DOI: 10.15514/ISPRAS-2026-38(3)-20

Статический анализатор для распознавания массивов в С-программах для задач фаззинга

Д.В. Кознов, ORCID: 0000-0003-2632-3193 <d.koznov@spbu.ru>
Д.А. Усачев, ORCID: 0009-0003-4863-6964 <usachev63@ro.ru>

Санкт-Петербургский государственный университет,
Россия, 199034, г. Санкт-Петербург, Университетская наб., д. 7–9.

Аннотация. В экосистеме тестирования ПО сетевых устройств крупной телекоммуникационной компании активно применяется фаззинг – подход к тестированию, где на вход тестируемой программе подаются случайные, неожиданные или некорректные входные данные. В связи с отсутствием в языке С динамических массивов как таковых для задач фаззинга С-функций оказывается полезной информация о массивах, которые принимаются на вход функциями. В данной работе предлагается специальный вид статического анализа для автоматического распознавания массивов, с которыми работают С-функции, а также определения их длины (аппроксимации). На основе предложенного метода был реализован предметно-ориентированный инструмент, нацеленный на конкретную кодовую базу компании, который при приемлемой производительности смог достичь значений метрик точности 79% и полноты 98% в распознавании массивов, при этом длина массива была правильно определена в 69% случаях. Интеграция нашего инструмента в экосистему тестирования компании позволила значительно улучшить качество фаззинга, увеличив метрику покрытия кодовой базы на 10%, а количество найденных ошибок – на 40%.

Ключевые слова: статический анализ; поиск динамических массивов; фаззинг; язык С; телекоммуникации.

Для цитирования: Кознов Д.В., Усачев Д.А. Статический анализатор для распознавания массивов в С-программах для задач фаззинга. Труды ИСП РАН, том 38, вып. 3, часит 2, 2026 г., стр. 33–48. DOI: 10.15514/ISPRAS–2026–38(3)–20.

Static Analyzer for Array Recognition in C Programs for Fuzzing

Dmitry Koznov, ORCID: 0000-0003-2632-3193 <d.koznov@spbu.ru>
Danila Usachev, ORCID: 0009-0003-4863-6964 <usachev63@ro.ru>

Saint-Petersburg State University,
7/9 Universitetskaya emb., St. Petersburg, 199034, Russia.

Abstract. Fuzzing is a method of software testing where random, unexpected or invalid data is provided as input to the program under test. It is extensively used for testing network device software at a large telecommunications company. Since the C programming language lacks dynamic arrays, information about arrays passed as input to C functions becomes useful for fuzzing problems. In this paper we propose a special static analysis method for automatic recognition of arrays used by C functions and their length approximation. With this method we have implemented a domain-specific tool, which attained 79% precision and 98% recall in array recognition as well as 69% accuracy in determining their length. Integrating our tool into company's testing ecosystem resulted in a significant improvement of fuzzing quality, increasing the code coverage metric by 10% and the number of found errors – by 40%.

Keywords: static analysis; dynamic array detection; fuzzing; C language; telecommunications.

For citation: Koznov D.V., Usachev D.A. Static analyzer for array recognition in C programs for fuzzing. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 3, part 2, 2026, pp. 33-48 (in Russian). DOI: 10.15514/ISPRAS-2026-38(3)-20.

1. Введение

Предметно-ориентированный анализ программного кода [1] подразумевает решение классических задач статического анализа, поиска клонов и других артефактов для одной кодовой базы, которая имеет значительные размеры (десятки миллионов строк кода). Такие кодовые базы требуют многочисленных средств поддержки для автоматического тестирования, рефакторинга, реинжиниринга, внедрения предметно-ориентированных языков (позволяющих поднять уровень абстракции для некоторой части кода с поддержкой его генерации и обратного проектирования) и так далее. Полученные алгоритмы и инструменты должны корректно работать не для всех возможных программ на выбранном языке программирования (C, Java и так далее), а только на данной кодовой базе, использующей некоторое подмножество стандартных языков и определённые шаблоны проектирования. Предметно-ориентированные инструменты по анализу программного кода в последнее время активно разрабатываются разными компаниями с использованием открытых (open source) инфраструктур, таких как Eclipse, LLVM, MS Visual Studio и других.

В данной работе рассмотрена задача статического определения факта использования С-функцией динамических массивов – то есть буферов памяти, размер которых неизвестен на момент компиляции программы, – а также задача по определению длины (аппроксимации) таких массивов. При этом в языке С синтаксически массивы отсутствуют, и часто процедуры получают их в виде нетипизированных указателей-параметров, и далее работают с этим указателем как с массивом. В общем виде задача является неразрешимой, поэтому предлагается её решать на основе выделенных шаблонов кода, а также статически аппроксимировать сверху длину таких массивов. Следует отметить, что статическая информация о динамических массивах необходима для фаззинга в изоляции С-функций: для каждой такой функции перед фаззингом нужно создать соответствующий контекст, в котором, в частности, должен быть создан динамический массив нужной длины и передан в качестве параметра в эту функцию. Если сделать это неправильно, то фаззинг аварийно завершится, но вовсе не из-за ошибки в функции.

Данная задача возникли в процессе сопровождения кодовой базы крупной телекоммуникационной компании. Данное ПО имеет значительный функционал по

обработке сетевых сообщений, которые оказываются буферами памяти, которые передаются по нетипизированным указателям в отдельные функции. При этом элементами таких буферов оказываются С-структуры, их поля могут содержать другие массивы, в ячейках которых, в свою очередь, могут содержаться другие структуры и так далее.

В рамках статье предлагается метод статического анализа для автоматического выявления массивов, принимаемых на вход С-функциями, и определения их длин. Метод анализирует указатели для моделирования объектов, с которыми работает функция (в том числе массивов), а также потоки данных для определения индексов и длин этих массивов. Метод был реализован в виде программного инструмента, созданного на основе Clang / LLVM. Инструмент был интегрирован в экосистему тестирования компании, в результате чего была повышена эффективность фаззинга.

Статья организована следующим образом. В разделе 2 описывается предложенный метод. Раздел 3 посвящён описанию инструмента, реализующего разработанный метод. В разделе 4 представлены эксперименты. Раздел 5 описывает близкие исследования. Наконец, раздел 6 является заключением, содержащим выводы и направления дальнейших исследований.

2. Метод

На вход методу поступает исходный (анализируемый) С-код. Далее, для каждой функции строится *сводка* – её промежуточное представление, которое используется для анализа на следующих шагах. Затем отдельные сводки связываются и создаётся *граф вызовов* для всей программы. После этого выполняется *восходящий анализ*, в результате которого для каждой функции выдаётся следующее: использует ли она или нет массив переменной длины, в случае использования выдаётся оценка его длины. Схема метода представлена на рис. 1.

