DOI: 10.15514/ISPRAS-2019-31(6)-2

Формальная модель обнаружения программных ошибок с помощью символьного исполнения программ

А.Ю. Герасимов, ORCID: 0000-0001-9964-5850 <agerasimov@ispras.ru>
Д.О. Куц, ORCID: 0000-0002-0060-8062 <kutz@ispras.ru>
А.А. Новиков, ORCID: 0000-0001-7567-0998 <a.novikov@ispras.ru>

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

Аннотация. Автоматическое обнаружение ошибок в программах является крайне востребованным направлением современных исследований и разработок в области обеспечения безопасности и устойчивости программного обеспечения. В рамках проекта 17-07-00702 Российского фонда фундаментальных исследований исследовались направления применения комбинированных методов анализа программ, совмещающих динамическое символьное исполнение, рандомизированное тестирование и статический анализ программ. Разработаны методы направленного анализа программ, основанные на совмещении статического анализа и динамического символьного исполнения, совмещения рандомизированного тестирования и динамического символьного исполнения программы. В данной статье рассматривается формальная модель обнаружения ошибок в программах методом символьного исполнения программ и её реализация для обнаружения ошибок выхода за границы буфера в памяти. Приводится формальная модель символьного исполнения программ, формулируется и доказывается теорема об обнаружении ошибки в программе, основанная на нарушении области определения операции вычислительной системы. Приводится описание реализации анализатора нарушения границ буфера в памяти в процессе динамического символьного исполнения программы и результаты применения реализованного прототипа анализатора на наборе программ из поставки Debian Linux, подтверждающих применимость предложенного метода обнаружения ошибок.

Ключевые слова: комбинированный анализ программ; динамическое символьное исполнение программ; обнаружение программных ошибок

Для цитирования: Герасимов А.Ю., Куц Д.О., Новиков А.А. Формальная модель обнаружения программных ошибок с помощью символьного исполнения программ. Труды ИСП РАН, том 31, вып. 6, 2019 г., стр. 21–32. DOI: 10.15514/ISPRAS–2019–31(6)–2

Благодарности: Исследование проведено при поддержке Российского фонда фундаментальных исследований. Проект 17-07-00702.

A formal model for program defect detection using symbolic program execution

A.Y. Gerasimov, ORCID: 0000-0001-9964-5850 <agerasimov@ispras.ru>
D.O. Kutz, ORCID: 0000-0002-0060-8062 <kutz@ispras.ru>
A.A. Novikov, ORCID: 0000-0001-7567-0998 <a.novikov@ispras.ru>

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

Abstract. An automatic program defect detection is extremely important direction of current research and development in the field of program reliability and security assurance. There were performed research of different ways of application for combined analysis methods which mix static source code analysis and dynamic symbolic execution, fuzz testing and dynamic symbolic execution as part of previous period of two years for project 17-07-00702 of the Russian Foundation for Basic Research. This paper presents elaboration of previously presented methods in form of formal model of program symbolic execution applied for program defect detection and implementation of analyzer of memory buffer bounds violation based on this model. The common theorem for program defect detection based on model of symbolic program execution and violation of definitional domain for computation system operation is formulated and proved. A special case theorem for buffer bounds violation detection is formulated and proved basing on common theorem and shadow memory model. As a practical application for theoretical basis an implementation of the analysis tool prototype description provided. Experimental results are received on the set of command line utilities of Debian Linux distribution, which shows applicability of proposed theoretical basis for solving practical tasks in the field of program reliability and security assurance.

Keywords: hybrid program analysis; dynamic symbolic execution; program defect detection

For citation: Gerasimov A.Y., Kutz D.O., Novikov A.A. A formal model for defect detection using symbolic program execution. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 6, 2019. pp. 21-32 (in Russian). DOI: 10.15514/ISPRAS-2019-31(6)-2

Acknowledgements. The research was supported by RFBR, grant 17-07-00702.

