алгоритмом слайсинга, обрабатываются с помощью механизма, который был описан в работе [9].

Следует отметить, что обрабатываемые вызовы библиотечных функций в трассе могут не соответствовать вызовам этих же функций в исходном коде анализируемой программы. Чаще всего несоответствие возникает из-за оптимизаций компилятора, встраивающих код функции в программу в месте вызова этой функции. В этом случае вызов функции в трассе не будет обработан с помощью описанной выше семантики, а вместо этого произойдёт обработка отдельных инструкций, которые соответствуют библиотечной функции. Это, в свою очередь, может привести к добавлению дополнительных ограничений на размер входных данных и ложноотрицательным результатам работы алгоритма.

Кроме вызовов библиотечных функций, специальным образом обрабатываются вызовы всех остальных функций. Если для вызова функции известна информация о размере кадра стека, множество выделенных буферов пополняется буфером, который описывает область памяти, соответствующую кадру стека. При обработке возврата из этой функции, буфер, соответствующий кадру стека, удаляется из множества выделенных буферов.

3.4 Завершение работы алгоритма

В процессе работы алгоритма проверяется нарушение доступа к памяти. Если при очередной проверке устанавливается факт нарушения, алгоритм

завершается. Результатом работы алгоритма является длина входного буфера, а также значение префикса.

4. Реализация дополнительных алгоритмов

В данном разделе описаны особенности реализации алгоритма разметки памяти и дополнение алгоритма выделения вызовов функций в рамках системы анализа бинарного кода.

4.1 Разметка памяти

В бинарном коде информация о переменных и буферах в памяти в явном виде отсутствует, поэтому для поиска выходов за границы буферов сначала нужно восстановить эту информацию. При реализации данного метода восстанавливалась информация о динамической и автоматической памяти.

Разметка динамической памяти. Составление карты динамической памяти основывается на использовании моделей функций [15]. Под моделью функции будем понимать функцию с описанными входными и выходными параметрами в виде ячеек памяти и регистров. Задаются три модели, каждая из которых соответствует функциям выделения (*alloc*), освобождения (*free*), и изменения размера уже выделенной памяти (*realloc*). Для каждой модели задаются параметры, имеющие определенную семантику. Для модели *alloc* задаётся размер (входной параметр) и адрес выделенной памяти (выходной параметр). Для модели *free* задаётся адрес освобождаемой памяти (входной параметр). Для модели *realloc* задаётся адрес буфера, размер которого будет изменён (входной параметр), новый размер (входной параметр) и адрес нового буфера (выходной параметр). После того, как модели заданы для каждого экземпляра в трассе, происходит обновление карты динамической памяти в соответствии с семантикой модели. Стоит отметить, что в исследуемой программе может быть несколько вложенных менеджеров памяти, для работы с которыми используются разные наборы функций. Для отличия областей выделенной памяти используется идентификатор менеджера памяти. Карта динамической памяти реализована в виде последовательности кортежей *{идентификатор менеджера памяти, шаг трассы при создании буфера, шаг трассы при удалении буфера, идентификатор процесса, идентификатор потока, адрес начала буфера, размер буфера}*.

Модель функции, связанная с конкретным вызовом этой функции в трассе, называется экземпляром модели этой функции.

Обработка экземпляра модели *alloc* добавляет кортеж в карту памяти, инициализируя все значения, кроме шага трассы на котором происходит удаление буфера.

Обработка экземпляра модели *free* добавляет в кортеж шаг трассы, на котором происходит удаление буфера.

Обработка экземпляра модели *realloc* является комбинацией обработки предыдущих двух моделей. Сначала записывается шаг трассы, на котором входной буфер удаляется, затем создаётся кортеж, описывающий новый буфер. Адрес и размер нового буфера соответствуют параметрам модели *realloc*.

С помощью обработки всех экземпляров моделей по описанным выше правилам составляется разметка для динамической памяти.

Разметка автоматической памяти. В случае с автоматической памятью буфером является кадр стека соответствующей функции заданного исполняемого модуля. Создание карты автоматической памяти происходит в два этапа:

- получение информации о кадрах стека для каждого исполняемого модуля при помощи IDA Pro;
- отображение полученной информации на трассу.

Информация, полученная с помощью IDA Pro, представляет собой последовательность кортежей вида:

- смещение адреса функции относительно базового адреса модуля;
- размер кадра стека;
- размер параметров функции расположенных на стеке.

