АНАЛИЗ И ТРАНСФОРМАЦИЯ ПРОГРАММ

С.С. Гайсарян, В.П. Иванников, А.И. Аветисян

Институт системного программирования РАН
109004, г. Москва, ул. Б. Коммунистическая, д. 25
(ssg@ispras.ru) (ivan@ispras.ru) (arut@ispras.ru)

Аннотация. Актуальность данного обзора состоит в том, что развитие многих современных направлений компьютерной индустрии невозможно представить без существенного развития методов анализа и трансформации программ. В обзоре рассматриваются наиболее значимые направления с анализом существующих технологий, исследований по разработке новых технологий и тенденций развития. Кроме того, в обзоре описываются фундаментальные понятия анализа и трансформации программ с учетом последних достижений в этой области и инфраструктура необходимая для организации исследований и реализации соответствующих технологий.

1. Введение

Данный обзор имеет целью рассмотреть новые результаты и направления в широкой области исследований и разработок, связанных с анализом и трансформацией программ. Первые исследования по трансформации программ были связаны с проблемами разработки оптимизирующих компиляторов, но впоследствии выяснилось, что эти исследования и технологии, использующие их результаты, могут успешно применяться не только при разработке методов компиляции и оптимизации программ, но и во многих других областях разработки программного обеспечения (ПО). В настоящее время анализ и трансформация программ успешно применяются в таких областях как обратная инженерия, синтез программ по их спецификациям, восстановление ПО, исследование и обеспечение различных аспектов безопасности ПО, рефакторинг, исследование и преобразование бинарного кода. Во всех перечисленных областях применение систем трансформации программ способствует повышению производительности труда разработчиков ПО, обеспечивая возможность работы с ПО на более высоком уровне абстракции, повышая удобство сопровождения и возможности повторного использования разработанного ПО.

В настоящее время исследования по методам анализа и трансформации программ развиваются особенно интенсивно, на их основе разрабатываются и успешно внедряются новые прогрессивные технологии разработки и верификации ПО. В данном обзоре рассматриваются следующие три категории исследований: (1) исследования по разработке и обоснованию новых подходов к трансформации программ, (2) исследования и практические работы по применению методов анализа и трансформации программ в новых технологиях, связанных с разработкой ПО; (3) создание новых инструментов и систем автоматизации разработки ПО, в которых используются методы анализа и трансформации программ.

Следует отметить, что методы анализа и трансформации программ применяются не только к исходному коду компонентов ПО, но и к их двоичному представлению. Поэтому в обзор включены работы по статическому и динамическому анализу двоичного кода, а также по методам дизассемблирования и декомпиляции.

Обзор имеет следующую структуру. В разделе 2 рассматриваются основные классы трансформаций программ, на основе которых реализуются остальные трансформации, и внутренние представления программ, удобные для реализации введенных классов трансформаций. В разделе 3 наиболее актуальные проблемы оптимизации программ, направленные на решение новых задач, связанных с учетом специфики современных платформ и тенденциями развития их архитектуры. Раздел 4 посвящен рассмотрению различных аспектов безопасности программного обеспечения и методов анализа и трансформации программ, применяемых для решения проблем, связанных с обеспечением безопасного функционирования ПО. Сюда относятся такие проблемы, как обнаружение дефектов в компонентах ПО, обнаружение недокументированных свойств у компонентов ПО, защита ПО от обратной инженерии и др. В разделе 5 обсуждаются модельно-ориетнированные подходы в разработке ПО, позволяющие свести до минимума «разрыв» между абстрактной моделью ПО и ее реализацией. Генерация выполнимого кода по моделям позволяет быстро создать прототип на ранних этапах проектирования системы. Выполнение такого прототипа помогает обнаружить архитектурные ошибки и, за счет раннего обнаружения, заметно сократить их стоимость. Наконец, в разделе 6 рассматриваются проблемы обратной инженерии и верификации ПО. Эти методы были особенно популярны в 90-е годы прошлого века в связи с проблемой переноса унаследованного кода (т.е. программ, разработанных в прошлые годы, но продолжающих интенсивно эксплуатироваться) на новую аппаратуру. Успехи в области обратной инженерии привели к необходимости разработки методов защиты ПО от обратной инженерии (этот аспект безопасности ПО рассматривается в разделе 4). В настоящее время обратная инженерия направлена на обнаружение дефектов в программах как разрабатываемых, так и готовых и на оптимизацию архитектур программных систем (в частности, это рефакторинг).

2. Теория и техника трансформации программ

Трансформация и анализ программ выполняются на различных уровнях: *на уровне исходного кода, на промежуточном уровне*, когда программа представляется как последовательность инструкций, похожих на ассемблерные, но сохраняет переменные (а не регистры и ячейки), имеющие тип, *на ассемблерном уровне, на уровне бинарного кода*. Перечисленные представления в том или ином виде реализованы во всех компиляторах. Так в компиляторе *Gcc* [1] внутреннее представление исходного уровня называется *Generic*, внутреннее представление промежуточного уровня – *Gimple,* внутреннее представление ассемблерного уровня – *RTL* [2]. Следует отметить, что в компиляторах уже давно решены почти все проблемы, связанные с построением внутреннего представления первых двух уровней, но это не значит, что реализация соответствующего ПО не составляет трудностей. Более того, разработка и реализация фаз компилятора, занимающихся построением этих представлений, требует значительных трудозатрат. Поэтому в исследовательских (а иногда и в коммерческих) проектах обычно используются соответствующие фазы свободно распространяемых компиляторов. Обычно их берут из *Gcc.*

Это связано и с тем, что в последнее время вокруг внутренних представлений различного уровня разрабатываются инфраструктуры, включающие *API*, упрощающие разработку программ, реализующих соответствующие трансформации (так, вокруг *Gimple* реализована инфраструктура *TreeSSA* [4], пользующаяся заслуженной популярностью у программистов). Вокруг нового представления *LLVM* [3], используемого в компиляторах компании *Apple,* которое разработано по мотивам *JavaVM* и имеет более низкий уровень, чем *Gimple*, но более высокий уровень, чем *RTL,* тоже реализована обширная инфраструктура, быстро завоевывающая популярность.

Наиболее предпочтительными для реализации и исследования новых методов и алгоритмов анализа и трансформации программ в настоящее время являются внутреннее представление *Gimple* и инфраструктура *TreeSSA* компилятора *Gcc*, исходя из следующих соображений:

- Инфраструктура *TreeSSA* содержит богатый набор реализованных алгоритмов и эвристик анализа и трансформации программ (анализ потока управления, анализ указателей, *SSA*-представление для регистровых переменных, памяти и циклов, анализ индуктивных переменных, межитерационные зависимости по данным и т.п.), над которыми легче строить новые трансформации.
- Промежуточное представление *Gimple* является языково-независимым и машинно-независимым, а следовательно, трансформации программ, реализованные на нем с помощью *Tree SSA*, будут работать для всех языков и целевых архитектур, поддерживаемых компилятором *Gcc*. В настоящий момент поддерживаются языки *C, C++, Fortran* 95*, Java, Ada* и некоторые другие, а также большинство популярных архитектур (*Intel x86, IBM PowerPC, Intel Itanium, Sun SPARC, ARM* и т.д.).
- Компилятор *Gcc* является стабильным компилятором промышленного уровня и используется как стандартный компилятор в *UNIX*-подобных системах, а также популярных дистрибутивах *ОС Linux*. Инфраструктура *TreeSSA* доступна в *Gcc* уже более трех лет и также является стабильной и хорошо отлаженной.
- Работа над инфраструктурой, промежуточным представлением, а также другими частями компилятора (переводчиком с языков программирования в промежуточное представление и кодогенератором) ведется группой высококвалифицированных программистов из *Red Hat, Novell, IBM, Intel, AMD, HP* и ряда других компаний. Задача поддержки альтернативной инфраструктуры, сравнимой по набору возможностей и стабильности с *TreeSSA,* требует очень больших затрат труда высококвалифицированных программистов и представляется невыполнимой.

Таким образом, использование внутренних представлений *Gcc* и инфраструктуры *TreeSSA* позволяет сконцентрироваться на реализации самой трансформации, не тратя времени на написание и доводку необходимой

компиляторной инфраструктуры, а также применить реализованную трансформацию к широкому классу программ.

Далее в разделе будут рассмотрены некоторые методы анализа и трансформации программ, завоевавшие популярность в последнее время и нашедшие применение в новых исследованиях и программных продуктах: трансформации, основанные на распространении атрибутов и на решении задач о потоке данных, анализ алиасов, а также подходы к обнаружению и устранению избыточного кода (нумерация значений, частичная избыточность) [5,6].

2.1. Типы трансформаций программ

В этом разделе мы будем рассматривать часто используемые трансформации программ, выполняющиеся на промежуточном уровне. Трансформации, выполняющиеся на ассемблерном уровне и необходимые для кодогенерации (выбор команд, планирование команд, распределение регистров), не рассматриваются. Некоторые вопросы планирования команд для современных архитектур будут рассмотрены в разделе 3.

Можно выделить следующие типы анализов и трансформаций: выполняющиеся над отдельными инструкциями, *локальные* – выполняющиеся в пределах одного базового блока, *глобальные* – выполняющиеся на всей процедуре, *межпроцедурные* – выполняющиеся на нескольких процедурах [5, с.705, 1061]. Примером трансформации, выполняющейся над отдельной инструкцией, является сворачивание констант (вычисление выражений, все элементы которых константы). Как правило, эта трансформация используется вместе с распространением констант и копий, хотя иногда ее применяют и отдельно.

В современных компиляторах большинство трансформаций являются глобальными. Некоторые трансформации могут выполняться как локально, так и глобально, в зависимости от того, на какой области программы выполнялся анализ. Среди глобальных трансформаций выделяют следующие:

- основанные на *анализе потока данных* (удаление частичной избыточности, унификация выражений) [5, §9.5];

- основанные на *распространении* атрибутов (удаление мертвого кода, распространение констант и копий, распространение диапазонов значений) [5, §§9.1, 9.4; 6, §12.5];
- основанные на *SSA*-представлении [6, §8.11] (нумерация значений [6, §12.4]);
- цикловые – трансформации, применяемые лишь к циклическим участкам кода программы [6, глава 14]. Можно выделить две группы цикловых трансформаций: (1) основанные на выделении *индуктивных* переменных (т.е. таких, которые на каждой итерации цикла изменяются на постоянную величину) – вынос инвариантных выражений из циклов [6, §13.2], сокращение сложности вычислений (strength reduction), и (2) преобразования циклов и гнезд циклов, как правило, для достижения наилучшего использования кэша [5, глава 11]. Преобразования циклов, применяемые для распараллеливания программ, представляют собой отдельный большой класс трансформаций, рассмотрение которых выходит за рамки настоящего обзора.

Межпроцедурные трансформации можно разделить на два класса: глобальные трансформации, расширившие контекст применения до нескольких процедур (так, существует межпроцедурный анализ алиасов, распространение констант, распределение регистров), и трансформации, возможные только на межпроцедурном уровне – здесь наиболее ярким примером является *встраивание процедур* и *оптимизация хвостовой рекурсии* [6, §§15.2, 15.3]. Кроме того, межпроцедурные трансформации могут применяться как к процедурам из одного модуля, так и к процедурам из всех модулей программы (например, при связывании объектных модулей в один исполняемый файл программы). В последнем случае межпроцедурные трансформации особенно эффективны, т.к. опираются на максимально точную информацию о программе, однако время их выполнения может оказаться неприемлемо большим для промышленного применения.

При дальнейшем изложении мы будем пользоваться понятием графа потока управления процедуры [5, §8.4]. *Базовым блоком* будем называть четверку ⟨*Bi*, *P*, *In*, *Out*⟩, где – номер (имя) базового блока, *P* – последовательность инструкций программы, составляющих блок *Bi* (по определению *P* может начать выполняться

только с первой, а закончить – только на последней инструкции), *In* – множество входных переменных (переменных, определенных вне блока *Bi*, но используемых в нем), *Out* – множество выходных переменных (переменных, определенных в блоке *Bi*, но используемых вне него); отметим, что в отечественных публикациях базовые блоки иногда называются линейными участками. *Графом потока управления* для некоторой процедуры программы будем называть четверку ⟨*V, E, Entry, Exit*⟩, где *V* – множество вершин графа, соответствующих базовым блокам процедуры, $E \subseteq V \times V$ – множество дуг, соответствующих передачам управления между базовыми блоками, *Entry* и *Exit* – две специально выделенных вершины, соответствующие базовым блокам, представляющим соответственно точку входа управления и точку выхода управления из процедуры.

2.2. Трансформации, основанные на анализе потока данных

Задачей анализа потока данных является получение информации о том, каким образом некоторая процедура обрабатывает свои данные. Например, на основе информации о том, что в некоторой точке (инструкции) процедуры значение некоторой переменной всегда постоянно, можно заменить использования переменной в этой инструкции на значения соответствующей константы. Как правило, анализом рассматривается множество некоторых объектов (констант, переменных, выражений и т.п.), при этом необходимо для каждой точки процедуры определить множество «корректных» в некотором смысле объектов.

Рассмотрим в качестве примера задачу *достигающих определений* (*reaching definitions*) [5, пункт 9.2.4]. Определением переменной назовем присваивание этой переменной некоторого значения. Некоторое определение достигает данной точки процедуры, если существует путь выполнения между определением и данной точкой такой, что в этой точке переменная все еще содержит значение, присвоенное при определении. Для решения этой задачи рассмотрим граф потока управления программы, и для каждой вершины *i* графа определим четыре множества определений переменных: *GEN*(*i*) –определения, которые генерируются в базовом блоке *Bi*; *KILL*(*i*) – определения, которые переопределяются в базовом блоке *Bi* (т.е. той же переменной

присваивается иное значение); *REACHin*(*i*) и *REACHout*(*i*) – определения, достигающие соответственно входа и выхода в базовый блок *Bi*. Тогда для каждой вершины графа потока управления будут верны следующие уравнения:

$$REACHout(i) = (REACHin(i) - KILL(i)) \cup GEN(i),$$

$$REACHin(i) = \bigcup_{j \in Pred(i)} REACHout(j).$$

Систему уравнений для всей процедуры можно решить, вычислив множества *GEN* и *KILL* для каждой вершины *Bi* по виду инструкции, а также положив для каждой вершины $REACHin(i)=\varnothing$. После этого достаточно итеративно вычислять множества *REACHin* и *REACHout* для каждой вершины графа в порядке сверху вниз до тех пор, пока они не перестанут изменяться. Полученные множества *REACHin* дадут решение исходной задачи о достигающих определениях.

Также результаты анализа достигающих определений удобно представлять с помощью *DU- и UD-цепочек* [6, §8.10] – структур данных, связывающих определение некоторой переменной с его использованиями, а также использование переменной с достигающими его определениями. *DU*/*UD*-цепочки являются эффективным способом представления информации о потоке данных через переменные программы, поэтому на их основе часто строят трансформации, как показано в следующем разделе.

Аналогичным образом можно формулировать и более сложные задачи потока данных: необходимо задать множества объектов *GEN* и *KILL* для каждой вершины графа потока управления процедуры, а также определить, как связаны множества интересующих объектов *DATAin* (на входе) и *DATAout* (на выходе из вершины). Однако описанный выше итерационный процесс сойдется и даст требуемое решение не для любой задачи. Достаточным условием является монотонность функций потока, описываемых уравнениями для множеств *DATAin*.

Различают *прямую* и *обратную* задачи анализа потока данных в зависимости от того, связывается ли входное множество вершины с выходными множествами её предков либо выходное множество вершины с входными множествами её потомков. Кроме того, в прямой задаче обновление множеств происходит сверху вниз по графу потока управления, начиная с вершины *Entry*, а в обратной – снизу вверх, начиная с вершины *Exit*. Например, при удалении частичной избыточности определяются

множества избыточных выражений в каждой точке процедуры, для чего решается четыре различных задачи потока данных, две из которых прямые и две – обратные [5, §9.5].

2.3. Трансформации, основанные на распространении атрибутов

Некоторые трансформации, пользующиеся анализами потока данных, удобнее формулировать не через решение систем уравнений потока, а с помощью распространения необходимых атрибутов по графу потока управления и *DU*/*UD*-цепочкам. При этом используются результаты уже выполненного анализа достигающих определений, а также можно более гибко управлять распространением атрибутов для ускорения сходимости.

Примером такой трансформации является удаление мертвого кода [5, пункт 9.1.6]. *Мертвой* называется инструкция, вычисляющая значения, не используемые на любых путях выполнения программы, начиная с данной инструкции. При удалении мертвого кода строится множество *полезных* инструкций (т.е. влияющих на результаты выполнения программы). Построение начинается с инструкций, заведомо изменяющих окружение программы (операторы возврата значений из функций, записи глобальных переменных и т.п.), далее помечаются как полезные инструкции, вычисляющие операнды уже помеченных (в том числе инструкции передачи управления) и т.д. Инструкции, не попавшие в окончательное множество полезных, удаляются.

Как видно, удаление мертвого кода состоит в распространении атрибута полезности по инструкциям программы и последующем удалении бесполезных инструкций. Вообще, трансформации, основанные на распространении атрибутов, удобно делить на два этапа. На первом этапе для каждой инструкции программы вычисляются интересующие нас атрибуты через *распространение* уже вычисленных атрибутов по графу потока управления программы, а также по *DU*-цепочкам (от определений переменных к использованиям) до того момента, пока вычисленные атрибуты не перестанут изменяться. На втором этапе выполняется трансформация инструкций с «интересными» атрибутами.