1. Введение

Первоочередной задачей для демонстрации ошибки в программе, найденной аналитиком или инструментами автоматической инспекции кода программ, является генерация внешних данных программы, проявляющих её ошибочное поведение. Наиболее популярным методом обнаружения ошибок во времена исполнения программы является метод рандомизированного тестирования [1], который тестирует программу на псевдослучайных внешних данных и проверяет устойчивость программы к неожидаемым внешним данным [2]. Метод рандомизированного тестирования программ обладает высокой производительностью в связи с тем, что наборы внешних данных программы генерируются с высокой скоростью, а программа подвергается легковесной инструментации (в случае рандомизированного тестирования с подкреплением), которая не приводит к значительному замедлению исполнения программы. При этом, обнаружение программных ошибок методом рандомизированного тестирования обладает случайным характером и не может быть использовано для достоверного подтверждения ошибки.

Альтернативным методом генерации тестового набора внешних данных программы является динамическое символьное исполнение [3]. Данный метод основывается на тяжеловесной инструментации кода программы с целью сбора трассы исполнившихся инструкций для последующего построения формулы, описывающей модель исполнения программы. На основе этой формулы вычисляются новые наборы внешних данных программы, которые приводят к исполнению программы по альтернативному пути или воспроизводят имеющийся дефект. В связи с тяжеловесной инструментацией кода

программы и высокой вычислительной сложностью решения формулы данный метод на современном уровне развития вычислительной техники не может быть применен для исчерпывающего тестирования программы.

Наиболее перспективным способом применения метода динамического символьного исполнения является применение направленного анализа программы [4], использующего информацию о положении потенциальных ошибок в программе для целенаправленного исполнения программы по путям, приводящим к местам потенциальных ошибок [5], с последующей проверкой реализации ошибочной ситуации в программе.

В данной статье приводится формальная модель символьного исполнения программы с целью проявления ошибок в программе, а также даётся пример реализации проверки ошибочной ситуации для ошибок выхода за границы буфера в памяти.

2. Модель символьного исполнения программы

Определим программу как четверку:

$$P = < F, S, s_I, S_T >, \quad (1)$$

где: S – множество состояний программы; s_I – начальное состояние программы; S_T – множество конечных состояний программы; F – множество операций, каждая из которых переводит программу из одного состояния в другое:

$$f : S \to S, f \in F, \quad (2)$$

Тогда исполнение программы можно определить как последовательность переходов между состояниями программы $s_i \in S$, осуществляемых операциями $f_i \in F$:

$$\{f_I(s_I) \to s_1, f_1(s_1) \to s_2, \dots, f_i(s_i) \to s_T\}, \quad (3)$$

где: $s_I \in S$ – начальное состояние программы; $s_T \in S_T$ – одно из конечных состояний программы; $f_i \in F$ – операция, принадлежащая множеству операций программы, переводящих одно состояние программы в другое.

Определим состояние программы как

$$s = < d, f > | d \in D, f \in F, \quad (4)$$

где d – подмножество множества данных, обрабатываемых программой; D – множество данных, обрабатываемых программой; f – следующая операция программы.

Тогда исполнение операции программы можно представить в виде

$$f_i(d_i) \to < d_j, f_j > | d_i, d_j \in D, f_i, f_j \in F. \quad (5)$$

Определение 1. *Ограничением пути исполнения в программе PC* (path condition) будем называть множество ограничений на значения данных программы, полученное путём преобразования операций, выполненных над данными программы на пути исполнения, в элементы множества ограничений и однозначно описывающее исполнение программы по пути, достигающем состояния s.

