НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
сентябрь–октябрь 2022 Том 22 № 5 http://ntv.ifmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
September–October 2022 Vol. 22 No 5 http://ntv.ifmo.ru/en/
ISSN 2226-1494 (print) ISSN 2500-0373 (online)

НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

doi: 10.17586/2226-1494-2022-22-5-982-991
УДК 004.052.42

Генерация слабейших предусловий программ с динамической памятью в символьном исполнении

Александр Владимирович Мисонижник[1], Юрий Олегович Костюков[2]✉, Михаил Павлович Костицын[3], Дмитрий Александрович Мордвинов[4], Дмитрий Владимирович Кознов[5]

[1,2,3,4,5] Санкт-Петербургский государственный университет, Санкт-Петербург, 199034, Российская Федерация

[1] misonijnik@gmail.com, https://orcid.org/0000-0002-5907-0324
[2] kostyukov.yurii@gmail.com✉, https://orcid.org/0000-0003-4607-039X
[3] michael.kosticyn@gmail.com, https://orcid.org/0000-0001-9982-6571
[4] mordvinov.dmitry@gmail.com, https://orcid.org/0000-0002-6437-3020
[5] d.koznov@spbu.ru, https://orcid.org/0000-0003-2632-3193

Аннотация

Предмет исследования. Символьное исполнение — широко известный метод систематического исследования путей исполнения программ. Метод позволяет решить ряд важных задач, связанных с контролем корректности: поиск ошибок и уязвимостей, автоматическая генерация тестов и др. Основная идея символьного исполнения — порождение и использование символьных логических выражений при анализе программ в прямом порядке, т. е. от входной точки к точкам интереса. Хорошо известен метод обратного символьного исполнения, когда условия попадания в точку интереса распространяются ко входной точке программы за счет итеративного вычисления слабейших предусловий. Реализовать этот метод, как правило, намного труднее, чем прямое символьное исполнение, так что даже артефакты последнего не удается использовать при реализации. **Метод.** Исследована связь между прямым и обратным символьными исполнениями на основе вычисления слабейших предусловий. В частности показано, как обратное исполнение может быть реализовано на базе прямого. **Основные результаты.** Приведено формальное описание процедуры символьного исполнения с ленивой инициализацией для программ с динамической памятью. Предложен алгоритм вычисления слабейших предусловий для произвольных ветвей исполнения программы. Механизм ленивой инициализации и алгоритм вычисления слабейших предусловий реализованы в символьной виртуальной машине KLEE, работающей на базе широко известной платформы LLVM. **Практическая значимость.** Представленный метод позволяет выполнять обратный символьный анализ при помощи прямого символьного исполнения. Это важно для реализации двунаправленного исполнения программ, которое может быть применено для верификации программ и автоматического порождения тестовых покрытий.

Ключевые слова

слабейшие предусловия, символьное исполнение, обратный символьный анализ, двунаправленный анализ, автоматическая генерация тестов

Благодарности

Данное исследование было поддержано грантом РНФ № 22-21-00697.

Ссылка для цитирования: Мисонижник А.В., Костюков Ю.О., Костицын М.П., Мордвинов Д.А., Кознов Д.В. Генерация слабейших предусловий программ с динамической памятью в символьном исполнении // Научно-технический вестник информационных технологий, механики и оптики. 2022. Т. 22, № 5. С. 982–991. doi: 10.17586/2226-1494-2022-22-5-982-991

Generation of the weakest preconditions of programs with dynamic memory in symbolic execution

Aleksandr V. Misonizhnik[1], Yurii O. Kostyukov[2]✉, Michael P. Kostitsyn[3], Dmitry A. Mordvinov[4], Dmitry V. Koznov[5]

[1,2,3,4,5] St. Petersburg State University (SPbSU), Saint Petersburg, 199034, Russian Federation

[1] misonijnik@gmail.com, https://orcid.org/0000-0002-5907-0324
[2] kostyukov.yurii@gmail.com✉, https://orcid.org/0000-0003-4607-039X
[3] michael.kosticyn@gmail.com, https://orcid.org/0000-0001-9982-6571
[4] mordvinov.dmitry@gmail.com, https://orcid.org/0000-0002-6437-3020
[5] d.koznov@spbu.ru, https://orcid.org/0000-0003-2632-3193

Abstract

Symbolic execution is a widely used method for the systematic study of program execution paths; it allows solving a number of important problems related to verification of correctness: searching for errors and vulnerabilities, automatic test generation, etc. The main idea of symbolic execution is generation and use of symbolic expressions in the program analysis in direct order, i.e., from the entry point to the points of interest. At the same time, since the time of E.W. Dijkstra, the method of backward symbolic execution has been popular when the conditions for hitting the point of interest are extended to the entry point of the program due to the iterative calculation of the weakest preconditions. This method is usually much more difficult to implement than direct symbolic execution, so even the artifacts of the latter cannot be used in the implementation. In this paper, the relationship between direct and backward symbolic execution based on the calculation of the weakest preconditions is investigated. In particular, it is shown that the latter can be implemented using the former. A formal presentation of symbolic execution with lazy initialization for programs with dynamic memory is given. An algorithm for calculating the weakest preconditions for arbitrary symbolic executed program branches is proposed. The lazy initialization mechanism and the algorithm for calculating the weakest preconditions are implemented in KLEE, a symbolic virtual machine for the well-known LLVM platform. The proposed method allows performing backward symbolic analysis using direct symbolic execution. This is important for the implementation of bidirectional program execution which can be used both for program verification and for automatic test generation.

Keywords

weakest preconditions, symbolic execution, backward symbolic analysis, bidirectional analysis, automatic test generation

Acknowledgments

The work is supported by the grant of RNF No. 22-21-00697.

