Алгоритмы анализа указателей для обнаружения дефектов в исходном коде программ*

В. М. Ицыксон
vlad@ftk.spbstu.ru

М. Ю. Моисеев
mm@aivt.ftk.spbstu.ru

М. Х. Ахин
akhin@aivt.ftk.spbstu.ru

А. В. Захаров
zakharov@aivt.ftk.spbstu.ru

В. А. Цесько
tsesko@aivt.ftk.spbstu.ru

Санкт-Петербургский государственный
политехнический университет

В последнее время все более актуальной становится задача повышения надежности программного обеспечения. Одним из способов решения этой задачи является автоматизация процесса обнаружения дефектов в исходном коде программ. В данной статье предлагаются алгоритмы статического анализа, обеспечивающие обнаружение ошибок в работе с указателями в программах на языке C. Основой этих алгоритмов является предложенная модель отношения «указывает-на», которая позволяет описать взаимосвязи между объектами реальной программы. Показано преимущество разработанных алгоритмов над алгоритмами, которые используются в существующих средствах обнаружения дефектов.

* Исследование выполнено в рамках работ по государственному контракту № 02.514.11.4081 «Исследование и разработка системы автоматического обнаружения дефектов в исходном коде программного обеспечения» Федерального агентства по науке и инновациям.

Введение

В настоящее время технические системы играют ключевую роль во многих сферах человеческой деятельности. Существует большой класс систем с повышенными требованиями к надежности и безопасности: бортовые управляющие системы, энергетические системы, системы военного назначения, банковские системы и др. Ошибки в подобных системах могут приводить к серьезным последствиям: причинению вреда здоровью людей, нанесению ущерба окружающей среде, крупным финансовым потерям.

Большинство современных технических систем имеют в своем составе программные подсистемы. Общая надежность таких систем определяется надежностью аппаратуры и программного обеспечения (ПО).

Основными причинами, снижающими надежность и безопасность программных систем (ПС), являются ошибки, допущенные разработчиками на различных стадиях жизненного цикла разработки программного обеспечения. Большинство программных ошибок вносится на этапе кодирования — будем называть такие ошибки программными дефектами. Обнаружение дефектов является трудоемкой задачей и может занимать существенную часть времени разработки ПС, поэтому автоматизация процесса обнаружения дефектов является актуальной задачей.

В этой статье рассматриваются вопросы автоматического обнаружения дефектов в приложениях на языке C на основе статического анализа исходного кода[1]. Выбор данного языка обусловлен тем, что он широко используется при разработке прикладного и системного ПО, а также ПО для встроенных систем. Другими причинами выбора языка C являются отсутствие автоматического управления ресурсами и контроля инициализации переменных, наличие «опасных» конструкций, таких как адресная арифметика, вызов функций через указатели, а также неконтролируемое приведение типов.

Для определения наиболее распространенных и «опасных» типов дефектов обратимся к существующим исследованиям в этой области. В сфере инженерии надежного и безопасного ПО авторитетным источником является организация CERT (Computer Emergency Response Team) при институте программной инженерии университета Карнеги-Меллон[2]. Согласно данным этой организации,

[1] http://www.open-std.org/jtc1/sc22/wg14/www/standards
[2] https://www.securecoding.cert.org/

а также результатам независимых исследований[3], наиболее серьезными типами дефектов в программах на языке C, с большой вероятностью приводящими к уязвимостям безопасности, являются ошибки отсутствия инициализации, управления памятью и выход за границы объекта. Данная статья посвящена решению задачи автоматического обнаружения перечисленных типов дефектов.

1. Состояние предметной области

Исследования в области обнаружения программных дефектов с использованием методов статического анализа ведутся более 15 лет. Имеются как фундаментальные теоретические работы в этой области, так и различные программные средства для анализа исходного кода. Рассмотрим наиболее существенные из них.

1.1. Анализ указателей и обнаружение дефектов

Для обнаружения перечисленных типов дефектов в программах на языке C необходимо проводить анализ указателей. Классические алгоритмы анализа указателей представлены в работах Андерсена [3] и Стеенсгарда [11]. Оба алгоритма основаны на выводе типов и решении систем неравенств. Недостатками алгоритма Андерсена являются потоконечувствительность и ограниченная контекстная чувствительность анализа. К недостаткам алгоритма Стеенсгарда относятся потоко- и контекстно-нечувствительность, а также отсутствие анализа сложных типов данных.

Еще один подход представлен в работе Шварцбаха [10], в которой рассматривается статический анализ программ с использованием аппарата теории решеток. В данной работе, в частности, демонстрируется возможность применения теории решеток для решения задач анализа указателей. Ограничениями представленного подхода являются использование искусственного языка TIP и недостаточная глубина проработки алгоритмов.

Помимо работ, посвященных анализу указателей, существует множество публикаций, рассматривающих решение задачи обнаружения дефектов в программах на языке C. В работе Хейне и Лэм [7] рассматривается поиск дефектов утечки и повторного освобождения памяти. Используется модель единственного владения (в каждый момент времени объектом владеет только один объект).

[3] http://scan.coverity.com/report/

Показана возможность успешного поиска дефектов указанных типов. Существенным недостатком данной работы является отсутствие анализа циклов и учета влияния входных данных программы.

Работа Лярошеля и Эванса [8] посвящена обнаружению ошибок выхода за границу массива. Предложенный в работе подход основан на использовании аннотаций кода и библиотечных функций. Аннотации оформляются в виде специальных комментариев (указываются размеры объекта, допустимость `NULL` и т. д.). Аннотации и соответствующие им ограничения связываются с объектами программы, после этого используются алгоритмы вывода типов. К недостаткам подхода относятся отсутствие полноты анализа и использование предположений о линейном характере итераторов в циклах.

В ряде работ авторы идут на некоторые допущения относительно использования указателей в анализируемых программах. В работе Эвотса, Далтона и Лившица [4] введение допущений о структуре и способах использования указателей в программе на языке C (Practical C Pointers) позволяет значительно сократить сложность алгоритмов. Анализ является контекстно-чувствительным, что достигается за счет клонирования контекстов в местах вызовов функций с последующим применением контекстно-нечувствительных алгоритмов. Существенным недостатком алгоритма является отсутствие полноты.

В результате проведенного анализа рассмотренных выше, а также ряда других работ были выделены основные недостатки существующих алгоритмов:

- отсутствие полных и точных потоко- и контекстно-чувствительных алгоритмов анализа указателей;
- использование упрощенных моделей программы;
- обнаружение лишь ограниченного подмножества ошибок работы с указателями.

Указанные недостатки не позволяют непосредственно использовать результаты рассмотренных работ для решения поставленных задач.

