МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ М.В.ЛОМОНОСОВА
Механико-математический факультет

На правах рукописи

Пучков Федор Михайлович

Методы и средства автоматизированного обнаружения уязвимостей в программах на языке C на основе статического анализа их исходных текстов

Специальность 05.13.19 — методы и системы защиты информации, информационная безопасность

АВТОРЕФЕРАТ
диссертации на соискание ученой степени
кандидата физико-математических наук

Москва — 2010

Работа выполнена на Механико-математическом факультете и в Институте проблем информационной безопасности Московского государственного университета имени М. В. Ломоносова.

Научный руководитель:	доктор физико-математических наук, профессор *Васенин Валерий Александрович.*
Официальные оппоненты:	доктор технических наук, профессор *Тимонина Елена Евгеньевна*;
	кандидат физико-математических наук, *Костюхин Константин Александрович.*
Ведущая организация:	Институт системного программирования РАН.

Защита состоится 24 марта 2010 г. в 16 час. 45 мин. на заседании диссертационного совета Д 501.002.16 при Московском государственном университете имени М. В. Ломоносова по адресу: Российская Федерация, 119991, Москва, ГСП-1, Ленинские горы, д. 1, Московский государственный университет имени М. В. Ломоносова, Механико-математический факультет, ауд. 14-08.

С диссертацией можно ознакомиться в библиотеке Механико-математического факультета Московского государственного университета имени М. В. Ломоносова.

Автореферат разослан 24 февраля 2010 г.

Ученый секретарь
диссертационного совета,
доктор физико-математических наук, доцент Корнев А. А.

Общая характеристика работы

Актуальность работы. Одним из важнейших условий успешного функционирования практически значимых объектов, потенциально уязвимых для кибератак, является защита используемого в их составе программного обеспечения от возможных сбоев в работе. Это условие становится особенно важным при проектировании автоматизированных систем с высоким уровнем требований к их защищенности, в том числе — при разработке и реализации программно-технических средств защиты информации, что отражено в соответствующих Руководящих документах ФСТЭК РФ.[1,2,3] Появление подобных сбоев зачастую связано с наличием *уязвимостей* (ошибок, дефектов) в программных средствах подобных систем. В их числе такие, использование которых злоумышленником может привести к нерегламентированному политикой информационной безопасности повышению его привилегий на том или ином узле системы, подлежащей защите. Кроме того, сбои в работе программ могут приводить к аварийному завершению работы важных программно-аппаратных компонентов, создавая угрозу отказа для системы в целом. Имея в виду существующие подходы к разработке и последующему сопровождению программного обеспечения,[4] в частности, на основе уже существующих средств с открытым исходным кодом, на первый план выходят следующие задачи:

- создание эффективных механизмов обнаружения потенциальных уязвимостей в программном обеспечении;
- применение указанных механизмов для обнаружения и устранения уязвимостей используемого программного обеспечения;
- обоснование отсутствия других уязвимостей (дефектов) в используемом программном обеспечении.

Актуальность выбора для анализа именно программ, написанных с использованием языка программирования C, обусловлена широкой распространенностью этого языка в среде программного обеспечения с открытым исходным кодом. Согласно проведенным исследованиям, на долю языка C приходится более 80%

[1]Защита от несанкционированного доступа к информации. Термины и определения. // Руководящий документ. Гостехкомиссия России, 1992.

[2]Средства вычислительной техники. Защита от несанкционированного доступа к информации. Показатели защищенности от несанкционированного доступа к информации. // Руководящий документ. Гостехкомиссия России, 1992.

[3]Средства вычислительной техники. Защита от несанкционированного доступа к информации. Часть 1. Программное обеспечение средств защиты информации. Классификация по уровню контроля отсутствия недекларированных возможностей. // Руководящий документ. Гостехкомиссия России, 1992.

[4]Липаев В. В. Программная инженерия. Методологические основы. — М.: Теис, 2006. — 608 с.

объема кода, используемого в современных программных средствах.[5,6] Дополнительным фактором такого выбора является и то обстоятельство, что обеспечение корректности операций с памятью и указателями в языке C является задачей разработчика программного обеспечения.

Следует отметить, что в настоящее время эффективных и широко распространенных автоматизированных средств, позволяющих проверять корректность программ на языке C и выявлять в них возможные уязвимости, не существует. Традиционные средства статического анализа (аудита) исходного кода недостаточно точно анализируют семантику программы, и, как следствие, — приводят к большому количеству ложных предупреждений. Например, программный комплекс ITS4[7] анализирует лишь наборы лексем, сопоставляя их с образцами «потенциально опасных конструкций», хранящимися в специальной базе данных. Подобный алгоритм проверки может выдавать до 100% ложных предупреждений, не обнаружив при этом в исследуемой программе ни одной реальной уязвимости. Программный комплекс Splint[8] позволяет находить лишь простейшие уязвимости такие, как, например, выход за границу статического массива при обращении по фиксированному индексу. Для обнаружения уязвимостей в более сложных ситуациях анализатору необходимо предоставить специальные аннотации, составление которых представляется весьма трудоемкой процедурой. Программный комплекс BOON[9] реализует один из простейших вариантов «интервального анализа» — для каждой целочисленной переменной x вычисляется интервал $[a, b]$ ее возможных значений такой, что в каждой точке программы справедливо соотношение $a \leq x \leq b$. К недостаткам алгоритма можно отнести следующие факторы, значительно снижающие точность анализа.

- Алгоритм анализа нечувствителен к потоку управления. Анализируются лишь инструкции присваивания, причем независимо от порядка, в котором они встречаются в исследуемой программе.
- Отсутствует анализ указателей.
- Не исследуются зависимости между различными переменными.

[5]Wheeler D. More Than a Gigabuck: Estimating GNU/Linux's Size. [Электронный ресурс]. Режим доступа: `http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html`, свободный. — Электрон. текст. док.

[6]Measuring Libre Software Using Debian 3.1 (Sarge) as A Case Study: Preliminary Results. [Электронный ресурс]. Режим доступа: `http://www.upgrade-cepis.org/issues/2005/3/up6-3Amor.pdf`, свободный. — Электрон. текст. док.

[7]ITS4: Software Security Tool. [Электронный ресурс]. Режим доступа: `http://www.cigital.com/its4/`, свободный.

[8]Splint — Secure Programming Lint. [Электронный ресурс]. Режим доступа: `http://www.splint.org`, свободный.

[9]BOON — Buffer Overrun detectiON. [Электронный ресурс]. Режим доступа: `http://www.eecs.berkley.edu/~daw/boon/`, свободный.

На рынке информационных технологий представлен также ряд коммерческих продуктов (Coverity,[10] PolySpace[11]), позволяющих (по мнению их разработчиков) проводить более точный анализ исходного кода программ. Однако, результаты использования указанных продуктов, а также применяемые методы и алгоритмы не были опубликованы.

Программный комплекс ASTRÉE[12] позволяет верифицировать программы на языке C, не содержащие операций динамического выделения/освобождения памяти, а также не использующие рекурсию.

Средства динамического анализа, применяемые непосредственно во время работы исследуемой программы (например, Valgrind[13]), могут быть эффективными для обнаружения «ошибок времени выполнения» (runtime errors). Однако, во-первых, они существенно замедляют работу программы, а во-вторых, указанные средства проверяют корректность выполнения программы лишь на конечном множестве наборов входных данных (тестов) и не могут гарантировать корректность работы исследуемой программы в процессе ее дальнейшей эксплуатации. По этой причине, методы динамического анализа не являются полноценной заменой статическим методам аудита исходного кода C-программ.