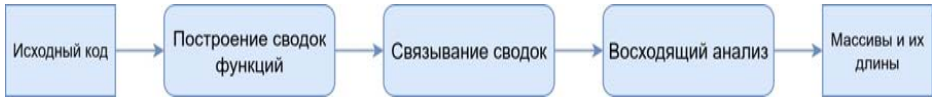

Рис. 1. Схема предложенного метода.
Fig. 1. Schema of the approach.

Рассмотрим подробнее каждый шаг метода.

2.1 Построение сводок функций

Пусть $Functions$ является множеством всех функций анализируемого С-кода. Определим сводку отдельной функции $f \in Functions$ следующим образом:

$$Summary_f = \{OG_f, CFG_f, Parameters_f, GlobalVariables_f, ArrayUsages_f, Constraints_f\},$$

где:

$OG_f = \{V_f, AccessEdges_f, ArithmeticEdges_f\}$ – *граф объектов* языка С (object graph), используемых в теле функции f (константы и литералы, переменные, динамически созданные объекты и пр.);

$CFG_f = \{BB_f, CFGEdges_f, Insn_f\}$ – граф потока управления для f;

$Parameters_f$ – список параметров функции f;

$GlobalVariables_f$ – список глобальных переменных, используемых в f;

$ArrayUsages_f$ – инструкции обращения к массиву в функции f;

$Constraints_f$ – множество ограничений над численными переменными в функции f.

Рассмотрим эти составляющие сводки более детально.

2.1.1 Граф объектов OG_f

Вершина $v \in V_f$ в графе объектов $OG_f = \{V_f, AccessEdges_f, ArithmeticEdges_f\}$ обозначает множество объектов функции f. Объектом, используемым в функции f, может быть: константа или литерал (строковый или составной); переменная (локальная, глобальная, статическая); динамически созданный объект; элемент другого объекта (поле структуры/объединения, элемент массива, взятый адрес объекта); результат вызова функции или приведения типов; результат применения оператора (оператора присваивания, в том числе составного присваивания, например, `+=`, оператора инкремента/декремента, арифметического/логического оператора) и других конструкций. При этом разные объекты могут быть представлены одной вершиной в графе объектов. Например, если указатель `p` в одном случае указывает на переменную `a`, а в другом – на переменную `b`, то объекты `a` и `b` будут представлены одной вершиной, поскольку их нецелесообразно различать в графе объектов. Рёбра в графе объектов бывают двух видов: рёбра доступа $AccessEdges_f$ и арифметические рёбра $ArithmeticEdges_f$. Ребро доступа $e = \{v_1, v_2, op\} \in AccessEdges_f$ означает, что к объекту $v_2 \in V_f$ может быть осуществлён доступ путём применения операции op к объекту $v_1 \in V_f$. Мы рассматриваем виды операции op, перечисленные в табл. 1.

Табл. 1. Виды операций на рёбрах доступа.
Table 1. Operation kinds for access edges.

Вид операции	**Обозначение**
Разыменование указателя `p`	`p[0]`
Доступ к полю `field` структуры (объединения) `s`	`s.field`
Доступ к элементу массива `a` по явному индексу `i`	`a[i]`
Доступ к элементу массива `a` по неизвестному индексу	`a[?]`
Смещение адреса `addr` на константу `i`	`addr.offset(i)`

Таким образом, рёбра доступа (в особенности, связывающие указатель и объект-указуемое) моделируют расположение в памяти объектов, с которыми работает функция. Касаемо рёбер доступа выполняются следующие правила:

- из вершины не могут исходить два ребра доступа с одинаковой операцией op;
- если из вершины исходит ребро доступа к массиву по неизвестному индексу `[?]`, то из неё не могут исходить рёбра доступа к массиву по явному индексу `[i]`.

В качестве примера можно рассмотреть код в листинге 1. Граф объектов для этого примера изображён на рис. 2. Он состоит из семи пронумерованных вершин, каждая вершина на рисунке помечена своим типом. Вершина v_1 соответствует единственному параметру `p` функции `field`, который является указателем. Из v_1 ведёт одно ребро разыменования в вершину v_2, которая представляет структуру `Point` и имеет три поля: `x`, `y` и `d`; для каждого из них имеется соответствующее ребро доступа. Поле `d`, являющееся указателем `void*` и представленное вершиной v_5, имеет ребро доступа по указателю, ведущее в вершину v_6 конкретного типа `Description`. Наконец, имеется ещё одно ребро «структура – поле», ведущее в вершину v_7 типа `int`.

Арифметические рёбра в графе объектов связывают объекты целочисленного типа. Арифметическое ребро $a = \{v_1, v_2, op, s\}$ обозначает тот факт, что после выполнения инструкции с номером s (про нумерацию см. далее) значения объектов $v_1 \in V_f$ и $v_2 \in V_f$ связаны соотношением $v_2 = op(v_1)$, где op является одной из унарных операций, представленных в табл. 2.

```c
typedef struct { int color; } Description;
typedef struct { int x; int y; void *d; } Point;

int foo(Point *p) {
  Description *d = (Description *)p->d;
  return d->color + p->x + p->y;
}
```

Листинг 1. Пример С-кода для иллюстрации графа объектов.
Listing 1. C code snippet for illustrating the object graph.

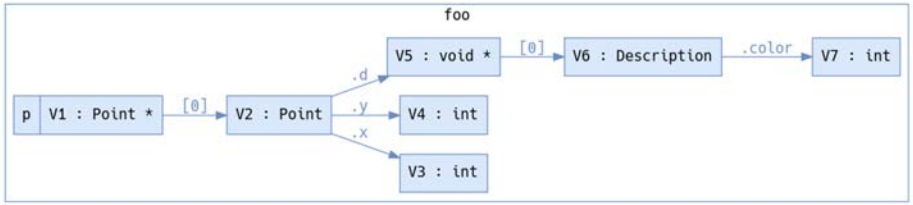

Рис. 2. Пример графа объектов.
Fig. 2. Object graph example.

Табл. 2. Виды операций на арифметических рёбрах.
Table 2. Operation kinds for arithmetic edges.