Для каждого вызова функции в трасе из заданного модуля создаётся кортеж в карте, аналогичный кортежу в карте динамической памяти. Шагом создания является шаг вызова функции, а шагом удаления является шаг возврата из функции.

Совокупность разметок автоматической и динамической памяти используется при дальнейшем анализе.

4.2 Разметка вызовов в Linux

Поиск ошибок в программах под ОС Linux усложняется из-за использования механизмов ленивого связывания. Во время первого вызова каждой библиотечной функции вызывается функция-заглушка из библиотеки ld.so, которая получает адрес вызываемой функции и изменяет код исполняемого файла таким образом, что при следующем вызове этой же библиотечной функции она будет вызвана напрямую. При этом выход из функции-заглушки происходит с помощью инструкции RET и приводит к передаче управления на код вызываемой библиотечной функции. Фактически, вызов библиотечной функции в этом случае происходит с помощью инструкции RET, что приводит к искажению результатов работы алгоритмов, выполняющих поиск вызовов функций. Это, в свою очередь, приводит к тому, что первый вызов каждой библиотечной функции не обрабатывается модулем символьного анализа. В то же время, многие ошибки, связанные с переполнением буфера на стеке в результате обработки параметров командной строки, происходят в результате копирования параметра с помощью строковых функций (strcpy, strcat) в самом

начале программы. Это приводит к ложноотрицательному результату при поиске ошибок в таких программах.

Для решения данной проблемы необходим более детальный анализ кадров стека. Рассмотрим пример вызова функции waitpid из библиотеки libc-2.19.so. Последовательность инструкций изображена на Рис. 2. Функция waitpid вызывается из программы bash с помощью пары инструкций CALL и JMP по адресам 0807F0C1 и 08059700 соответственно. Так как это первый вызов функции waitpid, управление передаётся на заглушку по адресу 08059706 и далее в библиотеку ld-2.19.so. Выход из библиотеки происходит с помощью инструкции RET 000Ch по адресу B7FF243B.

```
        bash 0807F0C1   CALL  08059700h
        bash 08059700   JMP   DWORD PTR [080F51B8h]
        bash 08059706   PUSH  00000358h
        bash 0805970B   JMP   08059040h
        bash 08059040   PUSH  DWORD PTR [080F5004h]
        bash 08059046   JMP   DWORD PTR [080F5008h]
  ld-2.19.so B7FF2420   PUSH  EAX
  ld-2.19.so B7FF2421   PUSH  ECX
                        …
  ld-2.19.so B7FF2434   MOV   DWORD PTR SS:[ESP], EAX
  ld-2.19.so B7FF2437   MOV   EAX, DWORD PTR SS:[ESP + 04h]
  ld-2.19.so B7FF243B   RET   000Ch
libc-2.19.so B7E398B0   CMP   DWORD PTR GS:[0Ch], 0
libc-2.19.so B7E398B8   JNZ   0B7E398DCh
```

Рис. 2. Последовательность инструкций при вызове функции waitpid.

Так как функция-заглушка сохраняет указатель стека, значение указателя стека после выполнения инструкции CALL (по адресу 0807F0C1), после выполнения инструкции JMP (по адресу 08059700) и после выполнения инструкции RET (по адресу B7FF243B) одинаковое. Анализ цепочки вызовов и инструкций RET в комбинации с анализом значений указателя стека позволяет установить факт вызова функции-заглушки и корректно определить вызов библиотечной функции.

5. Результаты практического применения

Предложенный метод был реализован в виде модуля-расширения среды анализа бинарного кода, он использует такие ее возможности, как повышение уровня представления, модель процессора общего назначения, слайс трассы.

Разработанный инструмент был опробован на 11 программах, работающих под управлением 32-разрядных ОС Windows XP SP2 и Arch Linux (по состоянию на март 2015). Среди программ были 4 с опубликованными уязвимостями. Список анализируемых программ приведён в табл. 2. Для

каждой тестовой программы была получена трасса нормального, безошибочного, выполнения на некотором наборе входных данных. Далее был запущен алгоритм поиска ошибок, с целью проверки, сможет ли он построить входные данные, реализующие дефект.

В табл. 3 приведены результаты работы алгоритма. Для всех исследуемых программ время работы SMT-решателя не превышало одной секунды. При получении результатов было добавлено дополнительное ограничение сверху на абстрактную длину буфера. Без этого ограничения, SMT-решатель может подобрать слишком большое значение для абстрактной длины буфера, которое не позволит создать в памяти буфер такого размера.