С учетом представленных выше доводов, задача создания методов и средств, позволяющих в автоматизированном режиме находить в программах на языке C потенциально уязвимые места, а именно — конструкции, выполнение которых может быть некорректным (неопределенным) согласно спецификации языка программирования C, а также гарантировать корректность выполнения других участков кода, является в настоящее время крайне актуальной.

Целью диссертационной работы является исследование и разработка математических моделей, алгоритмов и программных средств автоматизированного обнаружения дефектов (уязвимостей) в программах на языке C, которые основаны на использовании формальных методов верификации и статического анализа таких программ, ориентированных на применение к крупным программным комплексам с открытым исходным кодом, содержащим до 500 тыс. строк кода.

Научная новизна. Автором разработан новый, доказательно обоснованный метод статического анализа исходного кода программ на языке C, позволяющий *гарантированно* выявлять в таких программах некоторые классы дефектов (уязвимостей). Отличительными особенностями разработанного метода являются: строгая формализация задачи обнаружения программных дефектов в рамках

[10]Coverity. [Электронный ресурс]. Режим доступа: `http://www.coverity.com/`, свободный.

[11]PolySpace Embedded Software Verification. [Электронный ресурс]. Режим доступа: `http://www.mathworks.com/products/polyspace/`, свободный.

[12]The ASTRÉE Static Analyzer. [Электронный ресурс]. Режим доступа: `http://www.astree.ens.fr/`, свободный.

[13]Valgrind. [Электронный ресурс]. Режим доступа: `http://www.valgrind.org`, свободный.

построенной математической модели; использование этой модели для обоснования корректности представленного базового алгоритма; возможность корректного расширения базового алгоритма, позволяющего значительно повысить его эффективность (метод проверки индуктивных гипотез).

Практическая значимость. Автором диссертации совместно с К. А. Шапченко и О. О. Андреевым разработан прототип программного комплекса — «Статический анализатор программных дефектов», реализующий представленные в диссертации алгоритмы. В работе продемонстрированы функциональные возможности созданного прототипа, а также его эффективность для анализа существующих крупных программных комплексов.

На защиту выносятся следующие основные результаты диссертации:

- систематизирована исследуемая предметная область, введено понятие программного дефекта, предложена классификация существующих программных дефектов С-программ;
- предложен метод автоматизированного обнаружения дефектов в программах на языке С, основанный на преобразовании исходной программы в промежуточный формат «IAL» (от Intermediate Analyser Language) с последующей верификацией полученной IAL-программы;
- разработан язык IAL промежуточного представления С-программ, описаны его синтаксис и семантика, разработана математическая модель языка IAL, в рамках которой формализована задача верификации IAL-программ;
- разработан базовый алгоритм верификации программ в формате промежуточного представления, сформулирована и доказана его корректность, предложен не нарушающий корректности алгоритма способ проверки индуктивных гипотез, позволяющий повысить точность анализа;
- создан прототип программного комплекса автоматизированного обнаружения дефектов в программах на языке С, эффективность которого продемонстрирована как на ряде специально подобранных модельных примеров, так и на примере программы `xorg-server` — основной части подсистемы управления графикой в UNIX-подобных операционных системах.

Апробация работы. Основные результаты диссертации докладывались на международной научной конференции «Информационная безопасность регионов России (ИБРР-2003)», на международных конференциях «Математика и безопасность информационных технологий» (2004–2006), «Ломоносовские чтения» (2006–2007), на механико-математическом факультете МГУ имени М.В. Ломоносова на семинаре «Проблемы современных информационно-вычислительных систем» под руководством д.ф.-м.н., проф. В.А. Васенина (2004, 2009).

Публикации. По теме диссертации опубликовано 10 научных работ, в том числе в зарубежных журналах, из них одна статья [3] в журнале из перечня ВАК ведущих рецензируемых журналов, а также — 4 патента на изобретения. Материалы работы вошли в главу 7 опубликованной в 2008 году книги «Критически важные объекты и кибертерроризм. Часть 2. Аспекты программной реализации средств противодействия» под ред. д.ф.-м.н., проф. В.А. Васенина [6].

Личный вклад автора. Автором проведено исследование современного состояния в области автоматизации процессов верификации и обнаружения дефектов программного обеспечения; предложен метод обнаружения дефектов в программах на языке C, основанный на преобразовании исходной программы в промежуточный формат «IAL» (от Intermediate Analyser Language) с последующей верификацией полученной IAL-программы; разработан язык IAL (описаны его синтаксис и семантика); предложена математическая модель и алгоритм верификации IAL-программ, в рамках которой автором доказана корректность предложенного алгоритма; разработан модуль верификации IAL-программ, являющийся основой программного комплекса автоматизированного обнаружения дефектов в C-программах, созданного автором совместно с К. А. Шапченко и О. О. Андреевым.

Структура и объем диссертации. Работа состоит из введения, четырех глав, заключения, списка литературы. Общий объем диссертации — 120 страниц. Список литературы включает 77 наименований.

Содержание работы

Во **введении** описываются цели работы, обосновывается ее актуальность и практическая значимость, перечисляются основные результаты.

Первая глава является вводной и посвящена исследованию и систематизации рассматриваемой предметной области. В начале главы приводится ряд определений.

Определение 1.1. *Уязвимость вычислительной системы* — такое ее состояние, которое позволяет субъекту (потенциальному злоумышленнику) получить несанкционированный политикой информационной безопасности доступ к ее ресурсам, вызвать отказ в обслуживании, или каким-либо другим образом нарушить штатный режим функционирования.

Определение 1.2. *Уязвимость программного обеспечения (программы)* — такое ее свойство, которое позволяет субъекту (потенциальному злоумышленнику) привести систему в состояние уязвимости, воздействуя определенным образом на внешнее окружение, в котором выполняется программа.

Определение 1.3. Под *внешним окружением программы* будем понимать любые данные, получаемые ею из внешних источников (из файлов, из входного потока данных, из устройств ввода, из переменных окружения и так далее).

Определение 1.4. Под *внутренним окружением программы* будем понимать значения переменных и состояние памяти программы в момент ее выполнения.

В рамках диссертационной работы рассматриваются лишь такие программные уязвимости, которые возникают из-за наличия в исходном тексте так называемых «программных дефектов».

Определение 1.5. *Программным дефектом* (применительно к программам на некотором языке программирования) будем называть операцию в программе, результат выполнения которой (при определенном воздействии на внешнее окружение программы) приводит к аварийному завершению работы программы, либо является неопределенным согласно спецификации используемого языка программирования.

В разделе 1.1 рассматривается предложенная автором классификация программных дефектов. В ее рамках выделены следующие классы программных дефектов:

- некорректные операции с памятью;
- некорректные операции с целыми типами;
- некорректное использование функций стандартной библиотеки;
- операции чтения неопределенного значения;
- утечки памяти;
- другие.

Далее в соответствующих подразделах диссертационной работы перечисленные выше классы дефектов с демонстрирующими их примерами рассматриваются более подробно.

В разделе 1.2 проводится обзор существующих методов и средств обнаружения уязвимостей (дефектов) в программах на языке C.

В подразделе 1.2.1 на примере программных средств GNU grep,[14] Flawfinder,[15] ITS4[16] показана неэффективность использования только средств лексического и синтаксического анализа для решения задачи автоматизированного обнаружения уязвимостей в программах на языке C.

В подразделах 1.2.2–1.2.3 рассматриваются программные средства Splint,[17] BOON,[18] реализующие несложный семантический анализ. Основными их недо-

[14]The GNU Grep.3;3 [Электронный ресурс]. Режим доступа: `http://www.gnu.org/software/grep/`, свободный.