1.2. Программные средства обнаружения дефектов

Развитие индустрии разработки ПО привело к появлению большого числа программных средств статического анализа и обнаружения дефектов. В настоящее время на рынке имеется несколько

десятков анализаторов, часть из которых являются коммерческими продуктами, часть — свободно-распространяемым ПО. К наиболее известным относятся: IBM Rational Software Analyzer, Fortify Source Code Analysis, Splint, Klocwork Insight, Coverity, Frama-C, FlexLint. Кроме программных продуктов существуют также системы (frameworks) (SUIF, CIL, IRE, Mygcc и др.), позволяющие на основе предоставляемых парсеров и библиотек анализа конструировать специфические статические анализаторы, реализующие необходимую пользователю функциональность. Проведенный анализ показал, что наряду с достоинствами продуктов (интеграция с инструментами разработки, стабильность работы) и систем (гибкость, расширяемость) большинство средств обладает следующими существенными недостатками [2]:

- неполная поддержка стандартов языка C;
- отсутствие поддержки расширений языка C (в том числе расширений GNU);
- обнаружение ограниченного набора типов дефектов;
- упрощенный анализ сложных типов данных;
- большое число ложных обнаружений;
- существенное число необнаруживаемых дефектов;
- слабая расширяемость архитектуры.

1.3. Вывод

Обзор публикаций, посвященных анализу указателей, показал отсутствие полных и точных алгоритмов обнаружения дефектов. Исследование возможностей существующих программных средств показало их недостаточную эффективность. Таким образом, задача разработки новых качественных алгоритмов анализа указателей и обнаружения дефектов является актуальной.

2. Алгоритм анализа указателей

Разрабатываемый подход предусматривает обнаружение дефектов в два этапа. На первом этапе выполняются различные виды статического анализа: анализ указателей, интервальный анализ [1] и др. На втором этапе информация, полученная при помощи статического анализа, используется алгоритмами обнаружения дефектов. Этот подход используется в инструментальных средствах, раз-

рабатываемых на кафедре автоматики и вычислительной техники СПбГПУ.

2.1. Постановка задачи

Сформулируем требования к возможностям разрабатываемого алгоритма, учитывая особенности выбранных типов дефектов и принимая во внимание результаты анализа существующих алгоритмов:

- анализ больших программных проектов;
- анализ всех возможных значений указателей для каждой конструкции программы;
- анализ указателей на функции;
- контекстно-чувствительный межпроцедурный анализ с учетом рекурсий;
- анализ сложных объектов с точностью до каждого элемента.

Анализ больших программных проектов является ресурсоемкой задачей. Известно, что для программных систем, размер которых составляет сотни тысяч строк кода, время анализа даже на высокопроизводительных вычислительных системах может составлять до нескольких недель [5]. Исходя из этого, разрабатываемые алгоритмы должны обеспечивать одновременное обнаружение всех рассматриваемых дефектов, а также обладать устойчивостью к влиянию дефектов на анализ оставшейся части программы.

2.2. Представление объектов программы

Алгоритм анализа указателей извлекает информацию об отношении «указывает-на» в анализируемой программе. Каждый элемент программы может быть представлен парой (o, i), где o — объект, i — смещение внутри этого объекта. Отношение «указывает-на» связывает два элемента программы и представляется в виде кортежа $((o_{j_1}, k_1), (o_{j_2}, k_2))$, где (o_{j_1}, k_1) указывает на (o_{j_2}, k_2) (см. рис. 1).

Такое отношение отражает модель работы с памятью в реальной программе на языке C и позволяет выразить прямую и косвенную адресацию, а также представить объекты сложных типов.

Каждый объект обладает следующими атрибутами: тип, размер и владелец. Объект o может иметь один из следующих типов:

- динамический тип ($T^{dynamic}$) — такой объект представляет собой область динамической памяти, выделенной функцией $malloc()$ или другими функциями языка C;

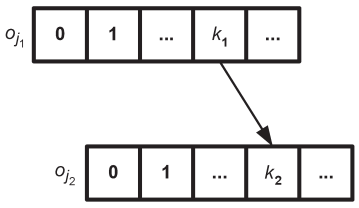

Рис. 1. Модель отношения «указывает-на».

- статический тип (T^{static}) — к таким объектам относятся глобальные и локальные переменные;
- функция (T^{func}) — введение такого типа позволяет анализировать указатели на функции и отличать их от указателей на объекты.

Дополнительно определены специальные объекты o^{null}, $o^{invalid}$ и $o^{universum}$.

Объект o^{null} представляет собой некорректный объект с нулевым адресом. На o^{null} указывают указатели, которые имеют значение $NULL$.

Объект $o^{invalid}$ представляет множество объектов с некорректными адресами — адресами, обращение по которым является ошибкой. На $o^{invalid}$ указывают неинициализированные указатели и указатели на объект, освобожденный функцией $free()$.

Допустимым смещением внутри объектов o^{null} и $o^{invalid}$ является только смещение 0. Эти объекты не могут стоять в левой части кортежа.

Объект $o^{universum}$ представляет все множество объектов с любыми адресами, в том числе допустимыми (явно созданные статические и динамические объекты), некорректными (объекты $o^{invalid}$) и $NULL$ (объект o^{null}). Объект $o^{universum}$ используется в случае анализа конструкции, результат выполнения которой не определён. Примерами таких конструкций являются разыменование неинициализированного указателя, прибавление к указателю неопределённого смещения. Объект $o^{universum}$ не может стоять в левой части кортежа — во всех таких конструкциях он заменяется на множество объектов, видимых в текущем состоянии.

Размер объекта o определяется при создании объекта (функция $sizeof(o)$) и не изменяется в дальнейшем (размер объекта измеряется в словах). Размер объекта, имеющего тип T^{func}, считается равным 1.

Владелец объекта o — $ownerof(o)$ — связывает статические объекты между собой. При выходе статического объекта из области видимости уничтожаются все объекты, владельцем которых он являлся.

Конструктор объекта $\langle type, size, owner\rangle$ создает новый объект необходимого типа $type$, размера $size$ и устанавливает владельца $owner$. В случае, когда объект не имеет владельца, он создается конструктором $\langle type, size\rangle$.

Смещение внутри объекта i используется для определения конкретного элемента объекта сложного типа, на который указывает переменная-указатель, что позволяет анализировать массивы указателей и другие сложные типы. В случае, если переменная-указатель указывает на первый элемент массива, смещение равно нулю. Смещение рассматривается совместно с объектом для повышения точности анализа.

Задачей алгоритма анализа указателей является извлечение информации о возможных отношениях «указывает-на» для всех конструкций анализируемой программы. Состояние программы в l-й конструкции — $\mathbb{S}_l$ — представляет собой множество кортежей, описывающих отношение «указывает-на» для всех объектов программы, видимых в данной конструкции.