Не ограничивая общности рассуждений, всё множество операций в программе можно разделить на три вида:

$$f_i(d_i) \to < d_j, f' > \begin{cases} d_i \neq d_j, \quad f_i \to f_j & \text{вычислительная операция} \\ d_i = d_j, f_i \to f_j & \text{безусловный переход} \\ d_i = d_j, \begin{cases} f_i \to f_j \mid p_i \\ f_i \to f_k, \mid \neg p_i \end{cases} p_i \in PC & \text{условный переход} \end{cases} \quad (6)$$

где:

- вычислительная операция изменяет состояние программы путём изменения множества данных программы $d_i \to d_j$ и переводит исполнение программы на следующую операцию $f_i \to f_j$;

- безусловный переход изменяет состояние программы путём перевода исполнения на следующую операцию $f_i \to f_j$;
- условный переход изменяет состояние программы путём перевода исполнения на операцию f_j, если подмножество предусловий состояния, являющееся условием перехода, вычисляется в истину, и на операцию f_k, если подмножество предусловий состояния, являющееся условием перехода, вычисляется в ложь.

Разделим множество данных программы D, на котором определено ограничение пути исполнения в программе PC, на два множества: V – множество внутренних данных программы; W – множество внешних данных программы. Тогда предусловие состояния s_i можно описать как функцию:

$$p(w, v) \mid w \subseteq W, v \subseteq V. \quad (7)$$

Заменим элементы множества W на элементы множества переменных X, где позиции каждого элемента множества внешних данных в потоке внешних данных программы соответствует переменная $x \in X$.

Определение 2. *Множеством свободных переменных* будем называть множество переменных, значения которых соответствуют значениям элементов множества внешних данных программы.

Определение 3. *Множество зависимых переменных* формируется из переменных, являющихся результатом исполнения операций, множество аргументов которых содержит хотя бы одну свободную или хотя бы одну зависимую переменную.

Определение 4. *Символьное ограничение пути* SPC, или *символьное предусловие* состояния s в программе, является множеством ограничений на значения зависимых и свободных переменных и внутренних данных программы, полученных путём преобразования операций над ними на пути исполнения программы, предшествующем состоянию s.

Множество значений данных программы удовлетворяющих символьному предусловию пути для состояния s_i описывается функцией $\pi(x_i, v_i)$:

$$D_{\pi_i} = \pi(x_i, v_i) \mid D_{\pi_i} \subseteq D. \quad (8)$$

Определение 5. *Состояние ошибки* это такое состояние в программы $s_{err} \in S$, при котором дальнейшее исполнение программы ошибочно.

Определение 6. *Множество определения операции* f – это такое подмножество данных программы $D' \subseteq D$, на котором выполнение операции f не приводит к достижению состояния ошибки.

Утверждение 1. Для каждой операции f, входящей в множество операции программы F, существует множество определения данной операции D':

$$\forall f \in F \; \exists \, D' \subseteq D, \quad (9)$$

Утверждение 2. Если множество значений аргументов операции f не входит в множество определения операции D' и множество D' не пусто, то исполнение операции приводит программу в состояние ошибки

$$f(d) \to s_{err} \mid f \in F, \forall \, d \notin D' \; \& \; D' \neq \varnothing, \quad (10)$$

Если множество определения операции пусто, то выполнение операции не зависит от данных программы, что соответствует определению операции на всём множестве данных программы, и, следовательно, либо всякое исполнение операции гарантированно приводит к достижению состояния ошибки в программе s_{err}, либо выполнение данной операции не может привести исполнение программы в состояние ошибки s_{err}.

Теорема 1. Если множество определения D' операции f_i не пусто и дополнение множества D' и множества значений данных программы D_{π_i}, ограниченных функцией $\pi(x_i, v_i) =$

D_{π_i}, не пусто, то существуют такие значения внешних данных, исполнение программы на которых приведёт к достижению ошибочного состояния s_{err} в программе:

$$f_i(d_i) \rightarrow s_{err} \mid \forall\, d_i \in D_{err},\ D_{err} = D_{\pi_i}\ D' \,\&\, D' \neq \varnothing. \quad (11)$$

Доказательство. Исходя из формулировки *утверждения 2*, достижение состояния ошибки в программе возможно при выходе значения аргументов операции f_i за пределы множества области определения операции D'. Если символьные ограничения пути исполнения SPC наложенные на значения внешних данных программы, описываемые функцией $\pi(x_i, v_i)$, формируют такое множество D_{π_i}, что значения аргументов f_i попадают в множество допустимых значений данных программы D_{π_i}, но не попадают в множество определения D' операции f_i, то, исходя из *утверждения 2*, программа достигнет состояния ошибки s_{err}, что и требовалось доказать.