[15]Flawfinder. [Электронный ресурс]. Режим доступа: `http://www.dweeler.com/flawfinder/`, свободный.

[16]ITS4: Software Security Tool. [Электронный ресурс]. Режим доступа: `http://www.cigital.com/its4/`, свободный.

[17]Splint — Secure Programming Lint. [Электронный ресурс]. Режим доступа: `http://www.splint.org`, свободный.

[18]BOON — Buffer Overrun detectiON. [Электронный ресурс]. Режим доступа: `http://www.eecs.berkley.edu/~daw/boon/`, свободный.

статками являются: недостаточная глубина и точность анализа; отсутствие математически строгого обоснования корректности используемых алгоритмов анализа; ориентированность на выявление узкого класса программных дефектов. Недостаточная глубина и точность анализа приводят к появлению большого количества ложных предупреждений, выдаваемых на выходе. По причине отсутствия математически строгого обоснования корректности используемых алгоритмов появляется возможность пропусков реальных уязвимостей (дефектов). Как следствие того, что перечисленные средства ориентированы на выявление узкого класса программных дефектов возрастает вероятность присутствия в исследуемой программе дефектов, относящихся к другим классам.

Во **второй главе** представлен разработанный автором метод обнаружения дефектов в программах на языке C. Во введении к главе (раздел 2.1) определяется свойство *полноты* алгоритма автоматизированного обнаружения дефектов, как возможности *гарантированного* обнаружения всех возможных дефектов, относящихся к определенному классу. При этом допускается наличие ложных тревог.

В разделе 2.2 излагается общий подход к решению задачи обнаружения программных дефектов. Метод, предлагаемый в диссертационной работе, состоит из трех этапов:

1) проведение лексического и синтаксического анализа C-программы, построение дерева разбора/абстрактного синтаксического дерева;

2) преобразование дерева разбора в промежуточное представление;

3) верификация программы на языке промежуточного представления.

Лексический и синтаксический анализ представляет собой процесс сопоставления линейной последовательности символов программы с формальной грамматикой языка C (согласно стандарта «C99»). Результатом является дерево разбора или абстрактное синтаксическое дерево.

Преобразование дерева разбора в промежуточное представление. Структура абстрактного синтаксического дерева C-программы является достаточно сложной. По этой причине на втором шаге осуществляется преобразование дерева в более простой формат, называемый «промежуточным представлением». Промежуточное представление является программой, в некотором смысле эквивалентной исходной, написанной на специально разработанном для этого языке IAL (от Intermediate Analyser Language). Подробное описание языка IAL представлено в разделе 2.3 диссертации. Рассмотрим наиболее значимые его особенности.

- Язык IAL представляет собой однопроцедурный формат. Программа представляет собой конечный упорядоченный набор инструкций. Функции, определенные в исходной C-программе, непосредственно встраиваются в код основной процедуры. Для C-программ, не содержащих рекурсивных вызовов

функций, возможность такого встраивания следует из ацикличности графа вызовов функций программы. Если же исходная C-программа содержит рекурсивные вызовы функций, то необходимо предварительно вызов каждой рекурсивной функции преобразовать в эквивалентную итерационную конструкцию, использующую динамически выделяемый массив для моделирования стека вызовов функций. Внешние по отношению к исследуемой C-программе функции (например, библиотечные) заменяются на соответствующие *аннотации* — наборы IAL-инструкций, адекватно отражающие выполненные в рамках вызова внешней функции операции с параметрами и возвращаемым значением.

- Переменные в IAL-программе играют роль «регистров». А именно, они позволяют хранить значения различных типов, однако для переменной отсутствует понятие адреса или участка памяти, содержащего ее значение. Участки памяти, доступные по указателю, выделяются в IAL-программе динамически инструкцией `new`, а освобождаются — инструкцией `del`.
- В языке IAL принята упрощенная система типов. Определено восемь целых типов, а именно: `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, а также тип указателя `ptr`. Для каждого из представленных типов в точности определена его разрядность, множество допустимых значений, семантика арифметических операций со значениями этого типа. Вместе с тем, отсутствуют типы чисел с плавающей точкой, массивы, структуры, объединения. Операции с объектами этих типов моделируются соответствующими операциями с участками памяти.
- Язык IAL представляет собой разновидность трехадресного кода. Сложные выражения представлены в виде последовательности «атомарных» инструкций, а управляющие конструкции преобразованы в эквивалентные с использованием операторов `if-goto` и `goto`.
- Семантика IAL-программы является строгой. Любая некорректная операция (например, арифметическое переполнение или обращение к памяти по недействительному указателю) должна немедленно приводить к аварийной остановке всей программы. Преобразование C-программы к виду программы на языке IAL обладает тем свойством, что если исходная C-программа содержит дефекты, проявляющиеся при определенном воздействии на внешнее окружение, то при аналогичном воздействии на внешнее окружение IAL-программы, ее выполнение завершится в одном из аварийных состояний. Таким образом, задача обнаружения дефектов в C-программе может быть сведена к задаче верификации соответствующей ей IAL-программы.

Верификация IAL-программы составляет основную часть разработанного метода. Целью верификации IAL-программы является выявление списка *опасных*

инструкций, в результате выполнения которых возможен переход программы в одно из аварийных состояний.

Определение 2.6. Алгоритм верификации будем называть *корректным*, если для любой заданной IAL-программы получаемый на выходе список инструкций содержит все ее опасные инструкции (и, возможно, некоторые другие инструкции).

Свойство корректности алгоритма верификации IAL-программ (в предположении корректности алгоритма преобразования C-программ в промежуточное представление) является необходимым и достаточным условием полноты всего алгоритма обнаружения программных дефектов в программах на языке C.

Для корректного решения задачи верификации IAL-программ используется подход, основанный на построении *инвариантов* с последующей проверкой в каждой точке программы выводимости условий корректности из инвариантов в соответствующей точке.

Определение 2.7. Пусть n — номер некоторой инструкции IAL-программы P. *Инвариантом программы* P в точке n будем называть логическую формулу φ, связывающую значения переменных и состояние памяти программы, истинную всякий раз, когда текущая выполняемая инструкция имеет номер n.

Определение 2.8. Пусть n — номер некоторой инструкции IAL-программы P. *Условием корректности программы* P в точке n будем называть логическую формулу ψ, связывающую значения переменных и состояние памяти программы, которая выражает необходимые и достаточные условия, при которых выполнение инструкции с номером n не приведет к аварийному состоянию.

Пусть φ — инвариант программы P в точке n, а ψ — условие корректности в этой же точке. Тогда, если формула $(\varphi \Longrightarrow \psi)$ истинна при любых значениях входящих в нее свободных переменных (или, что эквивалентно, формула $(\varphi \wedge \neg\psi)$ невыполнима), то корректность выполнения инструкции с номером n будем считать доказанной. Данный принцип положен в основу следующего алгоритма (через $|P.Stmt|$ обозначено число инструкций в программе P, а через $P.Stmt[n]$ — инструкция с номером n):

```
IAL-VERIFY(P)
1  cc := GENERATE-CC(P)              ▷ Сгенерировать массив условий корректности cc
2  inv := GENERATE-INV(P)            ▷ Сгенерировать массив инвариантов inv
3  wrn := ε                          ▷ Текущей список предупреждений wrn
4  for n := 1 to |P.Stmt| do         ▷ P.Stmt — массив инструкций программы P
5  begin
6     if CHECK-SAT(inv[n] ∧ ¬cc[n]) then
7        wrn := (wrn, P.Stmt[n])     ▷ Добавить инструкцию P.Stmt[n] в конец
                                     ▷ текущего списка предупреждений
8  end
9  return wrn
```

В разделе 2.3 диссертации приводится детализированное описание языка IAL, его синтаксиса и семантики.