Вид операции	**Соотношение**
Копирование	$v_2 = v_1$
Сложение с константой	$v_2 = v_1 + c$
Умножение на константу	$v_2 = v_1 * c$
Деление на константу	$v_2 = v_1/c$, c ≠ 0

2.1.2 Граф потока управления CFG_f

На уровне исходного кода определение функции f представлено в виде *дерева абстрактного синтаксиса* (abstract syntax tree, AST). Оно состоит из набора операторов/выражений, которые также могут содержать в себе подвыражения. Как будет видно далее, для более точного анализа полезно учитывать порядок исполнения операторов. В данном случае представление в виде дерева оказалось неудобным, так как для произвольных двух вершин, отвечающих двум операторам, трудно понять, какой оператор должен исполняться раньше. Поэтому для функции f создаётся другое промежуточное представление – *граф потока управления* (control flow graph) CFG_f.Построение графа выполняется следующим образом. Рассматриваются все операторы/выражения в функции f (в том числе и подвыражения), которые мы далее будем называть *инструкциями*. Метод разбивает инструкции на *базовые блоки* (basic block). Базовым блоком называется максимальная последовательность инструкций, удовлетворяющая следующим ограничениям: (i) поток управления функции может попасть в блок лишь войдя в первую инструкцию блока; (ii) инструкции в блоке выполняются последовательно без ветвлений: после первой обязательно выполняется вторая, после второй – третья и так далее до последней. Последняя инструкция в блоке называется *терминатором* и может выполнять выход из блока, переход в его начало, но не может

выполнять переход в середину блока. Также вводятся два дополнительных базовых блока, не содержащих инструкций: блок $EntryBB_f$ (вход в функцию) и блок $ExitBB_f$ (выход из функции). Множество базовых блоков BB_f назначается множеством вершин графа потока управления CFG_f. Все возможные переходы между базовыми блоками представлены множеством рёбер $CFGEdges_f$.

Для инструкций дополнительно вводится нумерация $Insn_f$ так, чтобы в каждом базовом блоке номера инструкций шли подряд. При этом номера инструкций в разных базовых блоках могут соотноситься произвольным образом, главное, чтобы они были различными.

2.1.3 Параметры функции $Parameters_f$

Список параметров функции f обозначим как $Parameters_f = \{p_1, p_2, \dots, p_n\}$. Параметр с номером i имеет вид $p_i = \{v, name\}$, где $v \in V_f$, а $name$ – имя параметра.

2.1.4 Глобальные переменные $GlobalVariables_f$

Множество $GlobalVariables_f$ содержит глобальные переменные, используемые функцией f. Каждая такая переменная представляется в виде пары $\{v, name\}$, где $v \in V_f$, а $name$ – имя переменной.

2.1.5 Инструкции обращения к массиву $ArrayUsages_f$

Для автоматического определения массивов и их индексов в сводку функции f добавляются все инструкции обращения к массивам внутри f. Каждая инструкция представляется в виде $u = \{v_{array}, i, v_{result}, s\} \in ArrayUsages_f$, что означает следующее: инструкция с номером s осуществляет доступ к массиву $v_{array} \in V_f$ по индексу i, и результирующий объект (элемент массива) представлен вершиной $v_{result} \in V_f$. Индекс имеет вид $i = p * v_{index} + q$, где $v_{index} \in V_f$ – объект целочисленного типа, а p и q – это рациональные коэффициенты. Довольно часто индексом является целая константа ($p = 0$) или вершина $v_{index} \in V_f$ ($p = 1$ и $q = 0$). Описанный общий вид индекса нужен для поддержки шаблонов кода, где длина массива нетривиально зависит от параметра функции.

2.1.6 Численные ограничения $Constraints_f$

Для того, чтобы автоматически аппроксимировать длину найденного в функции f массива, необходимо выяснить, насколько большими могут быть индексы, которые используются при обращении к нему, то есть для заданной инструкции обращения к массиву $u = \{v_{array}, i, v_{result}, s\} \in ArrayUsages_f$ необходимо найти численное ограничение сверху на индекс i, которое выполняется перед инструкцией с номером s. Для этого метод строит множество численных ограничений $Constraints_f$.

Каждое ограничение с $\in Constraints_f$ имеет вид $c = \{v_{lhs}, rhs, rel, b\}$ и означает следующее: в начале исполнения базового блока $b \in BB_f$ выполняется соотношение $v_{lhs} relrhs$, где $v_{lhs} \in V_f$ – объект целочисленного типа, $rel \in \{<, \leq, >, \geq, =, \neq\}$ – знак сравнения, а $rhs = p * v_{rhs} + q$, где $v_{rhs} \in V_f$ – объект целочисленного типа, а p и q – рациональные коэффициенты. Ограничения выделяются из условий операторов ветвления, входящих в f.

2.2 Связывание сводок функций

Для осуществления упрощённой версии межпроцедурного анализа, необходимой для решения нашей задачи, строится *граф вызовов*. Для функции $f \in Functions$ через $CallSites_f$

обозначим множество всех её *точек вызова*, то есть номеров соответствующих инструкций вызова функции (см. определение $Insn_f$). Вершинами графа вызовов $CallGraph = \{Functions, CGEdges\}$ являются все функции анализируемого С-кода, а каждое ребро имеет вид $e = \{f, c, g\} \in CGEdges$ и означает, что функция f в точке вызова $c \in CallSites_f$ может вызывать функцию g. Отметим, что функция f может вызывать функцию g в нескольких местах, и поэтому рёбра графа вызовов различаются по точке вызова c.

В предположении, что все вызовы в программе являются *прямыми*, то есть вызываемая функция известна во время компиляции, построение графа вызовов оказывается несложным. Сложность возникает в случае вызовов по указателю на функцию (так называемых *непрямых* вызовов). В таком случае функций g может быть много. Для построения графа вызовов с учётом непрямых вызовов используется алгоритм, предложенный в гл. 19 [2].

Далее, на основе графа вызовов вершины в сводках вызываемой и вызывающей функций связываются между собой *рёбрами возврата*, которое задаётся следующим образом. Пусть имеется ребро графа вызовов $e = \{f, c, g\} \in CGEdges$. Тогда соответствующие ему рёбра возврата имеют вид $\{v_g, c, v_f\}$, где $v_f \in V_f$, $v_g \in V_g$, и $c \in CallSites_f$. Множество всех рёбер возврата обозначим через $RetEdges$. Ребро возврата $\{v_g, c, v_f\}$ добавляется в $RetEdges$ в следующих случаях:

- если v_g отвечает параметру функции g, а v_f – соответствующему аргументу инструкции вызова c в функции f;
- если v_g отвечает аргументу оператора `return`, а v_f соответствует возвращаемому значению инструкции вызова c в функции f;
- если существует глобальная переменная G, которая используется как в функции f, так и в функции g, причём в функции f она представлена вершиной v_f, а в функции g – вершиной v_g.