Так, в инфраструктуре *TreeSSA* компилятора *Gcc* распространение копий, констант и диапазонов значений выполняется с помощью единой инфраструктуры *SSA-распространителя* [7]. Необходимо лишь определить, какие инструкции необходимо посетить при распространении, а также реализовать процедуру, вычисляющую необходимый атрибут по инструкции и определить, изменился ли атрибут со времени последнего вычисления. После этого обход необходимых инструкций будет выполнен автоматически. По окончании вычисления атрибутов происходит собственно трансформация: при распространении констант и копий – подстановка найденных констант, упрощение выражений, удаление мертвых копий; при распространении диапазонов значений – упрощение заведомо истинных/ложных условий и удаление мертвого кода.

2.4. Анализ алиасов

При описании задач анализа потока данных, рассмотренных выше, предполагалось, что по некоторой инструкции можно легко определить множество переменных, определяемых и используемых ею. Для большинства процедурных языков программирования это неверно из-за присутствия *алиасов* [6, глава 10]. Говорят, что если некоторую абстрактную ячейку памяти в программе можно именовать несколькими способами, то эти имена являются *алиасами*. Например, если `p=&x`, то `x` и `*p` являются алиасами. Источники алиасов зависят от языка программирования, но, как правило, большинство алиасов порождается указателями (объектными ссылками).

Анализ алиасов призван ответить на вопрос, какие имена являются алиасами. При наличии указателей и адресной арифметики статический анализ алиасов является алгоритмически неразрешимой задачей. Тем не менее, существуют алгоритмы анализа алиасов, строящие приблизительные множества объектов, являющимися алиасами. Так, различают анализ алиасов, основанный на *типах* (например, в корректной программе на языке C указатели типов `int*` и `float*` не могут указывать на один и тот же объект); анализ указателей (или points-to анализ), вычисляющий множество переменных, на которые потенциально может указывать некоторый указатель и некоторые другие.

Хороший анализ алиасов является необходимым в оптимизирующем компиляторе, так как при его отсутствии приходится считать, например, что запись по указателю может модифицировать любую переменную, что не позволяет проводить практически никакие трансформации. Поэтому стараются улучшить точность анализа алиасов, увеличивая область анализа (например, выполняя межпроцедурный анализ) либо вводя дополнительные средства в язык программирования, с помощью которых можно дать указания компилятору (например, в языке C указатели-параметры функции, помеченные ключевым словом `restrict`, не могут ссылаться на один и тот же объект [8, пункт 6.7.3.1]).

2.5. Трансформации, основанные на SSA-представлении

SSA-представление (статическое представление с единственными присваиваниями [6, §8.11; 9]) является промежуточным представлением программы, в котором каждой переменной присваивается значение ровно один раз. Такое представление является удобным для оптимизаций, т.к. *UD*-цепочки могут быть представлены явно – достигающее определение всегда будет ровно одно. Для построения *SSA*-представления каждое новое определение переменной вводит новое *поколение* этой переменной (обозначается нижним индексом, как x_2), и все инструкции, которых достигает это определение, используют введенное поколение вместо первоначальной переменной. В случае, когда некоторый базовый блок достигают несколько определений одной переменной, в начале этого блока создается т.н. φ-функция, которая вводит новое поколение переменной через поколения из этих определений, и далее в блоке используется введённое поколение (см. рис. 1).

Некоторые из уже рассмотренных трансформаций (распространение констант и копий, удаление частичной избыточности) удобнее выполнять на SSA-представлении, чем на обычном промежуточном представлении. Кроме того, существует ряд трансформаций, специально разработанных для выполнения над SSA-представлением. Примером такой трансформации является *нумерация значений* (value numbering) [6, глава 12.4; 10].

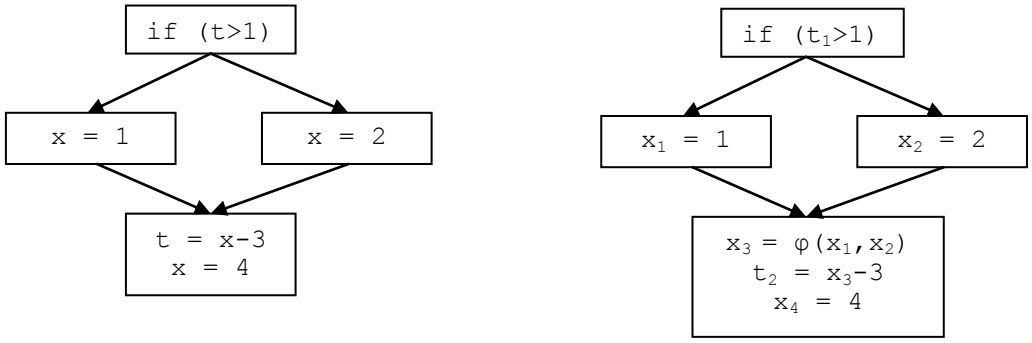

Рис. 1. Фрагмент программы (слева) и её SSA-представление (справа)

Идея нумерации значений заключается в присвоении переменным и выражениям номеров таким образом, что выражения с одинаковыми номерами обязательно вычисляют одинаковые значения. Как следствие, можно удалить избыточные вычисления, сохранив результат только одного из выражений с одинаковым номером во временную переменную и используя её вместо остальных таких выражений. Нумерация значений потенциально может найти больше избыточностей, чем классическая трансформация удаления избыточностей, т.к. ищет выражения, которые совпадают не по синтаксической записи, а значениями.

Выделяют два типа нумерации значений: на основе *хэширования выражений* и на основе конгруэтности в *графе значений*. Нумерация на основе хэширования значений является локальной трансформацией, т.е. обрабатывает по одному базовому блоку. Базовый блок обходится сверху вниз, и для каждого выражения вычисляется хэш-функции таким образом, что одинаковые с точностью до коммутативности выражения с операндами, имеющими одинаковые значения хэш-функции, в свою очередь получают одинаковые значения хэш-функции. Если выражение с таким значением хэш-функции уже было вычислено, это означает, что нет необходимости повторно вычислять его, а можно взять запомненное значение. Если значение некоторой переменной переопределяется, то выражения с её участием удаляются из списка выражений с известными значениями хэш-функции.

```
a = 4
b = 4
c = a + 2
d = b + 2
if c >=
```

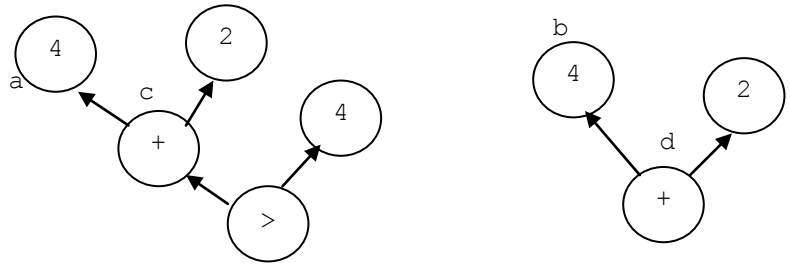

Рис. 2. Пример фрагмента программы и его графа значений
(левые стрелки соответствуют первым операндам, а правые – вторым).

Нумерация на основе графа значений является глобальной трансформацией. По *SSA*-представлению процедуры строится граф значений, в вершинах которого находятся константы и выражения, а дуги являются присваиваниями, направленными от выражения к его операндам и помеченными номером операнда в выражении. Вершины графа также именуются *SSA*-поколениями переменных, в которых записывается результат выражения (см. рис. 2). Две вершины из графа значений называются *конгруэнтными*, если они совпадают, либо значения их констант совпадают, либо они являются одинаковыми выражениями с конгруэнтными операторами (на рисунке 2 – вершины, соответствующие переменным *c* и *d*). Конгруэнтные вершины вычисляют одинаковые значения и могут быть оптимизированы.

Доказано, что различные нумерации значений на основе хэширования и графа значений несравнимы: для каждой из них можно найти тестовые программы, которые обрабатываются одной, но не другой. Кроме того, в последнее время подход на основе хэширования значений был расширен до глобального, в котором объединены лучшие стороны обоих подходов при меньшей вычислительной сложности. Новый подход уже реализован в рамках инфраструктуры *TreeSSA* компилятора *Gcc*.

3. Современные проблемы оптимизации программ

В данном разделе будут рассмотрены наиболее актуальные проблемы оптимизации программ, направленные на решение новых задач, связанных с учетом специфики современных платформ и тенденциями развития их архитектуры. В настоящее время происходит массовое внедрение новых архитектурных платформ (*Itanium, Cell* и др.) и специализированных систем (*ARM, DSP* и др.). Эти архитектуры предоставляют широкие возможности по параллельному выполнению команд (возможность одновременной выдачи нескольких команд, наличие нескольких функциональных устройств, конвейеры данных и команд), они позволяют обеспечить параллельное выполнение программы на уровне потоков (многоядерность). Это выдвигает на первый план разработку и внедрение в компиляторы новых методов оптимизации, обеспечивающих эффективное использование перечисленных свойств в прикладных программах.

Так, для архитектур с явно выраженным параллелизмом команд важную роль играют следующие трансформации: *условное выполнение*, позволяющее превращать небольшие ветвления в линейные участки кода, что уменьшает количество условных переходов и ненужных сбросов конвейера; *спекулятивное выполнение*, позволяющее выдавать команды тогда, когда со значительной вероятностью уже известно, что их выполнение необходимо, а их данные готовы (эта оптимизация будет рассмотрена в разделе 3.1); *конвейеризация циклов*, позволяющая составить такое расписание тела цикла, что параллельно будут выполняться команды с разных итераций цикла (более подробно конвейеризация циклов описана в разделе 3.2). При этом важно не только реализовать соответствующую трансформацию в компиляторе, но и обеспечить взаимодействие между ней и остальными частями компилятора. Например, для полноценной поддержки условного выполнения требуется отдельная трансформация по преобразованию ветвлений в линейный код, изменения в анализах потока данных, распределении регистров, планировании команд – т.е. значительная переработка структур данных, базовых алгоритмов и интерфейсов компилятора. Все вышеперечисленные оптимизации реализованы в промышленных компиляторах,

например, в компиляторе *Icc* компании *Intel* для платформы *Itanium*, а также в компиляторе *Gcc*.

Условия эксплуатации специализированных систем делают актуальными совершенно новый класс оптимизирующих преобразований, когда целью оптимизаций становится энергосбережение. Сама по себе оптимизация времени работы программы, а также оптимизация работы с памятью помимо ускорения программы также приводят к понижению энергопотребления всей системы. Тем не менее, активно исследуются специализированные энергосберегающие оптимизации, которые возможно проводить на нескольких уровнях: при разработке аппаратной части системы (уменьшение статических утечек), на уровне операционной системы (например, переход в спящий режим или понижение частоты процессора), на уровне компилятора (например, понижение частоты для тех участков программы, на которых большинство временных затрат приходится на ожидание данных из памяти; мы рассмотрим эту оптимизацию в разделе 3.3). В промышленных компиляторах такие оптимизации пока не получили распространения.

Оптимизация программ для многоядерных архитектур сейчас является наиболее актуальным вопросом компиляторных исследований, но никаких конкретных компиляторных решений, применимых к широкому классу прикладных программ, пока не предложено. Ведутся работы по разработке новых языков программирования, позволяющих естественно выражать параллелизм на уровне потоков, разработке новых архитектур. Для облегчения оценки результатов таких исследований предлагается выделить *паттерны* параллельного программирования, успешная реализация которых позволит судить об эффективности разработанной системы для прикладных программистов.

3.1. Спекулятивное выполнение команд

Спекулятивным выполнением (*speculative execution* [6, §17.2; и глава 11]) принято называть опережающее выполнение команд тогда, когда почти наверняка известно, что их выполнение необходимо, либо что их данные готовы. Техника спекулятивного выполнения команд широко используется в современных процессорах

при предсказании переходов, при этом процессор должен реализовать сложную функциональность, обеспечивающую выбор команд для спекулятивного выполнения и отмену результатов их работы в случае, если направление перехода предсказано неверно.

Другой подход к реализации спекулятивного выполнения предлагает архитектура с явно выраженным параллелизмом команд (*EPIC*), реализованная в процессорах семейства *Itanium* [12]. В этой архитектуре вся работа по выявлению и описанию параллелизма на уровне команд перенесена на компилятор: помимо явного указания команд, которые могут выполняться параллельно, компилятор может подсказывать направление условных переходов, управлять загрузкой данных в кэш, выдавать команды на спекулятивное и условное выполнение. Кроме того, в поиске кандидатов на спекулятивное выполнение компилятор может просмотреть больший объем кода – например, всю процедуру исходного кода программы. При этом компилятор оперирует значительными ресурсами архитектуры – большое количество функциональных устройств и регистров, большой объём кэш-памяти и т.п.

На рисунке 3 представлен пример использования спекулятивного выполнения на архитектуре Itanium. На рисунке 3а процессор ожидает результатов загрузки в строке 1, чтобы выполнить условный переход в строке 5. Для выдачи команд на спекулятивное выполнение необходимо перенести команды вверх через условный переход в одну из групп, при выполнении которой остаются свободные функциональные устройства – это группа в строке 1 либо в строке 3. Если компилятор установит, что перемещения команд не приведут к записи неправильных значений в регистры (например, регистры, в которых записываются значения, не используются на другой ветке условного перехода), то команды `mul` и `add` в строках 6 и 8 можно безопасно перенести вверх через условный переход на свободные места и выполнить спекулятивно, т.к. они не могут вызвать исключения.

а)

```
1   r2 = ld[r3]
2   ;;
3   p6 = cmp r2, 0
4   ;;
5   (p6) jmp label
6   mul r4, r4, r1
7   ;;
8   add r5, r5, r4
9   ;;
10  r6 = ld [r5]
```

б)

```
1   r2 = ld[r3]
2   mul r4, r4, r1
3   ;;
4   add r5, r5, r4
5   ;;
6   p6 = cmp.lt r2, 0
7   r6 = ld.s [r5]
8   ;;
9   (p6) jmp label
10  chk.s r6
```

Рис. 3. Спекулятивное выполнение на процессоре Itanium:
а) – первоначальный код, б) – измененный компилятором

Команда же загрузки в регистр `r6` (строка 10) может вызвать исключение, поэтому при её спекулятивном выполнении возникшее исключение должно быть подавлено аппаратурой. Компилятор сигнализирует процессору о необходимости использования спекулятивной версии команды загрузки с помощью суффикса "`.s`" (строка 7 на рисунке 3б). При возникновении исключения в момент спекулятивного выполнения команды процессор помечает её целевой регистр с помощью специального NaT-бита, означающего, что значение регистра не может использоваться в вычислениях. Перед использованием результатов спекулятивной загрузки необходимо проверить, установлен ли этот бит, и в случае положительного ответа возбудить исключение. Для этого процессором предоставляется специальная команда проверки `chk.s`, которую также должен сгенерировать компилятор. Результирующий код показан на рисунке 3б.

Архитектура Itanium предоставляет спекулятивные версии инструкций загрузки, которые позволяют выдавать их прежде, чем позволяют как зависимости по данным между операциями с памятью, так и по управлению. Возможность спекулятивной выдачи по данным возникает по сути из-за невозможности выполнить точный статический анализ алиасов, т.е. тогда, когда компилятор консервативно считает, что две команды используют одну и ту же ячейку памяти. Тем не менее, если есть возможность предполагать, что во время работы программы эти ячейки всегда (или почти всегда) будут разными, то можно применить спекулятивную выдачу по данным.

Спекулятивная выдача инструкций загрузки на Itanium позволяет также спекулятивно выполнить и другие инструкции, использующие загруженные данные. В этом случае все спекулятивно выполненные инструкции включаются в код восстановления, который должен сгенерировать компилятор. Кроме того, задачей компилятора является выбор момента для выдачи спекулятивной команды, поиск кандидатов на спекулятивное выполнение и выбор наилучшего кандидата. Как видно, сложность компилятора для EPIC-архитектуры значительно возрастает по сравнению с компилятором для обычной архитектуры.

Наиболее естественно спекулятивные команды генерируются в процессе планирования команд, так как компилятор обладает информацией о состоянии конвейера процессора на каждый момент планирования, а следовательно, может выдать спекулятивную команду в момент простоя конвейера. Планировщик команд для Itanium может выдать команду спекулятивно, если все ее зависимости по данным и по управлению могут быть устранены превращением команды в ее спекулятивный аналог. Важно подчеркнуть, что для успешной поддержки спекулятивного выполнения планировщик должен обладать информацией о том, какие зависимости по данным и по управлению более вероятны, а какие менее вероятны, т.к. цена выполнения кода восстановления в случае неправильной догадки велика. Поэтому компилятор должен использовать информацию о распределении вероятностей условных переходов в программе во время выполнения, получаемую с помощью профилирования. После этого возможно статическое проведение вероятностного анализа алиасов, аннотирующего каждый найденный алиас вероятностью того, что он выполнится во время работы программы [11].

В компиляторе GCC при выборе наилучшего кандидата на спекулятивную выдачу по управлению учитывается вероятность выполнения команды относительно данной точки планирования (на рисунке 3б это вероятность того, что переход на метку в строке 6 не будет произведен). При оценке кандидатов на спекулятивную выдачу по данным используется ряд эвристик, распознающих ситуации, в которых зависимости по данным скорее всего не будет (например, можно предполагать, что два указателя, переданных в процедуру в качестве разных параметров, не ссылаются на одну и ту же ячейку памяти) [13].