For citation: Misonizhnik A.V., Kostyukov Yu.O., Kostitsyn M.P., Mordvinov D.A., Koznov D.V. Generation of the weakest preconditions of programs with dynamic memory in symbolic execution. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2022, vol. 22, no. 5, pp. 982–991 (in Russian). doi: 10.17586/2226-1494-2022-22-5-982-991

Введение

Символьное исполнение — известный метод статического исполнения программ. Данный метод основан на исполнении программ в рамках *символьной памяти*, в которой, в противовес адресному пространству программы, располагаются символьные, а не конкретные данные, т. е. термы некоторого формального языка. Символьное исполнение позволяет эффективно решать ряд задач, связанных с проверкой корректности программ: поиск ошибок и уязвимостей, автоматическая генерация тестов, и др. [1]. К настоящему времени зарубежными и российскими исследователями разработано большое количество систем символьного исполнения, например, KLEE [2], Angr [3], Crest [4], Pex [5], SPF [6], Manticore [7], DART [8], SAGE [9], S2E [10], Sydr [11] и др.

В научных работах встречаются два основных вида символьного исполнения: прямое и обратное [1]. Прямое осуществляет исполнение программы от входной точки (например, процедуры main в С-программе) к точкам интереса — тем строкам и операторам программы, выполнение которых интересует программистов и тестировщиков. Такое исполнение реализовано в подавляющем числе символьных виртуальных машин [1]. Однако данная техника имеет фундаментальное ограничение — комбинаторный взрыв путей исполнения программы, поскольку число символьных состояний программы растет экспоненциально от числа выполненных инструкций.

Один из способов частичного преодоления этого ограничения заключается в обратном исполнении инструкций программы [12]. Обратное исполнение позволяет эффективно находить входные данные, приводящие исполнение программы в точку интереса. При этом обратный шаг традиционно производится за счет вычисления *слабейших предусловий*, что, однако, оказывается труднореализуемым в существующих инструментах [13]: шаг исполнения представляет собой вычисление слабейшего предусловия относительно текущей инструкции [14]. Слабейшее предусловие — такое слабейшее условие, которому должно удовлетворять состояние программы, чтобы после исполнения инструкции получилось состояние, удовлетворяющее постусловию.

В настоящей работе предложен способ вычисления слабейших предусловий при обратном символьном исполнении через прямое символьное исполнение. Это позволяет избавиться от необходимости разрабатывать отдельный модуль для обратного исполнения

инструкций, что является очень трудоемкой задачей для программных платформ (JVM, LLVM, .NET и пр.) из-за большого количества инструкций со сложной семантикой.

Представленный способ рассчитан на программы с динамической памятью, что важно для неакадемического, практического применения символьного исполнения. Это оказывается возможным ввиду использования известной техники *ленивой инициализации* [15], позволяющей символьно исполнять программы со сложными входными структурами данных. В работе представлена также реализация предложенного подхода в рамках известной системы символьного исполнения KLEE [2], которая предназначена для символьного исполнения байт-кода платформы LLVM. В свою очередь, LLVM используется компиляторами более десяти различных языков, включая C, C++, Go, Kotlin и др. Представленная реализация имеет открытый исходный код и может служить основой для создания эффективных инструментов контроля качества на основе KLEE.

Таким образом, основными задачами данной работы является следующее.

1. Формальное описание символьного исполнения с ленивой инициализацией.
2. Алгоритм вычисления слабейших предусловий относительно произвольных ветвей программы.
3. Масштабирование подхода на случаи использования в программах динамической памяти.
4. Реализация алгоритма вычисления слабейших предусловий в символьной виртуальной машине KLEE.

Обзор

Многие работы (например, [16–19]) направлены на улучшение символьного исполнения программ с динамической памятью путем применения абстракций для ускорения процесса символьного анализа программы, ведущих к потере информации о программе. Ленивая инициализация, используемая в настоящей работе, не является абстракцией такого рода. Предложенный подход оказывается ортогональным вышеупомянутым и может использоваться совместно с ними.

В [20–22] использована аналогичная основа, как и в настоящей работе. В работах [20] и [21] предложено отсекать пути в программе, которые могут быть поглощены с помощью резюме уже посещенных путей. В [21] резюме считается при помощи интерполяции, а в [20] — путем вычисления слабейшего предусловия пути. Вычисление слабейших предусловий в последнем случае полагается на трудно реализуемую трансформацию программ, которая должна быть нетривиально расширена для поддержки новых языковых конструкций. Такой проблемы нет в предложенном подходе, так как слабейшие предусловия порождаются из существующих артефактов символьного исполнения. В работе [21] при помощи интерполяции невыполнимой формулы, выражающей достижимость отдельного пути, вычислены аннотации, аппроксимирующие множество достижимых состояний программы сверху. Получаемый интерполянт в общем случае грубее слабейшего предусловия пути, а потому на практике целесообразно рассматривать символьные пути, которые можно было бы отсечь, пользуясь подходом настоящей работы. В [22] рассмотрены некоторые формальные аспекты ленивой инициализации и алгоритмы для ее вычисления. В представленной работе предложен способ применения ленивой инициализации для вычисления слабейших предусловий.

Слабейшие предусловия также используются в дедуктивной верификации программ, в методе обратного символьного анализа, основанного на идеях Э.В. Дейкстры [14]. Данный метод применен для доказательства корректности троек Хоара и позволяет порождать промежуточные логические аннотации, корректность которых затем проверена системой проверки доказательств. При этом циклы и рекурсивные вызовы можно аппроксимировать пользовательскими индуктивными инвариантами [23]. В некоторых случаях дедуктивная верификация способна доказывать корректность всех поведений программы, однако, как правило, она требует существенных усилий от пользователя. Отметим, что большинство дедуктивных верификаторов поддерживают лишь фрагменты языков программирования [24, 25]. В данной работе предложен альтернативный метод, который является полностью автоматическим, реализован в виртуальной машине KLEE и поддерживает все инструкции LLVM.

Демонстрационный язык

Опишем демонстрационный язык программирования с возможностью изменения динамически выделяемой памяти. В качестве примеров рассмотрим программы на этом языке программирования.