Следствие теоремы 1. Для того чтобы вычислить внешние данные программы, приводящие исполнение программы в состояние ошибки, необходимо и достаточно для каждого типа ошибки определить множество операций, исполнение которых может приводить к ошибке, и сформулировать условие выхода значений аргументов операции за пределы множества определения этих операций и при этом входящих в множество значений данных программы.

Определение 7. *Буфер в памяти* – область памяти, ограниченная адресом начала A_{begin} и адресом конца A_{end} блока данных программы в памяти.

Определение 8. Операцией *разыменования при доступе к буферу в памяти* будем называть операцию разыменования указателя, область определения которой D'_{buffer} ограничена адресом начала A_{begin} и адресом конца A_{end} буфера в памяти.

Подсистема управления динамической памятью операционной системы при выделении блока оперативной памяти программе производит выделение таким образом, что каждый новый выделенный блок памяти не пересекается по интервалу адресов начала и конца блока $< A_{begin},\ A_{end} >$ с другими выделенными блоками памяти на момент его выделения. Также менеджер может выделить блок памяти, включающий адреса, попадающие в интервал адресов ранее освобождённых блоков.

Для контроля доступа к выделенным и освобождённым блокам памяти заведём два множества: множество выделенных блоков памяти D_{alloc} и множество освобождённых блоков памяти D_{freed}, каждое из которых в качестве элементов содержит описание блоков памяти в виде интервала адресов $< A_{begin},\ A_{end} >$, где A_{begin} – адрес начала блока памяти, A_{end} – адрес конца блока памяти. При выполнении операции выделения блока в памяти интервал адресов $< A_{begin},\ A_{end} >$ добавляется в множество выделенных блоков памяти D_{alloc}. При выполнении операции освобождения блока в памяти интервал адресов $< A_{begin},\ A_{end} >$ удаляется из множества выделенных блоков памяти D_{alloc} и добавляется в множество освобожденных блоков памяти D_{freed}. Если возникает пересечение адресов нового выделенного блока с уже освобожденными блоками в множестве D_{freed}, то в множестве D_{freed} происходит преобразование интервалов адресов, которые пересекаются с выделенным блоком таким образом, чтобы интервал адресов выделенного блока не пересекался ни с одним из интервалов в множестве освобождённых блоков памяти D_{freed}.

Теорема 2. Для определения ошибки доступа к буферу в памяти по указателю f_{dbuf} необходимо и достаточно в символьное предусловие операции добавить условие проверки значения переменной указателя меньше адреса начала буфера A_{begin} и больше адреса конца буфера A_{end}:

$$f_{dbuf}\,(d) \rightarrow s_{err} \mid (SPC \,\&\, (d < A_{begin} \,||\, d > A_{end})) \neq \varnothing. \quad (12)$$

Доказательство. Множество определения операции доступа к буферу f_{dbuf} по *определению 8* ограничено адресом начала A_{begin} и адресом конца буфера A_{end}. Если символьное предусловие SPC операции разыменования f_{dbuf} позволяет вычисление значений множества внешних данных программы, таких, что аргумент операции разыменования принимает значение меньше адреса начала буфера A_{begin} или больше адреса конца буфера A_{end}, то дополнение разности множеств $(D_{\pi_i} \setminus D'_{buffer})$ не пусто и содержит адрес, выходящий за границы буфера. В соответствии с *определением 8* происходит нарушение множества определения операции разыменования указателя при доступе к буферу, что в соответствии с *теоремой 1* приводит к достижению состояния ошибки в программе s_{err}, что и требовалось доказать.