2.3. Предлагаемый подход

Исходными данными для алгоритма анализа указателей является модель исходного кода на основе SSA (Static Single Assignment form) [6], свойства которой приведены ниже:

- в местах входа в области видимости переменных добавляются специальные конструкции *declare* для каждой переменной;
- в местах выхода из областей видимости переменных добавляются специальные конструкции *undeclare* для каждой переменной;
- для каждого присваивания значения скалярной переменной используются уникальные версии переменной;
- в местах слияния потоков управления добавляются специальные конструкции — ϕ-функции;
- операторы цикла представляются с помощью оператора ветвления и безусловного перехода;
- сложные выражения разбиваются на более простые с использованием временных переменных, все выражения представлены в трехоперандной форме вида $a = b + c$.

Основное отличие используемой модели от классической модели SSA заключается в семантике ϕ-функций: ϕ-функции добавляются *всегда* при слиянии потоков управления вне зависимости от наличия разных версий переменных в объединяемых потоках управления.

В качестве основы алгоритма анализа указателей предлагается использовать подходы, базирующиеся на теории решеток [9, 10]. При этом оказывается необходимым определить операторы языка C и функции, интерпретируемые алгоритмом анализа. В анализируемое множество включаются все операторы и функции, влияющие на отношение «указывает-на».

Для всех интерпретируемых конструкций разработаны правила построения уравнений. Они описывают преобразование входного множества кортежей $\widehat{\mathbb{S}}$ в выходное множество $\mathbb{S}$.

Правила применяются к конструкциям анализируемой программы — в результате строится система уравнений. Неизвестными в этих уравнениях являются множества кортежей для конструкций программы. В общем случае уравнение для l-й конструкции можно представить в следующем виде: $\mathbb{S}_l = \mathcal{F}(\mathbb{S}_1, \mathbb{S}_2, \ldots, \mathbb{S}_N)$, где N — количество конструкций в программе.

Рассмотрим вид уравнений для различных конструкций, используемых в программе. В ϕ-функциях производится объединение состояний программы, полученных на всех входящих путях. Соответствующее уравнение выглядит следующим образом:

$$\mathbb{S} = \bigcup_{i \in \mathbb{E}} \mathbb{S}_i,$$

где $\mathbb{E}$ — множество конструкций, из которых есть прямой переход в данную ϕ-функцию.

Для остальных интерпретируемых операторов и функций уравнения имеют вид

$$\mathbb{S} = \widehat{\mathbb{S}} \setminus kill[statement] \cup gen[statement].$$

Функции $kill[statement]$ и $gen[statement]$ рассматриваются ниже. Для неинтерпретируемых конструкций уравнения имеют вид $\mathbb{S} = \widehat{\mathbb{S}}$.

Предлагаемый алгоритм анализа указателей является полным — полученное решение всегда включает в себя правильное решение, — но не является точным: полученное решение может включать лишние кортежи. Решение системы уравнений на решетке на-

чинается с пустого множества и расширяется до достижения наименьшей неподвижной точки (Least Fixed Point). Система уравнений решается с использованием известных алгоритмов теории решеток [9].

2.4. Правила для основных конструкций

Сформулируем правила для интерпретируемых конструкций программы. В целях упрощения частные случаи с объектами o^{null} и $o^{universum}$ не рассматриваются.

Объявление в программе переменной-указателя приводит к добавлению в текущее состояние программы кортежа для нового объекта-указателя и объекта $o^{invalid}$, при этом удаление кортежей из предыдущего состояния не производится:

$$gen[declare(p)] : ((p = \langle T^{static}, 1\rangle, 0), (o^{invalid}, 0));$$
$$kill[declare(p)] : \emptyset.$$

Объявление статического массива приводит к добавлению в текущее состояние программы кортежа, для которого создаются два новых объекта, — объект длины 1, который хранит адрес массива, и объект для элементов массива:

$$gen[declare(p[size])] :$$
$$((p = \langle T^{static}, 1\rangle, 0), (o_k = \langle T^{static}, size, p\rangle, 0)) \cup \quad (2.1)$$
$$\cup \bigcup_{i=0}^{size-1} ((o_k, i), (o^{invalid}, 0));$$
$$gen[declare(p[size])] : \quad (2.2)$$
$$((p = \langle T^{static}, 1\rangle, 0), (o_k = \langle T^{static}, size, p\rangle, 0))$$
$$kill[declare(p[size)] : \emptyset.$$

Правило (2.1) выполняется, если массив p объявлен как массив указателей, правило (2.2) — в остальных случаях. Необходимо отметить, что для массива длиной 1 будут созданы два разных объекта, в то время как при объявлении переменной, не являющейся массивом, — только один объект.

Выход локальной переменной из области видимости приводит к удалению всех кортежей с этой переменной из текущего состояния программы. Для всех указателей, указывающих на статиче-

ский объект, которым владела эта переменная, добавляется кортеж с объектом $o^{invalid}$:

$$gen[undeclare(p)]: \bigcup_{\substack{\forall i,j,k:((p,0),(o_j,0)),\\((q,i),(o_j,k))\in\widehat{\mathbb{S}}\ \wedge\\ \wedge\ (ownerof(o_j)=p)\ \wedge\ (q\neq p)}} ((q,i),(o^{invalid},0));$$

$$kill[undeclare(p)]: \bigcup_{\substack{\forall i,j,k:((p,0),(o_j,0)),\\((q,i),(o_j,k))\in\widehat{\mathbb{S}}\ \wedge\\ \wedge\ (ownerof(o_j)=p)}} ((q,i),(o_j,k)); \tag{2.3}$$

$$kill[undeclare(p)]: \bigcup_{\substack{\forall j,k:((p,0),(o_j,k))\in\widehat{\mathbb{S}}\ \wedge\\ \wedge\ (ownerof(o_j)\neq p)}} ((p,0),(o_j,k)). \tag{2.4}$$

Правило (2.3) действует для объекта, владельцем которого является p, правило (2.4) — для всех остальных объектов.

Действие этих правил проиллюстрировано на рис. 2.

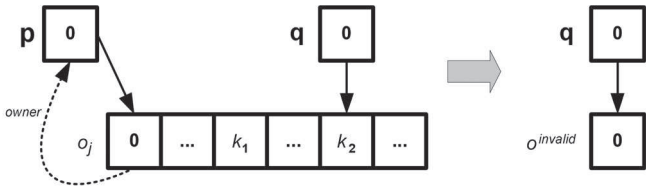

Рис. 2. Правила для функции *undeclare*().