Синтаксис демонстрационного языка представлен на листинге 1. Программа — нумерованная последовательность инструкций. Инструкции ***fail*** и ***halt*** вызывают остановку программы с ошибкой и без ошибки соответственно. Демонстрационный язык не содержит функций и циклов, однако включает в себя условный переход на инструкцию ***goto***. В условном переходе слева от стрелки записывается условие, а справа — номер инструкции, на которую будет осуществлен переход, если условие выполнено. Когда в программе веток условного перехода много, то они проверяются одна за

Program ::= (ℕ: *Statement*)*
Statement ::= ***fail*** | ***halt*** | *Location* := *Expr* | ***goto*** {Expr → ℕ}$^+$
Expr ::= ***true*** | ***false*** | ℤ | *Expr BinOpExpr* | *UnOp Expr* | *Location* | ***alloc*** ℕ
Location ::= *ident* | * *Expr*

Листинг 1. Синтаксис демонстрационного языка

List. 1. Demo language syntax

одной в том порядке, в котором они идут в тексте программы, и переход происходит только по одной из них. Если ни одно условие не было выполнено, исполняется следующая после условного перехода инструкция. Оператор присваивания := имеет ту же семантику, что и в языке C++. Другими словами, если слева от присваивания находится переменная, то запись ведется в память, выделенную под эту переменную, а если слева от присваивания стоит разыменование выражения, то сначала вычисляется именно оно, а затем происходит запись по адресу, соответствующему вычисленному выражению.

Демонстрационный язык содержит бинарные и унарные арифметические и булевы операторы *BinOp* и *UnOp*. Оператор ***alloc*** выделяет в памяти зануленный массив фиксированного размера, не пересекающийся с прежде выделенными массивами, а также возвращает указатель на его начало. Для обращения к памяти в массиве используем следующую нотацию, применяемую в C++: $p[e] \stackrel{\text{def}}{=} *(p + e)$.

В языке имеется два типа данных: булев и чисел. Рассмотрим только корректно типизированные программы: операторам, ожидающим числа/булевы значения, передаются, соответственно, только числа/булевы значения. Для любой ветки оператора ***goto***, встречающегося в программе, всегда существует инструкция с соответствующим номером, а имена идентификаторов и ключевые слова языка не пересекаются. Запись по константным адресам, в том числе по нулю (например, $*42 := 1$, $* := 2$), для простоты изложения будем считать корректной. Таким образом, имеется лишь один вид ошибок в программах — достижимость инструкции ***fail***.

Язык предусловий и постусловий

Пусть $\Sigma \stackrel{\text{def}}{=} \langle \mathbb{Z}, \Sigma_f, \Sigma_p \rangle$ – сигнатура арифметики, расширенная функциональным символом разыменования $* : \mathbb{Z} \to \mathbb{Z}$, т. е.

$$\Sigma_f \stackrel{\text{def}}{=} \{+, \times, -, *\}, \Sigma_p \stackrel{\text{def}}{=} \{=, <, \leq, >, \geq\}.$$

Семантика арифметических функций и предикатов является стандартной; семантика функции разыменования $*$ не фиксирована. Разыменование переменной запишем как $* x$, опуская скобки (как в языке C++). Символами $\top$ и $\bot$ обозначим истину и ложь. Пред- и постусловия выразим бесквантерными формулами в используемом языке. Для автоматической проверки выполнимости формул в этом языке могут быть использованы SMT-решатели, такие как Z3 [26].

```
1. p[0] := 1
2. q[0] := 2
3. i := 0
4. goto { !(i < n) -> 9 }
5. goto { p[i] == q[i] -> 8 }
6. i := i + 1
7. goto { true -> 4 }
8. fail
9. halt
```

Листинг 2. Пример программы на демонстрационном языке, которая проверяет, что в двух массивах по индексу с одним и тем же номером лежат различные значения

List. 2. An example of a code snippet in the demo language. The code checks that all indices of two arrays contain different values

Слабейшие предусловия

Согласно Дейкстре [14], часто важным оказывается не полная семантика программы, а лишь возможность обеспечить истинность конкретного постусловия, ради которого программа разработана. Так как вычисление слабейшего предусловия для всей программы *сразу* оказывается на практике слишком трудоемкой задачей [13], необходимо обеспечить истинность постусловия после исполнения *конкретного пути* в программе. Путем в программе будем называть конечную последовательность меток, следующих друг за другом в исходной программе инструкций, которые могут быть выполнены вычислителем в ходе исполнения программы. Заметим, что так как путь — *конечная* последовательность, его вычисление всегда завершится, а потому не нужно разделять понятия свободного и несвободного предусловия.

Тройка Хоара относительно пути в программе определяется аналогично классической тройке Хоара относительно всей программы [23]. А именно, тройка $\{P\}path\{R\}$ является *корректной*, когда каждое состояние программы, удовлетворяющее предусловию P, после исполнения на нем инструкций пути $path$ дает состояние, удовлетворяющее постусловию R. Например, для программы с листинга 2 корректна следующая тройка Хоара: $\{n = 0 \wedge i \geq 42\}$ 3, 4, 9 $\{n \geq 0\}$. Заметим, что всякая тройка вида $\{\bot\}path\{R\}$ всегда является корректной, а значит, $\bot$ является предусловием любого пути относительно любого постусловия, однако не несет в себе информации о программе. Потому на практике востребовано *слабейшее предусловие пути*.

Пусть имеется некоторый путь $path$ в программе и желаемое постусловие R, тогда слабейшее предусловие этого пути относительно постусловия обозначим как $wp(path, R)$. Оно должно удовлетворять двум ограничениям.

Во-первых, для любого пути $path$ и постусловия R корректна тройка $\{wp(path, R)\}path\{R\}$, т. е. $wp(path, R)$ действительно вычисляет предусловие пути. Например, для программы с листинга 2 можно проверить, какие начальные состояния после исполнения инструкций 3, 4, 9 дадут состояния, удовлетворяющие условию $n \geq 0$:

$$wp([3, 4, 9], n \geq 0) \equiv wp([3, 4], \neg (i < n) \wedge n \geq 0) \equiv$$
$$\equiv wp([3], \neg (i < n) \wedge n \geq 0) \equiv wp([\ \ ], \neg (0 < n) \wedge n \geq 0) \equiv$$
$$\equiv \neg (0 < n) \wedge n \geq 0 \Leftrightarrow n = 0.$$

Другими словами, только если в начальном состоянии n было равно нулю, то после исполнения инструкций 3, 4, 9 значение n все еще будет неотрицательно.