3.2. Конвейеризация циклов

Задачей конвейеризации цикла является составление расписания тела цикла, в котором команды “выстраиваются” как на конвейере, при этом каждая команда соответствует одной из стадий конвейера, а одна итерация цикла соответствует переходу на следующую стадию всех команд [5, глава 10.5]. Конвейеризация цикла позволяет выявить параллелизм между итерациями цикла: если тело цикла разделено на три стадии, то одновременно будут выполняться первая стадия *i*-й итерации, вторая стадия *i-1*-й итерации и третья стадия *i-2*-й итерации.

Мы рассмотрим конвейеризацию циклов на примере алгоритма интервального планирования (modulo scheduling) [14]. Это эвристический алгоритм, целью которого является максимизация частоты выдачи новых итераций, а также минимизация регистрового давления, т.е. количества одновременно живущих регистров в одной итерации цикла. *Интервалом запуска* (Initiation Interval, II) называется количество тактов между началом выполнения одной стадии и началом выполнения следующей. На рисунке 4 показан пример конвейеризованного цикл из пяти итераций, в теле которого три команды, при этом цикл разделен на три стадии и занимает четыре такта. Расстояние между командами одной итерации соответствует латентности верхней команды, индекс команды соответствует номеру итерации. *Ядро* цикла, соответствующее конвейеру, состоит из загрузки с *i*-й итерации, сложения с *i-2*-й итерации и записи в память с *i-3*-й итерации. *Прологом* (соответственно *эпилогом*) называются все команды, которые выполняются до (соответственно после) того, как конвейер выйдет на полную загрузку.

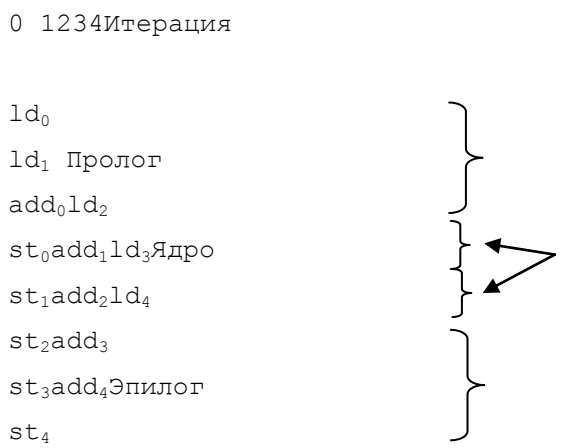

Рис. 4. Пример конвейеризованного цикла

Для выполнения конвейеризации циклов необходимо выполнить следующие шаги. Сначала строится граф зависимости по данным для цикла, причем находятся как внутри-, так и межитерационные зависимости (которые нужны, так как выполняется группировка команд с разных итераций). В графе зависимостей команды тела цикла являются вершинами, а взвешенными дугами обозначаются зависимости, при этом вес дуги соответствует латентности команды в тактах процессора, а для межитерационной зависимости – расстоянию в итерациях. Далее проводится оценка снизу длины интервала запуска в тактах, minII, исходя из ограничений на зависимости по данным и на ресурсы процессора. Рекуррентные зависимости по данным, а именно – длина циклов в графе зависимостей, дают первую оценку. Вторая оценка получается из того, что у процессора должно хватить ресурсов, чтобы выполнить все команды цикла за интервал запуска. Оценкой сверху на интервал запуска, maxII, является последовательное расписание тела цикла.

Наконец, для каждого значения интервала запуска между minII и maxII производится попытка уместить все команды из тела цикла в окно времени, длина которого равна этому значению. Все команды цикла анализируются в таком порядке, что предпочтение отдается командам с критических путей, а также командам, уменьшающим количество одновременно живых регистров. Для каждой команды выбирается место в окне планирования как можно ближе к её предкам либо потомкам по графу зависимостей. Если такого места не нашлось, исходя либо из ограничений по

зависимостям, либо по ресурсам процессора, то значение интервала запуска увеличивается на единицу, и процесс повторяется.

В случае, когда в результирующем расписании обнаружились регистры, срок жизни которых превышает значение интервала запуска, это означает, что такой регистр должен одновременно хранить два значения, что невозможно. Так, в примере на рисунке 4 результаты очередной загрузки перетрут значение, загруженное на предыдущей итерации, с которым еще не было выполнено сложение. Для этих регистров генерируются команды, сохраняющие их копии во временных регистрах, после чего каждое из использований регистров заменяется на использование соответствующей копии. Аппаратная поддержка *вращающихся* регистров (rotating registers) позволяет избежать вставки копий путем сдвига соответствия между логическими и физическими именами регистров на каждой итерации [5, глава 10.5.12]. Кроме того, вращающиеся предикатные регистры позволяют избежать вставки пролога и эпилога цикла, которые формируются из команд, подготавливающих данные для выполнения ядра.

В компиляторах *Icc* и *Gcc* реализована конвейеризация циклов через интервальное планирование [15,16]. Необходимо отметить, что эта оптимизация считается одной из самых сложных машинно-зависимых оптимизаций, т.к. помимо реализации, требует точного анализа зависимостей по данным, а также модели процессора для оценки ограничений по ресурсам. Тем не менее, ускорение, получаемое этой оптимизацией также может быть велико и достигать десятков процентов [17].

3.3. Оптимизации энергопотребления системы

При рассмотрении задачи об оптимизации энергопотребления системы разделяют *динамическое* и *статическое* энергопотребление [18, раздел 2]. Статическое энергопотребление возникает из-за тока утечки, протекающего через транзисторы схемы даже тогда, когда они выключены. Снижения статического потребления добиваются, как правило, при разработке схемы, уменьшая количество транзисторов либо используя транзисторы с лучшими физическими характеристиками. Подробное рассмотрение этого вопроса выходит за рамки настоящего обзора.

Основным источником динамического энергопотребления является переключение конденсаторов, т.е. зарядка и разрядка конденсаторов на выходах схем. Потребляемую при этом мощность можно можно представить следующим образом: $P_d \sim aCV^2 f$, где C – емкость, V – напряжение, f – тактовая частота, a – коэффициент активности, показывающий количество переходов от 0 к 1 или от 1 к 0 (*переключений битов*) на схеме. Из этого уравнения сразу следует несколько способов уменьшения энергопотребления.

Первым способом является уменьшение емкости элементов схемы (коэффициент C). Этого можно добиться лишь на этапе разработки схемы, уменьшая емкость транзисторов либо длину дорожек. Вторым способом является уменьшение коэффициента активности, чего можно добиться как при разработке процессора, например, запирая недействующие функциональные устройства от тактового сигнала (т.н. *clock gating*), так и программным путем, уменьшая количество переключений битов на шине команд. Для этого необходимо найти такой порядок команд, что битовые последовательности, кодирующие соседние команды, как можно меньше различных бит стоят в одинаковых позициях. Учитывать эти соображения можно при планировании команд в качестве одной из эвристик, определяющих оптимальный порядок команд. При этом основной трудностью является сопоставление инструкции внутреннего представления и битовой последовательности, кодирующей команду процессора, в которую перейдет эта инструкция.

Часто используемым способом уменьшения динамического энергопотребления является понижение напряжения на схемы, либо понижение тактовой частоты схемы, либо комбинация этих методов. Комбинированное понижение тактовой частоты и напряжения принято называть *динамическим понижением напряжения* (*DVS*) [18, раздел 3.5]. Как видно из вышеприведенного уравнения, теоретически этот подход должен позволить уменьшить энергопотребление кубически. Многие современные процессоры реализуют возможность программно изменять напряжение и тактовую частоту, включая последние процессоры *Intel*, популярные реализации архитектуры *ARM, Transmeta Crusoe* и другие. Тем не менее, теоретического обоснования подходов, основанных на *DVS*, недостаточно из-за сложности предсказания нагрузки процессора в реальном времени, а также неточности рассматриваемой модели энергопотребления и

недетерминированного времени работы процессора над конкретным приложением. Как правило, используются эмпирические подходы к понижению частоты процессора, например, операционной системой по результатам мониторинга загрузки процессора за определенный промежуток времени.

Рассмотрим один из статических подходов к *DVS*, пригодных для реализации в компиляторе [19]. Поставим задачу отыскать регионы в исходном коде программы, время выполнения которых определяется в основном ограничениями на вычислительные ресурсы, память, либо устройства ввода-вывода. Если на некотором участке программы в основном происходит ожидание записи или чтения данных из памяти, то, при условии отдельного подвода напряжения к памяти и процессору, можно уменьшить напряжение, подаваемое на процессор. При этом выполнение этого участка не замедлится, либо замедлится незначительно, а энергопотребление снизится. Пример такого участка приведен на рисунке 5: после понижения частоты затраты на загрузку из памяти не изменяются, а вычислительные затраты увеличиваются, но при этом не превосходят их.

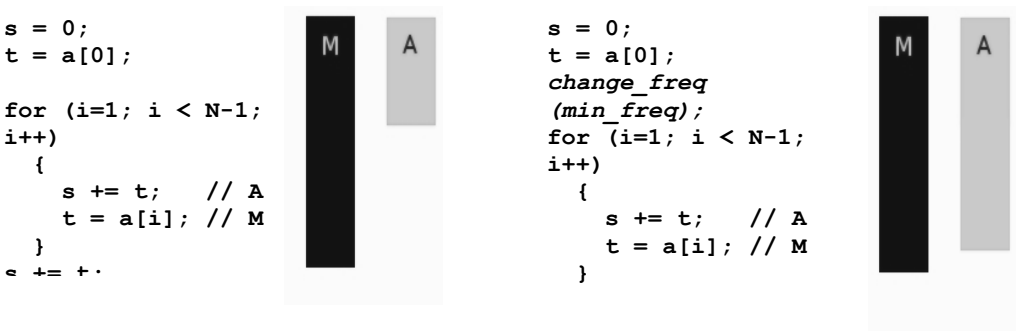

Рис. 5. Пример динамического понижения напряжения

Для того, чтобы выбрать регионы, являющиеся кандидатами на выполнение с пониженной частотой, необходимо узнать время работы регионов как на высокой, так и на низкой частоте (при условии, что для выбора доступны только две частоты), а также количество раз, которое выполнялся каждый регион. Для этого программа профилируется и запускается на целевой аппаратуре два раза, на высокой и на низкой частоте. Далее, предположим, что на минимальной частоте будет выполняться только один регион. Тогда достаточно для каждого региона сравнить энергопотребление всей

программы на максимальной частоте и потребление всей программы, кроме данного региона, на максимальной частоте, данного региона на минимальной частоте, и затрат на переключение частот:

$$\min_{R,f} P_f \cdot T(R, f) + P_{f_{\max}} \cdot T(P - R, f_{\max}) + 2P_{trans} \cdot N(R).$$

Здесь энергия выражается через мощность, потребляемую на каждой частоте, время работы регионов, и количество переключений частоты. Для того, чтобы последнюю величину можно было оценить, изначально выбираются регионы с одним входом и одним выходом, для которых количество переключений будет равно количеству раз, которое выполнялся регион, а эту величину можно получить из профиля программы.

Дополнительное условие, накладываемое на выбранный регион, заключается в том, что трансформированная программа должна замедлиться не более чем на *k%*: это ограничение часто используется во встраивамых системах. Кроме того, задачу минимизации можно усложнить, рассматривая несколько регионов как кандидаты на выполнение с пониженным напряжением.

Помимо компиляторной реализации профилирования программы, решения задачи о нахождении оптимальных регионов, и вставки системных вызовов, понижающих частоту процессора, необходимо реализовать сами эти вызовы. В операционной системе *Linux* новых версий для ряда процессоров *ARM* уже существует интерфейс, позволяющий менять тактовую частоту, который можно оформить как системный вызов [20]. Экспериментальные результаты использования этого интерфейса на платформе *ARM* показывают, что затраты на переключение частоты на встраиваемых системах являются значительными, и целесообразно выбирать как можно более крупные регионы, для чего необходимо проводить межпроцедурный анализ программы.

4. Проблемы безопасности программного обеспечения

Особенностью развития современной компьютерной индустрии является то, что благодаря повсеместному внедрению сетевых технологий (*Internet*/*Intranet*), практически все компьютеры, в том числе правительственные и корпоративные, доступны через глобальную сеть. Это открывает возможности для несанкционированного доступа, позволяя злоумышленникам получать конфиденциальную информацию, а также при необходимости парализовать работу жизненно важных информационных инфраструктур.

Существуют различные подходы к решению проблем безопасности: попытки защиты системы по «периметру» (*firewalls*, etc.), выполнение критических приложений в специальном окружении с целью предотвращения недокументированного поведения (песочница), сигнатурный поиск зловредных включений (вирусы). Эти подходы не используют методов анализа и трансформации программ и в данном обзоре не рассматриваются.

Безопасность ПО имеет много аспектов. Прежде всего, это безопасность функционирования программно-аппаратных систем: системное и прикладное ПО, управляющие такой системой, должны гарантировать правильную ее работу, чтобы она могла выполнить поставленные перед ней задачи. Есть и другие аспекты безопасности ПО, например, безопасность от несанкционированного доступа, безопасность от несанкционированного копирования (борьба с «пиратством»), безопасность от несанкционированной обратной инженерии (сохранение авторских прав) и др. Как правило, атаки связаны с внедрением программ атакующего на вычислительную систему, на которой выполняется ПО, с целью изменить работу системы в требуемом для атакующего направлении. Борьба с такими атаками имеет, по крайней мере, два аспекта: (1) обеспечение устойчивости ПО к внедрению программ атакующего, (2) обнаружение (и удаление) ранее внедренных программ атакующего.

Устойчивость ПО к внедрению программ атакующего связана с отсутствием в нем «уязвимостей», т.е. таких ошибок разработчика, которые трудно обнаружить на этапе тестирования, но которые влияют на устойчивость работы ПО. Прежде всего, это

уязвимости безопасности и критические ошибки. Такие ошибки естественно искать в исходном коде программ. Для обнаружения ошибок этого класса широко используются методы анализа и трансформации программ, рассматриваемые в данном обзоре.

Ранее внедренные программы атакующего (часто их называют «закладками» или «недокументированными свойствами ПО», еще одно название – Троянские кони, или «Трояны») имеют целью воздействовать на атакованную систему по сигналу атакующего или постоянно обеспечивать дополнительные функции системы, не заметные для ее администрации. Как правило, такие программы внедряются в двоичный код, что значительно затрудняет их обнаружение.

Статические методы анализа и трансформации программ в двоичном (бинарном) представлении также рассматриваются в данном обзоре. Кроме того, в обзоре рассматриваются существующие подходы к анализу бинарного кода с целью верификации работающих алгоритмов и выявления недокументированных свойств, в предположении, что бинарный код защищен, и при помощи статического анализа получить результат невозможно.

Наконец, рассматриваются методы трансформации программ (как исходных, так и двоичных), применяемые с целью такого изменения программы, при котором средства анализа программ не могут нормально работать. К этому классу относятся рассматриваемые в данном обзоре методы обфускации программ.

4.1. Статические методы поиска дефектов в исходном коде программ

Бурное развитие современных телекоммуникационных технологий позволило решить задачу доступа к информационным и вычислительным ресурсам вне зависимости от географического расположения поставщика и потребителя ресурсов. Сеть Интернет связывает миллионы компьютеров по всей планете. С другой стороны, именно общедоступность информационных ресурсов подняла на новый уровень требования к безопасности программного обеспечения. Необходимым условием обеспечения безопасности ПО является его корректная работа на всех возможных входных данных и всех других видах внешних по отношению к программе воздействий.

Следует отметить, что сформулированное требование сильнее, чем требование отсутствия в программе ошибок, если под ошибками понимать несоответствие действительного поведения программы специфицированному на указанном в спецификации множестве входных данных программы. Спецификация может определять поведение программы лишь на подмножестве множества всех возможных входных данных. Например, для программ, получающих данные от пользователя или из других неконтролируемых программой внешних источников реальное множество входных данных представляет собой просто множество всех возможных битовых строк вне зависимости от спецификации входных данных программы. Если программа является частью многопроцессной системы и взаимодействует с другими процессами и окружением, реальное множество входных данных зависит и от всех возможных темпоральных вариантов взаимодействия процессов, а не только от специфицированных.

Когда требование корректной работы программы на всех возможных входных данных нарушается становится возможным появление так называемых уязвимостей защиты (*security vulnerability*). Уязвимости защиты могут приводить к тому, что одна программа может использоваться для преодоления ограничений защиты всей системы, частью которой является данная программа, в целом. В особенности это относится к программам, обслуживающим различные общедоступные сервисы сети Интернет и к программам, работающим в привилегированном режиме.

В настоящее время сложилась некоторая классификация уязвимостей защиты в зависимости от типа программных ошибок, которые могут приводить к появлению уязвимости в программе. Рассмотрим примеры уязвимостей.