Выделение динамической памяти приводит к созданию кортежа с новым объектом и выполняется по следующим правилам:

$$gen[p = malloc(size)]: \left((p,0),(o_k = \langle T^{dynamic}, \frac{size}{sizeof(*p)}\rangle,0)\right) \cup \bigcup_{i=0}^{\frac{size}{sizeof(*p)}-1} ((o_k,i),(o^{invalid},0)); \tag{2.5}$$

$$gen[p = malloc(size)]: \left((p,0),(o_k = \langle T^{dynamic}, \frac{size}{sizeof(*p)}\rangle,0)\right); \tag{2.6}$$

$$kill[p = malloc(size)] : \bigcup_{\forall j,k:((p,0),(o_j,k))\in\widehat{\mathbb{S}}} ((p,0),(o_j,k)).$$

Правило (2.5) используется в случае выделения массива указателей, правило (2.6) — в остальных случаях.

Для функции освобождения динамической памяти используются следующие правила:

$$gen[free(p)] : ((p,0),(o^{invalid},0)) \cup$$
$$\cup \bigcup_{\substack{\forall i,j,k_1,k_2:((p,0),(o_j,k_1)),\\ ((q,i),(o_j,k_2))\in\widehat{\mathbb{S}}}} ((q,i),(o^{invalid},0));$$

$$kill[free(p)] : \bigcup_{\forall j,k:((p,0),(o_j,k))\in\widehat{\mathbb{S}}} ((p,0),(o_j,k)); \tag{2.7}$$

$$kill[free(p)] : \bigcup_{\substack{\forall i,j,k_1,k_2:((p,0),(o_j,k_1)),\\ ((q,i),(o_j,k_2))\in\widehat{\mathbb{S}}}} ((q,i),(o_j,k_2)). \tag{2.8}$$

Для переменной p все кортежи заменяются на кортеж с объектом $o^{invalid}$. Если указатель p указывает на несколько объектов, то кортежи для других указателей на эти объекты *дополняются* кортежем с объектом $o^{invalid}$ (правило (2.7)). В случае, когда p указывает на единственный объект, такие кортежи *заменяются* кортежем с объектом $o^{invalid}$ (правило (2.8)). Пример использования сформулированных выше правил показан на рис. 3.

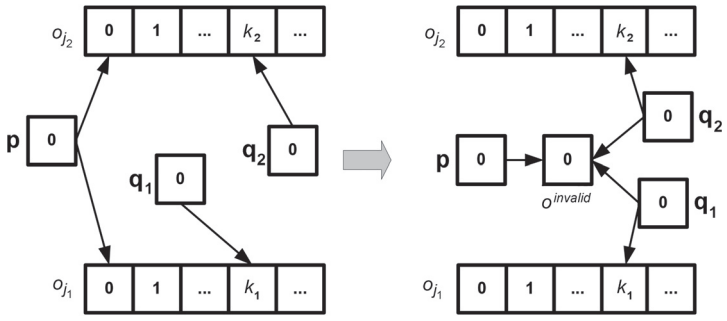

Рис. 3. Правила для функции $free()$.

Рассмотрим правило для операции адресной арифметики

$p = q + shift$:

$$gen[p = q + shift] : \bigcup_{\forall j,k:((q,0),(o_j,k))\in\widehat{\mathbb{S}}} ((p,0),(o_j,k + shift)); \quad (2.9)$$

$$kill[p = q + shift] : \bigcup_{\forall j,k:((p,0),(o_j,k))\in\widehat{\mathbb{S}}} ((p,0),(o_j,k)).$$

Правило (2.9) выполняется для всех объектов o_j, кроме $o^{invalid}$, — если для $(q,0)$ существует кортеж с $o^{invalid}$, то такой же кортеж добавляется и для $(p,0)$.

Рассмотрим различные конструкции, содержащие операции взятия адреса и разыменования указателя в правой части:

$$gen[p = \&q] : ((p,0),(q,0));$$

$$kill[p = \&q] : \bigcup_{\forall j,k:((p,0),(o_j,k))\in\widehat{\mathbb{S}}} ((p,0),(o_j,k));$$

$$gen[p = *q] : \bigcup_{\substack{\forall j_1,j_2,k_1,k_2((q,0),(o_{j_1},k_1)),\\ ((o_{j_1},k_1),(o_{j_2},k_2))\in\widehat{\mathbb{S}}}} ((p,0),(o_j,k)); \quad (2.10)$$

$$gen[p = *q] : ((p,0)(o^{invalid},0)); \quad (2.11)$$

$$kill[p = *q] : \bigcup_{\forall j,k:((p,0),(o_j,k))\in\widehat{\mathbb{S}}} ((p,0),(o_j,k));$$

$$[p = *(q + shift)] \Rightarrow tmp = q + shift;\ p = *tmp; \quad (2.12)$$

$$[p = q[m]] \Rightarrow tmp = q + m;\ p = *tmp. \quad (2.13)$$

Правило (2.11) дополняет правило (2.10) в том случае, если указатель q может указывать на $o^{invalid}$ ($\widehat{\mathbb{S}}$ содержит пару $((q,0),(o^{invalid},0))$). Способ построения правил для более сложных конструкций показан в (2.12) и (2.13). Индексы m и $shift$ могут быть набором интервалов, их значения определяются интервальным анализом.

Рассмотрим случаи операций над указателями в левой части:

$$gen[*p = q] : \bigcup_{\substack{\forall j_1,j_2,k_1,k_2:((p,0),(o_{j_1},k_1)),\\ ((q,0),(o_{j_2},k_2))\in\widehat{\mathbb{S}}}} ((o_{j_1},k_1),(o_{j_2},k_2));$$

$$kill[*p = q]: \bigcup_{\substack{\forall j_1,j_2,k_1,k_2:((p,0),(o_{j_1},k_1)),\\ ((o_{j_1},k_1),(o_{j_2},k_2))\in\widehat{\mathbb{S}}}} ((o_{j_1},k_1),(o_{j_2},k_2)); \qquad (2.14)$$

$$[p[m] = q] \Rightarrow tmp = p + m;\ *tmp = q.$$

Правило (2.14) срабатывает лишь в том случае, когда указатель p указывает на единственный объект o_{j_1} со смещением k_1, в противном случае кортежи не удаляются.

Рассмотренные выше правила применимы для объектов-функций. Необходимо сформулировать правило вызова функции через указатель. Для каждого определения функции вида

$$f(x_1, \ldots, x_n)\{\ldots\ return\ res\}$$

вызов функции $v = (id)(a_1, \ldots, a_n)$ (где id может указывать на функцию $f()$) порождает следующие правила:

$$gen[v =(id)(a_1, \ldots, a_n)]: \bigcup_{\forall i\in[1,n], f:((id,0),(f,0))\in\widehat{\mathbb{S}}} gen(x_i = a_i) \cup gen(v = res);$$

$$kill[v =(id)(a_1, \ldots, a_n)]: kill(v = res).$$

При построении уравнений учитывается влияние операторов ветвления. Выполняется анализ условных выражений, которые оперируют указателями на объекты и функции. Примерами таких выражений являются: $(p == q)$, $(p == NULL)$, $(p > q)$. Для операторов ветвления используются следующие правила:

$$gen[if(cond)]: \emptyset;$$

$$kill[if(cond)]: \widehat{\mathbb{S}} \cap \neg cond; \qquad (2.15)$$

$$kill[if(cond)]: \widehat{\mathbb{S}} \cap cond. \qquad (2.16)$$

Здесь операция $\cap$ производит усечение кортежей в соответствии с условием $cond$. Правило (2.15) действует для ветки $true$ оператора if, правило (2.16) — для ветки $false$.