В разделе 2.4 строится математическая модель языка IAL. Программа представляет собой тройку $P = (Var, Type, Stmt)$, где Var — множество переменных; $Type$ — отображение, сопоставляющее каждой переменной ее тип; $Stmt$ — конечное упорядоченное множество инструкций программы. С целью формализации процесса выполнения IAL-программы рассматривается система переходов состояний $\Sigma_P = (S,\ T,\ s_{init},\ E)$, где S — множество состояний системы (не обязательно конечное); $T \subseteq S \times S$ — отношение, определяющее возможность перехода в следующее состояние; s_{init} — начальное состояние; $E \subset S$ — множество *ошибочных* состояний, причем $s_{init} \in (S \setminus E)$. Состояния множества $(S \setminus E)$ называются *корректными*. В разделе 2.4 описан каждый из указанных компонентов.

В разделе 2.5 исследуется формальная логическая система $\mathcal{F}$, являющаяся основой для построения инвариантов и условий корректности. При этом инварианты и условия корректности являются формулами системы $\mathcal{F}$.

В разделе 2.6 приведен базовый алгоритм решения задачи генерирования инвариантов, сформулирована и доказана теорема о корректности базового алгоритма.

Базовый алгоритм генерирования инвариантов может быть представлен, как процесс последовательного обхода инструкций программы и генерирования в них инвариантов на основе ранее полученных инвариантов с учетом семантики проходимых инструкций.

Пусть задана программа P, и $G = (V, E)$ — ее управляющий граф, $start$ — начальная вершина. При этом предполагается, что

1) любая вершина графа G достижима из $start$;

2) $prev(start) = \emptyset$.

Для наглядности изложения предположим, что вершины графа покрашены в один из трех цветов: белый; серый; черный. Обход графа связан с последовательным их перекрашиванием. Изначально все вершины являются белыми, кроме начальной вершины $start$, покрашенной в серый цвет. В процессе работы алгоритма они могут изменять цвет следующим образом:

$$\text{белый} \rightarrow \text{серый} \rightarrow \text{черный}.$$

Множество черных вершин будем обозначать через Bl, множество серых вершин — через Gr.

Определение 3.1. *Раскраской* графа G будем называть пару (Bl, Gr), позволяющую определить цвет каждой его вершины.

Сформулируем правила перекрашивания вершин и укажем порядок их использования.

Правило A. Пусть v — белая вершина и пусть одна из соседних предшествующих вершин (то есть одна из вершин множества $prev(v)$) является черной. Тогда v может быть перекрашена в серый цвет (рис. 2.1).

Правило B. Пусть $v \in Gr$ и пусть $prev(v) \subseteq Bl$. Тогда v может быть перекрашена в черный цвет (рис. 2.2).

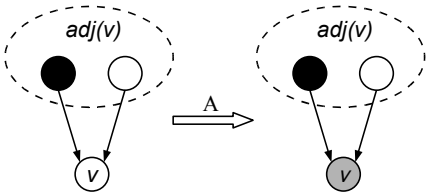

Рис. 2.1. Схема действия правила A.

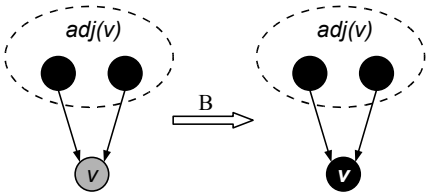

Рис. 2.2. Схема действия правила B.

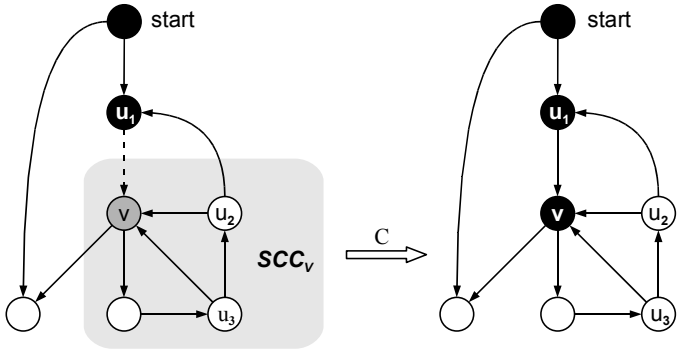

Рис. 2.3. Схема действия правила C. На рисунке обозначены множество ребер $Bl\text{-}Gr(v)$ (пунктирная стрелка) и подграф $SCC_v = SCC(v, Bl\text{-}Gr(v))$.

Для произвольного множества ребер $E' \subseteq E$ и произвольной вершины $v \in V$ рассматриваются:

$G'(E')$ — граф, получающийся из G удалением ребер множества E';

$SCC(v, E')$ — сильносвязанная компонента графа $G'(E')$, содержащая вершину v;

$Bl\text{-}Gr(v) = \{(u, v) \in E : \ u \in Bl\}$ — множество ребер, входящих в вершину v, другой конец которых окрашен в черный цвет.

Правило C. Пусть $v \in Gr$, и в графе $G'(Bl\text{-}Gr(v))$ не существует пути, ведущего из вершины $start$ в v. Тогда v может быть перекрашена в черный цвет (рис. 2.3).

Пусть E' — произвольное подмножество ребер, а V' — произвольное непустое подмножество вершин графа G. Рассмотрим граф $G'(E') = (V,\ E \setminus E')$. В графе $G'(E')$ существует такая сильносвязанная компонента R, что выполнены два свойства:

- множество $R \cap V'$ не пусто;
- для всех вершин u, v таких, что $u \in R \setminus V'$, а $v \in R \cap V'$, в графе $G'(E')$ не существует пути из вершины u в вершину v.

Обозначим такую компоненту через $SCC(V', E')$.

Правило D. Пусть $Gr \neq \emptyset$, $Bl\text{-}Gr(Gr) = \bigcup_{v \in Gr} Bl\text{-}Gr(v)$. Тогда вершины множества $SCC(Gr, Bl\text{-}Gr(Gr)) \cap Gr$ могут быть перекрашены в черный цвет (рис. 2.4).

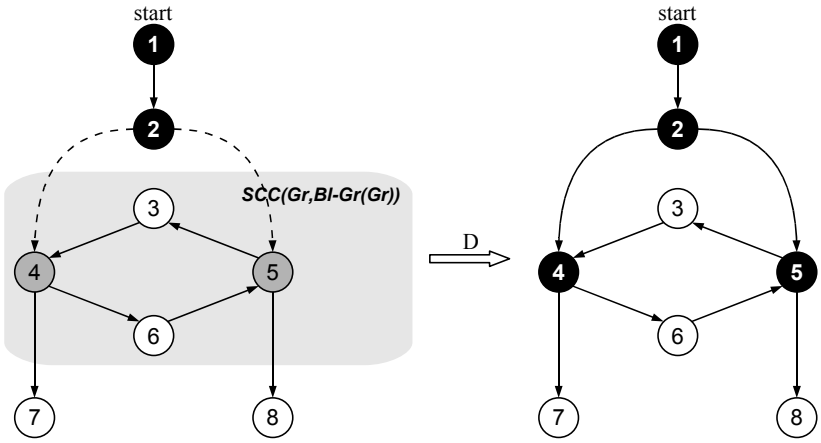

Рис. 2.4. Схема действия правила D. На рисунке пунктирными стрелками обозначены «черно-серые» ребра $Bl\text{-}Gr(Gr) = \{(2,4),\ (2,5)\}$, а также выделена компонента $SCC(Gr, Bl\text{-}Gr(Gr)) = \{3,4,6,5\}$.