Табл. 2. Список анализируемых программ.

Операционная система	Программа	Версия приложения	CVE / OSVDB
Linux	iwconfig	iwconfig v26	CVE: 2003-0947
Linux	get_driver	sysfsutils 2.1.0	–
Linux	mkfs.jfs	jfsutils 1.1.15	–
Linux	alsa_in	jack 0.124.1	–
Linux	OpenSSL	openssl 1.0.0f	CVE: 2014-0160 (Heartbleed)
Windows XP SP2	httpdx	httpdx 1.5.4	OSVDB-ID: 84454
Windows XP SP2	GoldMP4Player	GoldMP4Player 3.3	OSVDB-ID: 103826
Linux	faad	faad2 2.7	–
Linux	gencnval	icu 54.1	–
Linux	lou_checktable	liblouis 2.5.2	–
Linux	mysql_plugin	mariadb 10.0.17	–

Табл. 3. Результаты работы алгоритма.

ОС	Программа	Размер префикса, байт	Размер входных данных	Тип доступа к памяти	Память	Время работы, с.
Linux	iwconfig	32	93	запись	стек	3
Linux	get_driver	14	489	запись	стек	4
Linux	mkfs.jfs	31	436	запись	стек	2
Linux	alsa_in	34	355	запись	стек	4
Linux	OpenSSL	18	25	чтение	стек	5
WinXP SP2	httpdx	329	330	запись	куча	338

WinXP SP2	GoldMP4Player	36	505	запись	куча	255
Linux	faad	13	302	запись	стек	<1
Linux	gencnval	13	574	запись	стек	<1
Linux	lou_checktable	15	527	запись	стек	8
Linux	mysql_plugin	16	578	запись	стек	14

В двух программах удалось обнаружить переполнение буфера на куче. Для многих из исследуемых программ были обнаружены ошибки доступа к памяти, связанные с записью входных данных, размер которых можно контролировать, в буфер фиксированного размера. Однако, для некоторых программ (OpenSSL и httpdx) были обнаружены ошибки доступа к памяти другого характера.

На примере OpenSSL было продемонстрировано обнаружение уязвимости Heartbleed, которая заключается в чтении данных за границами выделенного буфера. Для этого примера алгоритм автоматически подбирает два значения размера: размер пакета с данными и значение поля внутри пакета, которое описывает размер передаваемых (и запрашиваемых) данных. В табл. 3 приведен размер пакета, а размер запрашиваемых данных входит в префикс и составляет 247 байт.

В программе httpdx пользователь может контролировать не размер входных данных, а размер буфера выделенной памяти. Программа получает HTTP-запрос и выделяет буфер с помощью функции malloc. Размер буфера берётся из значения поля Content-Length, содержащегося в HTTP-заголовке. Затем происходит копирование тела запроса в выделенный буфер. При достаточно маленьком размере буфера происходит его переполнение. На этом примере также демонстрируется возможность автоматического определения соответствия между строковым значением размера, содержащимся в поле Content-Length и численным значением, передаваемым в функцию malloc.

Для проверки работы алгоритма был проведен следующий эксперимент. Для каждой исследуемой программы был сгенерирован новый набор входных данных на основе префикса и размера входных данных, которые были получены в результате работы описанного алгоритма. Если размер входных данных был больше размера префикса, оставшиеся байты, следующие за префиксом, принимались равными значению 0x41. Для всех исследуемых программ полученные наборы входных данных привели к аварийному завершению, что указывает на отсутствие ложноположительных результатов на данном наборе исследуемых программ. Следует отметить, что данный способ дополнения входных данных может привести к ложноположительному результату в случае, когда исследуемая программа проверяет формат входных данных.

6. Обзор близких работ

Наиболее близкие результаты были показаны в работах Splat [11] и LESE [16]. Инструмент Splat позволяет автоматически анализировать исходный код на языке Си и генерировать входные данные, приводящие к нарушению доступа к памяти. В данном инструменте используется понятие абстрактной длины и символьная интерпретация некоторых функций. Основным ограничением инструмента является то, что он предназначен только для исходных текстов программ на языке Си, в отличие от предлагаемого метода, который анализирует бинарный код. В работе LESE описывается метод обработки циклов, позволяющий проанализировать поведение программы при выполнении произвольного числа итераций цикла. В данном подходе считается, что входные данные определяются известной грамматикой. Для каждого цикла заводится специальная символьная переменная-счётчик *(trip counter*), которая отвечает за количество итераций в этом цикле. Все индуктивные переменные циклов выражаются через такие счётчики. Также счётчики связываются с атрибутами входных данных: длина поля или количество итераций поля-счётчика. В отличие от Splat, подход, описанный в LESE, применим также к программам, распространяемым в виде бинарного кода. К сожалению, описанные в данных работах инструменты недоступны.