2.5. Сходимость алгоритмов решения системы уравнений

Для обеспечения сходимости алгоритмов решения системы уравнений функции, стоящие в правой части уравнений, должны

быть монотонны, а высота решетки, на которой заданы эти функции, должна быть конечной [10].

Функция F является монотонной, если

$$S_1 \subseteq S_2 \Rightarrow F(S_1) \subseteq F(S_2).$$

Для всех рассмотренных правил доказано, что полученные функции являются монотонными[4].

Высота решетки конечна, если ограничены количество и размер объектов. Множество объектов состоит из статических (локальные и глобальные переменные, функции) и динамических объектов. Множество статических объектов ограничено за счет явного ограничения глубины стека вызовов функций и использования контекстно-нечувствительного межпроцедурного анализа при достижении указанного ограничения [9].

Количество динамических объектов при выполнении программы потенциально может быть неограниченным. В предлагаемом подходе число объектов искусственно ограничивается за счет использования объекта $o^{universum}$. Размеры объектов определяются в результате интервального анализа и являются конечными.

3. Алгоритм обнаружения дефектов

Исходными данными для алгоритма обнаружения дефектов являются:

- результаты анализа указателей — кортежи вида $((p, i), (o_j, k))$;
- результаты интервального анализа — кортежи вида $((o_j, k), i)$, где i — интервал значений.

Рассмотрим правила обнаружения для различных видов дефектов.

3.1. Ошибки управления динамической памятью

Дефект утечки динамической памяти возникает в конструкциях присваивания нового значения указателю p и в конструкции $undeclare(p)$ в том случае, если указатель p в предыдущем состоянии программы $\widehat{\mathbb{S}}$ указывал на динамический объект и на данный

[4]Из-за ограниченного объема статьи доказательство не приводится.

объект не указывали другие указатели. Это описывается следующим правилом:

$$[undeclare(p) \mid p = \ldots] : \exists j :$$
$$(\exists((p,0),(o_j,k)) \in \widehat{\mathbb{S}} : typeof(o_j) = T^{dynamic}) \wedge$$
$$\wedge \ (\nexists((q,0),(o_j,k)) \in \widehat{\mathbb{S}} : q \neq p).$$

Утечка памяти также может возникнуть в случае, когда освобожденный при помощи функции $free()$ указатель был вершиной сложной структуры в динамической памяти и элементы этой структуры не были освобождены ранее. Обнаружение данного дефекта сводится к решению задачи достижимости всех динамических объектов из статических объектов-указателей в текущем состоянии.

Дефект освобождения некорректного указателя возникает в том случае, если указатель p, переданный в функцию $free()$, мог указывать на некорректный объект, статический объект или функцию. Эта ситуация описывается следующим правилом:

$$[free(p)] :$$
$$\exists((p,0),(o_j,k)) \in \widehat{\mathbb{S}} : \tag{3.1}$$
$$typeof(o_j) \neq T^{dynamic} \vee o_j = o^{invalid}.$$

3.2. Ошибки выхода за границы объекта

Ошибка обращения к массиву по неправильному индексу возникает при значении индекса, выходящем за границы массива. Данный дефект обнаруживается следующим правилом:

$$[p[m]] : \exists((p,0),(o_j,k)) \in \widehat{\mathbb{S}} : \tag{3.2}$$
$$(k + \min(\Gamma_{low}(m)) < 0) \vee (k + \max(\Gamma_{high}(m)) \geq sizeof(o_j)),$$

где $\Gamma_{low}(m)$ — множество всех нижних, а $\Gamma_{high}(m)$ — всех верхних границ интервалов значений для m. Возможные интервалы значений для m определяются интервальным анализом.

Ошибка разыменования указателя, выведенного за границу массива, аналогична ошибке выхода за границу массива. Этот дефект обнаруживается следующим правилом:

$$[*p] : \exists((p,0),(o_j,k)) \in \widehat{\mathbb{S}} : (k < 0) \vee (k \geq sizeof(o_j)).$$

3.3. Ошибки адресной арифметики

Одной из ошибок адресной арифметики является выполнение операции вычитания или сравнения указателей на разные объекты:

$$[p \Diamond q]: \\ \exists((p,0),(o_{j_1},k_1)),((q,0),(o_{j_2},k_2)) \in \widehat{\mathbb{S}} : o_{j_1} \neq o_{j_2},$$

где $\Diamond$ — операция адресной арифметики.

Другой ошибкой является выполнение арифметических операций с указателем на объект, не являющийся массивом, или с указателем на функцию:

$$[p = q + shift]: \\ \exists((q,0),(o_j,k)) \in \widehat{\mathbb{S}} : sizeof(o_j) = 1 \vee typeof(o_j) = T^{func}.$$

3.4. Ошибки инициализации указателей

Дефект разыменования неинициализированного, неконтролируемого или нулевого указателя возникает в конструкции $*p$ в том случае, если указатель мог указывать на некорректный объект $o^{invalid}$. Данная ситуация обнаруживается следующим способом:

$$[*\mathtt{p}] : \exists((p,0),(o^{invalid},0)) \in \widehat{\mathbb{S}}. \tag{3.3}$$

3.5. Расширение набора правил обнаружения дефектов

При необходимости обнаружения других связанных с указателями дефектов (контроль корректности аргументов функций работы с указателями, контроль возвращаемого значения функций и т. д.) данный набор правил может быть дополнен новыми правилами, учитывающими требования поставленной задачи обнаружения дефектов. Изменения алгоритма анализа указателей не требуется, так как предоставляемая им информация является достаточной для определения состояния указателей в любой точке программы.

4. Пример применения предложенных алгоритмов

Рассмотрим применение предложенных алгоритмов на примере простой программы, реализующей цифровой фильтр (листинг 1).