Базовый алгоритм генерирования инвариантов изображен на рис. 2.5.

С каждым из правил B, C, D связана соответствующая процедура определения инвариантов в перекрашенных в черный цвет вершинах:

```
Generate-Invariants-B;
Generate-Invariants-C;
Generate-Invariants-D.
```

Сформулируем некоторые свойства базового алгоритма, изображенного на рис. 2.5, которые используются в дальнейшем при обосновании его корректности.

Лемма 2.1. Приведенный на схеме алгоритм завершает свою работу за конечное число шагов. По завершении его выполнения все вершины графа будут черными.

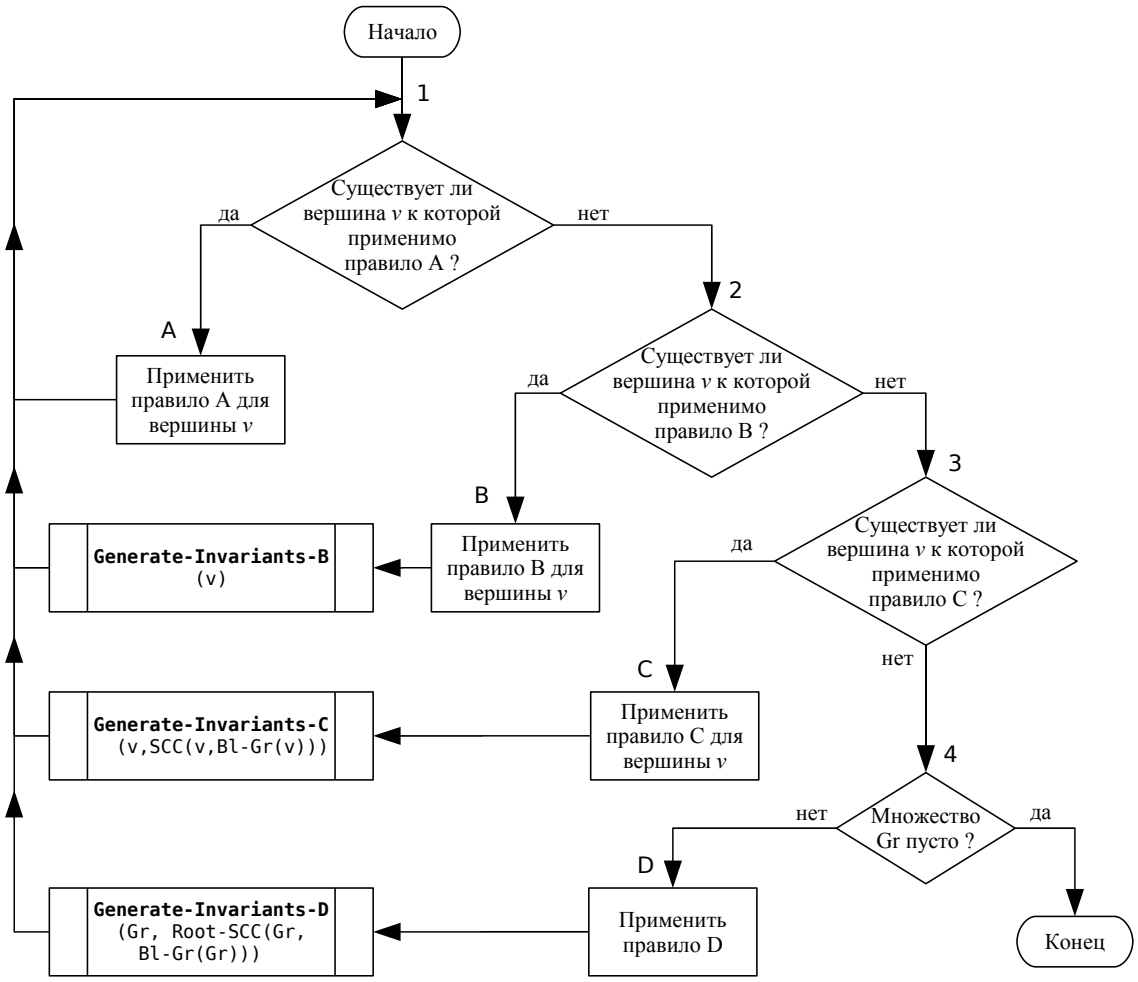

Рис. 2.5. Схема алгоритма решения задачи генерирования инвариантов. Определение применимости правил A, B, C, D и выделение соответствующих множеств выполняется на этапе обхода графа. Процедуры `Generate-Invariants-B`, `Generate-Invariants-C`, `Generate-Invariants-D` *отвечают за определение инвариантов в вершинах, перекрашенных в черный цвет.*

Лемма 2.2. Если при попадании в точку, обозначенную на схеме 2.6 цифрой 2, для вершины v выполнены условия применимости правила B, то любой путь (не обязательно простой) в графе G из вершины $start$ в вершину v имеет вид

$$start \to \ldots \to u \to v,$$

причем вершина u окрашена в черный цвет.

Лемма 2.3. Если при попадании в точку, обозначенную на схеме 2.6 цифрой 3, для вершины v выполнены условия применимости правила C, то любой путь в графе G из вершины $start$ в вершину v имеет вид

$$start \to \ldots \to u \to \underbrace{v \to \ldots \to v}_{C},$$

причем:

- вершина u окрашена в черный цвет;
- цикл C принадлежит компоненте $SCC(v,\ Bl\text{-}Gr(v))$.

Лемма 2.4. Если при попадании в точку, обозначенную на схеме 2.6 цифрой 4, выполнены условия применимости правила D (то есть $Gr \neq \emptyset$), то любой путь в графе G из вершины $start$ в любую вершину $v \in (Gr \cap SCC(Gr, Bl\text{-}Gr(Gr)))$ имеет вид

$$start \to \ldots \to v'' \to \underbrace{v' \to \ldots \to v}_{L'},$$

где $v' \in Gr,\ v'' \in Bl$, а путь L' принадлежит компоненте $SCC(Gr, Bl\text{-}Gr(Gr))$.

В подразделах 2.6.3 определена реализация процедур `Generate-Invariants-B`, `Generate-Invariants-C`, `Generate-Invariants-D`.

Пусть задана программа $P = (Var, Type, Stmt)$ и ее управляющий граф $G(P)$. Для произвольной вершины $v \in V$ введем следующие обозначения:

- $inv[v]$ — формула, выражающая инвариант в точке v;
- $h[v]$ — формула, выражающая априорно (наперед) заданный инвариант в точке v.

Реализация процедуры Generate-Invariants-B. Пусть v — перекрашиваемая вершина, $prev(v) = \{u_1, \ldots, u_k\}$ — множество предшествующих ей (черных) вершин. Тогда инвариант в точке v определяется следующим образом:

$$inv[v] = \Big(\bigvee_{i=1}^{k} update(inv[u_i],\ Stmt[u_i])\Big) \wedge h[v].$$

Реализация процедуры Generate-Invariants-C. Пусть v — перекрашиваемая вершина, $prev(v) = \{u_1, \ldots, u_k, u_{k+1}, \ldots, u_{k+l}\}$ — множество предшествующих соседних вершин, причем вершины $u_1, \ldots, u_k$ — черные, а вершины $u_{k+1}, \ldots, u_{k+l}$ — белые или серые.