Также стоит отметить отдельный класс программных инструментов, основанных на символьном исполнении, которые реализуют управляемый фаззинг для поиска дефектов: BitFuzz [17], FuzzBall [18], SAGE [19], Avalanche [20].

7. Заключение

В статье представлен метод поиска дефектов, приводящих к нарушению доступа к памяти. Метод основан на символьной интерпретации бинарной трассы, он позволяет абстрагироваться от конкретной длины входных данных и за счёт этого вычислять длину входных данных, при которой проявляется дефект. Метод был реализован в виде программного инструмента, являющегося частью среды анализа бинарного кода.

Входными данными для анализа выступает набор трасс, обеспечивающий достаточное покрытие кода. Анализ набора трасс производится автоматически, что позволяет совмещать его с другой деятельностью, например, восстановлением алгоритма из бинарного кода [3, 12].

Для последующей оценки критичности найденных ошибок следует воспользоваться методом, описанным в работе [9]. Входные данные, реализующие дефект, используются для получения новой трассы. Ее анализ позволяет оценить возможности эксплуатации найденного дефекта.

Дальнейшие работы предполагают автоматизацию определения точек получения входных данных и поддержку более широкого класса библиотечных функций.

Список литературы

[1]. Common Weakness Enumeration, a community-developed dictionary of software weakness types. https://cwe.mitre.org Дата обращения: 8.04.2015

[2]. К. Батузов, П. Довгалюк, В. Кошелев, В. Падарян. Два способа организации механизма полносистемного детерминированного воспроизведения в симуляторе QEMU. // Труды Института системного программирования РАН, том 22, 2012 г. Стр. 77-94.

[3]. Андрей Тихонов, Арутюн Аветисян, Вартан Падарян. Методика извлечения алгоритма из бинарного кода на основе динамического анализа. // Проблемы информационной безопасности. Компьютерные системы. №3, 2008. Стр. 66-71

[4]. Андрей Тихонов, Вартан Падарян. Применение программного слайсинга для анализа бинарного кода, представленного трассами выполнения. // Материалы XVIII Общероссийской научно-технической конференции «Методы и технические средства обеспечения безопасности информации». 2009. стр. 131

[5]. А.Ю.Тихонов, А.И. Аветисян. Комбинированный (статический и динамический) анализ бинарного кода. // Труды Института системного программирования РАН,том 22, 2012 г. стр. 131-152.

[6]. Alexander Getman, Vartan Padaryan, and Mikhail Solovyev. Combined approach to solving problems in binary code analysis. // Proceedings of 9th International Conference on Computer Science and Information Technologies (CSIT'2013), pp. 295-297.

[7]. Довгалюк П.М., Макаров В.А., Романеев М.С., Фурсова Н.И. Применение программых эмуляторов в задачах анализа бинарного кода. // Труды Института системного программирования РАН, том 26, 2014 г. Выпуск 1. Стр. 277-296. DOI: 10.15514/ISPRAS-2014-26(1)-9.

[8]. King J.C. Symbolic execution and program testing. // Commun. ACM. – 1976. – No 19.

[9]. Падарян В.А., Каушан В.В., Федотов А.Н. Автоматизированный метод построения эксплойтов для уязвимости переполнения буфера на стеке. // Труды Института системного программирования РАН, том 26, 2014 г. Выпуск 3. Стр. 127-144. DOI: 10.15514/ISPRAS-2014-26(3)-7.

[10]. E. J. Schwartz, T. Avgerinos, D. Brumley. // All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). // IEEE Symposium on Security and Privacy, May 2010, pp. 317–331.

[11]. Ru-Gang Xu, Patrice Godefroid, Rupak Majumdar. // Testing for Buffer OverFlows with Length Abstraction. // ISSTA, 2008

[12]. В.А. Падарян, А.И. Гетьман, М.А. Соловьев, М.Г.Бакулин, А.И. Борзилов, В.В. Каушан, И.Н. Ледовских, Ю.В. Маркин, С.С. Панасенко. // Методы и программные средства, поддерживающие комбинированный анализ бинарного кода. // Труды Института системного программирования РАН, том 26, 2014 г. Выпуск 1. Стр. 251-276. DOI: 10.15514/ISPRAS-2014-26(1)-8.