(1) *Переполнение буфера* (*buffer overflow*). Эта уязвимость возникает как следствие отсутствия контроля или недостаточного контроля за выходом за пределы массива в памяти. Языки C/C++, чаще всего используемые для разработки системного ПО, не выполняют автоматического контроля выхода за пределы массива во время выполнения программы. По месту расположения буфера в памяти процесса различают переполнения буфера в стеке (*stack buffer overflow*), куче (*heap buffer overflow*) и области статических данных (*bss buffer overflow*). Все три вида переполнения буфера могут с успехом быть использованы для выполнения произвольного кода уязвимым

процессом. Рассмотрим более детально уязвимость переполнения буфера в стеке как наиболее простую. Рассмотрим программу:

```
#include < stdio.h>
int main(int argc, char **argv){
  char buf[80];
  gets(buf);
  printf(«%s», buf);
  return 0;
}
```

Как известно, функция `gets` не ограничивает длину строки, вводимой из стандартного потока ввода. Вся введённая строка до символа '`\n`', кроме него самого, будет записана в память по адресам, начинающимся с адреса массива `buf`. При этом если длина введённой строки превысит 80 символов, то первые 80 символов строки будут размещены в памяти, отведённой под массив `buf`, а последующие символы будут записаны в ячейки памяти, непосредственно следующие за `buf`. То есть будут испорчены сначала сохранённые регистры и локальные переменные, затем адрес предыдущего стекового фрейма, затем адрес возврата из функции `main` и т. д. В момент, когда функция `main` будет завершаться с помощью оператора `return`, процессор выполнит переход по адресу, хранящемуся в стеке, но этот адрес испорчен в результате выполнения функции `gets`, поэтому переход произойдёт совсем в другое место, чем стандартный код завершения процесса.

Чтобы использовать такое переполнение буфера, необходимо подать на вход программе специальным образом подготовленную строку, которая будет содержать небольшую программу, выполняющую действия, нужные атакующему (это так называемый *shellcode*, который в простейшем случае просто выполняет вызов стандартного командного интерпретатора `/bin/sh`). Кроме того, нужно так подобрать размер подаваемых на вход данных, чтобы при их чтении на место, где размещается адрес возврата из `main`, попал адрес начала *shellcode*. В результате в момент завершения работы функции `main` произойдёт переход на начало фрагмента *shellcode,* в результате чего будет запущен интерпретатор командной строки. Интерпретатор

командной строки будет иметь полномочия пользователя, под которым работал уязвимый процесс, кроме того, стандартные средства аутентификации оказываются обойденными.

Для предотвращения выполнения произвольного кода в случае использования переполнения буфера используются такие приёмы, как запрет выполнения кода в стеке, отображение стандартных библиотек в адресное пространство процесса со случайных адресов, динамический контроль барьерных данных и так далее. Но не один из этих приёмов не может гарантировать предотвращения использования уязвимости переполнения буфера в стеке, поэтому ошибки приводящие к переполнению буфера должны быть устранены непосредственно в исходном коде.

(2) Ошибки форматных строк (*format string vulnerability*). Этот тип уязвимостей защиты возникает из-за недостаточного контроля параметров при использовании функций форматного ввода-вывода `printf`, `fprintf`, `scanf`, и т. д. стандартной библиотеки языка C. Эти функции принимают в качестве одного из параметров символьную строку, задающую формат ввода или вывода последующих аргументов функции. Если пользователь программы может управлять форматной строкой (например, форматная строка вводится в программу пользователем), он может сформировать её таким образом, что по некоторым ячейкам памяти (адресами которых он может управлять) окажутся записанными указанные пользователем значения, что открывает возможности, например, для переписывания адреса возврата функции и исполнения кода, заданного пользователем.

Уязвимость форматных строк возникает, по сути, из-за того, что широко используемые в программах на C функции, интерпретируют достаточно мощный язык, неограниченное использование возможностей которого приводит к нежелательным последствиям. Как следствие, в безопасной программе не должно быть форматных строк, содержимое которых прямо или косвенно зависит от внешних по отношению к программе данных. Если же такое невозможно, при конструировании форматной строки она должна быть тщательно проверена. В простейшем случае из пользовательского ввода должны «отфильтровываться» опасные символы «`%`» и «`$`».

(3) *Уязвимость «испорченного ввода»* (*tainted input vulnerability*). Уязвимости испорченного ввода могут возникать в случаях, когда вводимые пользователем данные

без достаточного контроля передаются интерпретатору некоторого внешнего языка (например, языку *Unix shell* или *SQL*). В этом случае пользователь может таким образом задать входные данные, что запущенный интерпретатор выполнит совсем не ту команду, которая предполагалась авторами уязвимой программы. Рассмотрим следующий пример:

```
#include < stdio.h>
#include < stdlib.h>
int main(void)
{
 char buf[80], cmd[100];
 fgets(buf, sizeof(buf), 80);
 snprintf(cmd, sizeof(cmd), «ls -l %s», buf);
 system(cmd);
 return 0;
}
```

В этом примере ожидается, что пользователь программы вводит имя файла, а программа вызывает стандартную программу `ls`, которая печатает информацию о введённом файле. При этом для вызова программы `ls` командная строка передаётся интерпретатору командной строки `/bin/sh`. Это можно использовать если ввести в программу строку, содержащую, например, символ `;` (точка с запятой), например «`myfile ; rm -rf /`». Строка, фактически переданная интерпретатору командной строки будет равна «`ls -l myfile ; rm -rf /`», то есть фактически будет состоять из двух команд интерпретатора *shell*, а не из одной, при этом вторая команда - это запрос на удаление всей файловой системы.

Как и в случае уязвимости форматной строки, достаточное условие отсутствия уязвимости типа испорченного ввода в программе состоит в том, что «опасные» аргументы «опасных» функций никак не должны зависеть от внешних по отношению к программе данных.

Кроме перечисленных здесь типов уязвимостей защиты существуют и другие типы, например - уязвимости как следствие синхронизационных ошибок (*race*

conditions), некорректная работа с временными файлами, слабое шифрование и другие классы уязвимостей.

Уязвимости защиты – это один из видов дефектов ПО. Другие дефекты связаны с ошибками при работе с динамической памятью (утечка памяти, разыменование нулевых указателей и др.), неосвобождением памяти и других ресурсов, используемых программой, использованием неинициализированных переменных. [21]

Дефекты в исходном коде создают серьезные проблемы при разработке и эксплуатации ПО. Наличие дефектов может приводить к возникновению уязвимостей, краху программы и повреждению пользовательских данных. Поиск и устранение дефектов в ПО требует больших трудозатрат, при этом многие дефекты могут остаться незамеченными. По данным исследования, проведенного по заказу Национального Института Стандартов и Технологий США, убытки, возникающие из-за недостаточно развитой инфраструктуры устранения дефектов в ПО, составляют от 22 до 60 миллиардов долларов в год [22]. Часто дефекты являются причиной переноса сроков выпуска программ. Стоимость устранения дефекта, пропущенного на этапах разработки и тестирования, может возрасти после поставки программы от 2 до 100 раз [23].

Методы статического поиска дефектов позволяют обнаруживать дефекты по исходному коду программы. При этом поведение программы оценивается на всех путях исполнения одновременно. За счет этого, возможно обнаружение дефектов, проявляющихся лишь на необычных путях исполнения при редких входных данных. Эта особенность позволяет дополнить методы тестирования, повышая надежность программ. Системы статического анализа могут использоваться при проведении аудита исходного кода, для систематического устранения дефектов в существующих программах, и встраиваться в цикл разработки, автоматически обнаруживая дефекты в создаваемом коде.

Основные ограничения на область применимости систем статического анализа возникают из-за невозможности проведения точного автоматического анализа реальных программ. При обнаружении конструкций, которые могут потенциально указывать на дефект, во многих случаях не удается установить, возможен ли в действительности путь исполнения программы и входные данные, приводящие к

ошибке. Выдача всех предупреждений в таких ситуациях приводит к тому, что большая их доля оказывается ложной, делая систему статического анализа малополезной для многих приложений. При этом даже некоторые истинные предупреждения могут диагностироваться в форме, не позволяющей отличить их от ложных. Например, в предупреждении сообщается только, что данная операция может привести к переполнению массива, но не указано, каким образом, т.к. сама система анализа не может это определить, что в данном случае и явилось причиной вывода предупреждения.

Примером обладающей этим недостатком системы является *BOON* [24], выдающая предупреждения для всех мест потенциального переполнения буфера. В случае, когда полный исходный код программы недоступен, возникает неопределенность, не связанная с неточностью анализа: в зависимости от свойств недоступного при анализе кода, данная операция может как приводить, так и не приводить к ошибке. Например, для библиотечного кода часто возможно построение некорректного вызова из пользовательского кода, приводящего к выполнению некорректной операции в коде библиотеки, но при отсутствии кода этого вызова ошибка в коде библиотеки как правило диагностироваться не должна.

Улучшение алгоритмов анализа или использование более тяжеловесных видов анализа, требующих больших вычислительных ресурсов, повышает точность анализа, но в силу принципиальных ограничений не позволяет достичь достаточно высокого его качества. В результате, область применимости систем статического анализа определяется в первую очередь ограничениями, накладываемыми на анализируемую программу, требованиями к результатам анализа и приемлемым объемом работы по подготовке программы к анализу, выполняемой вручную.

В разделе будут рассмотрены следующие классы систем обнаружения дефектов, определяемые используемыми методами и областью применимости: системы автоматизации экспертного аудита, системы верификации ограниченного исходного кода, системы проверки корректности пользовательских аннотаций, и системы автоматического поиска дефектов.

Простые системы, частично автоматизирующие процесс экспертного аудита, основаны на поиске мест использования потенциально опасных библиотечных

функций. Такие системы служат двум основным целям: во-первых, они позволяют избежать ручного просмотра исходного кода, указывая только на строки, содержащие потенциально опасные вызовы и не пропуская их; во-вторых, они включают в себя обширную базу потенциально опасных функций, с описанием для каждой из них условий, при которых возможно возникновение ошибки, и методов безопасного их использования. Использование этих систем позволяет повысить производительность и снизить требования к квалификации оператора для выполнения систематического аудита.

Для поиска вызовов потенциально опасных функций в исходном коде используется синтаксический анализ. Требование к исчерпывающему поиску вызовов в сочетании с поверхностным характером такого анализа приводит к большому количеству ложных предупреждений о возможных дефектах. Эта особенность требует большого количества трудозатрат при применении таких систем. Выдаваемые предупреждения разделяются по примерной степени риска, создаваемого каждым из обнаруженных мест вызова потенциально опасных функций, в зависимости от используемой функции и особенностей её конкретного вызова, которые возможно установить при помощи синтаксического анализа. Кроме того, как правило предусматривается возможность отметки в исходном коде мест выдачи ложных предупреждений. К сожалению, за счет этой возможности реальные дефекты могут быть помечены как ложные предупреждения, и никогда более не проявляться при статическом анализе [25]. Эти особенности позволяют сосредоточиться на наиболее важных предупреждениях и упростить последующие циклы проведения аудита, а также использовать систему непосредственно при разработке. К описанному классу относятся системы *ITS4* [26], *RATS* [27] и *Flawfinder* [25]. Таким образом, системы автоматизации экспертного аудита позволяют обнаруживать существенно ограниченный класс дефектов, требуя больших трудозатрат на просмотр выданных предупреждений, но при этом не накладывают ограничений на анализируемую программу и не требуют её предварительной подготовки.

Системы верификации ограниченного исходного кода накладывают на анализируемую программу специальные требования, выполнение которых позволяет произвести точный анализ, доказывающий отсутствие определенного класса дефектов

(верификацию). При использовании таких систем исходный код приводится в состояние, в котором система верификации не выдает предупреждений, что может предполагать существенную переработку программы (подготовку к анализу) для выполнения всех необходимых требований, или даже разработку, специально ориентированную на использование определенной системы верификации (что возможно только в некоторых случаях). Использование таких систем может быть внедрено в цикл разработки, при этом выдаваемые предупреждения по своей роли становятся аналогичными ошибкам компиляции, указывая новые дефекты по мере их возникновения. К этому классу относятся системы *PolySpace* [28] (см. сравнение с другими системами в [29]) и *ASTRÉE* [30], основанные на абстрактной интерпретации [31] и позволяющие доказать отсутствие таких ошибок, как переполнение буфера и разыменование нулевого указателя. Эти системы применялись для обеспечения надежности встраиваемого ПО в авиации. Для их использования требуется ограничение некоторых возможностей языка C, наличие исходного кода всех используемых библиотечных функций и ограниченный объем исходного кода анализируемых программ. Системы *BLAST* [32] и *SLAM* [33], применяемые для верификации драйверов, идут по другому пути, используя автоматическую проверку на модели. Для их работы требуется детальная спецификация используемых библиотечных функций, и при работе так же накладывается ограничение на объем исходного кода (порядка десятков тысяч строк).

Системы проверки пользовательских аннотаций позволяют выполнять ограниченную верификацию отсутствия (или поиск) сложных дефектов в программах, не накладывая на них таких строгих ограничений, как системы предыдущего класса. Это достигается за счет написания пользователем вручную аннотаций, указывающих требования к свойствам исходного кода. Роль системы проверки аннотаций состоит в том, чтобы проверить соответствие исходного кода указанным в аннотации требованиям. Такие системы выполняют роль, аналогичную проверке корректности типизации программы компилятором. При этом производится проверка только свойств, описанных аннотациями, и только в местах, где эти аннотации применяются, что позволяет выполнять анализ неполных программ и программ большого объема. Использование аннотаций позволяет постепенно внедрять статический анализ в цикл

разработки программы, но не позволяет производить исчерпывающий анализ на наличие определенных видов дефектов и требует написания большого количества аннотаций (подготовки исходного кода к анализу) для получения существенной отдачи от использования таких систем. Некоторые из систем проверки аннотаций ограничиваются поддержкой дополнительных спецификаторов типа, отсутствующих в языке программирования, указываемых при объявлении переменных и предназначенных для обнаружения определенного вида дефектов. Так, система *cqual* [34] производит проверку аннотаций, расширяющих систему типов языка C, для поиска уязвимостей форматной строки. Уязвимость форматной строки (*format string vulnerability*) возникает при возможности передачи управляемой пользователем строки в качестве форматного ввода функциям, таким как `printf`, что позволяет атакующему получить доступ к памяти. Более сложные системы *LCLint* [35], *Splint* [36] и *CSSV* [37] позволяют использовать пользовательские аннотации для обнаружения уязвимостей переполнения буфера и других дефектов.

Системы автоматического поиска дефектов предназначены для поиска сложных дефектов в большом объеме исходного кода. Эти системы требуют минимальной предварительной подготовки программы к анализу и стремятся выдавать как можно меньшее количество ложных предупреждений. Такие требования не позволяют производить точный анализ реальных программ, поэтому как правило приходится отказаться от строгости и консервативности анализа, в пользу поддерживаемых эвристическими алгоритмами методов, нацеленных на поиск только тех дефектов, наличие которых удается установить на основании собранной в результате анализа информации [38]. Одним из подходов в этом классе систем является поиск большого количества специальных шаблонов, описывающих возможные ситуации проявления дефектов или нарушения принятых для повышения качества кода соглашений. Шаблоны подбираются вручную таким образом, что ложные предупреждения для них выдаются достаточно редко, и даже в случае, когда предупреждение ложное, или шаблон не описывает непосредственно некоторый дефект, найденные конструкции в исходном коде являются стилистическим нарушением и подлежат исправлению. Таким образом, даже если предупреждения не всегда соответствуют реальным дефектам, большинство предупреждений не оказываются бесполезными и внесение

соответствующих исправлений повышает качество кода. К таким системам относятся *FindBugs* [39] и *Klocwork* [40]. Качество анализа в этом случае может быть повышено за счет добавления небольшого количества пользовательских аннотаций для функций, играющих роль в обнаруживаемых дефектах, и указания шаблонов, специфичных для анализируемой программы. Другие системы не требуют составления шаблонов для частных случаев и производят поиск дефектов на основании их семантики, как в случае верификации свойств исходного кода, но выдают только наиболее точные из возможных предупреждений. Системы [41], *PREfix* [42] и *Airac* [43, 44] оценивают точность полученных в результате анализа предупреждений на основании эвристик и указывают её в результатах, позволяя выделить наиболее качественные предупреждения. Системы *Splint* [36] и *ARCHER* [45] используют эвристики при выполнении нестрогого анализа таким образом, чтобы повысить качество выдаваемых предупреждений за счет пропуска некоторых из дефектов. Системы *MC* [46] и *Saturn* [47] используют поиск внутренних противоречий в программе, что позволяет избегать ложных предупреждений во многих случаях даже при существенной неточности анализа. Система *Coverity* [48] является коммерческим развитием системы *MC* и обнаруживает дефекты и уязвимости безопасности при помощи поиска противоречий и достаточно общих шаблонов. Система *CodeSonar* [49] обладает похожими возможностями.

4.2. Обфускация программ

Задача *обфускации программ* заключается в разработке таких преобразований, которые сохраняют функциональные характеристики программ, но при этом делают невозможным или чрезвычайно трудоемким извлечение из открытого текста программы полезной информации об устройстве алгоритмов и структур данных, содержащихся в исходной программе.