Каждую сводку $Summary_f$ можно представить в виде отдельной плоскости, содержащей вершины V_f, связанные между собой *горизонтальными рёбрами*, то есть рёбрами доступа и арифметическими рёбрами. Рёбра возврата $RetEdges$, в свою очередь, назовём *вертикальными рёбрами*. Для иллюстрации рассмотрим пример на листинге 2, состоящий из двух функций – `foo` и `bar`. Графы объектов этих функций вместе с рёбрами возврата изображены на рис. 3.

```
int bar(int x, int y) {
  if (x < y) {
    return 1;
  } else {
    return 0;
  }
}
void foo(int *p) {
  int y = 2;
  int z = bar(*p, y);
}
```

Листинг 2. Пример С-кода для иллюстрации связывания сводок.
Listing 2. C code snippet for illustrating summary binding.

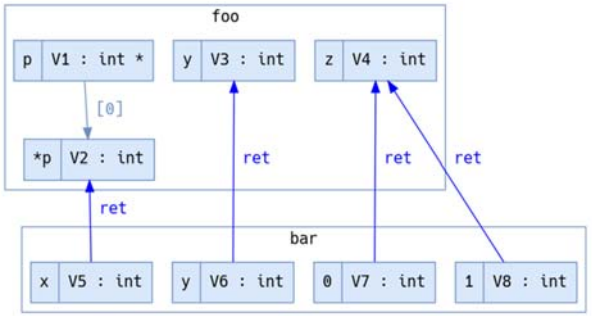

Рис. 3. Связывание сводок функций `foo` *и* `bar` *рёбрами возврата.*
Fig. 3. The binding of summaries for functions `foo` *and* `bar` *using return edges.*

Параметры `x` и `y` функции `bar` (вершины v_5 и v_6) связываются ребром возврата с выражением `*p` и переменной `y` (вершинами v_2 и v_3) соответственно. Кроме того, в функции `bar` есть две точки возврата, и соответствующие возвращаемые значения (вершины v_7 и v_8) связываются с переменной `z`.

2.3 Восходящий анализ

На третьем шаге метода выполняется упрощённый межпроцедурный анализ.

Функции рассматриваются «снизу вверх», поэтому данный анализ называется *восходящим*. А именно, метод упорядочивает все функции анализируемого кода в последовательность $f_1, f_2, \ldots, f_n$ так, что вызывающая функция всегда стоит раньше вызываемой. Затем каждая функция анализируется по очереди, начиная с последней. Не рассматриваются рекурсивные вызовы, поскольку они запрещены в целевой кодовой базе.

Для очередной функции f_i выполняется анализ, который состоит из следующих шагов:

(i) «протягивание» инструкций обращения к массиву вдоль арифметических рёбер;
(ii) поиск массивов;
(iii) определение имён массивов и длин;
(iv) построение решётки; и
(v) «подъём» информации по рёбрам возврата в функции, которые вызывают f_i.

Результатом анализа функции f_i является список из всех найденных массивов, а также оценки длины для каждого массива. Восходящий порядок гарантирует, что перед анализом функции f_i её сводка дополнена информацией, «поднятой» из всех вызываемых функций (которые стоят в последовательности правее f_i).

2.3.1 «Протягивание» инструкций обращения к массиву вдоль арифметических рёбер

Этот шаг является основным (и, фактически, единственным) случаем применения арифметических рёбер в методе и необходим для того, чтобы поддержать шаблоны кода, где длина массива не равна параметру функции f_i, а выражается линейной функцией (например, $l + 1$ или $3 * l/2 - 3$, где l – параметр f_i). «Протягивание» на основе имеющейся информации в сводке добавляет новые инструкции обращения к массиву во множество $ArrayUsages_{f_i}$, что позволяет найти больше массивов и правильно определить их длину. Шаг «протягивания» можно опустить, не нарушая метод: в таком случае метод не найдёт нужных

массивов для вышеуказанных шаблонов кода, однако массивы будут найдены (а их длина – правильно определена) для всех остальных требуемых шаблонов.

«Протягивание» выполняется так. Пусть в сводке функции f_i имеется инструкция обращения к массиву $u = \{v_{array}, index, v_{result}, s_u\} \in ArrayUsages_{f_i}$, где $index = p_u * v_1 + q_u$, а $p_u \neq 0$. Пусть также имеется арифметическое ребро $a = \{v_2, v_1, op, s_a\} \in ArithmeticEdges_{f_i}$, ведущее в вершину v_1. Пусть также известно, что инструкция s_a исполняется раньше, чем инструкция s_u. Если все эти условия выполнены, то метод «протягивает» u вдоль a: он добавляет в множество $ArrayUsages_{f_i}$ новую инструкцию обращения к массиву $u' = \{v_{array}, index', v_{result}, s_a\}$, где $index' = p_u * op(v_2) + q_u$. Фактически, метод создаёт альтернативное описание той же инструкции обращения к массиву. Если в инструкции u индекс выражен как линейная функция от вершины v_1, то в добавленной инструкции u' индекс выражен уже в терминах вершины v_2.

Процесс «протягивания» может продолжаться до достижения «неподвижной точки», то есть такого состояния, когда никакое дальнейшее «протягивание» не создаёт новый элемент множества $ArrayUsages_{f_i}$. В реальности данный процесс может оказаться долгим или даже не сойтись, поэтому в реализации процесс ограничивается во избежание длительного времени работы и большого потребления памяти.

Для проверки того, что инструкция s_1 исполняется раньше, чем инструкция s_2 (для произвольно взятых s_1 и s_2) рассматриваются соответствующие им базовые блоки b_1 и b_2 в графе потока управления CFG_{f_i}. Проверяется, что базовый блок b_1 *доминирует* базовый блок b_2, то есть любой путь от входного базового блока $EntryBB_{f_i}$ до базового блока b_2 проходит через блок b_1. Если инструкции расположены в одном блоке (то есть $b_1 = b_2$), то дополнительно проверяется, что они идут в правильном порядке, то есть $s_1 < s_2$. Проверка доминируемости осуществляется путём построения *дерева доминаторов* для графа потока управления (для этого использован алгоритм из [3]).

2.3.2 Поиск массивов в сводке функции

Наиболее сложным при анализе функции f_i является поиск массивов в её сводке. Результатом этого поиска является множество $Arrays_{f_i}$, элементы которого имеют вид $a = \{v_{array}, l\} \in Arrays_{f_i}$. Здесь вершина $v_{array} \in V_{f_i}$ отвечает массиву, а l представляет собой длину этого массива: $l = p * v_{length} + q$, где $v_{length} \in V_{f_i}$, а p и q – это рациональные коэффициенты.