Листинг 1. Пример программы с дефектами.

```
 1 #define N 5
 2 #define M 2
 3 #define SIZE 512
 4 int needfilter = (SIZE > N);
 5
 6 struct datarecord {
 7   double *pdata;
 8   int size;
 9 } *psrc, *pfiltered;
10
11 int main(void)
12 {
13   double filter[N] = {0.1, 0.25, 0.3, 0.25, 0.1};
14   char answer = 'N';
15   printf("Do you want to filter the data? [Yes/No]\n");
16   scanf("%c", &answer);
17   needfilter = (answer == 'Y') || (answer == 'y');
18   psrc = malloc(sizeof(struct datarecord));
19   psrc->size = SIZE;
20   psrc->pdata = malloc(SIZE * sizeof(double));
21   if (needfilter)
22   {
23     pfiltered = malloc(sizeof(struct datarecord));
24     pfiltered->size = SIZE;
25     pfiltered->pdata = malloc(SIZE * sizeof(double));
26   }
27   for (int i = 0; i < SIZE; i++)
28     psrc->pdata[i] = 5.0 * (i - SIZE/2);
29   if (needfilter)
30   {
31     for (int i = 0; i < SIZE; i++)
32     {
33  pfiltered->pdata[i] = 0.0;
34  if ((i >= M) && (i <= SIZE-M))
35  {
36    for (int j = 0; j < N; j++)
```

```
37      pfiltered->pdata[i] =
38        pfiltered->pdata[i] + psrc->pdata[i-M+j]*filter[j];
39  }
40     }
41   }
42   free(psrc);
43   free(pfiltered->pdata);
44   free(pfiltered);
45   return 0;
46 }
```

Данная программа содержит несколько искусственно внесенных дефектов:

- выход за границу массива `psrc` → `pdata` (строка 37);
- утечка памяти в `psrc` → `pdata` (строка 42);
- разыменование некорректного указателя `pfiltered` (строка 43);
- освобождение невыделенной памяти в `pfiltered` (строка 44).

На рис. 4 приведена модель рассматриваемой программы, в которой для экономии места некоторые операторы представлены не в трехоперандной форме.

Рассмотрим применение алгоритма анализа указателей. На рис. 4 конструкции, интерпретируемые алгоритмом анализа указателей, пронумерованы. Система уравнений для анализируемой программы имеет следующий вид:

$$\mathbb{S}_0 = \{\};$$

$$\mathbb{S}_1 = \mathbb{S}_0 \ \cup \ \{((psrc, 0), (o^{invalid}, 0))\};$$

$$\mathbb{S}_2 = \mathbb{S}_1 \ \cup \ \{((pfiltered.1, 0), (o^{invalid}, 0))\};$$

$$\mathbb{S}_3 = \mathbb{S}_2 \ \cup \ \{((pfiltered.2, 0), (o^{invalid}, 0))\};$$

$$\mathbb{S}_4 = \mathbb{S}_3 \ \cup \ \{((pfiltered.3, 0), (o^{invalid}, 0))\};$$

$$\mathbb{S}_5 = \mathbb{S}_4 \setminus \bigcup_{\forall i, j_1, k_1 : ((psrc, i), (o_{j_1}, k_1)) \in \mathbb{S}_4} ((psrc, i), (o_{j_1}, k_1)) \cup$$

$$\cup \ \{((psrc, 0), (o_{j_1} = \langle T^{dynamic}, 2\rangle, 0)), ((o_{j_1}, 1), (o^{invalid}, 0))\};$$

$$\mathbb{S}_6 = \mathbb{S}_5 \setminus \bigcup_{\forall j_1 : ((psrc, 0), (o_{j_1}, 0)) \in \mathbb{S}_5} ((o_{j_1}, 1), (o_{j_2}, k_2)) \cup$$

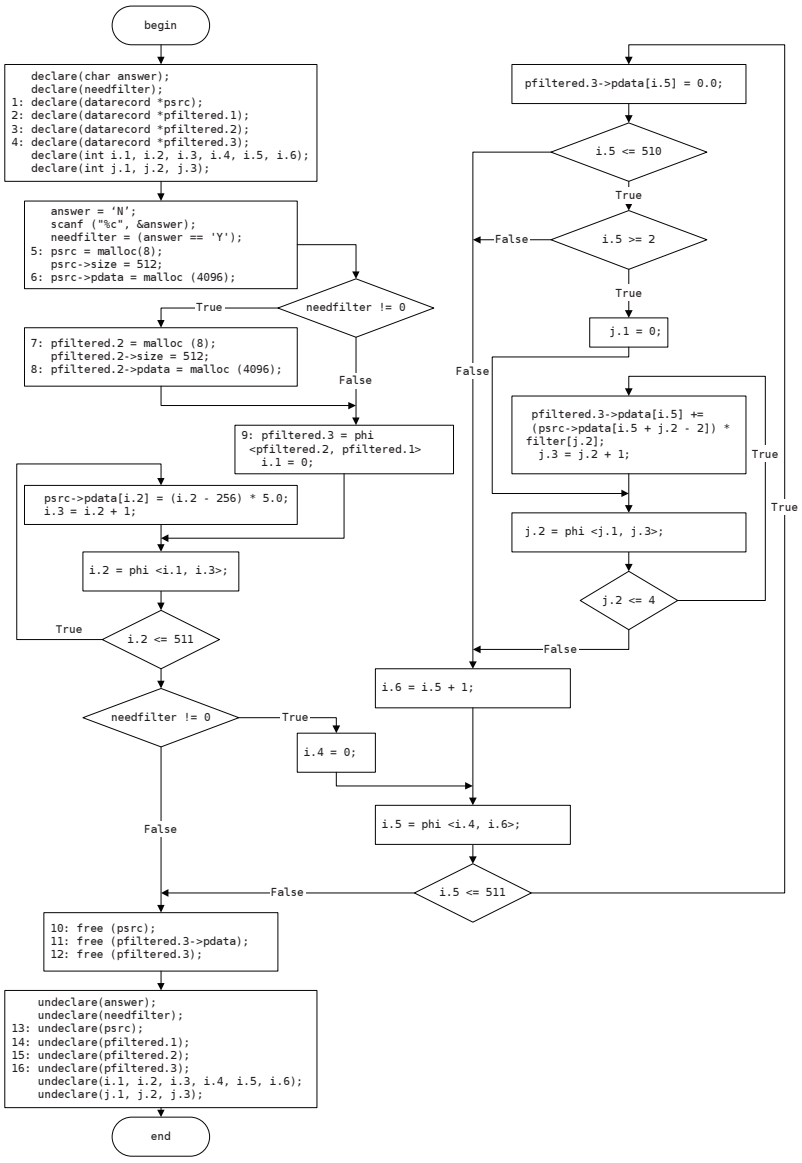

Рис. 4. Модель рассматриваемой программы.