Простейшим методом обфускации является переименование объектов программы (переменных, структур и их полей, функций и т.п.). Казалось бы такая обфускация не вызовет серьезных трудностей для понимания программы. Однако если при переименовании объектов программы использовать необычные для программы

символы («_», «@», «~» и т.п.), что и делается в большинстве обфускаторов, то в результате такой простейшей обфускации программа, представленная на рис. 6, может превратиться в текст с рис. 7.

```
void primes(int cap) {
  int i, j, composite, t = 0;
  while(t < cap * cap) {
    i = t / cap;
    j = t++ % cap;
    if(i <= 1);
    else if(j == 0)
      composite = 0;
    else if(j == i && !composite)
      printf("%d\t",i);
    else if(j > 1 && j < i)
      composite += !(i % j);
  }
}

int main() {
  primes(100);
}
```

Рис.6. Пример программы до обфускации

```
_(__,___,____){___/__<=1?_(__,___+1,____):!(___%__)?_(__,___+1
,0):___%__==___/
__&&!____?(printf("%d\t",___/__),_(__,___+1,0)):___%__>1&&___%
__<___/__?_(__,1+
___,____+!(___/__%(___%__))):___<__*__?_(__,___+1,____):0;}mai
n(){_(100,0,0);}
```

Рис. 7. Программа с рис. 6 после обфускации

Так же как и шифрование, обфускация предназначена для информационной защиты программ. Однако, в отличие от шифрования, обфускирующие преобразования оставляют функциональность программ неизменной. Именно это особое качество обфускации открывает ей широкие потенциальные возможности применения в криптографии и компьютерной безопасности. Впервые о криптографических приложениях обфускации было упомянуто в работе Диффи и Хеллмана [50], но лишь спустя 20 лет в работе Колберга [51] было проведено первое подробное исследование проблемы обфускации программ.

В настоящее время можно выделить два основных направления исследований и разработок по обфускации программ. *Первое направление* – это теоретические работы по обфускации. Они ставят своей целью определить место обфускации в теории программирования, исследовать различные методы и алгоритмы, применяемые в обфускации, оценить их качество (в частности, стойкость) и сложность, найти связи обфускации с другими направлениями теории программирования и прикладной математики. *Второе направление* связано с разработкой и реализацией обфускаторов и обфускирующих компиляторов, имеющих целью обеспечить безопасность реальных программ и программных систем. Рассмотрим эти направления.

4.2.1. Теоретические исследования обфускации

Обфускация программ может быть применена для решения многих криптографических задач, например, преобразования криптосистем с секретным ключом в криптосистемы с открытым ключом (см. [50, 52-54]). Для этого достаточно подвергнуть обфускации программу, реализующую алгоритм шифрования с вставленным в нее секретным ключом. Преобразованную таким образом программу можно использовать в качестве программы шифрования криптосистемы с открытым ключом. Если обфускация является стойкой, то извлечение из открытого текста обфускированной программы встроенного в нее секретного ключа становится трудной задачей. Подобным же образом обфускацию программ можно применять для конструирования систем вычислений над зашифрованными данными [52, 55], для замещения модели случайного оракула в криптографических протоколах [52, 55], для обеспечения безопасности в иерархических системах распределенного доступа [55], для

создания верифицируемых систем тайного голосования [56], для обеспечения конфиденциальности в поисковых системах [57], базах данных [65] и др. Во всех перечисленных криптографических приложениях от обфускации программ требуется столь же высокая стойкость, как и от традиционных криптографических систем, используемых для решения подобных задач.

Не менее обширна и область возможных приложений обфускации для решения проблем компьютерной безопасности. Обфускирующие преобразования могут быть использованы для защиты интеллектуальной собственности на программное обеспечение [51, 58, 59], для информационной защиты мобильных агентов [60]. К сожалению, оказалось, что техника обфускации программ может быть также использована и в злонамеренных целях для разработки трудно обнаруживаемых «вирусов» [61 - 66], а также для сокрытия плагиата программного обеспечения [67]. Перспективы успешной разработки и применения обфускирующих преобразований в этом направлении представляются более определенными, поскольку для большинства коммерческих приложений проведение обфускации программ становится целесообразным, если издержки, которые вынужден понести противник для преодоления последствий обфускации, существенно превосходят издержки, вызванные применением самой обфускации.

Тем не менее ключевой проблемой при разработке обфускирующих преобразований является обеспечение стойкости обфускации. Впервые строгое математическое определение стойкости обфускации программ было предложено в работе [52]: обфускация считается стойкой, если всякий противник может извлечь из текста обфускированной программы за разумное (полиномиальное) время не больше информации, чем можно было бы получить, проводя тестовые испытания этой программы как «черного ящика». В этой же работе было установлено существование таких программ, для которых подобная стойкость обфускации в принципе не достижима. Впоследствии в ряде работ [68-72] были предложены и другие, менее требовательные определения стойкости обфускации; для большинства из них была показана невозможность построения эффективного транслятора, гарантирующего стойкую обфускацию произвольных программ. Вместе с тем, в работах [55, 69-71] было показано, что для отдельных классов функций (т.н. «точечных функций») стойкая

обфускация вычисляющих их программ возможна при тех или иных стандартных криптографических предположениях. Наиболее значительное продвижение было достигнуто в работе [73]; ее авторы доказали осуществимость стойкой обфускации программ перешифрования сообщений. В целом, однако, результаты исследований в этом направлении не дают больших оснований для оптимизма: до сих пор не удалось обнаружить достаточно сложной криптографической функции, программы вычисления которой допускают доказуемо стойкую обфускацию.

Несколько лучше обстоят дела с оценкой стойкости обфускирующих преобразований, применяемых для обеспечения компьютерной безопасности, поскольку здесь для многих приложений достаточно ограничиться экспериментальными исследованиями. Обфускация может быть признана приемлемо стойкой, если она способна противодействовать современным автоматическим средствам анализа программ. Если противнику открыт доступ лишь к исполняемому двоичному коду программы, то деобфускация программы начинается с преобразования ее двоичного кода в описание программы на языке высокого уровня. Эта процедура называется *обратной инженерией*; она состоит из двух этапов – *дизассемблирования* и *декомпиляции*. Некоторые методы обфускации программ используют особенности устройства двоичных программных кодов с целью затруднить дизассемблирование и декомпиляцию исполняемых файлов. Главная особенность двоичных кодов состоит в том, что машинное слово в зависимости от контекста может восприниматься как команда или как фрагмент данных. Обфускация исполняемого кода состоит в том, что обфускатор перемежает машинные команды с данными или несущественными словами («мусором»), и это затрудняет применение статических методов дизассемблирования. Метод обфускации машинных кодов, предложенный в работе [74], приводит к тому, что свыше 60% машинных команд обфускированной программы восстанавливаются дизассемблерами неправильно. В последующих работах [64-66, 75-79] развернулось настоящее исследовательское соревнование между проектировщиками дизассемблеров и разработчиками обфускаторов двоичных машинных кодов. Систематическое описание методов обфускации двоичных машинных кодов представлено в работе [77]. Эти работы свидетельствуют о том, что обфускаторы способны эффективно противодействовать статическим методам дизассемблирования. Однако использование

динамического дизассемблирования, при котором ассемблерный код программы восстанавливается в процессе ее выполнения, существенно снижает стойкость обфускации.

Наиболее изощренные методы обфускации программ используют т.н. непроницаемые предикаты, значения которых легко вычисляются на этапе компиляции и обфускации программ, но трудны для вычисления на этапе их анализа. Как правило, эти предикаты строятся на основе вычислительно трудных комбинаторных задач [66, 80-84]) или криптографических примитивов [85]. Более простые методы обфускации программ предполагают проведение специальных эквивалентных преобразований, приводящих программу к такому виду, при котором ее понимание становится затруднительным. Классификация этих преобразований проведена в работе [96, 86]. Эти методы разделяются на две группы.

- *Обфускация потока управления программы.* Цель обфускации – скрыть наиболее существенные индивидуальные особенности программ. Для этого управляющие структуры программ (потоки управления) приводятся к единообразной форме. Тело программы (главная функция) представляет собой переключатель (*switch*), моделирующий работу конечного автомата. На вход автомата поступают значения логических условий, в зависимости от которых управление передается той или иной функции, выполняющей линейную последовательность простейших операторов. Таким образом, у всех программ, имеющих одно и то же число линейных участков, графы потоков управления выглядят одинаково. Это подход был впервые применен в работе [81] и развит в работах [82, 87, 88].
- *Обфускация потоков данных программы.* Здесь целью обфускации является сокрытие зависимости по данным между переменными и функциями программы. Это достигается, например, за счет интенсивного использования переменных-указателей, функций-указателей и адресной арифметики. Поскольку даже простейшие задачи выявления синонимии переменных и оценки диапазона их значений имеют большую вычислительную сложность [89, 90], методы статического анализа не способны выявить зависимости по

данным между разными фрагментами обфускированных программ. Этот подход был систематически исследован в работе [92].

В работе [93] были предложены численные меры оценки качества обфускирующих преобразований на основе метрик сложности [94]. В серии работ [95-97] было показано, каким образом можно использовать теорию абстрактных интерпретаций программ для автоматического построения «непроницаемых» предикатов. Другой подход к оценке стойкости практически используемых обфускирующих преобразований связан с оценкой сложности проверки эквивалентности программ в алгебраических моделях программ.

Особое место занимают методы информационной защиты программ, близкие к методам обфускации и рассчитанные на использование высоконадежных вычислительных устройств. В этих методах исходная программа разбивается на две компоненты, одна из которых устанавливается на высокопроизводительном вычислительном устройстве, не имеющем надежных средств защиты, а другая - на высоконадежном (но, быть может, значительно менее производительном) вычислительном устройстве, которое обеспечивает достаточный уровень защиты от любого противника. Впервые этот подход к информационной защите программ был исследован в работе [98], где было показано, что с небольшими вычислительными издержками доступ к открытой памяти можно организовать так, что противнику по ходу вычисления не будет открыта никакая информация о выполняемой программе. В работах [99, 100] описаны способы построения сечений программ; сечение представляет собой небольшой фрагмент программы, который, будучи скрытым от противника, не позволяет ему получить никакой информации об устройстве всей программы. В работе [101] показано, что для некоторых вычислительных схем можно выбрать подходящий способ шифрования данных, который позволяет проводить вычисления над зашифрованными данными. Шифрование и дешифрование данных проводится на защищенном устройстве, а сама схема вычислений над зашифрованными данными считается открытой. Показано, что в этом случае вероятность правильного восстановления программы по ее открытой компоненте является пренебрежимо малой величиной.

Для противодействия динамическим методам дизассемблирования программ применяются специальные методы обфускации программ, приводящие к тому, что код программы постоянно видоизменяется в процессе выполнения. Впервые этот подход к защите программного обеспечения был представлен в работе [102]. Суть метода состоит в том, что фрагменты программы постоянно расшифровываются и перешифровываются в процессе ее выполнения, не позволяя тем самым противнику увидеть весь код программы целиком. В работе [103] этот подход был усилен тем, что код программы в процессе выполнения не только постоянно шифруется, но и изменяет свою собственную структуру (мутирует), используя генераторы случайных чисел. Этот метод обфускации активно используется создателями трудно диагностируемых полиморфных вирусов для того, чтобы скрыть характерные признаки вирусов («подписи») от антивирусных сканеров.

В настоящее время создано несколько действующих развитых систем (де)обфускации программ [88, 106], в которых используются различные комбинации описанных выше методов обфускации. Индивидуальные особенности каждой из этих систем определяются стратегией применения обфускирующих преобразований.

4.2.2. Обфускаторы и обфускирующие компиляторы

Наибольшее количество обфускаторов реализовано для программ на языках *Java* и *C#* (*.NET*) Это связано как с большой популярностью этих языков, так и с тем, что к программам на этих языках легко применимы методы обратной инженерии. Действительно, простота этих языков и наличие интерпретаторов существенно упрощают составление и отладку программ на этих языках, но эти же свойства существенно упрощают декомпиляцию программ, позволяя «пиратам» использовать их, либо их фрагменты в своих программах. Обфускаторы для этих языков призваны охранять интеллектуальную собственность авторов программ. Кроме того, они сообщают программе устойчивость к внешним воздействиям: обфускированную программу трудно модифицировать (например, для того, чтобы скрыть факт ее несанкционированного использования).

Типичным обфускатором является обфускатор *Spices* для среды *.NET* [142]. В нем используются как простые методы обфускации (переименование классов, их полей

и методов, внесение изменений в поток управления каждого метода, добавление в методы мертвого кода, шифрование строк и другие простые преобразования, рассмотренные в [51]), так и более сложные, основанные на использовании непрозрачных предикатов [51]. Кроме того, применяются оригинальные, запатентованные фирмой, методы обфускации: *анонимизация* кода (преобразование кода, после которого его становится невозможно использовать после дизассемблирования или декомпиляции) и вставка анонимных заглушек [142]. Большой популярностью пользуется и обфускатор *DashO* [143] для программ на языке *Java* и его аналог *Dotfuscator* [144] для среды *.NET*. Эти обфускаторы, выпускаемые компанией *PreEmptive Solutions*, используют все традиционные методы обфускации. Обфускаторы *DashO* и *Dotfuscator* не изменяют исходного кода обфускируемых программ, они изменяют их объектный код, обеспечивая защиту от декомпиляции. В рассмотренных обфускаторах реализована возможность указать фрагменты программы, которые необходимо обфускировать, так как обфускация всей программы может ухудшить ее эксплуатационные характеристики, в частности, время выполнения.

Можно упомянуть также обфускаторы для скриптовых языков *JavaScript* [145] и *Perl* [146], обфускаторы для языков C/C++ [147], [148] и др.

В заключение отметим, что постоянно растет интерес специалистов к теории и технике обфускации, во многих научных центрах проводятся исследования по этой тематике, постоянно появляются на рынке новые обфускаторы.

4.3. Обратная инженерия бинарного кода

Задача обратной инженерии бинарного кода состоит в восстановлении выполняющихся в программе алгоритмов и представлении прикладному аналитику описания этих алгоритмов на языке ассемблера, и, если это окажется возможным, на языке высокого уровня (например, на языке *C*). Отметим, что осуществить обратную инженерию бинарного кода автоматически зачастую оказывается невозможно. В этих случаях используются программные инструментальные средства, которые помогают специалисту (назовем его системным аналитиком) получить описание исследуемого бинарного кода на языке высокого уровня. Для этого системному аналитику

необходимее решить такие задачи, как разбиение исследуемого бинарного кода на процедуры, получение графа потока управления каждой процедуры, восстановление типов используемых переменных, восстановление арифметических и логических выражений, определение используемых в параметров процедур и др. Получив (вручную или с помощью программных инструментов) решения перечисленных задач, системный аналитик может построить высокоуровневое описание исследуемого бинарного кода, пригодное для изучения прикладным аналитиком.

Следует отметить, что анализируемый бинарный код мог быть получен в результате трансляции ассемблерного кода, либо в результате компиляции программы, написанной на высокоуровневом языке, таком как C/C++, либо представлять собой код виртуальной машины, выполняющей программу на интерпретируемом языке (примером такой машины является *JavaVM*). В последнем случае исследуемый код будет содержать бинарные константы, представляющие собой код интерпретируемой программы, описание которой и нужно получить.

В каждом из рассмотренных случаев бинарный код обладает специфическими свойствами. Например, ассемблерный код позволяет задействовать в программе недокументированные машинные команды. Бинарный код, полученный из компилятора языка C++, может реализовывать различные методы вызова функций, в частности, через таблицы виртуальных методов (если такие в программе имеются), вызов так называемых «голых» функций, не имеющих пролога и эпилога и т.п.

Более того, бинарный код может быть снабжен средствами защиты от обратной инженерии (в последнее время таких случаев становится все больше). Для такого кода необходимо решить дополнительную задачу – преодоление защиты.

Рассмотрим некоторые виды защиты кода, противодействующие статическому и динамическому анализу.

(1) *Упаковка кода*. Код программы хранится в сжатом (зашифрованном) виде, вспомогательная программа распаковывает код и помещает его в оперативную память, после чего передает на него управление. Данный подход делает неприменимым непосредственное дизассемблирование кода, хранящегося на диске.

(2) *Противодействие отладке*. Существует ряд приемов, позволяющих обнаружить присутствие отладчика. В случае обнаружения со стороны программы

предпринимаются действия: меняется работа алгоритмов, программа перестает работать, «портятся» данные отладчика. Противодействие отладке преодолевается применением потактовых симуляторов. В этом случае обнаружение отладки возможно только из-за ошибок в симуляторе, приводящих к отличному от аппаратной платформы поведению.

Наиболее простым для анализа является незащищенный бинарный код, полученный компиляцией программы с языка высокого уровня. Существует целый ряд методов, позволяющих идентифицировать в коде конструкции исходного языка. Перечислим некоторые такие конструкции.

Идентификация функций является наиболее важным методом поднятия абстракции, поскольку позволяет уйти с уровня машинных инструкций. Структуризация происходит за счет анализа перекрестных ссылок, выявления пролога/эпилога функции, сопоставления заранее известной сигнатуры, идентификации таблиц виртуальных функций. Помимо того, известны специализированные подходы для выявления точек входа, конструкторов и деструкторов.

Идентификация переменных предлагает различные подходы для выявления локальных и глобальных переменных, переменных, расположенных в «куче». Следует отметить, что успешно применятся техника заимствованная у оптимизирующих компиляторов: анализ указателей и его более общий вариант – *shape*-анализ [108].

Идентификация управляющих структур весьма близка к структурному анализу, проводимому при компиляции программ. Однако в случае бинарного кода так же используется иск характерных последовательностей команд, например для оператора switch, ищется реализация метода обрезки ветвей или таблица переходов.

Таким образом, восстановление алгоритма требует от системного аналитика, помимо владения языком ассемблера, языками C/C++, целый перечень специфических знаний, таких как недокументированные свойства аппаратных платформ, особенности работы операционных систем и сетевых протоколов и т.п. Однако успешная работа, как правило, обеспечивается еще и целым рядом инструментальных программ, избавляющих аналитика от монотонной работы и делающих некоторые виды анализа в принципе осуществимыми.