Множество $Arrays_{f_i}$ строится следующим образом. Рассматривается каждая инструкция обращения к массиву $u = \{v_{array}, index, v_{result}, s_u\} \in ArrayUsages_{f_i}$, $index = p_u * v_{index} + q_u$. Если индекс является константой, то есть $p_u = 0$, то в $Arrays_{f_i}$ добавляется элемент $\{v_{array}, q_u + 1\}$, а поиск переходит к следующей инструкции обращения к массиву. Если $p_u > 0$ (случай $p_u < 0$ рассматривается симметрично), то метод ищет численное ограничение для вершины v_{index}, которое выполняется перед инструкцией s_u и имеет вид $c = \{v_{index}, rhs, <, b_c\} \in Constraints_{f_i}$, где $rhs = p_c * v_{rhs} + q_c$. Проверку выполнимости ограничения перед инструкцией мы обсудим ниже. Ограничения вида $\{v_{index}, rhs, \leq, b_c\}$ рассматриваются методом так же, как и ограничения $\{v_{index}, rhs + 1, <, b_c\}$. Если для инструкции обращения к массиву u метод нашёл подходящее ограничение c, то в $Arrays_{f_i}$ добавляется элемент $\{v_{array}, p_u * (p_c * v_{rhs} + q_c) + q_u\}$. В самом простом случае имеется инструкция обращения к массиву $u = \{v_{array}, v_{index}, v_{result}, s_u\}$ и выполняющееся перед инструкцией s_u ограничение $v_{index} < v_{length}$. Это означает, что в данной точке функции f_i выполнено обращение к массиву v_{array} по индексу, про который известно, что он меньше v_{length}. Таким образом массив v_{array} имеет длину v_{length}.

Для того, чтобы выяснить, должно ли ограничение c выполняется перед инструкцией s_u, выполняется проверка того, что в графе потока управления CFG_{f_i} базовый блок b_c

доминирует базовый блок b_s инструкции s_u. Данный способ является приближённым и не даёт гарантий того, что ограничение действительно выполняется в данной точке программы. Например, при исполнении базового блока b_s значение численной переменной v_{index} могло измениться до инструкции s_u. Тем не менее данная проверка позволяет исключить нахождения большого количества ложноположительных динамических массивов.

2.3.3 Определение имени массива

Найденные на предыдущем шаге массивы и их длины выражены в терминах вершин графа объектов функции f_i. Для того, чтобы сделать полученный результат пригодным для выдачи пользователю, следует вычислить *имена* соответствующих вершин. Каждое имя представляется в виде так называемого *пути*, который начинается с имени параметра функции f_i или глобальной переменной, и опционально продолжается несколькими возможными операциями: разыменованием указателя; взятием поля; доступом к элементу массива по константному индексу; доступом к элементу массива по неизвестному индексу; смещением адреса. Данные операции соответствуют возможным операциям на рёбрах доступа. Таким образом, метод строит отображение $Names_{f_i}$, которое сопоставляет вершинам графа объектов OG_{f_i} (возможно, не всем) их имена.

Для определения имён метод выполняет обход графа объектов. Обход начинается с вершин, соответствующих параметрам функции и глобальным переменным; то есть это вершины из множеств $Parameters_{f_i}$ и $GlobalVariables_{f_i}$. Для них в $Names_{f_i}$ добавляются их имена. Предположим, обход сейчас находится в вершине $v_1 \in V_{f_i}$, которой сопоставлено имя $name_1 = Names_{f_i}(v_1)$, а также имеется ребро доступа $e = \{v_1, v_2, op\} \in AccessEdges_{f_i}$, причём вершине v_2 пока не сопоставлено имя в отображении $Names_{f_i}$. Тогда метод добавляет имя для вершины v_2: $Names_{f_i}(v_2) := name_1 op$, после чего обход продолжается из вершины v_2. Обход завершается, когда всем достижимым вершинам сопоставлены имена. Возможно, что процесс обхода может найти несколько путей до одной и той же вершины, в нашем случае мы выбираем любой.

Имея отображение $Names_{f_i}$, метод собирает и экспортирует найденные массивы в требуемый формат. Рассматривается каждый массив $a = \{v_{array}, l\} \in Arrays_{f_i}$, $l = p * v_{length} + q$, и если для обеих вершин есть имена, то добавляется окончательный результат: массив $name_{array}$ в функции f_i, имеющий длину $p * name_{length} + q$ (где $name_{array} = Names_{f_i}(v_{array})$, и $name_{length} = Names_{f_i}(v_{length})$).

2.3.4 Построение решётки

До того, как завершить анализ функции f_i, метод должен «поднять» необходимую информацию «наверх», в сводки вызывающих функций, чтобы далее было возможно качественно проанализировать оставшиеся функции $f_1, f_2, \ldots, f_{i-1}$. Рассмотрим функцию g, вызывающую функцию f_i в точке вызова c – $\{g, c, f_i\} \in CGEdges$. Для осуществления «подъёма» информации в функцию g метод сначала добавляет ряд горизонтальных и вертикальных рёбер следующим образом. Предположим, в сводке функции f_i имеется горизонтальное ребро $h = \{v_1, v_2, op\} \in AccessEdges_{f_i}$, а также есть вертикальное ребро $r = \{v_1, c, v_1'\} \in RetEdges$. Тогда метод «поднимает» ребро h, то есть в сводку $Summary_g$ добавляется новое горизонтальное ребро $h' = \{v_1', v_2', op\}$, которое ведёт в новую вершину v_2'. Добавление ребра не происходит, если из вершины v_1' уже ведёт ребро доступа с операцией op. Затем метод добавляет новое вертикальное ребро $r' = \{v_2, c, v_2'\}$, тем самым формируя «ячейку» из двух горизонтальных и двух вертикальных рёбер. Этот процесс продолжается до тех пор, пока не останется горизонтального ребра, которое можно поднять. Набор сформированных «ячеек», которые состоят из горизонтальных рёбер в сводках

функций f_i и g и из вертикальных рёбер между этими двумя функциями, образует так называемую *решётку*. В свою очередь, вышеописанный шаг называется *построением решётки*.

Вместе с построением решётки метод «поднимает» используемые глобальные переменные из множества $GlobalVariables_{f_i}$. Пусть в функции f_i (или в вызванной функции) используется глобальная переменная G, которая представлена в виде $\{v, name\} \in GlobalVariables_{f_i}$. Предположим, что в функции g не используется переменная G. Тогда метод добавляет в граф объектов OG_g новую вершину v', которая соответствует переменной G: $\{v', name\} \in GlobalVariables_g$. После этого метод добавляет новое вертикальное ребро $r = \{v, c, v'\}$ во множество $RetEdges$.