[13]. Падарян В. А., Соловьев М. А., Кононов А. И. // Моделирование операционной семантики машинных инструкций. // Программирование, No 3, 2011 г. Стр. 50-64.

[14]. Nikolaj Bjørner, Leonardo de Moura. // Z3: Applications, Enablers, Challenges and Directions/ // Sixth International Workshop on Constraints in Formal Verification Grenoble, 2009.

[15]. А. И. Аветисян, А. И. Гетьман. // Восстановление структуры бинарных данных по трассам программ. // Труды Института системного программирования РАН, том 22, 2012 г. Стр. 95-118.

[16]. Prateek Saxena, Pongsin Poosankam, Stephen McCamant, Dawn Song. // Loop-Extended Symbolic Execution on Binary Programs. // ISSTA, 2009.

[17]. J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song. Input generation via decomposition and re-stitching: Finding bugs in malware. In Proc. of the ACM Conference on Computer and Communications Security, Chicago, IL, October 2010.

[18]. L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. // In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems, London, UK, Mar. 2012.

[19]. P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. // In Proc. of the Network and Distributed System Security Symposium, Feb. 2008.

[20]. Исаев, И. К., Сидоров, Д. В., Герасимов, А. Ю., Ермаков, М. К. (2011). Avalanche: Применение динамического анализа для автоматического обнаружения ошибок в программах использующих сетевые сокеты. Труды Института системного программирования РАН, том 21, 2011 г., стр. 55-70.

Memory Violation Detection Method in Binary Code

V. V. Kaushan <korpse@ispras.ru>
A. YU. Mamontov <mamontov@ispras.ru>
V. A. Padaryan <vartan@ispras.ru>
A. N. Fedotov <fedotoff@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004

Abstract. In this paper memory violation detection method is considered. This method is applied to program binaries without requiring debug information. It allows to find such memory violations as out-of-bound read or write. The technique is based on dynamic analysis and symbolic execution. Instead of representing input buffer as a symbolic variable of fixed size, we track only the prefix of buffer symbolically and a special symbolic variable that represents the length of input buffer. The symbolic length variable allows to interpret functions with known semantics such as string library or memory allocation functions. While interpreting these functions using symbolic length variables we assert some constraints on buffer bounds. Such constraints allow to find memory violations. If violation is located, concrete values of buffer prefix and final input buffer length are provided. To apply this method to binary code we have to recover buffer bounds. So we developed some methods that recover buffer bounds in heap and stack memory. We present a tool implementing the method. We used this tool to find 11 bugs in both Linux and Windows programs, 7 of which were undocumented at the time this paper was written. This tool was able to detect known Heartbleed vulnerability which couldn't be found by simple fuzzers in crash absence.

Keywords: bug finding; symbolic execution; binary code; dynamic analysis.

DOI: 10.15514/ISPRAS-2015-27(2)-7

For citation: Kaushan V. V., Mamontov A. YU., Padaryan V. A., Fedotov A. N. Memory Violation Detection Method in Binary Code. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 105-126 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-7.

References

[1]. Common Weakness Enumeration, a community-developed dictionary of software weakness types. https://cwe.mitre.org Date of treatment: 8.04.2015

[2]. K. Batuzov, P. Dovgalyuk, V. Koshelev, V. Padaryan. Dva sposoba organizatsii mehanizma polnosistemnogo determinirovannogo vosproizvedeniya v simulyatore QEMU.[Two methods full-system deterministic replay in QEMU]// *Trudy ISP RAN [The Proceedings of ISP RAS]*, vol. 22, 2012, pp. 77-94 (in Russian)

[3]. Tikhonov A.Yu., Avetisyan A.I., Padaryan V.A., Metodika izvlecheniya algoritma iz binarnogo koda na osnove dinamicheskogo analiza [Methodology of exploring of an algorithm from binary code by dynamic analysis]. *Problemy informatsionnoj bezopasnosti. Komp'yuternye sistemy [Informations security aspects. Computer sistems]*, 2008, №3. pp. 66-71 (in Russian)

[4]. Tikhonov A.Yu., Padaryan V.A., Primenenie programmnogo slaysinga dlya analiza binarnogo koda, predstavlennogo trassami vyipolneniya.[Using program slicing for bynary code represented by execution traces] *Materialyi XVIII Obscherossiyskoy nauchno-tehnicheskoy konferentsii «Metodyi i tehnicheskie sredstva obespecheniya bezopasnosti informatsii». [The Proceedings of XVIII Russian science technical conference "Methods and technical information security tools"]* 2009. pp 131 (In Russian).