$$\cup \bigcup_{\forall j_1:((psrc,0),(o_{j_1},0))\in\mathbb{S}_5} ((o_{j_1},1),(o_{j_2}=\langle T^{dynamic},512\rangle,0));$$

$$\mathbb{S}_7=\mathbb{S}_6\setminus \bigcup_{\forall j_1,k_1:((pfiltered.2,0),(o_{j_1},k_1))\in\mathbb{S}_6} ((pfiltered.2,0),(o_{j_1},k_1))\cup$$

$$\cup\ \{((pfiltered.2,0),(o_{j_1}=\langle T^{dynamic},2\rangle,0)),((o_{j_1},1),(o^{invalid},0))\};$$

$$\mathbb{S}_8=\mathbb{S}_7\setminus \bigcup_{\forall j_1:((pfiltered.2,0),(o_{j_1},0))\in\mathbb{S}_7} ((o_{j_1},1),(o_{j_2},k_2))\cup$$

$$\cup \bigcup_{\forall j_1:((pfiltered.2,0),(o_{j_1},0))\in\mathbb{S}_7} ((o_{j_1},1),(o_{j_2}=\langle T^{dynamic},512\rangle,0));$$

$$\mathbb{S}_9=(\mathbb{S}_8\cup\mathbb{S}_6)\setminus$$

$$\setminus \bigcup_{\forall j_1,k_1:} ((pfiltered.3,0),(o_{j_1},k_1))\cup$$

$$\cup \bigcup_{\forall j_1,k_1:((pfiltered.1,0),(o_{j_1},k_1))\in\mathbb{S}_8\cup\mathbb{S}_6} ((pfiltered.3,0),(o_{j_1},k_1))\cup$$

$$\cup \bigcup_{\forall j_1,k_1:((pfiltered.2,0),(o_{j_1},k_1))\in\mathbb{S}_8\cup\mathbb{S}_6} ((pfiltered.3,0),(o_{j_1},k_1));$$

$$\mathbb{S}_{10}=\mathbb{S}_9\setminus \bigcup_{\forall j_1:((psrc,0),(o_{j_1},0))\in\mathbb{S}_9} ((psrc,0),(o_{j_1},0))\cup$$

$$\cup \bigcup_{\forall j_1:((psrc,0),(o_{j_1},0))\in\mathbb{S}_9} ((psrc,0),(o^{invalid},0));$$

$$\mathbb{S}_{11}=\mathbb{S}_{10}\setminus \bigcup_{\forall j_1:((pfiltered.3,0),(o_{j_1},0))\in\mathbb{S}_{10}} ((o_{j_1},1),(o_{j_2},k_2))\cup$$

$$\cup \bigcup_{\forall j_1:((pfiltered.3,0),(o_{j_1},0))\in\mathbb{S}_{10}} ((o_{j_1},1),(o^{invalid},0));$$

$$\mathbb{S}_{12}=\mathbb{S}_{11}\setminus$$

$$\setminus \bigcup_{\substack{\forall i,j_1:((pfiltered.3,0),(o_{j_1},0)),\\ ((p_i,k_i),(o_{j_1},k))\in\mathbb{S}_{11}}}^{\substack{((pfiltered.3,0),(o_{j_1},0)),\\ ((p_i,k_i),(o_{j_1},k))}} \cup$$

$$\cup \bigcup_{\substack{\forall i,j_1:((pfiltered.3,0),(o_{j_1},0)),\\ ((p_i,k_i),(o_{j_1},k))\in\mathbb{S}_{11}}}^{\substack{((pfiltered.3,0),(o^{invalid},0)),\\ ((p_i,k_i),(o^{invalid},0))}};$$

$$\mathbb{S}_{13} = \mathbb{S}_{12} \setminus \bigcup_{\forall j_1,k_1:((psrc,0),(o_{j_1},k_1))\in\mathbb{S}_{12}} ((psrc,0),(o_{j_1},k_1));$$

$$\mathbb{S}_{14} = \mathbb{S}_{13} \setminus \bigcup_{\forall j_1,k_1:((pfiltered.1,0),(o_{j_1},k_1))\in\mathbb{S}_{13}} ((pfiltered.1,0),(o_{j_1},k_1));$$

$$\mathbb{S}_{15} = \mathbb{S}_{14} \setminus \bigcup_{\forall j_1,k_1:((pfiltered.2,0),(o_{j_1},k_1))\in\mathbb{S}_{14}} ((pfiltered.2,0),(o_{j_1},k_1));$$

$$\mathbb{S}_{16} = \mathbb{S}_{15} \setminus \bigcup_{\forall j_1,k_1:((pfiltered.3,0),(o_{j_1},k_1))\in\mathbb{S}_{15}} ((pfiltered.3,0),(o_{j_1},k_1)).$$

Решение системы уравнений приводится только для конструкций, содержащих дефекты:

$\mathbb{S}_9$:

$((psrc,0),(o_1,0))$,
$((o_1,1),(o_2,0))$,
$((pfiltered.2,0),(o_3,0))$,
$((pfiltered.3,0),(o_3,0))$,
$((o_3,1),(o_4,0))$,
$((pfiltered.1,0),(o^{invalid},0))$,
$((pfiltered.3,0),(o^{invalid},0))$.

$\mathbb{S}_{10}$:

$((o_1,1),(o_2,0))$,
$((pfiltered.2,0),(o_3,0))$,
$((pfiltered.3,0),(o_3,0))$,
$((o_3,1),(o_4,0))$,
$((psrc,0),(o^{invalid},0))$,
$((pfiltered.1,0),(o^{invalid},0))$,
$((pfiltered.3,0),(o^{invalid},0))$.

$\mathbb{S}_{11}$:

$((o_1,1),(o_2,0))$,
$((pfiltered.2,0),(o_3,0))$,
$((pfiltered.3,0),(o_3,0))$,
$((o_3,1),(o^{invalid},0))$,

$((psrc, 0), (o^{invalid}, 0)),$

$((pfiltered.1, 0), (o^{invalid}, 0)),$

$((pfiltered.3, 0), (o^{invalid}, 0)).$

$\mathbb{S}_{12}$:

$((o_1, 1), (o_2, 0)),$

$((psrc, 0), (o^{invalid}, 0)),$

$((o_3, 1), (o^{invalid}, 0)),$

$((pfiltered.1, 0), (o^{invalid}, 0)),$

$((pfiltered.2, 0), (o^{invalid}, 0)),$

$((pfiltered.3, 0), (o^{invalid}, 0)).$

Применение правила обнаружения дефектов (3.2) к результатам анализа указателей и результатам интервального анализа ($i.5 = [2, 510],\ j.2 = [0, 4]$) позволяет выявить ошибку выхода за границу массива:

$$[pfiltered.3 \rightarrow pdata[i.5] += (psrc \rightarrow pdata[i.5 + j.2 - 2]) * filter[j.2]] :$$
$$\exists((pfiltered.3, 0), (o_3, 0)), ((o_3, 1), (o_4, 0)) \in \mathbb{S}_9 :$$
$$(0 + \max(\Gamma_{high}(i.5)) + \max(\Gamma_{high}(j.2)) - 2) = 512 > 511.$$