3. Реализация проверки нарушения предикатов безопасности для ошибок выхода за границы буфера в памяти

Количество программных ошибок, обнаруженных инструментами автоматического анализа программ и связанных с ошибками доступа к памяти, достаточно велико [6]. Данные ошибки связаны как с обращением к памяти за границами выделенных блоков, так и с нарушением прав доступа в пределах выделенных блоков памяти (на чтение, на запись, на исполнение). В случае нарушения прав доступа к выделенным блокам памяти или обращению к адресам за границами страниц памяти выделенных исполняющемуся процессу, происходит ошибка сегментации (segmentation fault) в операционной системе семейства Linux или ошибка нарушения доступа к блоку памяти (access violation) в операционной системе семейства Windows.

Некорректный доступ к памяти является причиной таких ошибок, как разыменование нулевого указателя (CWE-476 [7]), переполнение буфера на стеке или куче (CWE-121 [8], CWE-122 [9]), запись значения по произвольному адресу (CWE-123[10]), запись и чтение за пределами выделенного буфера (CWE-125 [11], CWE-787 [12]). В наборе инструкций архитектуры Intel x86 и x86-64 количество инструкций доступа к памяти достаточно велико, поэтому для подтверждения предложенного подхода было введено ограничение на набор инструкций, используемых в прототипе инструмента анализа программ. Учитывались инструкции помещения или чтения значения из блока памяти *mov* (*movzx, movsx* и др.), инструкции работы со строками (*cmps, movs, scas, lods, ins, outs*), инструкции сравнения *cmp* и другие.

Перечисленные виды инструкций работают с несколькими операндами, но у всех них есть операнд-источник данных для работы и операнд-приемник, куда помещается результат работы. В качестве операндов для инструкций могут выступать регистры, ячейки памяти и константы (immediate operand) – значения, закодированные в самой инструкции. С точки зрения проверки свойств безопасности программы, наиболее интересными являются случаи, когда инструкция работает с памятью – если одним из двух перечисленных операндов инструкции (то есть осуществления чтения или записи) является память, то потенциально эта инструкция может стать причиной ошибки доступа. Важным условием для возможности появления ошибки доступа к памяти является возможность использования внешних данных программы при вычислении адреса доступа. В связи с этим при построении символьной модели вычислений программы учитывались только инструкции, операнды которых явно или косвенно зависели от внешних данных программы.

Реализация алгоритмов обнаружения ошибок доступа к памяти проводилась на базе инструмента символьного исполнения программ Anxiety [13], разрабатываемого в ИСП РАН. Инструмент Anxiety производит построение символьной формулы, описывающей

исполнение программы, только для помеченных инструкций, то есть таких инструкций, значения операндов которых явно или косвенно зависят от внешних данных программы. Для этого, во-первых, в инструменте Anxiety производится анализ помеченных данных (dataflow taint analysis), в рамках которого реализованы алгоритмы распространения помеченности операндов инструкций. Для операций доступа к памяти в формулу попадают предикаты только для тех инструкций, операнды которых помечены в результате распространения пометок. Если как минимум один операнд инструкции помечен, то это означает, что адрес операции доступа к памяти вычисляется с использованием внешних данных программы явно или косвенно.

Во-вторых, для проверки наличия дефекта на заданных на предыдущем шаге инструкциях необходимо описать состояние безопасности программы при выполнении этих инструкций и сформулировать ограничения, накладываемые на данные программы, для обеспечения этого безопасного состояния. Безопасным состоянием программы будем считать корректное исполнение инструкций – доступ к памяти осуществляется по разрешенным адресам и с разрешённым уровнем доступа. Условие корректного исполнения инструкций будем называть предикатом безопасности [14]. Инвертирование предиката безопасности в процессе символьного анализа исполнения программы если ограничения совместны должно привести к генерации входных данных, приводящих программу в небезопасное состояние, что означает возможность реализации ошибки в программе. При реализации прототипа инструмента были разработаны предикаты безопасности для обнаружения нарушения доступа к памяти.