Рассмотрим множество $Q = \{q_1, \ldots, q_s\}$ переменных, значения которых могут быть изменены в цикле $SCC(v, Bl\text{-}Gr(v))$. Тогда положим:

$$\begin{aligned}
inv_0[v] &= \Big(\bigvee_{i=1}^{k} update(inv[u_i],\ Stmt[u_i])\Big);\\
inv_1[v] &= update\text{-}var(inv_0[v],\ q_1);\\
inv_2[v] &= update\text{-}var(inv_1[v],\ q_2);\\
&\ldots\\
inv_s[v] &= update\text{-}var(inv_{s-1}[v],\ q_s);\\
inv[v] &= inv_s[v] \wedge h[v].
\end{aligned}$$

Реализация процедуры Generate-Invariants-D. Пусть $P = Gr \cap SCC(Gr, Bl\text{-}Gr(Gr))$ — множество перекрашиваемых в черный цвет вершин. Рассмотрим множество $Q = \{q_1, \ldots, q_s\}$ переменных, значения которых могут быть изменены в цикле $SCC(Gr, Bl\text{-}Gr(Gr))$. Тогда для каждого $v \in P$ in положим:

$$\begin{aligned}
inv_0[v] &= \Big(\bigvee_{(u,v) \in Bl\text{-}Gr(P)} update(inv[u],\ Stmt[u]) \Big); \\
inv_1[v] &= update\text{-}var(inv_0[v],\ q_1); \\
inv_2[v] &= update\text{-}var(inv_1[v],\ q_2); \\
&\ldots \\
inv_s[v] &= update\text{-}var(inv_{s-1}[v],\ q_s); \\
inv[v] &= inv_s[v] \wedge h[v].
\end{aligned}$$

В таких условиях оказывается возможным сформулировать и доказать теорему о корректности алгоритма генерирования инвариантов. Таким образом, свойство корректности алгоритма верификации IAL-программ, а вместе с ним — свойство полноты алгоритма обнаружения программных дефектов в C-программах, будет формально обосновано.

Теорема 2.13 Алгоритм генерирования инвариантов является корректным.

В разделе 2.7 предложен не нарушающий корректности базового алгоритма способ проверки индуктивных гипотез, позволяющий повысить точность решения задачи верификации.

Базовый алгоритм генерирования инвариантов, рассмотренный в разделе 2.6, является достаточно общим, и он не ориентирован непосредственно на решение указанных специальных задач. Однако, для проверки условий корректности, он позволяет использовать любые априорно (наперед) заданные инварианты. В связи с изложенным, для повышения эффективности работы алгоритма генерирования инвариантов предлагается подход, включающий следующие шаги.

1. Формирование некоторого количества «пробных» инвариантов (гипотез) в различных вершинах управляющего графа.

2. Проверка истинности гипотез с использованием описываемого далее метода. В результате, гипотезы, истинность которых подтвердится, будут добавлены к множеству априорно заданных инвариантов, в то время, как ложные гипотезы отбрасываются.

3. Запуск алгоритма генерирования инвариантов с обновленным множеством априорно заданных инвариантов и проверка истинности условий корректности.

При использовании данного подхода класс выводимых инвариантов может быть существенно расширен. Далее рассматривается простейший способ проверки истинности формируемых гипотез.

Пусть проверяемая в точке v гипотеза (полученная каким-либо способом) имеет вид формулы φ. Рассмотрим лишь случай, когда при применении алгоритма генерирования инвариантов вершина v перекрашивается в черный цвет по правилу C. Таким образом, вершина v является «входом» в цикл $SCC_v = SCC(v, Bl\text{-}Gr(v))$.

Проверка гипотезы осуществляется по принципу индукции. На первом шаге проверяется, что «начальные» значения переменных при входе в цикл SCC_v удовлетворяют формуле φ. Для достижения данной цели построим вспомогательную формулу:

$$inv_0[v] = \Big(\bigvee_{(u,v)\in Bl\text{-}Gr(v)} update(inv[u],\ Stmt[u]) \Big) \wedge h[v].$$

Для проверки базы индукции достаточно исследовать выполнимость формулы $(inv_0[v] \wedge \neg\varphi)$. Если указанная формула является невыполнимой, то базу индукции будем считать доказанной.

На втором шаге осуществляется проверка индуктивного перехода, для чего строится вспомогательный управляющий граф, состоящий из вершин множества SCC_v и новой вершины v' (рис. 2.6). Все ребра компоненты $SCC(v, Bl\text{-}Gr(v))$ сохраняются, за исключением ребер, входящих в вершину v. Каждое ребро указанной компоненты вида (w, v) заменяется новым ребром (w, v').

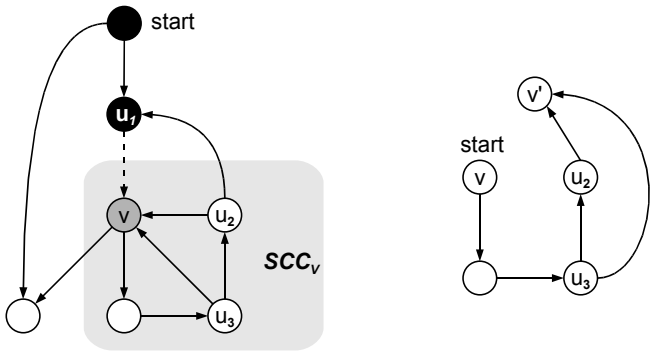

Рис. 2.6. Построение вспомогательного управляющего графа.

В построенном графе начальной объявляется вершина v, а конечной — вершина v'. Инструкции, приписанные вершинам SCC_v нового управляющего графа, остаются прежними.

Рассмотрим задачу анализа новой программы. Априорно заданные инварианты в каждой вершине сохраняются, однако в вершине v наперед задается инвариант φ. Задача анализа программы будет заключаться в проверке истинности формулы φ в точке v'.

Если можно гарантировать истинность формулы φ в вершине v', то индуктивную гипотезу будем считать доказанной.

В **третьей главе** рассматриваются аспекты программной реализации средства автоматизированного обнаружения дефектов в программах на языке C. Основным объектом исследования является программное средство «Статический анализатор программных дефектов» (далее — просто «Анализатор»).

В разделе 3.1 приведены требования к программному комплексу автоматизированного обнаружения дефектов в программах на языке C.

1. Метод должен гарантировать обнаружение всех дефектов.

2. Результатом анализа является список предупреждений об обнаруженных возможных дефектах, содержащий следующую информацию:

   - месторасположение «подозрительной» точки кода;
   - тип возможного дефекта;
   - условия, при которых возможно возникновение ошибки.

3. Метод должен быть применим для анализа крупных программных комплексов, содержащих до 500 тысяч строк кода.

4. Метод должен быть по-возможности наиболее эффективным. Критериями эффективности являются:

   - время работы;
   - количество ложных срабатываний на строку кода;
   - способность находить реальные дефекты в программах.

Схематично процедура статического анализа исходного кода C-программ и выявления потенциальных уязвимостей представлена на рис. 3.1. В ней можно выделить следующие этапы:

- этап лексического и синтаксического анализа;
- этап, осуществляющий трансляцию абстрактного синтаксического дерева в промежуточное представление, называемое IAL-представлением (от Intermediate Analyser Language);
- этап анализа программы в виде промежуточного представления.

Рассмотрим задачи, которые решаются на каждом из указанных этапов.