В настоящее время оснащение системного аналитика состоит из отладчика, дизассемблера, шестнадцатеричного редактора. Кроме того, существуют различные программные инструменты, такие как распаковщики, дамперы, редакторы ресурсов, «шпионы», мониторы. В практической работе, как правило, используется связка из нескольких инструментов, определяемая особенностями решаемой задачи. Рассмотрим некоторые из таких инструментов.

Отладчик *SoftICE* [109, 110] – наиболее широко используемый отладчик для платформы *x86*/*Windows*, поддерживающий отладку в режиме ядра. Проект был закрыт в 2007 г.

Главная особенность отладчика *SoftICE* заключается в использовании аппаратных возможностей отладки, доступных начиная с процессора 80386 (группа отладочных регистров *DR0-DR7*). Аппаратная отладка позволяет задавать четыре линейных адреса памяти, при попадании счетчика инструкций на какой-либо из этих адресов срабатывает аппаратное прерывание. В случае, когда включен страничный режим адресации, перевод линейных адресов в физические осуществляется средствами процессора. Таким образом, *SoftICE* перехватывает управление на аппаратном уровне, что позволяет вести отладку драйверов устройств и ядра операционной системы. В литературе [111] были рассмотрены примеры, когда средствами *SoftICE* удалось избежать краха операционной системы при сбое драйвера.

Помимо того, *SoftICE* может использоваться и как обычный отладчик пользовательского уровня. Особенностью являются интерфейсы данного отладчика, пользовательский интерфейс практически не менялся на протяжении всего проекта, будучи основанным на текстовом отображении информации и «горячих клавишах», для разработчика доступно развитое *API*, позволяющее автоматизировать выполнение определенных сценариев работы. Пользовательский интерфейс *SoftICE* с минимальными изменениями был задействован в другом отладчике уровня ядра *Linice* [112], работающем на платформе *x86*/*Linux* на базе тех же аппаратных средств отладки.

Одним из известных дополнений к *SoftICE* является утилита *IceExt* [113]. Данная утилита позволяет сохранять дампы памяти в файл и позволяет обходить ряд «противоотладочных» приемов, используемых для защиты программ. Сохранение дампа памяти позволяет обходить защиту бинарного кода запаковкой. Однако

воссозданные из дампа памяти исполняемые файлы могут работать неустойчиво, т.к. при распаковке кода происходит привязка к фиксированным адресам виртуальной памяти. Кроме того, защита, предусматривающая распаковку исполняемого кода в память по частям, по мере необходимости, не может быть таким образом преодолена. Примером распаковщика, использующим такой подход, является *Armadillo* / *Software Passport* [114]. Данный паковщик хранит код приложения в зашифрованном виде, и загружает его в память постранично. Поддерживается также и режим, когда расшифровке и выполнению подвергаются отдельные инструкции, но такой подход страдает из-за значительной потери производительности и невозможности применить устойчивые к взлому механизмы шифрования.

Помимо *SoftICE* следует отметить свободно распространяемый отладчик *OllyDbg* [115], позволяющий вести отладку только на прикладном уровне. Как и у *SoftICE, OllyDbg* имеется механизм подключения модулей-расширений. На данный момент существует сообщество энтузиастов ведущих разработку расширений и дополнений для данного отладчика. Среди дополнений доступны средства скрытия отладчика от средств защиты, средства определения оригинальной точки входа (*OEP*) в упакованной программе и т.д.

С середины 2005 года китайскими разработчиками ведется проект *Syser* [116] – отладчик уровня ядра. Одна из особенностей данного отладчика – поддержка *SMP*-машин.

Следует отметить также отладчик уровня ядра *WinDbg*, выпускаемый компанией *Microsoft*, и входящий в состав набора программных инструментов *Debugging Tools* [117]. Отладчик весьма требователен к квалификации аналитика, но в основной функциональности не превосходит возможности *SoftICE*.

Дизассемблер *Ida PRO* [118] – самое широко распространенное средство дизассемблирования программ для различных процессорных архитектур. Является наиболее развитым программным средством статического анализа программ. Предоставляет гибкие возможности навигации по коду, выполняя структурирование, производит анализ перекрестных ссылок, автоматически идентифицирует локальные переменные. Результат дизассемблирования представляется с учетом графа потока управления. В отличие от остальных дизассемблеров *Ida PRO* рассчитана на

интерактивную работу, что позволяет в обходить некоторые виды защиты, в частности переход в середину инструкции. Помимо того, *Ida PRO* предоставляет *API* расширения, позволяющее разрабатывать в рамках этой системы собственные анализаторы.

Примером такого расширения является декомпилятор *HexRays* [119]. Данное средство генерирует С-подобный пвсевдокод, выполняя при этом восстановление управляющих структур и выражений, идентифицирует встроенные в код строковые функции. При построении описания функций в коде идентифицируются формальные параметры; пользователю доступна визуализация отображения локальных переменных уровня псевдокода на регистры в соответствующих машинных инструкциях. Помимо того, декомпилятор *HexRays* способен заменять управляющие конструкции на выражения короткой логики.

Другие известные декомпиляторы, такие как *Boomerang* [120], *Interactive Decompiler* [121] обладают гораздо более скромными возможностями и меньшей по сравнению с *HexRays* областью применимости.

Таким образом, в случае незащищенного кода среда *Ida PRO* позволяет получать описание алгоритма пригодное для передачи прикладному аналитику.

В случае защищенного (в частности, обфускированного) кода приходится использовать динамический анализ, чтобы выявить истинный код, который необходимо дизассемблировать. Это существенно усложняет задачу дизассемблирования и декомпиляции. Следует отметить, что обфускация программ в настоящее время активно внедряется в практику разработки программ. Так, в состав среды разработки *Microsoft .NET*, включен программный продукт *Dotfuscator* [122], который представляет собой обфускирующий компилятор языка *C#*. Развитие обфускирующих компиляторов приведет к тому, что практически все программы будут иметь защищенный код. Результатом этого будет принципиальная неприменимость средств статического анализа кода, основанных на дизассемблировании полученных тем или иным способом машинных инструкций. Проблемный аналитик в этом случае будет сталкиваться с ситуацией, когда даже наличие машинного кода не позволяет восставить пригодное для прикладного анализа описание алгоритма, поскольку выполнившееся защитное преобразование кода является односторонним в том смысле, что обратное преобразование либо не существует, либо является очень сложным.

Для таких случаев перспективным представляется подход, развиваемый в Институте системного программирования РАН, когда бинарный код программы подвергается динамическому анализу. Каждая полученная трасса программы отфильтровывается с учетом зависимостей по данным между инструкциями и структурируется. После этого строится граф потока управленбия и над инструкциями бинарного кода выполняются некоторые из оптимизирующих преобразований, рассмотренных в разделе 2: удаление мертвого кода, упрощение выражений, распространение констант и копий и др. В результате получается компактное описание выполнившегося алгоритма. Последующее применение дизассемблера и декомпилятора позволит поднять уровень абстракции, описывая суть выполнявшихся преобразований данных.

5. Генерация кода по спецификациям

С самого появления модельно-ориентированных подходов в разработке ПО разработчики стараются свести до минимума «разрыв» между абстрактной моделью и ее реализацией. Генерация кода по модели позволяет сократить время разработки и сократить количество несоответствий между моделью и конечным продуктом. Кроме того, генерация выполнимого кода по моделям позволяет быстро создать прототип на ранних этапах проектирования системы. Выполнение такого прототипа помогает обнаружить архитектурные ошибки и, за счет раннего обнаружения, заметно сократить их стоимость.

С появлением объектно-ориентированного подхода в разработке ПО возникла потребность в методологии и инструментах для проектирования. К началу 90-х годов было написано множество книг по этой тематике, однако каждый автор использовал свои собственные определения, терминологию и нотацию. В 1995 году Якобсон, Рамбо и Буч совместными усилиями разработали Унифицированный Язык Моделирования (*Unified Modeling Language*) [123]. После этого стандартизацией *UML* занялся концерн *OMG*, и в 1997 году была выпущена первая версия языка.

На ранних этапах проектирования программного продукта используются диаграммы вариантов использования. Они позволяют выделить границы системы, ввести необходимую терминологию и определить внешних «пользователей» системы (как людей, так и другие системы, взаимодействующие с данной). Рассматриваемая система в терминологии *UML* называется субъектом. Внешние сущности, взаимодействующие с системой, называются экторами (actor). Так же варианты использования разрабатываются для компонентов системы – в этом случае субъектом будет компонент системы, а акторами – другие компоненты (или, опять таки, внешние сущности).

Диаграмма классов позволяет описывать классы системы, их структуру – атрибуты, их операции и взаимоотношения между ними – наследование и ассоциации. На диаграмме объектов описываются экземпляры классов. На диаграмме состояний описывается поведение компонента системы в виде конечного автомата – множества состояний и переходов между ними. На диаграмме последовательностей описывается

последовательность сообщений, передаваемых между компонентами системы в процессе некоторой транзакции. На диаграмме активностей определяется последовательность шагов в ходе выполнения некоторой задачи. Основными элементами диаграммы являются действия (actions), связаные между собой переходами.

Инфраструктура языка *UML* описывается самим *UML*. Это описание называется метамоделью. Например, возможность определения литерала для типа данных определяется наличием ассоциации между элементами *DataType* и *Literal*.

Профили позволяют приспосабливать *UML* к конкретным предметным областям, не изменяя при этом метамодель языка. Существует три основных типа конструкций для расширения: ограничения, стереотипы и теговые значения. Ограничения описываются в виде формального или неформального текста. Стереотип позволяет вводить новые типы элементов модели на основе существующих. Теговые значения содержат некоторую дополнительную информацию, относящуюся к некоторому элементу модели. Набор ограничений и стереотипов, предназначенный для некоторой предметной области, является профилем.

5.1. Генерация кода по статическим моделям

Для генерации кода по *UML* не существует никаких стандартов, да и вряд ли они когда-нибудь появятся. Это связано с тем, что не во всех языках есть поддержка всех концепций *UML* (например, *Java* не поддерживает множественное наследование классов, которое допустимо в *UML*). Кроме того, генерации кода должна учитывать требования пользователя при выборе вариантов сгенерированного кода. Например, одни пользователи могут потребовать, чтобы агрегация между классами представлялась указателем, а другие – ссылкой. Атрибут с множественными значениями может быть представлен как вектором, так и списком. Для этого необходимо предусмотреть настройку кодогенерации.

Различные инструменты решают эту задачу по-разному. Некоторые (*Borland Together* [127]) предоставляют стандартное отображение модели *UML* в язык программирования. При этом пользователю не позволяется создавать модели, которые

не могут быть отображены в целевой язык. Другие инструменты (*BOUML* [130]) предлагают пользователю полностью определять, как будет генерироваться код как для типа элемента (например, для интерфейса), так и для конкретного элемента модели. Третьи системы (*Telelogic Tau* [124], *Objecteering Modeler* [126]) используют промежуточный вариант: способ кодогенерации определяется специальным компонентом. В комплект инструмента входят стандартные компоненты для генерации C++, *Java*, C# и т.д. Если стандартный компонент не удовлетворяет пользователя, то есть возможность создать свой компонент, генерирующий такой код, который пользователь хочет видеть.

Во всех случаях, когда пользователю предоставляется возможность описания кода, генерируемого по модели, он должен оперировать элементами метамодели. Например, для генерации операции в системе *BOUML* пользователь пишет текст, содержащий специальные переменные, соответствующие элементам метамодели (`class ${name} {…}`). Система *Telelogic Tau* поддерживает расширение кодогенерации при помощи агентов – пользовательских функций, принимающих элемент модели и использующих *API* инструмента для доступа к его свойствам.

Так как процесс моделирования, как правило, является итеративным, код на целевом языке перегенерируется много раз. При этом важно не потерять изменения, внесенные пользователем в сгенерированный код. Причем изменения в коде могут быть двух типов: в одном случае изменяется часть кода, представленная в *UML* модели (например, тип аттрибута), в другом – часть кода, не присутствующая в модели (например, тело функции).

Для первого случая изменения, сделанные в сгенерированном коде, должны быть внесены в исходную модель. Инструмент должен по целевому коду построить *UML* модель и «слить» ее с существующей. Такой подход называется *round trip*. Свойства модели, не представимые в целевом языке (например, тип ассоциации), должны либо игнорироваться, либо для них должны генерироваться специальные конструкции на целевом языке. Например, *Telelogic Tau* для множественных атрибутов генерирует тип `String<T>` (где `T` – тип атрибута), и при внесении изменений в модель атрибут, типизированый `String<T>` становится множественным атрибутом с типом `T`.

Инструмент *Borland Together* использует специальные поля в *Javadoc*-комментариях для сохранения информации о модели.

```
public class Class1 {
...
/**
 * @link aggregation
 */
private Class2 lnkClass2;
...
}
...
public class Class2 {
...
}
```

5.2. Генерация кода по динамическим моделям

Генерация кода по динамическим моделям *UML* может служить как для верификации модели путем ее симуляции, так и для получения рабочего кода, который будет использован впоследствии либо как прототип, либо как часть целевой системы.

Стандарт *UML* не предусматривает описания в модели управляющих конструкций (*statements*), как в языках программирования. Однако многие инструменты дают возможность их описания. *Telelogic Tau* расширяет язык UML текстовой нотацией, в которой можно описывать такие управляющие конструкции, как циклы, условные операторы и т.д. В *Telelogic Rhapsody* пользователь определяет действия, используя синтаксис целевого языка. В *Borland Together* и *Objecteering Modeler* пользователь может редактировать тело метода прямо в сгенерированном коде.

Когда в сгенерированный код вносятся изменения, не относящиеся к *UML* модели (например, тело метода), при последующей генерации они должны быть сохранены. В *Borland Together* эта проблема решается средствами среды *Eclipse*, которые позволяют строить синтаксическую модель по коду на *Java*. В других

инструментах при генерации пользовательский код обрамляется специальными комментариями.

В тех случаях, когда реализационный код пишется пользователем на целевом языке, его синтаксическую и семантическую корректность может проверить только компилятор целевого языка. В таких системах, как *Borland Together*, генерация кода и его синтаксическая и семантическая проверка производятся «на лету», и таким образом пользователь видит свои ошибки сразу после того, как он их внес.

В системе *Telelogic Tau*, которая использует свой синтаксис для управляющих конструкций, проверка синтаксиса и семантики производится до кодогенерации. Это обеспечивает генерацию корректного кода, однако усложняет саму кодогенерацию – управляющие конструкции требуется корректно отобразить в данный целевой язык, что не всегда тривиально. Так же требуется обратное преобразование при round trip разработке.

5.3. Генерация кода по конечным автоматам

Большая часть инструментов, поддерживающих кодогенерацию по *UML* моделям, не предоставляют возможность генерации целевого кода по конечным автоматам. Причиной этому является, по видимому, неформальность конечных автоматов *UML* – нет четкого описания семантики их исполнения. Тем не менее, такие инструменты, как *Telelogic Tau*, *SmartState* [128] и *PathMate* [129] определяют собственную семантику исполнения и позволяют определять поведение компонентов *UML* модели в виде конечных автоматов с последующей генерацией исполнимого кода.

Инструмент *Telelogic Tau* для генерации кода на C использует семантику исполнения языка *SDL*. *UML* модель трансформируется в эквивалентную модель на *SDL*, после чего средствами поддержки *SDL* производится генерация кода на C.

В случае других систем, как и при генерации *C++* или *Java* в *Telelogic Tau*, используются соответствующие библиотеки поддержки времени исполнения. Такая библиотека должна обеспечивать планировщик (причем выполнение компонентов модели может быть сериализовано или производиться в разных потоках управления) и средства коммуникации исполняемых компонентов (очереди сообщений). При этом

конечные автоматы трансформируются в управляющие конструкции, поддерживаемые целевым языком.

5.4. Симуляция модели

Описание модели в виде конечного автомата достаточно наглядна, однако она может содержать различные ошибки – например, взаимную блокировку взаимодействующих процессов. Для проверки правильности модели может быть произведена ее симуляция. На основе модели генерируется код, который затем собирается в исполнимый модуль. Полученная программа генерирует события, соответствующие переходам в конечном автомате, посылке сообщений и т.д.

Система *SmartState* предоставляет возможность анализа выполнения сгенерированного приложения про помощи журнала, который содержит записи о происшедших событиях (переход между состояниями, выполнение действий, соответствующих входу в состояние и выходу из него и т.д.).

Система *Telelogic Tau* для выполнения симуляции дает пользователю наглядное представление исполнения симулируемой модели, возможность пошагового выполнения, а так же возможность посылать сигналы компонентам модели как извне, так и от имени других компонентов. Поддержка времени исполнения сгенерированного кода обеспечивает средства для коммуникации выполняемого кода со средой, а генератор кода – возможность инструментации генерируемого кода с использованием этих средств.

В режиме симуляции среда *Telelogic Tau* предоставляет возможность пошагового выполнения модели. При этом на диаграмме состояний помечаются выполняемые переходы между состояниями.

Также результат выполнения может быть представлен системой в виде диаграммы последовательностей, чтобы пользователь мог оценить правильность событий.