Адрес доступа к памяти может составляться в инструкции различными способами:

- хранение адреса в регистре;
- формирование адреса из значений базы, индекса и смещения.

```
mov eax -> (ecx)                : address = %ecx
mov ecx -> 0xa4(ebp)            : address = 0xa4 + %ebp
mov edx -> 0xc8(ebp, eax, 0x8)  : address = 0xc8 + %ebp + %eax * 0x8
```

Рис. 1. Варианты формирования адреса в архитектуре x86
Fig. 1. Options for address generation in x86 architecture

Для того чтобы адрес считался помеченным, нужно чтобы была помеченной хотя бы одна из его составных частей. В результате адрес в функции предикатов должен быть представлен в виде выражения над символьными переменными, которое установит взаимосвязь символьной модели программы с конкретными значениями входных данных программы. Предикат безопасности, нацеленный на выявление нарушения доступа к памяти, осуществляет проверку корректности построения значения адреса, по которому производится операция. Функция предиката безопасности принимает на вход адрес в виде выражения над символьными переменными и составляет условие принадлежности этого адреса тем регионам памяти в программе, для которых производимая в инструкции операция разрешена. Следовательно, если для инвертированного предиката возможен адрес, выходящий за границы этих регионов, то на полученных входных данных произойдёт выход за границы разрешенного региона и сгенерируется прерывание ошибки доступа или произойдёт «тихая» ошибка, не приводящая к аварийному завершению программы.

Все адресное пространство программы разделено на области с различными правами доступа – чтение (R), запись (W), исполнение (X) и их комбинации. Про некоторые области памяти неизвестно ничего, в таком случае можно считать их неразмеченными. В прототипе реализован подход, который собирает информацию об известных размеченных областях памяти и проверяющий попадание вычисленного адреса в неразмеченные

области памяти или в размеченные области памяти с ограничением доступа, не позволяющим проведение определённой операции. Для составления предиката безопасности доступа к памяти нужна информация о разметке адресного пространства программы. Получить эту информацию можно несколькими способами:

- отслеживания загрузки модулей в адресное пространство;
- перехват системных вызовов;
- специальные файлы операционной системы.

В связи с тем, что информация из этих источников пересекается, то для корректной реализации проверки предиката безопасности операций реализован механизм управления информацией о разметке адресного пространства в виде теневой памяти, который позволяет отслеживать информацию о выделенных буферах в памяти, разрешать конфликты при пересечении буферов и прав доступа объединять смежные области и т.п.

Реализация модели теневой памяти была осуществлена на основе инструмента динамической бинарной инструментации DynamoRIO [15], в частности, использовались возможности по отслеживанию загрузки исполняемых модулей и перехвату системных вызовов mmap, munmap и mprotect. Системный вызов mmap осуществляет отображение файла в адресное пространство процесса с указанными правами на доступ. Как правило, в вызове уже передаются начальное расположение сегмента в памяти и размер, но в некоторых случаях определение стартового адреса возложено на операционную систему. В таких случаях требуется осуществлять перехват не только вызова, но и возвращаемого значения, которое содержит начальный адрес нового сегмента и его размер. Системный вызов munmap используется для выгрузки региона и освобождения памяти, информацию из данного вызова следует использовать для удаления сегмента. Вызов mprotect осуществляется для изменения прав доступа уже существующего региона памяти.

Ещё одним способом получить информацию о разметке адресного пространства является обращение к специальным файлам в операционной системе. В операционных системах Unix информация об адресном пространстве процесса хранится и постоянно обновляется в специальном файле /proc/[pid]/maps, где [pid] – уникальный идентификатор процесса. Этот файл содержит наиболее полную информацию обо всех сегментах памяти, включая неразмеченные области. Однако из-за большого количества таких сегментов это увеличивает размер самого предиката безопасности. С другой стороны, такой наиболее полный предикат безопасности позволяет наиболее точно задать область запрещенных адресов для конкретной операции.