Лексический и синтаксический анализ представляет собой процесс сопоставления линейной последовательности символов программы с формальной грамматикой языка C (согласно стандарта ISO/IEC 9899:1999 «Programming

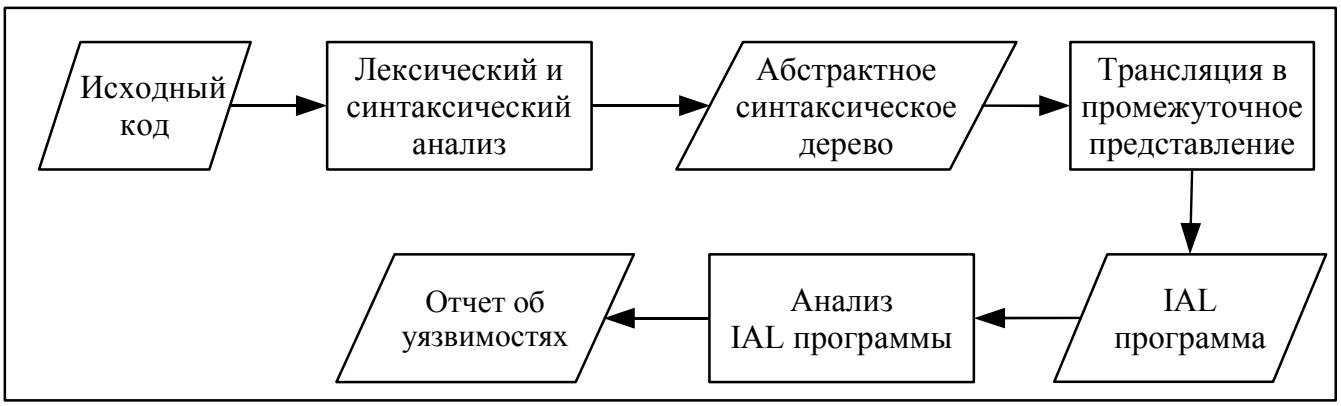

Рис. 3.1. Этапы статического анализа программы: лексическй и синтаксический анализ, трансляция в промежуточное представление, анализ IAL-программы.

Languages — C»). Результатом является дерево разбора или абстрактное синтаксическое дерево.

Трансляция абстрактного синтаксического дерева в промежуточное представление. Структура абстрактного синтаксического дерева, соответствующего формальной грамматике языка C, является очень сложной. По этой причине перед выполнением основной части анализа абстрактное синтаксическое дерево переводится в более простой формат, называемый «промежуточным представлением». Промежуточное представление является программой, семантически эквивалентной исходной, написанной на специально разработанном языке IAL (от Intermediate Analyser Language).

Этап трансляции абстрактного синтаксического дерева в IAL-программу носит, в основном, рутинный характер. К преобразованиям, выполняемым на данном этапе, относятся:

- устранение неявных преобразований типов в соответствии с правилами стандарта C99;
- разворачивание сложных выражений (преобразование сложных выражений в последовательность простых с использованием временных переменных);
- вычисление и подстановка константных выражений (включая выражения, содержащие операторы `sizeof`);
- разворачивание структур управления, таких как циклы и условные выражения;
- разворачивание вызовов функций.

Анализ программы на языке промежуточного представления. Анализ IAL-программы является главным этапом разрабатываемого метода. Основными задачами, которые решаются на данном этапе, являются: проведение статического анализа программы; выявление свойств программы, связывающих значения пере-

менных и состояние памяти в различных точках выполнения; проверка корректности операций в исходном коде; сбор и вывод сообщений о возможных дефектах. Основная часть диссертационной работы посвящена описанию используемых на данном этапе алгоритмов и обоснованию их корректности.

В заключительной части главы рассматриваются способы повышения надежности разработанного прототипа программного комплекса, а также обсуждаются вопросы соответствия разработанного прототипа программного комплекса формальным алгоритмам, представленным в предыдущей главе.

В **четвертой главе** исследуется эффективность разработанного программного комплекса для автоматизированного обнаружения программных дефектов. Объектом исследования является программное средство «Статический анализатор программных дефектов» (далее — просто «Анализатор»). Примеры, на которых производится тестирование «Анализатора», разделены на две группы, к которым относятся модельные примеры и реальные приложения.

Под *модельным примером* будем понимать пару небольших программ на языке C, одна из которых содержит некоторую разновидность программного дефекта, а в другой этот дефект устранен. Каждый модельный пример ориентирован на тестирование определенной функциональной возможности «Анализатора». В разделе 4.1 рассматриваются модельные примеры `stack-overflow`, `heap-overflow`,направленные, соответственно, на определение возможности выявления переполнения буфера в стеке, переполнение буфера в куче.

В разделе 4.2 диссертационной работы исследуется применение «Анализатора» к программе `xorg-server` (основной части графической подсистемы X.Org). Программа `xorg-server` выбрана в качестве примера для исследования и демонстрации масштабируемости разработанного прототипа. Ее исходный код содержит 552 модуля с исходным кодом, что составляет около 470 тыс. строк кода.

Далее рассматриваются следующие величины:

- LOC (от Lines Of Code) — количество строк кода в исследуемой программе на языке C;
- PVL (от Potentially Vulnerable Locations) — количество потенциально уязвимых участков кода;
- WRN (от WaRNings) — количество выданных предупреждений;
- TP (от True Positives) — количество реальных дефектов, обнаруженных «Анализатором»;
- FP (от False Positives) — количество ложных срабатываний «Анализатора».

Для оценки эффективности применения «Анализатора» к программе `xorg-server` введем две дополнительные величины, а именно — *плотность ложных срабатываний* (DFP) и *эффективность верификации* (EFV).

Определение 4.1. Под *плотностью ложных срабатываний* будем понимать относительную величину

$$DFP = \frac{FP}{LOC} \cdot 1000.$$

Она определяет среднее количество ложных срабатываний «Анализатора» на каждые 1000 строк исследуемой программы.

Определение 4.2. Под *эффективностью верификации* будем понимать относительную величину

$$EFV = \frac{PVL - FP}{PVL} \cdot 100\%.$$

Она показывает процентное соотношение числа ложных срабатываний «Анализатора» среди общего количества потенциально уязвимых участков.

Полный синтаксический разбор исходного кода `xorg-server` и преобразование абстрактного синтаксического дерева в промежуточное представление заняло около 80 минут. Всего в исходном коде программы присутствовало 29 310 потенциально уязвимых участков, 4 058 из которых были в результате анализа отмечены, как возможные дефекты. Для остальных 25 252 участков «Анализатором» была доказана их корректность. В следующих таблицах представлены результаты проведенного эксперимента:

№ ит.	LOC	PVL	WRN	TP	FP	DFP	EFV
1	469 537	29 310	4 573	$TP_1 \geq 0$	$FP_1 \leq 4573$	$DFP_1 \leq 9.74$	$EFV_1 \geq 84.40\%$
2	469 537	29 310	4 058	$TP_2 \geq 0$	$FP_2 \leq 4058$	$DFP_2 \leq 8.64$	$EFV_2 \geq 86.16\%$
3	469 537	29 310	4 058	$TP_3 \geq 0$	$FP_3 \leq 4058$	$DFP_3 \leq 8.64$	$EFV_3 \geq 86.16\%$

Таблица 4.1. Результат анализа программы `xorg-server`. Функциональные характеристики.

№ ит.	Время, мин.	Память, МБ
1	315	1 819
2	324	1 830
3	324	1 832

Таблица 4.2. Результат анализа программы `xorg-server`. Показатели ресурсоемкости.

Замечание. В связи с тем, что ручной анализ кода программы `xorg-server` в данной работе не проводился (по причине большого объема исследуемого кода), точные значения величин TP, FP, а следовательно, и DFP, и EFV на итерациях 1, 2, 3 неизвестны. По этой причине для величин TP и FP используются лишь их оценки $0 \leq TP$, $FP \leq WRN$. Исходя из этих неравенств, а также принимая во внимание тождество $TP + FP = WRN$, получаются оценки для величин DFP, EFV. Указанные оценки для величин TP, FP, DFP, EFV на каждой из трех проведенных итераций представлены в таблице 4.1 вместо точных значений.