Диаграммы активности задают более абстрактную спецификацию поведения системы, чем конечные автоматы. Однако, и такие модели могут быть верифицированы подобным образом, и система *Telelogic Tau* содержит средства для этого. Диаграмма

активности преобразуется в конечный автомат с поведением, эквивалентным поведению, описанному при помощи диаграммы активностей. По полученному конечному автомату создается исполнимая программа, которая может быть запущена симулятором выполнения модели. События, происходящие при выполнении программы, отображаются системой в переходы на диаграмме активностей и демонстрируются пользователю.

5.5. Другие задачи кодогенерации

Задачей не последней важности является читабельность сгенерированного кода – в особенности, если предполагается дальнейшая его модификация. Так же удобство в разработке добавляет возможность навигации между элементами модели и кодом на целевом языке.

В зависимости от целей кодогенерации для пользователя могут быть важны такие свойства сгенерированного кода, как производительность, компактность, так же должны учитываться ограничения платформы и инструментов последующей сборки. Эта проблема решается возможностью настройки кодогенератора. Пользователь может добавлять текст перед и после сгенерированных элементов, или даже предоставить полностью свой способ представления элементов модели на целевом языке.

6. Обратная инженерия и верификация программ

В современной компьютерной индустрии разработка программных систем строится в соответствии с некоторой моделью процесса разработки программного продукта. Наиболее известными и распространенными процессами разработки программных систем является водопадный (*waterfall*) и итеративный (*agile*) процессы разработки.

Водопадный процесс характеризуется свойством последовательного однократного выполнения этапов жизненного цикла программного продукта: сбор требований, проектирование, реализация, тестирование, эксплуатация. При проектировании создается модель программной системы, которая в процессе реализации уже не изменятся.

Итеративные процессы характеризуются свойством неоднократного последовательного повторения этапов сбора требований, проектирования, реализации, тестирования и ввода в эксплуатацию. В процессе каждой итерации программный продукт проходит фазы сбора требований, проектирования, реализации, тестирования и ввода в эксплуатацию новой версии программного продукта. Итеративные процесы разработки программного продукта позволяют достаточно быстро реализовать и запустить в эксплуатацию наиболее важные функции программной системы, а в последствии добавить к ней второстепенные или изменить поведение программной системы.

Применение различных процессов, как правило, зависит от типа программного продукта и области, в которой он будет применяться. Если требования к программному продукту фиксированы и не будут меняться в процессе его создания, то лучше подходит водопадный процесс. Примером таких продуктов может быть создание управляющих программ (драйверов) для различного рода устройств. С другой стороны, если требования не четки и могут изменяться или уточняться в процессе жизненного цикла программного продукта, то наиболее подходящим процессом будет итеративный процесс разработки программного продукта. К таким программным продуктам можно отнести, например, программы управления предприятием, в процессе жизненного

цикла которых изменяются или добавляются правила или области автоматизации производственных операций.

Независимо от используемого процесса разработки программного обеспечения практикуется как реализация проекта по разработке программного обеспечения внутри организации, так и передача проекта на реализацию сторонним организациям. Более того, нередко проект передается от одного подрядчика к другому.

В результате, в процессе жизненного цикла программных систем возникают следующие задачи:

- Анализ архитектуры реализованной программной системы;
- Моделирование изменения программной системы с последующей верификацией реализованного изменения на соответствие модели;
- Автоматический перенос изменений модели программной системы на исходный код;
- Исследование реализованной программной системы на наличие программных дефектов и уязвимостей;

Решение перечисленных задач возможно путем восстановления моделей реализованной программной системы из исходного кода (обратная инженерия, *reverse engineering*) и последующего анализа этих моделей.

Рассмотрим различные варианты использования моделей программной системы.

(1) *Архитектурный анализ*. Анализ визуального представления модели человеком с целью понимания фактического внутреннего устройства программной системы. Программные инструменты, обеспечивающие решение подобных задач включены в такие продукты как *IBM Rational Software Architect* [131] из пакета программного обеспечения *Rational Rose* компании *IBM, Klocwork Architectural Analysis* из пакета программного обеспечения *Klocwork InSight* [132] компании *Klocwork, Coverity Structure 101* [133] компании *Coverity*. Эти инструменты позволяют:

- осуществлять автоматический поиск архитектурных шаблонов в модели программного обеспечения с целью упрощения модели и облегчения анализа; (*IBM Rational Rose , Klocwork Architectural Analysis*)

- использовать извлеченную модель для сравнения с моделью изменения исходной программной системы; (*Coverity Structure 101, IBM Rational Software Architect, Klocwork Architectural Analysis*)
- модифицировать исходный код в соответствии с изменениями на извлеченной модели;
- осуществлять статический анализ модели программной системы с целью выявления различного рода дефектов в коде программной системы. (*Coverity Prevent, Klocwork InSight, Parasoft C++Test, Parasoft JTest*)

Анализ представления модели программной системы обычно возникает при изучении унаследованной программной системы новой командой разработчиков или при изучении программной системы сторонних разработчиков с целью понимания внутреннего устройства программной системы. Подобная модель может являться дополнением к спецификации и документации программной системы.

Для упрощения понимания внутреннего устройства программной системы человеком (как правило, это архитектор системы, отвечающий за высокоуровневую организацию программной системы) применяют различного вида графические представления архитектурных моделей. [132]

(2) *Анализ подсистем и рефакторинг.* В процессе анализа производится исследование архитектурных блоков и связей между блоками. В качестве примера можно рассматривать как блок верхнего уровня систему целиком; промежуточный блок – библиотека, класс; блок самого нижнего уровня – метод или функция программной системы. Связи между блоками: вызывает, наследует, содержит. На основании анализа связей между блоками возможно разделение программной системы на слои и подсистемы. Подобное разделение программной системы на подсистемы упрощает её анализ путем уменьшения количества информации (архитектурных блоков и связей между ними), необходимого для анализа определенной части системы.

Если оказывается возможным разделение системы на основании некоторого правила (шаблона), то возможен автоматический поиск подсистем, соответствующих этим правилам (шаблонам). Автоматический поиск подсистем позволяет ускорить анализ исходной программной системы, например благодаря автоматическому извлечению общеизвестных отношений между подсистемами – шаблонов

проектирования [133] и их выделения или сворачивания в один архитектурный блок на диаграмме архитектурной модели. [134]

Приведенные выше подходы к анализу архитектуры программных систем позволяют не только упрощать понимание внутреннего устройства программной системы, но и позволяют осуществить *рефакторинг* системы, т.е. произвести моделирование архитектуры программной системы с целью ее улучшения: избегать неоправданного дублирования подсистем, обнаруживать сильно связанные компоненты и прочее. Возможные подходы к решению задач рефакторинга описаны в [135]

(3) *Контроль соответствия предварительной и окончательной архитектур*. Интересной с точки зрения контроля развития программной системы является задача контроля соответствия архитектуры созданной на этапе проектирования программной системы, архитектуре реализованной программной системы. В рамках решения этой задачи изменение исходной системы сначала моделируется на архитектурной модели исходной программной системы. Например, архитектурная модель исходной программной системы изменяется в соответствии с новыми требованиями к системе или с целью улучшения архитектуры (рефакторинга). Далее изменения вносятся в исходный код программной системы, извлекается модель измененной программной системы и производится сравнение моделей измененной и извлеченной после изменения исходного кода. Данный подход позволяет на раннем этапе выявить несоответствие реализации на уровне архитектурной модели и в наглядном виде представить различия.

(4) *Возвратная разработка*. Задачу моделирования изменений на извлеченных архитектурных моделях можно развить в направлении применения изменений на модели обратно на исходный код. Такой подход носит название круговой или возвратной разработки (*round-trip engineering*). В данном случае вместо выдачи инструкций разработчикам на изменение исходного кода системы архитектор, после изменения модели, может автоматически изменить существующий или сгенерировать новый программный код, соответствующий изменению на модели (с определенными ограничениями на вид изменения модели). [136]

Одновременно с задачами архитектурного анализа и рефакторинга при восстановлении модели программных систем могут решаться задачи обнаружения различного рода дефектов в программном обеспечении.

(5) *Метрики*. При разборе исходного кода программной системы происходит не только извлечение архитектурной модели программной системы, но и подсчет всевозможных метрик, описывающих анализируемую программную систему и её компоненты (количество строк кода, сложность функций и методов и др.). Кроме того, можно выполнить извлечение других моделей (дерево синтаксического разбора, граф потока управления и др.), на которых можно производить поиск логических дефектов и уязвимостей программной системы.

(6) *Верификация*. При верификации ПО ставятся следующие цели:

- Проверка программной системы на соответствие спецификации (тестирование)
- Проверка на надежность (устойчивость к взлому, наработка на отказ и др.)
- Проверка качества реализации (отсутствие дублирующихся частей кода, переносимость на другую платформу, стиль кодирования и др.)

Существует несколько подходов к верификации программ. [137] Их можно разделить по видам анализа (динамический и статический). К динамическому анализу можно отнести такие методы верификации программ, которые связаны с запуском программы на выполнение в некоторой среде с одновременным анализом реакции программной системы на различные наборы входных данных. К статическим методам анализа относят всевозможные методы анализа моделей программ без запуска их на выполнение или эмуляции выполнения (например, на специальной виртуальной машине).

У каждого из рассмотренных подходов есть слабые и сильные стороны. В процессе динамического анализа программной системы есть возможность точно определить наличие проблемы (несоответствие спецификации, нестабильную работу программы при определенном наборе входных данных и др.), а также динамический анализ позволяет проводить измерение производительности работы программной системы при различных входных данных. Однако стоит заметить, что динамический анализ не дает гарантии покрытия всех возможных путей исполнения программной

системы. Попытка проведения полного анализа (анализа всех путей при всех возможных значениях входных параметров) может привести к «комбинаторному взрыву» и, как следствие, резкому увеличению времени анализа.

С другой стороны статический анализ (анализ модели программной системы) позволяет управлять точностью и временем анализа, но не всегда может гарантировать как нахождение проблемы, так и её действительное наличие. В данном контексте выделяют такие критерии качества анализа как время анализа [140], наличие и количество ложных сообщений о дефектах, а также наличие и количество не обнаруженных реальных дефектов в программе.

(7) *Использование моделей программной системы для ее верификации*. Статический анализ можно проводить по разным моделям программной системы. Обычно выделяют два класса анализируемых моделей: анализ дерева синтаксического разбора и анализ потоков данных.

Анализ дерева синтаксического разбора программной системы позволяет эффективно обнаруживать такие классы дефектов как:

- дефекты соответствия типов операндов;
- семантические дефекты;
- дефекты оформления исходного кода (соглашение о наименовании, стиль кодирования) и др.

Поиск дефектов на дереве синтаксического разбора программной системы сводится к поиску шаблонов поддеревьев, вершины которых обладают определенными свойствами.

Другие классы дефектов удобнее находить, используя модель потоков данных программной системы. Модель потоков данных отражает изменение состояния переменных программной системы в различных точках графа потока управления. Граф потока управления описывает порядок выполнения инструкций программы. Анализ распространения данных системы при переходе между вершинами графа потока управления позволяет находить шаблоны ошибочного или потенциально опасного поведения программной системы. При этом возможно указание места, где происходит инициирование ошибочной ситуации, места её фактического возникновения и полного

пути по графу потока управления между ними с отображением значений всех промежуточных условий.

Необходимо отметить, что в процессе реализации анализаторов программного кода приходится искать компромисс между следующими противоречивыми требованиями:

- *Обеспечение точности анализа*. Зависит от правильности описания шаблона поиска дефекта и точности модели, а именно: количества анализируемых данных, точности вычисления условий и диапазонов значений. В связи с этим анализатор может утверждать о наличии дефекта в программной системе с различной степенью достоверности.
- *Полнота анализа*. Количество обнаруживаемых классов дефектов из множества известных классов дефектов.
- *Производительность*. Время работы анализатора и количество потребляемой при анализе оперативной памяти напрямую зависит от точности и полноты анализа.

Статические и динамические методы верификации программ иногда применяются вместе. Например, при помощи методов статического анализа вычисляются условия обхода путей исполнения по графу потока управления и подбираются тестовые сценарии (наборы входных данных, для которых вычисляется результат работы системы), которые позволяют построить полное покрытие путей в программе или пройти определенный путь. В последствии эти тестовые сценарии позволяют за ограниченное время проводить тестирование программной системы в целом или её отдельных компонент. [141]

В совокупности перечисленные варианты верификации программных систем на базе методов обратной инженерии позволяют построить формальный процесс контроля качества программных систем с учетом различных аспектов и показателей.

7. Заключение

В данном обзоре рассмотрены различные аспекты применения методов анализа и трансформации программ – одного из важнейших направлений системного программирования. В последние годы идет бурное развитие этого направления системного программирования, что связано с несколькими факторами: (1) необходимость массового внедрения новых архитектур, в том числе встроенных систем, требующих новых технологий оптимизации программ (параллелизм на разных уровнях, энергосбережение); (2) возможность существенного повышения качества и безопасности программного обеспечения в ситуации, когда все ПО так или иначе доступно по сети (Интернет/Интранет) и существующие модели и методы защиты (например, защита по «периметру») не в состоянии отвечать на новые вызовы; (3) обеспечение существенного роста производительности труда разработчиков прикладного ПО, что является необходимым условием дальнейшего развития индустрии ПО и др.

Следует отметить, что высококвалифицированных специалистов по данному направлению недостаточно (это мировая тенденция), еще меньше коллективов разработчиков, способных решать конкретные задачи, в том числе разрабатывать новые технологии. Наш опыт показывает, что даже при наличии квалифицированных специалистов создание такого коллектива требует от трех до пяти лет. Таким образом, необходимо поддерживать фундаментальные исследования и прикладные разработки в рассмотренных областях программирования с целью как получения новых технологий, так и создания соответствующих коллективов.

Кроме того, авторы надеются, что обзор поможет ориентироваться в многочисленных исследованиях и публикациях на эту тему и может быть полезен при поиске наиболее актуальных проблем, решение которых связано с применением методов анализа и трансформации программ.

Литература

1. Компилятор Gcc. http://gcc.gnu.org
2. Внутренняя документация компилятора Gcc. http://gcc.gnu.org/onlinedocs/gccint
3. Компиляторная инфраструктура LLVM. http://llvm.org/
4. Diego Novillo. Design and Implementation of Tree SSA. 2004 GCC Developers' Summit, pp. 119-130, Ottawa, Canada, June 2004.
5. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы: принципы, технологии, инструментарий. Москва, Вильямс, 2008 г. Стр. 719-760.
6. Steven Muchnick. Advanced compiler design and implementation. Morgan Kaufmann, 3rd ed., 1997.
7. Diego Novillo. A Propagation Engine for GCC. 2005 GCC Developers' Summit, pp. 175-184, Ottawa, Canada, June 2005.
8. Стандарт языка Си ISO/IEC 9899:TC2. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf
9. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In ACM Transactions on Programming Languages and Systems, 13, 4 (Oct. 1991), pp. 451-490.
10. K. Cooper and L.T. Simpson. SCC-based value numbering. Tech. Rep. CRPC-TR95636-S, Rice University, October 1995.
11. А.А.Белеванцев, С.С.Гайсарян, В.П.Иванников. Построение алгоритмов спекулятивных оптимизаций. Журнал Программирование, N3 2008, с. 1-22.
12. Michael S. Schlansker, B. Ramakrishna Rau. EPIC: An Architecture for Instruction-Level Parallel Processors. HP Laboratories Palo Alto Technical Report HPL-1999-111, February 2000.
13. Andrey Belevantsev, Alexander Chernov, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik. Improving GCC Instruction Scheduler for IA64. GCC Summit 2005, June 2005, Ottawa, Canada.

14. Richard A. Huff. Lifetime-sensitive modulo scheduling. In Proc. of the SIGPLAN '93 Conf. on Programming Language Design and Implementation, pages 258--267, Albuquerque, N. Mex., Jun. 23--25, 1993.
15. Mostafa Hagog and Ayal Zaks. Swing Modulo Scheduling in GCC. In Proceedings of the GCC Developer's Summit 2004, pp 55-64, Ottawa, Canada.
16. Carole Dulong, Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, John Ng, and David Sehr. An overview of the Intel IA-64 compiler. Intel Technology Journal, Q4, 1999.
17. Kalyan Muthukumar. Compiling for the Intel Itanium Processor. Presented on Gelato GCC Workshop, Geneva, Switzerland, 2005.
18. Vasanth Venkatachalam and Michael Franz. Power reduction techniques for microprocessor systems. ACM Computer Surveys 37, 3, September 2005, pp. 195-237.
19. Chung-Hsing Hsu, Ulrich Kremer, and Michael Hsiao. Compiler-directed dynamic frequency and voltage scaling. In Workshop on Power-Aware Computer Systems, Cambridge, MA, 2000.
20. Linux Kernel CPUFreq Subsystem. http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufreq.html
21. Hovemeyer, D. and Pugh, W. 2004. Finding bugs is easy. SIGPLAN Not. 39, 12 (Dec. 2004), 92-106.
22. M. P. Gallaher and B. M. Kropp. Economic impacts of inadequate infrastructure for software testing. Technical report, RTI International, National Institute of Standards and Technology, US Dept of Commerce, May 2002.
23. Forrest Shull, Vic Basili, Barry Boehm, Winsor A. Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. In International Software Metrics Symposium. 2002. Ottawa, Canada., 2002.
24. David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In Network and Distributed System Security Symposium, pages 3–17, San Diego, CA, February 2000.