Таким образом, в анализ был добавлен предикат безопасности, позволяющий проверить инструкции доступа к памяти на возможность обращения к адресам, расположенным в недоступных регионах памяти, тем самым вызывая исключение нарушения доступа к памяти. Если в ходе символьного исполнения при инвертировании данного предиката SMT-решатель смог найти решение для всего предиката пути и построить входные данные, то во время запуска на них программа должна аварийно завершиться или проявить «тихую» ошибку. Если этого не произошло, значит предикат был составлен не точно, и подобранное решение для предиката на самом деле относится к валидной области памяти.

4. Экспериментальная проверка предложенного подхода

Испытания разработанного метода проводились на наборе программ с открытым исходным кодом из комплекта поставки Debian Linux. Результаты тестирования приведены в табл. 1.

Табл. 1. Результаты испытаний прототипа
Tab. 1. The results of the tests of the prototype

Приложение	Итерации	Дефекты без предикатов	Дефекты с предикатами	Время (с)
faad	73	1	1	5400
pnmhistmap	236	1	2	5402
i686-w64-mingw32-objdump	71	1	2	5401
m17n-dump	65	0	1	5402
vde_autoilnk	25	2	4	5401
swig2.0	116	1	1	5403
eperl	70	0	0	5403
sg_unmap	64	1	2	5402
xapian-chert-update	158	2	2	5403
yodlpost	76	0	0	5401
hdp	46	0	2	5403

Каждая программа анализировалась на протяжении полутора часов. В качестве входных данных были подобраны файлы и опции в ожидаемом для каждой программы формате, так как это способствует значительному ускорению анализа – инструменту не нужно тратить время на обход начальных условий, обрабатывающих корректность вводимых данных. Для проверки корректности работы предикатов безопасности набор программ был проанализирован за тот же промежуток времени без их использования.

В табл. 1 значения обнаруженных дефектов указаны за два запуска: первое значение – запуск без предикатов безопасности, второй – с предикатами. В результате запусков проектов с предикатами безопасности общее количество обнаруженных дефектов возросло, более детальный анализ показал, что в ряде программ действительно удалось обнаружить новые уязвимости. Но в большинстве случаев это оказываются известные ранее уязвимости, для которых с помощью предикатов безопасности подобрались новые входные данные, расположенные на альтернативных уязвимых путях исполнения программы.

5. Заключение

В данной работе предложена символьная модель и прототипная реализация метода вычисления внешних данных программы на основе символьного исполнения программ, которые позволяют демонстрировать ошибки в программах. Реализованный прототип подтверждает применимость теоретической модели на практике для реализации инструментов анализа программ с целью обнаружения ошибок в программах при помощи динамического символьного исполнения программ.

Дальнейшее развитие работы может проводиться в направлении реализации инструментов для обнаружения ошибок различного типа, таких как утечка ресурсов операционной системы, использования неинициализированных переменных и др.

Список литературы / References

[1]. B. P. Miller, L. Fredriksen, B. So. An empirical study of the reliability of UNIX utilities. Communications of the ACM, vol. 33, issue 12, 1990, pp. 32- 44.