Пример построения решётки приведён на рис. 4 (соответствующий пример кода см. в листинге 3). Для данного С-кода метод строит две сводки – сводка $Summary_{\text{foo}}$, состоящая из одной вершины v_1 (параметр p), и сводка $Summary_{\text{bar}}$ с тремя следующими вершинами: v_2 (параметр q), v_3 (выражение *q) и v_4 (выражение q->x). Также имеется одно ребро возврата $r = \{v_2, c, v_1\}$, где c – единственная точка вызова функции bar. При построении решётки в сводку $Summary_{\text{foo}}$ добавляются две новые вершины v_5, v_6, два горизонтальных ребра $\{v_1, v_5, [0]\}$, $\{v_5, v_6, .x\}$, а также два вертикальных ребра $\{v_3, c, v_5\}$, $\{v_4, c, v_6\}$. На рис. 4 добавленные вершины и рёбра изображены красным цветом.

```
struct Pt { int x; int y; };

void bar(struct Pt *q) {
  q->x = 0;
}
void foo(struct Pt *p) {
  bar(p);
}
```

Листинг 3. Пример С-кода для иллюстрации построения решётки.
Listing 3. C code snippet for illustrating lattice construction.

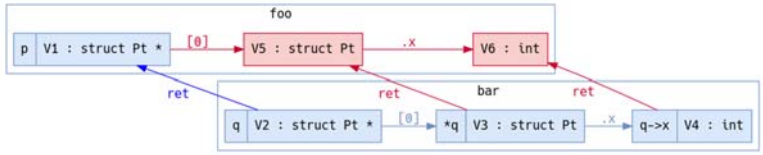

Рис. 4. Построение решётки.
Fig. 4. Lattice construction.

«Подъём» горизонтальных рёбер нужен для того, чтобы учесть информацию о том, какие объекты используются в вызываемых функциях и как с ними происходит работа.

2.3.5 «Подъём» информации по рёбрам возврата

Наконец, метод завершает анализ функции f_i «подъёмом» остальной информации в сводки всех функций, которые вызывают f_i. Пусть функция g вызывает функцию f_i в точке вызова c. Метод «поднимает» следующие сущности: арифметические рёбра $ArithmeticEdges_{f_i}$, инструкции обращения к массиву $ArrayUsages_{f_i}$ и массивы $Arrays_{f_i}$.

Для примера рассмотрим процесс «подъёма» в сводку $Summary_g$ инструкции обращения к массиву $u = \{v_{array}, i, v_{result}, s\} \in ArrayUsages_{f_i}$, $i = p * v_{index} + q$. Метод рассматривает три ребра возврата: $\{v_{array}, c, v_{array}'\}$, $\{v_{length}, c, v_{length}'\}$ (если $p = 0$, то это ребро не

рассматривается) и $\{v_{result}, c, v_{result}'\}$. Если какое-то из этих рёбер отсутствует, инструкция u не «поднимается». Иначе в сводку $Summary_g$ добавляется новая инструкция обращения к массиву $u' = \{v_{array}', i', v_{result}', c\}$, где $i' = p * v_{index}' + q$, а c – это номер инструкции вызова в нумерации $Insn_g$.

Остальная информация «поднимается» аналогично: для каждой вершины, входящей в состав сущности (арифметического ребра/массива) рассматривается ребро возврата, помеченное точкой вызова c, и каждая вершина заменяется на конец своего ребра. Номер инструкции, если он есть, заменяется на номер c. Таким образом, все инструкции в функции f_i «схлопываются» в одну инструкцию c в функции g.

«Подняв» информацию во все вызывающие функции, за ненадобностью метод удаляет информацию о текущей функции f_i для уменьшения потребления памяти, после чего метод переходит к анализу следующей функции f_{i-1}. Восходящий анализ продолжается до тех пор, пока не будут проанализированы все функции.

3. Программная реализация метода (инструмент)

Инструмент был реализован на основе компилятора Clang [4] (часть инфраструктуры LLVM). Это популярный компилятор для языков семейства С с открытым исходным кодом реализован на C++ и разрабатывается open source сообществом более 20 лет. Clang обладает качественной документацией, хорошо проработанной архитектурой и высокой производительностью. С его помощью было создано множество инструментов разного назначения: линтеры (clang-tidy), форматтеры (clang-format), всевозможные сервисы для IDE, например, языковой сервер clangd, а также статические анализаторы общего назначения (Clang Static Analyzer [5]). Лицензия Clang позволяет на его основе создавать проприетарное ПО (в отличие, например, от GCC), что является одним из требований к данному решению. По этим причинам Clang был выбран для данной задачи. Для хранения результатов анализа была выбрана СУБД SQLite [6], так как она легко интегрируется в виде С-библиотеки и не требует развёртывания сервера базы данных. Таким образом, результаты анализа хранятся в одном файле, с которым в дальнейшем работает фаззер.

Архитектура инструмента изображена на рис. 5.

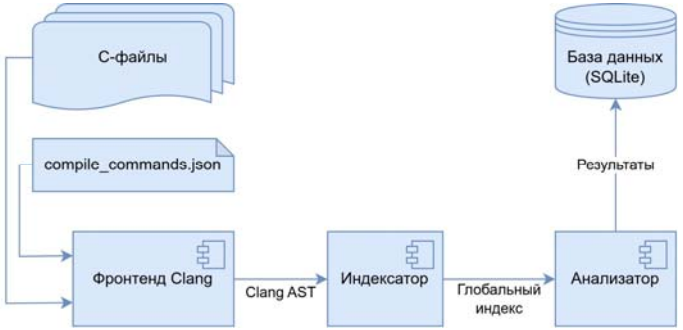

Рис. 5. Архитектура инструмента распознавания массивов.
Fig. 5. Architecture of the array recognition tool.

На вход инструмент принимает файл compile_commands.json [7], в котором перечислены все входные С-файлы, а также опции компиляции. На выход инструмент записывает результат анализа в базу данных SQLite, которая в итоге для каждой функции содержит информацию о выявленных в ней массивах (если таковые имеются) и их длинах.

В начале работы инструмент осуществляет *индексирование* каждого С-файла. Индексирование начинается с построения дерева абстрактного синтаксиса (Clang AST [8]), которое строится в фронтенде Clang. С помощью рекурсивного обхода этого дерева наш инструмент строит *индекс* данного С-файла – его минималистичное представление, содержащее информацию об определённых в этом файле символах (функциях, типах, глобальных переменных и других языковых конструктах) и информацию, необходимую для метода, то есть сводки всех функций в данном файле.