Во втором эксперименте автором было преднамеренно внесено два реальных дефекта в исходный код программы `xorg-server`. Операции, корректность которых ранее была доказана «Анализатором», были заменены на некорректные.

Затем проект был повторно исследован «Анализатором». Как видно из результатов эксперимента (табл. 4.3, 4.4), заключительный отчет об уязвимостях содержит ровно на 2 предупреждения больше, чем в предыдущем эксперименте, причем обе некорректные операции были успешно обнаружены. Результаты эксперимента свидетельствуют о возможности «Анализатора» находить реальные уязвимости, в том числе, при анализе больших программ.

№ ит.	LOC	PVL	WRN	TP	FP	DFP	EFV
1	469 537	29 310	4 576	$TP_1 \geq 2$	$FP_1 \leq 4574$	$DFP_1 \leq 9.74$	$EFV_1 \geq 84.40\%$
2	469 537	29 310	4 060	$TP_2 \geq 2$	$FP_2 \leq 4058$	$DFP_2 \leq 8.64$	$EFV_2 \geq 86.16\%$
3	469 537	29 310	4 060	$TP_3 \geq 2$	$FP_3 \leq 4058$	$DFP_3 \leq 8.64$	$EFV_3 \geq 86.16\%$

Таблица 4.3. Результат анализа модифицированной программы `xorg-server`*. Функциональные характеристики.*

№ ит.	Время, мин.	Память, МБ
1	315	1 819
2	324	1 830
3	324	1 832

Таблица 4.4. Результат анализа модифицированной программы `xorg-server`*. Показатели ресурсоемкости.*

Результаты проведенных экспериментов позволяют слелать следующие выводы.

Проведенные эксперименты на модельных программах свидетельствуют о том, что разработанный прототип в полной мере удовлетворяет предъявленным к нему функциональным требованиям. Для всех тестовых программ, содержащих тот или иной тип дефекта, указанные дефекты были «Анализатором» обнаружены. Для всех корректных программ «Анализатором» была доказана их корректность. Отметим, что обзор существующих средств аудита С-программ, представленный в главе 1, показал, что такие средства уже на модельных примерах зачастую выдают ложные срабатывания или пропускают реальные дефекты.

Проведенные эксперименты свидетельствуют о применимости на практике разработанного прототипа для анализа крупных программных проектов: время анализа всего проекта является приемлемым (не превышает 20 часов); плотность количества ложных срабатываний не превышает 10 (на 1000 строк исходного кода); эффективность верификации составила около 86%. Отметим, что в известных автору аналогах «Анализатора» по функциональному значению эффективность верификации достигала лишь 52%.[19]

[19]Ellenbogen R. Fully Automatic Verification of Absence of Errors via Interprocedural Integer Analysis. Master's thesis, School of Computer Science, Tel-Aviv University, Tel-Aviv, Israel, December 2004.

В заключении перечислены основные результаты диссертационной работы.

1. Систематизирована исследуемая предметная область, введено понятие программного дефекта, предложена классификация существующих программных дефектов C-программ.

2. Предложен метод автоматизированного обнаружения дефектов в программах на языке C, основанный на преобразовании исходной программы в промежуточный формат «IAL» (от Intermediate Analyser Language) с последующей верификацией полученной IAL-программы.

3. Разработан язык IAL промежуточного представления C-программ, описаны его синтаксис и семантика, разработана математическая модель языка IAL, в рамках которой формализована задача верификации IAL-программ.

4. Разработан базовый алгоритм верификации программ в формате промежуточного представления, сформулирована и доказана его корректность, предложен не нарушающий корректности алгоритма способ проверки индуктивных гипотез, позволяющий повысить точность анализа.

5. Создан прототип программного комплекса автоматизированного обнаружения дефектов в программах на языке C, эффективность которого продемонстрирована как на ряде специально подобранных модельных примеров, так и на примере программы `xorg-server` — основной части подсистемы управления графикой в UNIX-подобных операционных системах.

Благодарности. Автор выражает глубокую благодарность своему научному руководителю доктору физико-математических наук, профессору Валерию Александровичу Васенину за постановку задачи и постоянное внимание к работе. Автор также благодарит К. А. Шапченко и О. О. Андреева за участие в разработке прототипа программного комплекса автоматизированного обнаружения уязвимостей в программах на языке C.

Список литературы

[1] Галатенко А. В., Пучков Ф. М., Шапченко К. А. *Способ анализа программ на наличие угроз переполнения буферов* // Информационная безопасность регионов России (ИБРР-2003): Материалы конференции, — Санкт-Петербург, 2003. — С. 33. (Ф. М. Пучкову принадлежат результаты по описанию метода статического анализа блок-схем).

[2] Пучков Ф. М., Шапченко К. А. *К вопросу о выявлении возможных переполнений буферов посредством статического анализа исходного кода программ.* // Материалы конференции МаБИТ-04. — М.: МЦНМО, 2005. —

С. 347–359. (Ф. М. Пучкову принадлежат результаты по описанию метода статического анализа блок-схем).

[3] Пучков Ф. М., Шапченко К. А. *Статический метод анализа программного обеспечения на наличие угроз переполнения буферов.* // Программирование. — 2005. — N.4. — С. 19–34. (Ф. М. Пучкову принадлежат результаты по описанию математической модели и алгоритмов статического анализа блок-схем, доказательствам основных утверждений).

[4] Puchkov, F., Shapchenko, K. *Static Analysis Method for Detecting Buffer Overflow Vulnerabilities* // Programming and Computer Software, Volume 31, Number 4, July 2005, pp. 179–189(11).

[5] Пучков Ф. М., К. А. Шапченко, О. О. Андреев. *К созданию автоматизированных средств верификации программного кода.* // Материалы Второй международной научной конференции по проблемам безопасности и противодействия терроризму. Пятая общероссийская научная конференция «Математика и безопасность информационных технологий» (МаБИТ-06). МГУ имени М.В. Ломоносова, 25-26 октября 2006 г. — М.: МЦНМО, 2007. — С. 401–439. (Ф. М. Пучкову принадлежат результаты систематизации рассматриваемой предметной области, в том числе классификации существующих уязвимостей программного обеспечения; исследования и анализа существующих подходов к верификации программного обеспечения; описания метода трансляции дерева разбора С-программы в блок-схему; описания метода статического анализа блок-схемы и проверки условий корректности).

[6] Пучков Ф. М. *Средства аудита программного обеспечения на предмет обнаружения уязвимостей.* // Критически важные объекты и кибертерроризм. Часть 2. Аспекты программной реализации средств противодействия. / О. О. Андреев и др. Под ред. В. А. Васенина. — М.: МЦНМО, 2008, 607 с. — С. 375–455.

[7] *Способ генерации баз данных для систем верификации программного обеспечения распределенных вычислительных комплексов и устройство для его реализации.* / Пучков Ф. М., Шапченко К. А. // Патент на изобретение № 2364929. — 2009.

[8] *Способ генерации баз знаний для систем верификации программного обеспечения распределенных вычислительных комплексов и устройство для его реализации.* / Пучков Ф. М., Шапченко К. А. // Патент на изобретение № 2364930. — 2009.

[9] *Способ генерации баз данных и баз знаний для систем верификации программного обеспечения распределенных вычислительных комплексов и*

устройство для его реализации. / Пучков Ф. М., Шапченко К. А. // Патент на изобретение № 2373569. — 2009.

[10] *Способ верификации программного обеспечения распределенных вычислительных комплексов и система для его реализации.* / Пучков Ф. М., Шапченко К. А. // Патент на изобретение № 2373570. — 2009.