[2]. M. Zalewski. Symbolic execution in vuln research. Available at: https://lcamtuf.blogspot.com/2015/02/symbolic-execution-in-vuln-research.html, 23.12.2019.
[3]. R.S. Boyer, B. Elspas, K.N. Levitt. SELECT – F Formal System for Testing and Debugging Programs by Symbolic Execution. In Proc. of the International Conference on Reliable software, 1975, pp. 234-245.
[4]. А.Ю. Герасимов, Л.В. Круглов. Вычисление входных данных для достижения определенной функции в программе методом итеративного динамического анализа. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 159-174 / A.Y. Gerasimov, L.V. Kruglov. Input data generation for reaching specific function in program by iterative dynamic analysis method. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 159-174 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-10.
[5]. Герасимов А.Ю., Круглов Л.В., Ермаков М.К., Вартанов С.П. Подход определения достижимости программных дефектов, обнаруженных методом статического анализа программ, при помощи динамического анализа. Труды ИСПРАН, том 29, вып. 5, 2017 г., стр. 111-134 / Gerasimov A.Y., Kruglov L.V., Ermakov M.K., Vartanov S.P. An approach of reachability confirmation for static analysis defects with help of dynamic symbolic execution. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017. pp. 111-134 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-7.
[6]. Godefroid P., Levin M. Y., Molnar D. SAGE: whitebox fuzzing for security testing. Communications of the ACM. vol. 55, issue 3, 2012, pp. 40-44.
[7]. CWE-476: NULL Pointer Dereference. Available at: https://cwe.mitre.org/data/definitions/476.html, accessed: 23.12.2019.
[8]. CWE-121: Stack-based Buffer Overflow. Available at: https://cwe.mitre.org/data/definitions/121.html, accessed: 23.12.2019.
[9]. CWE-122: Heap-based Buffer Overflow. Available at: https://cwe.mitre.org/data/definitions/122.html, accessed: 23.12.2019.
[10]. CWE-123: Write-what-where Condition. Available at: https://cwe.mitre.org/data/definitions/123.html, accessed: 23.12.2019.
[11]. CWE-125: Out-of-bounds Read. Available at: https://cwe.mitre.org/data/definitions/125.html, accessed: 23.12.2019.
[12]. CWE-787: Out-of-bounds Write. Available at: https://cwe.mitre.org/data/definitions/787.html, accessed: 23.12.2019.
[13]. A. Gerasimov, S.Vartanov, M. Ermakov, L. Kruglov, D. Kutz, A. Novikov, S. Asryan. Anxiety: a dynamic symbolic execution framework. In Proc. of the 2017 Ivannikov ISPRAS Open Conference, 2017, pp. 16-21. DOI: 10.1109/ISPRAS.2017.00010
[14]. Федотов А.Н., Каушан В.В., Гайсарян С.С., Курмангалеев Ш.Ф. Построение предикатов безопасности для некоторых типов программных дефектов. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 151-162 / Fedotov A.N., Kaushan V.V., Gaissaryan S.S., Kurmangaleev Sh.F. Building security predicates for some types of vulnerabilities. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 151-162 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-8.
[15]. D. Bruening, T. Garnett, S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In Proc. of the International Symposium on Code Generation and Optimization, 2003, pp. 265-275.

Информация об авторах / Information about authors

Александр Юрьевич Герасимов – кандидат физико-математических наук, старший научный сотрудник ИСП РАН. Сфера научных интересов: автоматический анализ программ, статический анализ программ, динамический анализ программ, комбинированные методы анализа программ, управление научными исследованиями и разработками, жизненный цикл разработки безопасного ПО.

Alexander Yurievich GERASIMOV – PhD in Computer Sciences, senior researcher. Research interests: automatic program analysis, static program analysis, dynamic program analysis, combined methods for program analysis, R&D management, SSDLC.

Александр Андреевич НОВИКОВ – аспирант ИСП РАН. Его научные интересы включают динамическое символьное исполнение программ, символьный анализ многопоточных программ, методы обнаружения программных дефектов.

Alexander Andreevich NOVIKOV – PhD Student at ISP RAS. Research interests: dynamic symbolic execution, symbolic execution of multithreaded programs, program defect detection.

Даниил Олегович КУЦ – аспирант ИСП РАН. Научные интересы включают информационную безопасность, динамический анализ программ, задача решения формул в теориях.

Daniil Olegovich Kuts – PhD Student at ISP RAS. Research interests: information security, dynamic program analysis, SMT.