Во-вторых, для любого предусловия P, если корректна тройка $\{P\}path\{R\}$, то $P \Rightarrow wp(path, R)$.

При помощи слабейшего предусловия пути можно найти ошибки в программе и доказать их отсутствие. Например, для программы с листинга 2 проверим достижимость инструкции ***fail***, рассчитав слабейшее предусловие пути из начала программы:

$$wp([1, 2, 3, 4, 5, 8], \top) \equiv p = q \wedge 0 < n.$$

Это предусловие утверждает, что падение программы в инструкцию ***fail*** возможно, если изначально p и q указывали на одну и ту же память. Так как эта формула выполнима, то инструкция ***fail*** оказывается достижимой. С другой стороны, можно проверить, что нельзя достичь инструкции ***fail*** таким образом, чтобы размер массивов данных n оказался меньше $i - 1$:

$$wp([1, 2, 3, 4, 5, 8], n < i - 1) \equiv \bot.$$

Символьное исполнение программ с динамической памятью

Представим символьное исполнение с ленивой инициализацией программ с динамической памятью. Это широко известный подход, который применяется для автоматического поиска ошибок в программе. Основная идея символьного исполнения состоит в том, чтобы исполнять инструкции программы на *символьной памяти*.

Символьной памятью назовем конечное отображение из символьных локаций *loc* в символьные термы *term*, определенные на листинге 3. Пусть Σ — множество всех таких отображений, $\epsilon \in \Sigma$ — пустая символьная память (пустое отображение).

Символьное состояние памяти позволяет представлять (потенциально бесконечное) множество конкретных состояний программы. Например, после символьного исполнения программы с листинга 4 на пустой символьной памяти ϵ, символьная память будет выглядеть следующим образом:

$$\{x \mapsto 42; y \mapsto z; p \mapsto *(z + 10); r \mapsto 0;$$
$$*(r + 1) \mapsto u; *(r + 2) \mapsto 0\}.$$

Эта символьная память представляет бесконечное число конкретных состояний, которые могут быть получены из него, например, подстановкой конкретных чисел вместо z или u.

Чтобы иметь возможность символьно исполнять произвольные инструкции, требуются следующие операторы: $\_[\![\_]\!] : \Sigma \times term \to term$ — подстановки в терм значений из символьной памяти; $read : \Sigma \times loc \to term$ — чтения из символьного памяти; $write : \Sigma \times loc \times term \to \Sigma$ — записи в символьную память по некоторой локации некоторого терма; $alloc : \Sigma \times loc \times \mathbb{N} \to \Sigma$ — выделения символьной памяти фиксированного размера по некоторой локации.

term ::= ***true*** | ***false*** | ℕ | *term BinOp term* | *UnOp term* | *loc*
loc ::= *ident* | ∗ *term*

Листинг 3. Грамматика символьных выражений

List. 3. Symbolic expressions grammar

```
1. x := 42
2. y := z
3. p := y[10] // *(y + 10)
4. r := alloc 3
5. r[1] := u
```

Листинг 4. Фрагмент программы, меняющей динамическую память

List. 4. A fragment of a program that changes dynamic memory

Эти операторы имеют естественные свойства. Например, для символьной памяти $\sigma \in \Sigma$, локаций $x, y \in loc$, идентификатора $z \in loc$, чисел $i, n \in \mathbb{N}$ и символьных термов $t, t_1, t_2 \in term$, справедливо следующее:

$$x \in dom(\sigma) \Rightarrow read(\sigma, x) = \sigma(x),$$
$$x \notin dom(\sigma) \Rightarrow read(\sigma, x) = x,$$
$$read(write(\sigma, x, t), x) = t,$$
$$x \neq y \Rightarrow read(write(\sigma, y, t), x) = read(\sigma, x),$$
$$\sigma[\![z]\!] = read(\sigma, z),$$
$$\sigma[\![t_1\ BinOp\ t_2]\!] = \sigma[\![t_1]\!] BinOp\ \sigma[\![t_2]\!],$$
$$\sigma[\![*t]\!] \Rightarrow read(\sigma, *\sigma[\![t]\!]),$$
$$0 \le i < n = read(alloc(\sigma, x, n), *(x + i)) = 0.$$

На листинге 5 представлен алгоритм символьного исполнения. *Цветом выделена модификация для исполнения выделенного пути в программе, которая будет использована в разделе «Слабейшие предусловия в символьном исполнении» для вычисления слабейшего предусловия*. Алгоритм оперирует *символьными состояниями* — тройками $\langle s, \pi, r \rangle$, где s — фрагмент символьной памяти, π = c.pc — условие пути (англ. *path condition*) и r = c.path — список исполненных инструкций. *Условие пути* является, с одной стороны, формулой в языке пред- и постусловий, а с другой стороны — символьным термом булевого типа. В условии пути накапливаются логические ограничения из операторов ветвления **goto**, т. е. условия, которые должны быть выполнены для того, чтобы текущее состояния было достижимо.

Символьное исполнение работает следующим образом. Поддерживается очередь символьных состояний, которые требуется исполнить. Из этой очереди выбирается следующее символьное состояние согласно стратегии выбора, реализованной в searcher.pick (например, выбирать состояние с наименьшим или наибольшим числом пройденных инструкций). Для выбранного состояния исполняется следующая инструкция. Шаг символьного исполнения потенциально может породить больше одного символьного состояния. Например, когда исполняется оператор ветвления **goto**, то в отличии от обычного исполнения, где переход произойдет в *одну* инструкцию, в символьном исполнении будет выполнен переход *по всем веткам* исполнения сразу. Однако есть и другой потенциальный источник недетерминизма: инструкции, изменяющие динамическую память.