После индексирования всех С-файлов построенные индексы объединяются в *глобальный индекс* программы. Имея множество сводок всех функций, инструмент связывает их и выполняет восходящий анализ в соответствии с предложенным методом. Завершив анализ очередной функции, инструмент записывает информацию о найденных массивах в базу данных, и после этого восходящий анализ переходит к рассмотрению следующей функции.

4. Экспериментальное исследование

Для оценки эффективности инструмента были поставлены эксперименты, которые исследовали:

(i) производительность инструмента;

(ii) эффективность инструмента применительно к задачам фаззинга; и

(iii) точность метода поиска массивов и аппроксимации длин.

Эксперимент по производительности проводился на 8 модулях целевой кодовой базы (1200 файлов, что соответствует 860 тыс. строк кода[1]). Использовалось типовое облачное окружение с операционной системой EulerOS (Linux) на 8 ядрах (16 потоков), 32 ГБ оперативной памяти. Время работы оказалась приемлемым и составило 4,5 мин. при потреблении памяти 4,38 ГБ.

Во втором эксперименте была исследована эффективность фаззинга до и после внедрения нашего инструмента в экосистему тестирования кодовой базы. Оказалось, что покрытие кодовой базы фаззером при внедрении инструмента увеличилось на 10%, а количество найденных фаззером ошибок за 3 месяца эксплуатации инструмента увеличилось на 40% по сравнению с предыдущим аналогичным периодом до внедрения.

Последний эксперимент имел цель численно оценить точность нашего метода поиска массивов и проводился на открытом телекоммуникационном проекте DMM [9]. Проект DMM был выбран как пример открытого сетевого С-фреймворка, похожего по стилю на целевую кодовую базу. Был рассмотрен фрагмент проекта (119 функций, 18 тыс. строк кода), на котором был достигнуты значения точности (precision) 79% и полноты (recall) 98%. При этом длина массива была определена правильно в 69% случаях.

Ложноположительные результаты метод произвёл в следующих случаях: 1) когда подаваемый на вход указатель не использовался, а был перезаписан функцией; 2) когда один объект был распознан как массив из одного элемента; 3) когда С-структура была инициализирована функцией memset и метод распознал её как массив с типом элемента char. Наш метод можно улучшить, чтобы данные случаи обрабатывались правильно. Например, для первого случая можно представить исходное и новое значение указателя разными вершинами в графе объектов, используя SSA. Ложноотрицательных результатов оказалось немного, и большинство из них можно решить, поддержав дополнительные стандартные функции и системные вызовы, такие как send и recv.

Отметим, что применительно к задачам фаззинга метрика полноты нашего метода важнее, чем точность, поскольку при ложноположительном результате в контексте функции будет создан лишний массивов (он не повлияет на фаззинг), а при ложноотрицательном нужный

[1] Для сравнения: исходный код СУБД SQLite составляет 100 тыс. строк.

массив не будет создан, и фаззинг может аварийно завершиться, не покрыв значительную часть функции. Результаты третьего эксперимента доступны на GitHub [10].

5. Обзор существующих подходов

Существует ряд подходов к анализу динамической памяти в языках программирования. Можно упомянуть про классические алгоритмы анализа потока управления, анализа потока данных и анализа указателей [2]. Также имеются подходы в рамках символьного исполнения программ, моделирующие динамическую память при статическом исполнении программ [11]. Однако задача управления динамической памятью в языке С является алгоритмически неразрешимой, поскольку по нетипизированному указателю в общем случае невозможно понять, куда именно он указывает, а из-за адресной арифметики, в свою очередь, трудно отследить происхождение фрагмента памяти, на которую он указывает. В частности, это влечёт неразрешимость задачи автоматической сборки мусора для языков С/С++ [12]. Тем не менее, все эти подходы, решая так или иначе задачу идентификации массивов (анализ указателей), оставляют в стороне задачу аппроксимации длины динамических массивов.

Важной задачей в статическом анализе С-программ является детектирование переполнений буфера. Существует ряд анализаторов, реализующих автоматические проверки такого рода.

Clang Static Analyzer [5] – это инструмент статического анализа с открытым исходным кодом, который разрабатывается сообществом компилятора Clang [4] и предназначается для поиска ошибок в С/С++-программах. Он работает на уровне AST и использует метод символьного исполнения. Один из более сотни реализованных в инструменте детекторов находит потенциальный выход за границу массива. Данный детектор поддерживает массивы, длина которых известна статически, а наш метод также может выявлять массивы, длина которых хранится в переменной.

Коммерческий статический анализатор Svace [13-14] (создан в ИСП РАН) предназначен для внедрения процесса безопасной разработки в компаниях. Целью инструмента является предоставление универсальной единой инфраструктуры анализа, которая: обеспечивает масштабируемость на крупные приложения; поддерживает множество языков программирования (не только язык С), компиляторов, платформ; предоставляет единый пользовательский интерфейс и так далее. На базе межпроцедурного чувствительного к путям исполнения анализа в Svace реализованы качественные детекторы переполнения буфера, которые в том числе поддерживают динамически выделенные буферы с символьной длиной. В то время, как подход и реализованный движок анализа выглядит перспективным для решения задачи автоматического определения длин массивов, детекторы Svace неотделимы от всей крупной инфраструктуры статического анализа. По этой причине существующие фрагменты нецелесообразно повторно использовать и интегрировать в итоговое решение для фаззинга С-программ в телекоммуникациях.

Статический анализатор cooddy [15] разрабатывался в компании Huawei для внутреннего применения и имел детектор выхода за границу массива. Более того, он поддерживал пользовательские аннотации для функций, в особенности для тех, которые не определены внутри анализируемого проекта. Одной из аннотаций является аннотация буфера и его длины, с её помощью cooddy может искать больше случаев выхода за границу массива. Наш метод позволяет осуществить автоматическую генерацию аннотаций «буфер-длина», что потенциально может улучшить точность анализатора.