25. David A. Wheeler. Flawfinder. http://www.dwheeler.com/flawfinder/, апрель 2008.
26. J. Viega, J. T. Bloch, Y. Kohno, and G. Mcgraw. Its4: a static vulnerability scanner for c and c++ code. In Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference, pages 257–267, 2000.
27. Fortify Software, Inc. RATS - Rough Auditing Tool for Security. http://www.fortify.com/security-resources/rats.jsp, апрель 2008.
28. PolySpace Technologies. Embedded Software Verification products. http://www.polyspace.com/, март 2008.
29. Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, pages 97–106, New York, NY, USA, 2004. ACM.
30. Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. volume 3444/2005, pages 21–30. Springer Berlin / Heidelberg, 2005.
31. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 238–252. ACM Press, 1977.
32. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. SIGPLAN Not., 37(1):58–70, January 2002.
33. Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con Mcgarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. SIGOPS Oper. Syst. Rev., 40(4):73–85, October 2006.
34. Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium, pages 16–16, Berkeley, CA, USA, 2001. USENIX Association.

35. David Evans, John Guttag, James Horning, and Yang M. Tan. Lclint: a tool for using specifications to check code. SIGSOFT Softw. Eng. Notes, 19(5):87–96, December 1994.
36. David Evans and David Larochelle. Improving security using extensible lightweight static analysis. IEEE Software, 19(1):42–51, /2002.
37. Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. SIGPLAN Not., 38(5):155–167, May 2003.
38. Patrice Godefroid. The soundness of bugs is what matters (position statement). In BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools), 2005.
39. David Hovemeyer and William Pugh. Finding bugs is easy. SIGPLAN Not., 39(12):92–106, December 2004.
40. Klocwork. Системы анализа исходного кода. http://www.klocwork.com/products/, апрель 2008.
41. Ted Kremenek and Dawson Engler. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations, pages 1075–1075. 2003.
42. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. Softw. Pract. Exper., 30(7):775–802, 2000.
43. Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis, pages 203–217. 2005.
44. Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. An empirical study on classification methods for alarms from a bug-finding static c analyzer. Inf. Process. Lett., 102(2-3):118–123, 2007.
45. Yichen Xie, Andy Chou, and Dawson R. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In ESEC / SIGSOFT FSE, pages 327–336, 2003.
46. Dawson R. Engler, David Y. Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In Symposium on Operating Systems Principles, pages 57–72, 2001.

47. Isil Dillig, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. In PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, pages 435–445, New York, NY, USA, 2007. ACM.
48. Coverity. Static source code analysis solutions. http://www.coverity.com/, апрель 2008.
49. GrammaTech, Inc. CodeSonar. http://cayuga.grammatech.com/products/codesonar/, апрель 2008.
50. *Diffie W., Hellman M.* New directions in cryptography // IEEE Transactions om Information Theory, IT-22(6), 1976, 644—654.
51. *Collberg C., Thomborson C., Low D.* A taxonomy of obfuscating transformations // Tech. Report, N 148, Univ. of Auckland, 1997.
52. *Barak B., Goldreich O., Impagliazzo R., Rudich S., Sahai A., Vedhan S., Yang K.* On the (Im)possibility of obfuscating programs // Lecture Notes in Computer Science, v. 2139, 2001, p. 1-18.
53. *Goldwasser S., Tauman Kalai Y.* On the impossibility of obfuscation with auxiliary input // Proc. of the 46 th IEEE Symp. on Foundations of Computer Science, 2005, 553-562.
54. *Hofheinz D., Malone-Lee J., Stam M.* Obfuscation for cryptographic purposes // Lecture Notes in Computer Science, v. 4392, 2007, p. 214-232.
55. *Lynn B., Prabhakaran M., Sahai A.* Positive results and techniques for obfuscation // Lecture Notes in Computer Science, v. 3027, 2004, p. 20-39.
56. *Adida B., Wikström D.* Obfuscated ciphertext mixing // IACR. Eprint Archive, N 394, 2005.
57. *Ostrovsky R., Skeith W.E.* Private searching on streaming data //Advances in Cryptology - CRYPTO-2005, Lecture Notes in Computer Science, v. 3621, 2005, 223—240.
58. *Collberg C, Thomborson C.* Watermarking, tamper-proofing, and obfuscation - tools for software protection // IEEE Transactions on Software Engineering, v. 28, N 6, 2002.

59. *Collberg C, Thomborson C. Townsend G.* Dynamic graph-based software fingerprinting // ACM Transactions on Programming Languages and Systems, 2007, v. 29, N 6, Article 35.
60. *D'Anna L., Matt B., Reisse A., Van Vleck T. , Schwab S., LeBlanc P.* Self-protecting mobile agents obfuscation report // Report #03-015, Network Associates Laboratories, 2003.
61. *Chess D., White S.* An undetectable computer virus. // 2000 Virus Bulletin Conference, 2000.
62. *Szor P., Ferrie P.* Hunting for metamorphic // 2001 Virus Bulletin Conference, 2001, 123-144.
63. *Christodorescu M., Jha S.* Static analysis of executables to detect malware patterns // Usinex Security Symposium, 2003.
64. *Christodorescu M., Jha S., Seshia S.A., Song D., Bryant R.E.* Semantic-aware malware detection // Proc. of the 2005 IEEE Symp.on Security and Privacy (Oakland 2005), 2005.
65. *Dalla Preda M., Christodorescu M., Jha S.,Debray S.* Semantic-based approach to malware detection // Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symp.on Principles of Programming Languages, 2007, p. 377-388.
66. *Moser A., Kruegel C., Kirda E.* Limits of static analysis for malware detection // Proc. Of the 23rd Computer Security Applications Conference, 2007.
67. *Tamada H., Nakamura M., Monden A., Matsumoto K.* Design and evaluation of birthmarks for detecting theft of java programs // In *Proc. IASTED International Conference on Software Engineering (IASTED SE 2004), 2004, p. 569–575.*
68. *Varnovsky N.P.* A note on the concept of obfuscation // Труды Института системного программирования, т. 6, 2004.
69. *Wee H.* On obfuscating point functions // Proc. of 37th ACM Symp. on Theory of Computing, 2005, p. 523-532.
70. *Goldwasser, S., Rothblum G.* On best-possible obfuscation // Lecture Notes in Computer Science, v. 4392, 2007, p. 253-272.
71. *Varnovsky N.P., Zakharov V.A.* On the possibility of provably secure obfuscating programs // Lecture Notes in Computer Science, v. 2890, 2003, p. 91-102.

72. *Kuzurin N.N., Shokurov FA.V., Varnovsky N.P., Zakharov V.A.*On the concept of software obfuscation in computer security // Lecture Notes in Computer Science, v. 4779, 2007, p. 281-298.
73. *Hohenberger S., Rothblum G. N., Shelat A., Vaikuntanathan V.* Securely obfuscating re-encryption // Lecture Notes in Computer Science, v. 4392, 2007, p. 233-252.
74. *Linn C., Debray S.* Obfuscation of executable code to improve resistance to static disassembly // Proc. of the 10th ACM Conf. on Computer and Communications Security, Oct. 2003.
75. *Stroulia E., Systa T.* Dynamic analysis for reverse engineering and program understanding // ACM SIGAPP Applied Computing Review, v. 10, 2002, N 1, p. 8-17.
76. *Heffner K., Collberg C.* The obfuscation executives // Tech. Rep. TR 04-03, Dep. of computer science, Univ. of Arizona, USA, 2003.
77. *Kruegel C., Robertson W., Valeur F., Vigna G.* Static disassembly of obfuscated binaries // Proc. of USENIX Security, 2004, p. 255-270.
78. *Udupa S. K., Madou M., Debray S.* Deobfuscation: reverse engineering obfuscated code // Proc. of the 12th Working Conf. on Reverse Engineering, 2005, p. 45-54.
79. *Madou M., Anckaert B., De Bus B., De Bosschere K., Cappaert J., Preneel B.* On the effectiveness of source code transformations for binary obfuscation // Proc. of the Int. Conf. on Software Engineering Research and Practice (SERP06), June. 2006.
80. *Collberg C., Thomborson C., Low D. C.* Manufacturing cheap, resilient and stealthy opaque constructs // Symp. on Principles of Programming Languages, 1998, p. 184-196.
81. *Wang C., Hill J., Knight J,. Davidson J.* Software tamper resistance: obstructing static analysis of programs // Tech. Rep., N 12, Dep. of Comp. Sci., Univ. of Virginia, 2000.
82. *Chow S., Gu Y., Johnson H., Zakharov V.A.* An approach to the obfuscation of control-flow of sequential computer programs // Lecture Notes in Computer Science, v. 2200, 2001, p.144-155.

83. *Dalla Preda M., Giacobazzi R., Madou M., de Bosschere B.* Opaque predicate detection by means of abstract interpretations // Lecture Notes in Computer Science, v. 4019, 2006, p. 81-95.
84. *Majumdar A., Thomborson C.* Manufacturing opaque predicates in distributed systems for code obfuscation // Proc. of the 29th Australasian Computer Science Conf., 2006, p. 187-196.
85. *Fukushima K, Kiyomoto S., Tanaka T., Sakurai K.* Analysis of program obfuscation schemes with variable encoding technique // IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences 2008 E91-A(1):316-329.
86. *Wroblewski G.* General method of program code obfuscation // Proc. of the Int. Conf. on Software Engineering Research and Practice (SERP), 2002, p.153-159.
87. *Madou M., van Put L., De Bosschere K.* Understanding obfuscated code // Proc. of the 14th IEEE International Conf. on Program Comprehension, 2006, p. 268-274.
88. Ertaul L. Venkatesh S. Jhide – a toolkit for code obfuscation // Proc. of the 8th IASTED Int. Conf. on Software Engineering and Applications, 2004.
89. *Hind M., Pioli A.* Which pointer analysis should I use // Proc. of the 2000 ACM SIGSOFT Int. Symp. on Software Testing and Analysis, 2000, p 113-123.
90. Horwitz S. Precise flow-insensitive may-alias analysis is NP-hard // University of Wisconsin-Madison, August 1996.
91. *Collberg C., Thomborson C., Low D. C.* Breaking abstractions and unstructuring data structures // Proc. of the 1998 Int. Conf. on Computer Languages, 1998, p. 28-38.
92. *Sakabe Y., Soshi M., Miyaji A.* Java obfuscation with a theoretical basis for building secure mobile agents // Proc. of the 7th IFIP Conf. on Communication and Multimedia Security, 2003.
93. Anckaert, B.*;* Madou, M.*;* De Sutter, B.*;* De Bus, B.*;* De Bosschere, K.*;* Preneel, B. Program Obfuscation: A Quantitative Approach. Proceedings of the 2007 ACM workshop on Quality of protection. ACM Press. 2007. pp. 15-20
94. *McCabe T.J.* A complexity measures // IEEE Trans. Software Engineering, 1976, v. 2, N 4, p. 308-320.

95. *Cousot P., Cousot R.* An abstract interpretation based framework for software watermarking // Symp. on Principles of Programming Languages, 2003, p. 311-324.
96. *Dalla Preda M., Giacobazzi R.* Semantic-based code obfuscation by abstract interpretation // Lecture Notes in Computer Science, v. 3580, 2005, p. 1325-1336.
97. *Dalla Preda M., Giacobazzi R., Madou M., de Bosschere B.* Opaque predicate detection by means of abstract interpretations // Lecture Notes in Computer Science, v. 4019, 2006, p. 81-95.
98. *Goldreich O., Ostrovsky R.*. Software protection and simulation on oblivious RAMs.// Journal of the ACM, v. 43, N 3, 1996, p.431-473.
99. *Mana A., Pimentel A.* An efficient software protection scheme // Proc. of the 16th Int. Conf. on Information Security, 2001, p. 385-401.
100. *Zhang X. , Gupta* R. Hiding program slices for software security // Proc. of the First Annual IEEE/ACM Int. Symp. on Code Generation and Optimization, 2003, p. 325-336.
101. *Sander T., Tchudin C.* On software protection via function hiding. // Lecture Notes in Computer Science, v. 1525, 1998.
102. *Aushmith D.* Tamper resistant software: an implementation // Lecture Notes in Computer Science, v. 1174, 1996, p.317-333.
103. *Madou M., Anckaert B. , Moseley P., Debray S., De Sutter B., De Bosschere K.* Software protection through dynamic code mutation // Information Security Applications, 2005, p. 371-385.
104. *Madou M., Anckaert B., De Sutter B., De Bosschere K.* Hybrid static-dynamic attacks against software protection mechanisms // Proc. of the 5th ACM Workshop on Digital Rights Management, 2005, p. 75-82.
105. *Collberg, C.S.* Myles *G.*, Huntwork *A.* Sandmark - a tool for software protection research // IEEE Security & Privacy, v. 1, N 4, 2003, p. 40-49.
106. *Madou, M.; Van Put, L.; De Bosschere, K.* Loco: an interactive code (de)obfuscation tool // Proc. of ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation, 2006.

107. *Narayanan A., Shmatikov V.* Obfuscated database and group privacy // Proceedings of the 12th ACM conference on Computer and communications security, 2005, p. 102-111.

108. Mooly Sagiv, Thomas Reps, Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. // ACM Transactions on Programming Languages and Systems (TOPLAS) **24** (3): 217–298. ACM. doi:10.1145/292540.292552. May 2002.

109. SoftICE System Debugger Tool. http://www.compuware.com/

110. Reverse Engineering Tools. http://www.woodmann.com/crackz/Tools.htm

111. Крис Касперски, Ева Рокко. Искусство дизассемблирования. / BHV, 2007. 896 стр.

112. Linux Kernel Debugger – Linice. http://www.linice.com/

113. IceExt – SoftICE extension. http://sourceforge.net/projects/iceext/

114. Software Passport / The Armadillo Software Protection System. http://www.siliconrealms.com/software.shtml

115. OllyDbg assembler. http://www.ollydbg.de/

116. Syser Kernel Debugger. http://www.sysersoft.com/

117. Debugging Tools for Windows. http://www.microsoft.com/whdc/devtools/debugging/default.mspx

118. IDAPro disassembler, http://www.datarescue.com/idabase/

119. HexRays decompiler, http://www.hex-rays.com/index.shtml

120. Boomerang Decompiler. http://boomerang.sourceforge.net/

121. Interactive Decompiler. http://sourceforge.net/projects/idc/

122. .NET Obfuscator, Code Protector, and Pruner. http://www.preemptive.com/dotfuscator.html

123. Object Management Group. http://www.omg.org/

124. Telelogic Tau. http://www.telelogic.com/products/tau/

125. Telelogic Rhapsody. http://www.telelogic.com/products/rhapsody/

126. Objecteering Modeler. http://www.objecteering.com/

127. Borland Together. http://www.borland.com/us/products/together/

128. SmartState. http://www.smartstatestudio.com/

129. PathMate. http://www.pathfindermda.com/

130. BOUML. http://bouml.free.fr/

131. IBM Architecture Management: http://www-306.ibm.com/software/rational/offerings/design.html

132. Klocwork InSight: http://www.klocwork.com/products/insight.asp

133. Coverity Structure 101: http://www.coverity.com/html/prod_structure101.html

134. G. Kingston «Software Design Reviews Using the Software Architecture Analysis Method: A Case Study» *Defence Science and Technology Organization of Australian Department of Defence. Information Technology Division Electronics and Surveillance Research Laboratory DSTO-RR-0170, February 2000*

135. E. Gamma, R. Helm, R. Johnson, J. Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software» *Addison-Wesley, 1995*

136. G. Guo, J. Atlee, R. Kazman, «A Software Reconstruction Architecture Method*, Software Architecture» (Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)), (San Antonio, TX), February 1999, 15-33*

137. Igor Ivkovic, Kostas Kontogiannis, «A Framework for Software Architecture Refactoring using Model Transformations and Semantic Annotations», *csmr, pp. 135-144, Conference on Software Maintenance and Reengineering (CSMR'06), 2006*

138. N. Medvidovic, A. Egyed , D. S. Rosenblum. «Round-Trip Software Engineering Using UML: From Architecture to Design and Back». *Proceedings of 2nd Workshop on Object-Oriented Reengineering (WOOR) co-located with the 7th European Software Wngineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Toulouse, France, September 1999, pp. 1-8*

139. D. Hovemeyer, W. Pugh. «Finding bugs is easy». *In Companion of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, BC, October 2004*

140. M. Das, S. Lerner, M. Seigle. «ESP: Path-sensitive program verification in polynomial time». *In Proceedings of the ACM SIGPLAN 2002 Conference on*

Programming Language Design and Implementation, pages 57–68. ACM Press, 2002.

141. Laycock, G. T. (1993). «The Theory and Practice of Specification Based Software Testing». *Dept of Computer Science, Sheffield University, UK*
142. Spices.Net Obfuscator. http://www.9rays.net/Products/Spices.Obfuscator/
143. DashO for Java: Java Obfuscator, Java Code Protector, and Pruner. http://www.preemptive.com/dasho-java-obfuscator.html
144. Dotfuscator Product Family http://www.preemptive.com/dotfuscator-product-family.html
145. Jasob JavaScript Obfuscator. http://www.jasob.com/
146. Stunnix Perl-obfus - the obfuscator for Perl source code. http://www.stunnix.com/prod/po/
147. Introduction to C and C++ Obfuscator. http://www.stunnix.com/prod/cxxo/overview.shtml
148. Mangle-It C++ Obfuscator. http://www.top-shareware.net/MangleIt_C_Source_Code_Obfuscator.html