Например, несмотря на то, что в программе с листинга 6 отсутствуют конструкции ветвления, у нее есть два различных сценария поведения, зависящих от

```
function ExecuteSymbolically(state, path):
    Ввод:  state — начальное символьное состояние
           path — список инструкций, которые нужно посетить

    searcher.queue = {state}
    while searcher.queue is not empty do
        s = searcher.pick(path)
        children = step(s)
        for all c in children do
            if c.pc is sat then
                if c.path = path then yield c
                else searcher.add(c)
```

Листинг 5. Алгоритм символьного исполнения с (выделенной цветом) модификацией для исполнения выделенного пути в программе

List. 5. Symbolic execution algorithm with a (highlighted) modification for specific program path execution

того, указывают ли p и q на одну и ту же память или нет. Потому, если символьно исполнить эту программу на пустом символьном состоянии, то исполнение первой инструкции даст одно новое состояние, а исполнение второй инструкции разветвит символьное исполнение, как показано на рисунке.

Корректная поддержка этой формы недетерминизма программ с динамической памятью возможна в символьном исполнении благодаря механизму **ленивой инициализации**. Он работает следующим образом. Пусть в символьном состоянии есть n символьных локаций. Тогда при доступе к новой символьной локации порождается $n + 1$ символьное состояние: в n из них новая символьная локация приравнена одной из старых локаций в условии пути, и в одном символьном состоянии новая локация принимается отличной от всех старых. Таким образом, например, получаются символьные состояния на рисунке, соответствующие листингу 6. Применяя механизм ленивой инициализации, можно обработать все возможные состояния памяти программы и *только их*. На практике ленивую инициализацию проводят с учетом типов данных, доступных ресурсов и т. п., чтобы уменьшить число перебираемых локаций.

```
1p[0] := 1
1q[0] := 2
```

Листинг 6. Линейный фрагмент программы с листинга 2, требующий разветвления символьного исполнения

List. 6. A linear fragment of the program from Listing 2, which requires symbolic execution branching

Для каждого состояния, порожденного шагом символьного исполнения, проведем проверку выполнимости условия пути. Если условие пути невыполнимо (как формула логики первого порядка), состояние недостижимо, и оно отбрасывается. Если условие пути выполнимо, состояние добавляется алгоритмом символьного исполнения в очередь обработки состояний. Проверку выполнимости условия пути можно проводить при помощи логического решателя, например SMT-решателя Z3 [26].

Таким образом, в ходе символьного исполнения получены символьные состояния, которые представляют классы различных сценариев поведения программы. Покажем, как это свойство символьных состояний может быть использовано для вычисления слабейших предусловий.

Слабейшие предусловия в символьном исполнении

Рассмотрим новый алгоритм вычисления слабейшего предусловия пути в программе. Он основан на простой модификации алгоритма символьного исполнения, *выделенной цветом* на листинге 5, позволяющей символьно исполнять отдельный путь в программе. Модификация превращает алгоритм в сопрограмму, дающую символьные состояния, чьи пути совпадают с требующимися. Для эффективной работы алгоритма следует также модифицировать searcher таким образом, чтобы метод pick выбирал только те состояния, которые идут по требуемому пути (т. е. путь внутри которых является префиксом требуемого пути).

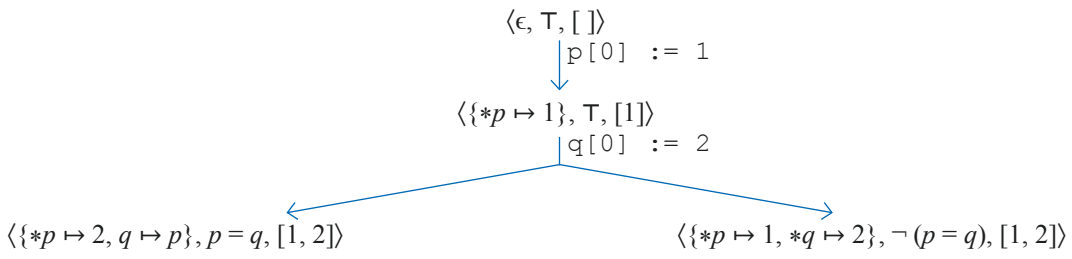

Рисунок. Граф разветвления символьного исполнения на программе с листинга 6

Figure. Symbolic execution graph for the program from Listing 6

```
function wpSE(path, R):
    Ввод: path — список номеров инструкций программы
          R — постусловие
    Вывод: слабейшее предусловие пути path относительно R

    states := ExecuteSymbolically(⟨ϵ, ⊤, [ ]⟩, path)
    return ⋁_{⟨s, π, _⟩ ∈ states} (π ∧ s ⟦R⟧)
```

Листинг 7. Алгоритм вычисления слабейшего предусловия при помощи символьного исполнения

List. 7. Algorithm for calculating the weakest precondition using symbolic execution

Полученный алгоритм вычисления слабейшего предусловия пути при помощи символьного исполнения представлен на листинге 7.

Этот алгоритм принимает список номеров инструкций программы и постусловие и возвращает слабейшее предусловие полученного пути относительно полученного постусловия. На первом шаге алгоритм, посредством вызова сопрограммы `ExecuteSymbolically` на пустом символьном состоянии $\langle \epsilon, \top, [\ ] \rangle$ и входном пути, получает множество символьных состояний `states`. Данное множество представляет все возможные достижимые состояния программы, которые могут быть получены в результате обычного исполнения инструкций заданного пути. Далее алгоритм вычисляет дизъюнкцию по всем символьным состояниям $\langle s, \pi, \_ \rangle \in states$ конъюнкций из условия пути π и подстановки $s\ [\![R]\!]$ значений переменных из состояния s в постусловие R.

Например, вычисление функции $wpSE([1,2,3,4,5,8], \top)$ для примера с листинга 2 даст множество из одного символьного состояния $states = \{\langle s, \pi\ [1,2,3,4,5,8] \rangle\}$, где

$$s \equiv \{*p \mapsto 2;\ q \mapsto p;\ i \mapsto 0\},$$
$$\pi \equiv p = q \wedge 0 < n \wedge 2 = 2.$$

Тогда $wpSE([1,2,3,4,5,8], \top) \equiv \pi \wedge s[\![\top]\!] \equiv \pi \wedge \top \Leftrightarrow \pi$, где $2 = 2$ и получено путем подстановки в условие $p[i] = q[i]$ из кода значений состояния s:

$$s[\![*(p+i) = *(q+i)]\!] \equiv s[\![*(p+i)]\!] = s[\![*(q+i)]\!] \equiv$$
$$\equiv s(*\ (s[\![p]\!] + s[\![i]\!])) = s(*\ (s[\![q]\!] + s[\![i]\!])) \equiv$$
$$\equiv s(*\ (p+0)) = s(*\ (p+0)) \equiv 2 = 2.$$

Таким образом, функция $wpSE$ возвращает формулу, эквивалентную слабейшему предусловию $wp([1,2,3,4,5,8], \top) \equiv p = q \wedge 0 < n$.