KLEEF [16] – это инструмент с открытым исходным кодом для символьного исполнения С/С++-программ, разрабатываемый R&D Toolchain Labs (исследовательская лаборатория им. П.Л. Чебышева). KLEEF выполняет две прикладные задачи: автоматическую генерацию качественного тестового покрытия и автоматическую верификацию трасс ошибок (найденных статическими анализаторами) для фильтрации ложных срабатываний. Важной составляющей любого движка символьного исполнения является *символьная память*, которая сопоставляет символам конкретные или символьные значения. Одной из

особенностей KLEEF, которая отсутствовала в оригинальном проекте KLEE [17], является поддержка в символьной памяти сложных структур данных: связных списков, деревьев, и, в особенности, динамически созданных массивов (массивов с символьной длиной). Тем самым, инструмент KLEEF вполне применим для качественного автоматического распознавания динамических массивов с аппроксимацией длины. В сравнении с нашим методом KLEEF имеет две технические тонкости, которые усложняют его интеграцию в экосистему тестирования телекоммуникационного ПО. Во-первых, символьное исполнение требует больше вычислительных ресурсов, ведь для каждой функции в анализируемом С-коде необходимо пройти множество различных путей исполнения. Это создаёт риск неприемлемого времени работы для большой кодовой базы. Наш подход более простой и более легковесный. Во-вторых, движок KLEEF работает на уровне промежуточного представления LLVM [18], тем самым, исходный С-код необходимо прежде скомпилировать в это представление. Это создаёт дополнительный нетривиальный шаг в процедуре управления качеством нашей кодовой базы. Для нашего инструмента достаточно иметь исходный код и файл compile_commands.json.

7. Заключение

В данной работе представлен специальный метод статического анализа для автоматического распознавания массивов, который состоит из:

1) построения сводок С-функций, состоящих из графа объектов, графа потока управления и других структур данных;
2) создания графа вызовов для всей С-программы;
3) упрощённого межпроцедурного анализа.

Для тестирования кодовой базы был разработан инструмент на основе Clang, реализующий данный метод. Инструмент был интегрирован в систему фаззинга компании, что позволило увеличить метрику покрытия кодовой базы на 10%, а количество найденных ошибок – на 40%. На релевантном открытом проекте инструмент достиг точности 79% и полноты 98% в распознавании массивов, а длина была правильно определена в 69% случаях.

Приоритетным направлением дальнейшего развития инструмента представляется правильное распознавание массивов для более сложных шаблонов кода, встречающихся в целевой кодовой базе. Так, имеется запрос на поддержку следующих шаблонов:

1) цикл, смещающий указатель на элемент массива;
2) массив с длиной, равной произведению двух параметров функции;
3) сообщение в формате type-length-value (TLV) [19].

Другим возможным улучшением является устранение ложноположительных результатов путём использования промежуточного представления SSA, в котором разные значения одного указателя будут представлены разными вершинами в графе объектов.

Список литературы / References

[1]. Кознов Д.В. Метод предметно-ориентированного анализа программного кода. Научно-технический вестник информационных технологий, механики и оптики. Принята к публикации. 2026.
[2]. Muchnick S.S. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997, 856 p.
[3]. Lengauer T., Tarjan R. E. A Fast Algorithm for Finding Dominators in a Flowgraph. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 1, no. 1, 1979, pp. 121-141. DOI: 10.1145/357062.357071.
[4]. Clang: A C language family frontend for LLVM. Available at: https://clang.llvm.org/, accessed 22.09.2025.

[5]. Clang Static Analyzer. Available at: https://clang.llvm.org/docs/ClangStaticAnalyzer.html, accessed 22.09.2025.
[6]. SQLite Home Page. Available at: https://sqlite.org/, accessed 22.09.2025.
[7]. JSON Compilation Database Format Specification – Clang documentation. Available at: https://clang.llvm.org/docs/JSONCompilationDatabase.html, accessed 22.09.2025.
[8]. Introduction to the Clang AST. Available at: https://clang.llvm.org/docs/IntroductionToTheClangAST.html, accessed 22.09.2025.
[9]. DMM (Dual Mode, Multi-protocol, Multi-instance). Available at: https://github.com/Huawei/DMM, accessed 22.09.2025.
[10]. Array Tool Evaluation, 2025. Available at: https://github.com/usachev63/arraytool-results, accessed 19.10.2025.
[11]. Морозов И.А., Мисонижник А.В., Мордвинов Д.А., Кознов Д.В., Иванов Д.А. Симкретная модель памяти с ленивой инициализацией и объектами символьного размера в символьной виртуальной машине KLEE. Труды ИСП РАН, 2023, том 35, вып. 3, стр. 91-108 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–7.
[12]. Berezun D., Boulytchev D. Precise Garbage Collection for C++ with a Non-Cooperative Compiler. Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia, 2014, no. 15, pp. 1-8. DOI: 10.1145/2687233.2687244.
[13]. Ivannikov, V.P., Belevantsev, A.A., Borodin, A.E. et al. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, 2014, vol. 40, pp. 265–275 (2014). DOI: 10.1134/S0361768814050041.
[14]. A. Belevantsev et al. "Design and Development of Svace Static Analyzers", 2018 Ivannikov Memorial Workshop (IVMEM), Yerevan, Armenia, 2018, pp. 3-9, DOI: 10.1109/IVMEM.2018.00008.
[15]. Герасимов А.Ю., Канахин А.А., Привалов П.А., Жуков А.А., Каминский Е.А. Применение статического анализа исходного кода для поиска проблем с производительностью: примеры из практики. Труды ИСП РАН, 2022, том 34, вып. 4, стр. 7-20. DOI: 10.15514/ISPRAS-2022-34(4)-1.
[16]. Misonizhnik A., Morozov S., Kostyukov Y., Kalugin V., Babushkin A., Mordvinov D., Ivanov D. KLEEF: Symbolic Execution Engine (Competition Contribution). Fundamental Approaches to Software Engineering: 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, pp. 314-319. DOI: 10.1007/978-3-031-57259-3_18.
[17]. Cadar C., Dunbar D., Engler D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08), 2008, pp. 209-224.
[18]. LLVM Language Reference Manual. Available at: https://llvm.org/docs/LangRef.html, accessed 18.10.2025.
[19]. Devopedia. TLV Format. Version 2, February 18, 2023. Available at: https://devopedia.org/tlv-format, accessed 04.11.2025.

Информация об авторах / Information about authors

Дмитрий Владимирович КОЗНОВ – доктор технических наук, профессор кафедры системного программирования Санкт-Петербургского государственного университета, Сфера научных интересов: программная инженерия, модельно-ориентированная разработка программного обеспечения, программные данные, машинное обучение

Dmitry Vladimirovich KOZNOV – Dr. Sci. (Tech.), Assoc. Prof., Professor St. Petersburg State University (SPbSU). Research interests: software engineering, model-driven software development, program data, machine learning.

Данила Александрович УСАЧЕВ – бакалавр Санкт-Петербургского государственного университета, С++-разработчик. Сфера научных интересов: статический анализ кода, фаззинг.

Danila Aleksandrovich USACHEV – bachelor of St. Petersburg State University (SPbSU), C++ developer. Research interests: static analysis, fuzzing.