[5]. Tikhonov A.Yu., Avetisyan A.I. Kombinirovannyj (staticheskij i dinamicheskij) analiz binarnogo koda. [Combined (static and dynamic) analysis of binary code]. *Trudy ISP RAN [The Proceedings of ISP RAS]*, vol. 22, 2012, pp. 131-152 (in Russian).

[6]. Alexander Getman, Vartan Padaryan, and Mikhail Solovyev. Combined approach to solving problems in binary code analysis. // Proceedings of 9th International Conference on Computer Science and Information Technologies (CSIT'2013), pp. 295-297.

[7]. Dovgalyuk P.M., Makarov V.A., Romaneev M.S., Fursova N.I. Primenenie programmyih emulyatorov v zadachah analiza binarnogo koda.[Applying program emulators for binary code analysis] // *Trudy ISP RAN [The Proceedings of ISP RAS]*, vol. 26, issue 2014, pp. 277-296. DOI: 10.15514/ISPRAS-2014-26(1)-9.

[8]. King J.C. Symbolic execution and program testing. // Commun. ACM. – 1976. – No 19.

[9]. Padaryan V.A., Kaushan V.V., Fedotov A.N. Avtomatizirovannyiy metod postroeniya eksploytov dlya uyazvimosti perepolneniya bufera na steke.[Automated exploit generaton method for stack buffer overflow vulnerabilities] // *Trudy ISP RAN [The Proceedings of ISP RAS]*, vol. 26, issue 3, 2014, pp.. 127-144. DOI: 10.15514/ISPRAS-2014-26(3)-7

[10]. E. J. Schwartz, T. Avgerinos, D. Brumley. // All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). // IEEE Symposium on Security and Privacy, May 2010, pp. 317–331.

[11]. Ru-Gang Xu, Patrice Godefroid, Rupak Majumdar. // Testing for Buffer OverFlows with Length Abstraction. // ISSTA, 2008

[12]. V.A. Padaryan, A.I. Getman, M.A. Solovyev, M.G. Bakulin, A.I. Borzilov, V.V. Kaushan, I.N. Ledovskich, U.V. Markin, S.S. Panasenko. Metody i programmnye sredstva, podderzhivayushhie kombinirovannyj analiz binarnogo koda [Methods and software tools for combined binary code analysis]. *Trudy ISP RAN [The Proceedings of ISP RAS]*, 2014, vol. 26, no. 1, pp. 251-276 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-8

[13]. Padaryan V.A., Solov'ev M.A., Kononov A.I. Modelirovanie operatsionnoy semantiki mashinnyih instruktsiy. [Simulation of operational semantics of machine instructions]. *Programming and Computer Software*, May 2011, Volume 37, Issue 3, pp 161 – 170 , DOI 10.1134/S0361768811030030 (In Russian)

[14]. Nikolaj Bjørner, Leonardo de Moura. // Z3: Applications, Enablers, Challenges and Directions/ // Sixth International Workshop on Constraints in Formal Verification Grenoble, 2009.

[15]. Avetisyan A.I., Getman A.I. Vosstanovlenie struktury binarnykh dannykh po trassam program [Recovery the structure of binary data on the program traces]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 95-118 (in Russian)

[16]. Prateek Saxena, Pongsin Poosankam, Stephen McCamant, Dawn Song. // Loop-Extended Symbolic Execution on Binary Programs. // ISSTA, 2009.

[17]. J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song. Input generation via decomposition and re-stitching: Finding bugs in malware. In Proc. of the ACM Conference on Computer and Communications Security, Chicago, IL, October 2010.

[18]. L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. // In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems, London, UK, Mar. 2012.

[19]. P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. // In Proc. of the Network and Distributed System Security Symposium, Feb. 2008.

[20]. Isaev, I. K., Sidorov, D. V., Gerasimov, A. YU., Ermakov, M. K. (2011). Primenenie dinamicheskogo analiza dlya avtomaticheskogo obnaruzheniya oshibok v programmakh ispol'zuyushhikh setevye sokety [Using dynamic analysis for automatic bug detection in software that use network sockets]. *Trudy ISP RAN [The Proceedings of ISP RAS]*, 2011, vol. 21, pp. 55-70 (In Russian).