При помощи того же символьного состояния s можно вычислить предусловие пути из другого примера:

$$wpSE([1,2,3,4,5,8], n < i - 1) \equiv \pi \wedge s[\![n < i - 1]\!] \equiv$$
$$\equiv p = q \wedge 0 < n \wedge 2 = 2 \wedge n < 0 - 1 \Leftrightarrow \bot.$$

Таким образом, алгоритм $wpSE$ вычисляет слабейшее предусловие пути программ с динамической памятью.

Теорема 1. Пусть *path* — последовательность меток кода и R — постусловие. Тогда алгоритм $wpSE$ из листинга 7 вычисляет слабейшее предусловие *path* относительно R. Другими словами,

$$\boldsymbol{wpSE}(\boldsymbol{path}, \boldsymbol{R}) \Leftrightarrow \boldsymbol{wp}(\boldsymbol{path}, \boldsymbol{R}).$$

Этот факт легко доказывается разбором случаев аналогично [27].

Простота алгоритма $wpSE$ указывает на то, что вся содержательная часть вычислений осуществляется алгоритмом символьного исполнения. Это является **главным преимуществом** предлагаемого подхода по сравнению с подходами, реализующими вычисление слабейшего предусловия непосредственно по тексту программы. Алгоритм символьного исполнения для программ с динамической памятью проще реализовать, чем вычисление слабейшего предусловия напрямую, так как последний требует *обратного* исполнения программы, что весьма затруднительно реализовать для программ с динамической памятью. Символьное же исполнение подразумевает *прямое* исполнение инструкций, при этом все возможные поведения программы корректно отслеживаются при помощи механизма ленивой инициализации. Более того, если кроме работы с динамической памятью потребуется поддержать другие возможности языка, такие как работа со статической памятью, виртуальными вызовами, исключениями и т. п., реализация обратного анализа станет еще сложнее, в то время как символьное исполнение все еще будет легко расширяемо на эти случаи. Таким образом, предложенный алгоритм можно легко расширить на более сложные виды программ с динамической памятью, используя существующие расширения символьного исполнения.

Реализация в KLEE

KLEE — символьная виртуальная машина, построенная на основе инфраструктуры компилятора LLVM. Для реализации алгоритма вычисления слабейших предусловий в виртуальную машину KLEE внесем два ключевых изменения.

1. Добавим возможность исполнять любую последовательность инструкций в изоляции. Для этого в KLEE реализована ленивая инициализация регистров и объектов в динамической памяти. Ленивая инициализация регистра вызывается, если при исполнении инструкции в изоляции в читаемом регистре еще нет значения, а ленивая инициализация объектов происходит при разыменовании символь-

ного указателя. После перебора всех объектов в памяти проверяется выполнимость условия пути, при котором символьный указатель не может указывать ни на один из объектов в памяти. Если условие выполнимо, в память добавляется новый символьный объект с адресом, равным адресу указателя.

2. Для вычисления слабейшего предусловия реализован оператор подстановки в терм значений из символьной памяти. В KLEE символьные значения регистров подставлены из значений регистров в символьном состоянии. Для подстановки символьных объектов из памяти необходимо сначала подставить значения в терм указателя на объект, а затем разыменовать этот указатель в символьном состоянии.

Заключение

В работе предложен алгоритм вычисления слабейшего предусловия пути в программе относительно заданного постусловия, построенный на базе символьного исполнения с ленивой инициализацией данных. Этот результат важен для реализации символьных машин, которые можно использовать на практике. Такая символьная машина может использовать как прямой, так и обратный анализ, так как последний позволяет анализировать программы, недоступные для прямого анализа. Например, обратный анализ хорошо зарекомендовал себя в решении задач автоматического тестирования и валидации ошибок.

Проблема обратного анализа заключается в том, что его тяжело реализовать так, чтобы он поддерживал реальные программы, так как сложно описать обратную семантику большинства существующих языковых конструкций, таких как, например, операции с динамической памятью. В результате работы сделан важный шаг для построения реалистичных символьных машин: показано, как осуществлять обратный анализ при помощи прямого символьного анализа программ.

В дальнейшем планируется исследование стратегий двунаправленного символьного исполнения путем совместного применения прямого и обратного символьного анализа. Эффективная стратегия позволит бороться с проблемой взрыва путей исполнения на практике.

Литература

1. Baldoni R., Coppa E., D'Elia D.C., Demetrescu C., Finocchi I. A survey of symbolic execution techniques // ACM Computing Surveys. 2018. V. 51. N 3. https://doi.org/10.1145/3182657
2. Cadar C., Dunbar D., Engler D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs // Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation. 2008. P. 209–224.
3. Wang F., Shoshitaishvili Y. ANGR – the next generation of binary analysis // Proc. of the IEEE Cybersecurity Development (SecDev). 2017. P. 8–9. https://doi.org/10.1109/SecDev.2017.14
4. Burnim J., Sen K. Heuristics for scalable dynamic test generation // Proc. of the ASE 2008 — 23rd IEEE/ACM International Conference on Automated Software Engineering. 2008. P. 443–446. https://doi.org/10.1109/ASE.2008.69
5. Tillmann N., Halleux J. Pex-white box test generation for .NET // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2008. V. 4966. P. 134–153. https://doi.org/10.1007/978-3-540-79124-9_10
6. Păsăreanu C.S., Rungta N. Symbolic PathFinder: symbolic execution of Java bytecode // Proc. of the 25th IEEE/ACM International Conference on Automated Software Engineering. 2010. P. 179–180. https://doi.org/10.1145/1858996.1859035
7. Mossberg M., Manzano F., Hennenfent E., Groce A., Grieco G., Feist J., Brunson T., Dinaburg A. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts // Proc. of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2019. P. 1186–1189. https://doi.org/10.1109/ASE.2019.00133
8. Godefroid P., Klarlund N., Sen K. DART: Directed automated random testing // Proc. of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI). 2005. P. 213–223.
9. Godefroid P., Levin M.Y., Molnar D. SAGE: whitebox fuzzing for security testing // Communications of the ACM. 2012. V. 55. N 3. P. 40–44. https://doi.org/10.1145/2093548.2093564
10. Chipounov V., Kuznetsov V., Candea G. S2E: a platform for in-vivo multi-path analysis of software systems // ACM SIGPLAN Notices. 2011. V. 46. N 3. P. 265–278. https://doi.org/10.1145/1961296.1950396
11. Vishnyakov A., Fedotov A., Kuts D., Novikov A., Parygina D., Kobrin E., Logunova V., Belecky P., Kurmangaleev S. Sydr: Cutting edge dynamic symbolic execution // Proc. of the 2020 Ivannikov