С помощью правила для обнаружения утечек памяти в сложной структуре можно обнаружить дефект утечки памяти, на которую ссылается $psrc \rightarrow pdata$:

$$[free(psrc)] :$$
$$(\exists((o_1, 1), (o_2, 0)) \in \mathbb{S}_{10} : typeof(o_2) = T^{dynamic}) \wedge$$
$$\wedge\ (\nexists((q, i), (o_2, k)) \in \mathbb{S}_{10}).$$

Использование правила обнаружения дефектов (3.3) позволяет выявить дефект разыменования некорректного указателя:

$$[free(pfiltered.3 \rightarrow pdata)] : \exists((pfiltered.3, 0), (o^{invalid}, 0)) \in \mathbb{S}_{11}.$$

Применение правила для обнаружения ошибок освобождения некорректного указателя (3.1) позволяет выявить соответствующий дефект управления памятью:

$$[free(pfiltered.3)] : \exists((pfiltered.3, 0), (o^{invalid}, 0)) \in \mathbb{S}_{12}.$$

Использование разработанных алгоритмов позволяет обнаруживать *все* внесенные в программу дефекты.

5. Сравнение с существующими средствами

Для оценки синтезированных алгоритмов сравним результаты их применения с результатами работы существующих средств обнаружения дефектов.

Для проведения экспериментальных исследований были выбраны следующие средства [2]:

- IBM Rational Software Analyzer 7.0.0,
- Gimpel FlexeLint 9.0,
- Microsoft Code Analysis 2008,
- Fortify Source Code Analyzer 5.2,
- Splint 3.1.2.

Экспериментальные исследования проводились с целью выявления особенностей алгоритмов анализа указателей, используемых в перечисленных средствах. Для этого тестовый пример, рассмотренный выше (см. листинг 1), был проанализирован указанными средствами. В результате четыре из пяти выбранных средств не обнаружили ни одного из внесенных дефектов, средство Splint обнаружило один из четырёх дефектов — утечку динамической памяти в строке 42.

Дополнительно было проведено исследование на наборе искусственных тестовых программ, в которых используются массивы, структуры, линейные списки, статические массивы указателей, массивы указателей в динамической памяти, указатели на указатели. Тестовый набор состоит из 15 программ, содержащих 21 дефект. Исходные коды тестовых программ доступны на сайте проекта[5]. Оценивалось количество правильно обнаруженных дефектов, результаты исследования приведены в табл. 1.

Таблица 1. **Результаты исследования средств обнаружения дефектов**

	IBM	**FlexeLint**	**MS**	**Fortify**	**Splint**	**Алгоритм**
Итого	0%	10%	0%	0%	24%	100%

Все средства, кроме FlexeLint и Splint, не обнаружили ни одного дефекта в тестовых программах. Splint обнаружил часть де-

[5] http://s2a.ftk.spbstu.ru

фектов, связанных с указателями в структурах. FlexeLint обнаружил дефект выдачи локального объекта из области видимости и дефект, связанный с вызовом функции $free()$ для статического объекта. Все рассмотренные средства показывают худшие результаты по сравнению с предложенным алгоритмом.

Заключение

В данной статье представлены алгоритмы обнаружения дефектов, связанных с некорректным использованием указателей. Основой алгоритма анализа указателей является предложенная модель отношения «указывает-на», которая позволяет описать связи между объектами программы. В статье подробно рассмотрены отдельные стадии алгоритма анализа указателей, сформулированы правила анализа для основных конструкций языка C, представлены правила обнаружения типичных дефектов, связанных с указателями.

Достоинствами синтезированных алгоритмов являются: полнота получаемого решения, отсутствие фундаментальных ограничений на анализируемые программы, простота создания правил для обнаружения других типов дефектов.

Для оценки разработанных алгоритмов были проведены экспериментальные исследования существующих коммерческих средств и средств с открытым исходным кодом на наборе тестовых программ. Сравнение полученных результатов показало, что предложенные алгоритмы значительно превосходят алгоритмы, используемые в существующих средствах обнаружения дефектов.

Описанные в статье алгоритмы можно распространить на другие языки программирования, в первую очередь на язык C++. Для этого потребуется модификация правил для учета дополнительных возможностей языка C++, таких как механизм исключений, полиморфизм объектов, перегрузка операций и др.

В рамках выполняемых исследований разрабатывается экспериментальный образец, который позволит применить предложенные алгоритмы для обнаружения дефектов в реальных программных системах.

Список литературы

[1] *Ицыксон В. М., Моисеев М. Ю., Цесько В. А., Захаров А. В., Ахин М. Х.* Алгоритм интервального анализа для обнаружения де-

фектов в исходном коде программ // Информационные и управляющие системы. № 2. СПб.: Политехника, 2009. С. 34–41.

[2] *Ицыксон В. М., Моисеев М. Ю., Цесько В. А., Карпенко А. В.* Исследование систем автоматизации обнаружения дефектов в исходном коде программ // Научно-технические ведомости СПбГПУ. № 5. СПб.: Изд-во Политехн. ун-та, 2008. С. 119–127.

[3] *Andersen L.* A Program Analysis and Specialization for the C Programming Language. Ph.D. thesis, Copenhagen, 1994. 311 p.

[4] *Avots D., Dalton M., Livshits B. et al.* Improving Software Security with a C Pointer Analysis // Proceedings of the 27th International Conference on Software Engineering (ICSE'05). 2005. ACM. P. 332–341.

[5] *Chen K., Wagner D.* Large-scale Analysis of Format String Vulnerabilities in Debian Linux // Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security (PLAS'07). 2007. ACM. P. 75–84.

[6] *Cytron R., Ferrante J., Rosen B. et al.* Efficiently Computing Static Single Assignment Form and the Control Dependence Graph // ACM Transactions on Programming Languages and Systems. Vol. 13. Issue 2. 1991. ACM. P. 451–490.

[7] *Heine D., Lam M.* A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector // Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03). 2003. ACM. P. 168–181.

[8] *Larochelle D., Evans D.* Statically Detecting Likely Buffer Overflow Vulnerabilities // Proceedings of the 10th Conference on USENIX Security Symposium (SSYM'01). 2001. ACM. P. 14.

[9] *Nielson F., Nielson H., Hankin C.* Principles of Program Analysis. Springer, 2005. 452 p.

[10] *Schwartzbach M.* Lecture Notes on Static Analysis. 58 p. http://www.brics.dk/m̃is/static.pdf

[11] *Steensgaard B.* Points-to Analysis in Almost Linear Time // Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96). 1996. ACM. P. 32–41.