References

1. Baldoni R., Coppa E., D'Elia D.C., Demetrescu C., Finocchi I. A survey of symbolic execution techniques. *ACM Computing Surveys*, 2018, vol. 51, no. 3. https://doi.org/10.1145/3182657
2. Cadar C., Dunbar D., Engler D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008, pp. 209–224.
3. Wang F., Shoshitaishvili Y. ANGR – the next generation of binary analysis. *Proc. of the IEEE Cybersecurity Development (SecDev)*, 2017, pp. 8–9. https://doi.org/10.1109/SecDev.2017.14
4. Burnim J., Sen K. Heuristics for scalable dynamic test generation. *Proc. of the ASE 2008 — 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 443–446. https://doi.org/10.1109/ASE.2008.69
5. Tillmann N., Halleux J. Pex-white box test generation for .NET. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008, vol. 4966, pp. 134–153. https://doi.org/10.1007/978-3-540-79124-9_10
6. Păsăreanu C.S., Rungta N. Symbolic PathFinder: symbolic execution of Java bytecode. *Proc. of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 179–180. https://doi.org/10.1145/1858996.1859035
7. Mossberg M., Manzano F., Hennenfent E., Groce A., Grieco G., Feist J., Brunson T., Dinaburg A. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. *Proc. of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1186–1189. https://doi.org/10.1109/ASE.2019.00133
8. Godefroid P., Klarlund N., Sen K. DART: Directed automated random testing. *Proc. of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2005, pp. 213–223.
9. Godefroid P., Levin M.Y., Molnar D. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 2012, vol. 55, no. 3, pp. 40–44. https://doi.org/10.1145/2093548.2093564
10. Chipounov V., Kuznetsov V., Candea G. S2E: a platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices*, 2011, vol. 46, no. 3, pp. 265–278. https://doi.org/10.1145/1961296.1950396
11. Vishnyakov A., Fedotov A., Kuts D., Novikov A., Parygina D., Kobrin E., Logunova V., Belecky P., Kurmangaleev S. Sydr: Cutting edge dynamic symbolic execution. *Proc. of the 2020 Ivannikov Ispras Open Conference (ISPRAS)*, 2020, pp. 46–54. https://doi.org/10.1109/ISPRAS51486.2020.00014

Ispras Open Conference (ISPRAS). 2020. P. 46–54. https://doi.org/10.1109/ISPRAS51486.2020.00014
12. Dinges P., Agha G. Targeted test input generation using symbolic-concrete backward execution // Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering. 2014. P. 31–36. https://doi.org/10.1145/2642937.2642951
13. Chandra S., Fink S.J., Sridharan M. Snugglebug: a powerful approach to weakest preconditions // ACM SIGPLAN Notices. 2009. V. 44. N 6. P. 363–374. https://doi.org/10.1145/1543135.1542517
14. Dijkstra E.W. A Discipline of Programming. Englewood Cliffs, N.J.: Prentice-Hall, 1976. XVII, 217 p.
15. Khurshid S., Păsăreanu C.S., Visser W. Generalized symbolic execution for model checking and testing // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2003. P. 553–568. https://doi.org/10.1007/3-540-36577-X_40
16. Anand S., Păsăreanu C.S., Visser W. Symbolic execution with abstraction // International Journal on Software Tools for Technology Transfer. 2009. V. 11. N 1. P. 53–67. https://doi.org/10.1007/s10009-008-0090-1
17. Lin Y. Symbolic execution with over-approximation: Ph.D. Thesis. The University of Melbourne, 2017.
18. Rungta N., Mercer E.G., Visser W. Efficient testing of concurrent programs with abstraction-guided symbolic execution // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2009. V. 5578. P. 174–191. https://doi.org/10.1007/978-3-642-02652-2_16
19. Berdine J., Calcagno C., O'Hearn P.W. Symbolic execution with separation logic // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2005. V. 3780. P. 52–68. https://doi.org/10.1007/11575467_5
20. Yi Q., Yang Z., Guo S., Wang C., Liu J., Zhao C. Postconditioned symbolic execution // Proc. of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST). 2015. P. 1–10. https://doi.org/10.1109/ICST.2015.7102601
21. McMillan K.L. Lazy annotation for program testing and verification // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2010. V. 6174. P. 104–118. https://doi.org/10.1007/978-3-642-14295-6_10
22. Deng X., Lee J., Robby. Efficient and formal generalized symbolic execution // Automated Software Engineering. 2012. V. 19. N 3. P. 233–301. https://doi.org/10.1007/s10515-011-0089-9
23. Hoare C.A.R. An axiomatic basis for computer programming // Communications of the ACM. 1969. V. 12. N 10. P. 576–580. https://doi.org/10.1145/363235.363259
24. Nepomniaschy V.A., Anureev I.S., Promskii A.V. Towards verification of C programs: axiomatic semantics of the C-kernel language // Programming and Computer Software. 2003. V. 29. N 6. P. 338–350. https://doi.org/10.1023/B:PACS.0000004134.24714.e5
25. Blazy S., Leroy X. Mechanized semantics for the Clight subset of the C language // Journal of Automated Reasoning. 2009. V. 43. N 3. P. 263–288. https://doi.org/10.1007/s10817-009-9148-3
26. De Moura L., Bjørner N. Z3: an efficient SMT solver // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2008. V. 4963. P. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
27. Ball T., Daniel J. Deconstructing dynamic symbolic execution // Proc. of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering. IOS Press, 2015.

12. Dinges P., Agha G. Targeted test input generation using symbolic-concrete backward execution. *Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014, pp. 31–36. https://doi.org/10.1145/2642937.2642951
13. Chandra S., Fink S.J., Sridharan M. Snugglebug: a powerful approach to weakest preconditions. *ACM SIGPLAN Notices*, 2009, vol. 44, no. 6, pp. 363–374. https://doi.org/10.1145/1543135.1542517
14. Dijkstra E.W. *A Discipline of Programming*. Englewood Cliffs, N.J., Prentice-Hall, 1976, XVII, 217 p.
15. Khurshid S., Păsăreanu C.S., Visser W. Generalized symbolic execution for model checking and testing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2003, pp. 553–568. https://doi.org/10.1007/3-540-36577-X_40
16. Anand S., Păsăreanu C.S., Visser W. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer*, 2009, vol. 11, no. 1, pp. 53–67. https://doi.org/10.1007/s10009-008-0090-1
17. Lin Y. *Symbolic execution with over-approximation*. Ph.D. Thesis. The University of Melbourne, 2017.
18. Rungta N., Mercer E.G., Visser W. Efficient testing of concurrent programs with abstraction-guided symbolic execution. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2009, vol. 5578, pp. 174–191. https://doi.org/10.1007/978-3-642-02652-2_16
19. Berdine J., Calcagno C., O'Hearn P.W. Symbolic execution with separation logic. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2005, vol. 3780, pp. 52–68. https://doi.org/10.1007/11575467_5
20. Yi Q., Yang Z., Guo S., Wang C., Liu J., Zhao C. Postconditioned symbolic execution. *Proc. of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10. https://doi.org/10.1109/ICST.2015.7102601
21. McMillan K.L. Lazy annotation for program testing and verification. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010, vol. 6174, pp. 104–118. https://doi.org/10.1007/978-3-642-14295-6_10
22. Deng X., Lee J., Robby. Efficient and formal generalized symbolic execution. *Automated Software Engineering*, 2012, vol. 19, no. 3, pp. 233–301. https://doi.org/10.1007/s10515-011-0089-9
23. Hoare C.A.R. An axiomatic basis for computer programming. *Communications of the ACM*, 1969, vol. 12, no. 10, pp. 576–580. https://doi.org/10.1145/363235.363259
24. Nepomniaschy V.A., Anureev I.S., Promskii A.V. Towards verification of C programs: axiomatic semantics of the C-kernel language. *Programming and Computer Software*, 2003, vol. 29, no. 6, pp. 338–350. https://doi.org/10.1023/B:PACS.0000004134.24714.e5
25. Blazy S., Leroy X. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 2009, vol. 43, no. 3, pp. 263–288. https://doi.org/10.1007/s10817-009-9148-3
26. De Moura L., Bjørner N. Z3: an efficient SMT solver. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008, vol. 4963, pp. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
27. Ball T., Daniel J. Deconstructing dynamic symbolic execution. *Proc. of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering*. IOS Press, 2015.

Авторы

Мисонижник Александр Владимирович — студент, Санкт-Петербургский государственный университет, Санкт-Петербург, 199034, Российская Федерация, SC 57203100912, https://orcid.org/0000-0002-5907-0324, misonijniki@gmail.com

Костюков Юрий Олегович — аспирант, Санкт-Петербургский государственный университет, Санкт-Петербург, 199034, Российская Федерация, SC 57219623001, https://orcid.org/0000-0003-4607-039X, kostyukov.yurii@gmail.com

Костицын Михаил Павлович — инженер-исследователь, Санкт-Петербургский государственный университет, Санкт-Петербург,

Authors

Aleksandr V. Misonizhnik — Student, St. Petersburg State University (SPbSU), Saint Petersburg, 199034, Russian Federation, SC 57203100912, https://orcid.org/0000-0002-5907-0324, misonijnik@gmail.com

Yurii O. Kostyukov — PhD Student, St. Petersburg State University (SPbSU), Saint Petersburg, 199034, Russian Federation, SC 57219623001, https://orcid.org/0000-0003-4607-039X, kostyukov.yurii@gmail.com

Michael P. Kostitsyn — Research Engineer, St. Petersburg State University (SPbSU), Saint Petersburg, 199034, Russian Federation, https://orcid.org/0000-0001-9982-6571, michael.kosticyn@gmail.com

199034, Российская Федерация, https://orcid.org/0000-0001-9982-6571, michael.kosticyn@gmail.com
Мордвинов Дмитрий Александрович — кандидат физико-математических наук, доцент, доцент, Санкт-Петербургский государственный университет, Санкт-Петербург, 199034, Российская Федерация, SC 57199323753, https://orcid.org/0000-0002-6437-3020, mordvinov.dmitry@gmail.com
Кознов Дмитрий Владимирович — доктор технических наук, доцент, профессор, Санкт-Петербургский государственный университет, Санкт-Петербург, 199034, Российская Федерация, SC 8885649400, https://orcid.org/0000-0003-2632-3193, d.koznov@spbu.ru

Dmitry A. Mordvinov — PhD (Physics & Mathematics), Associate Professor, Associate Professor, St. Petersburg State University (SPbSU), Saint Petersburg, 199034, Russian Federation, SC 57199323753, https://orcid.org/0000-0002-6437-3020, mordvinov.dmitry@gmail.com

Dmitry V. Koznov — D. Sc., Associate Professor, Professor, St. Petersburg State University (SPbSU), Saint Petersburg, 199034, Russian Federation, SC 8885649400, https://orcid.org/0000-0003-2632-3193, d.koznov@spbu.ru

Статья поступила в редакцию 31.05.2022
Одобрена после рецензирования 19.08.2022
Принята к печати 23.09.2022

Received 31.05.2022
Approved after reviewing 19.08.2022
Accepted 23.09.2022