О некоторых задачах анализа и трансформации программ*

С.С. Гайсарян, А.В. Чернов, А.А. Белеванцев, О.Р. Маликов, Д.М. Мельник, А.В. Меньшикова

Аннотация. В настоящей статье обсуждаются некоторые перспективные направления исследований, проводимые в отделе компиляторных технологий Института системного программирования РАН. Методы анализа и трансформации программ, ранее применявшиеся в основном в оптимизирующих компиляторах, в настоящее время находят применение при решении множества смежных задач, таких как обеспечение безопасности программ, генерация тестов для программ и т. д.

1. Введение

В настоящей статье обсуждаются некоторые перспективные направления исследований, проводимые в отделе компиляторных технологий Института системного программирования РАН. Методы анализа и трансформации программ, ранее применявшиеся в основном в оптимизирующих компиляторах, в настоящее время находят применение при решении множества смежных задач, таких как обеспечение безопасности программ, генерация тестов для программ и т. д.

В отделе ведётся работа и в традиционной области оптимизации программ. Упор делается на разработку новых методов анализа указателей в программах на языке Си. Также проводятся исследования так называемых «полустатических» (profile-based) методов оптимизации программ. Такие методы заключаются в использовании на стадии оптимизации кода информации, накопленной с предварительных её запусков.

Данная работа посвящена рассмотрению трёх направлений. Во-первых, это так называемая маскировка программ, преследующая цель, полностью сохранив пользовательское поведение программы, изменить её текст так, что обратная инженерия или повторное использование ее фрагментов или программы целиком становится сложным. Во-вторых, это задачи автоматической оптимизации программы для работы на многопроцессорных системах с общей памятью путём разбиения её на нити. В-третьих, это задача автоматического выявления уязвимостей в программе.

* Работа поддержана РФФИ, грант 03-01-00880

Для поддержки работ по исследованию методов анализа и трансформации программ в отделе разработана интегрированная инструментальная среда (Integrated Research Environment, IRE), которая содержит большое количество различных алгоритмов анализа и трансформации программ и предоставляет удобный интерфейс пользователя.

Данная работа имеет следующую структуру: в разделе 2 мы рассматриваем задачу автоматического разбиения программы на нити, в разделе 3 рассматриваются задачи маскировки программ, а в разделе 4 задача автоматического выявления уязвимостей. Далее в разделе 5 кратко описывается интегрированная среда IRE, её состав и внутреннее представление MIF, используемое всеми компонентами IRE. Наконец, в разделе 6 мы формулируем выводы данной работы и даём направления дальнейших исследований.

2. Разбиение программ на нити и повышение локальности

В настоящее время широко распространены рабочие станции и персональные компьютеры, содержащие несколько центральных процессоров. Массовые многопроцессорные системы обычно содержат 2, 4 или 8 процессоров, работающих над общей памятью с одинаковым для всех процессоров временем доступа (SMP). Для максимального использования возможностей SMP-систем в вычислительно-интенсивных приложениях необходимо максимально использовать «легковесные» процессы (нити). В этом случае накладные расходы на коммуникацию минимизированы, так как все нити разделяют одно адресное пространство, а синхронизационные операции выполняются проще и быстрее, чем для обычных («тяжелых») процессов.

Известно, что большинство программ при работе демонстрируют хорошую локальность, т.е. работают над близко расположенными в памяти данными, или выполняют одни и те же инструкции. На этом наблюдении основана работа процессорных кэшей. Для наиболее полного использования возможностей кэша необходимо улучшать локальность программы.

В данном разделе мы представим новый алгоритм для разделения программы на нити, который улучшает локальность программы в целом. Полученные экспериментальные результаты показывают оправданность применения нового алгоритма для разбиения на нити программ без чёткой циклической структуры, которые не могут быть разбиты на нити традиционными методами. Основным выводом работы является то, что соображения локальности должны приниматься во внимание при разделении программы на нити для небольшого числа процессоров.

Системы с разделяемой памятью наиболее удобны для программиста параллельных приложений. Более того, часть работы по распараллеливанию последовательного кода может быть выполнена компилятором. Существует много исследований по автоматическому распараллеливанию циклов и

рекурсивных процедур на таких системах. Некоторые разработки реализованы в промышленных компиляторах, например, IBM Visual Age C++, Intel C++ Compiler, SGI MIPSPro, REAPAR и других.

В последнее время проводятся исследования по автоматическому распараллеливанию любого последовательного кода. Предложено несколько подходов, таких, как управление выполнением нитей (thread-level speculation) [6], коммутативный анализ, динамическое распределение задач на нити (dynamic task scheduling) [5], автоматическое разделение на нити на этапе компиляции. Часть предложенных алгоритмов проверена авторами на эмуляторах, часть реализована в существующих исследовательских компиляторах, например, в компиляторе SUIF Стенфордского университета [7].

Формализация понятия локальности проведена в [8]. Рассматривается два вида событий локальности:

- *Событие временной локальности* происходит при повторном доступе к ячейке памяти, уже имеющейся в кэше.
- *Событие пространственной локальности* происходит при доступе к ячейке памяти, расположенной в блоке, уже загруженном в кэш при обращении к какой-либо другой ячейке.

Для увеличения количества событий локальности в последнее время предложено большое количество оптимизирующих преобразований программы. Основными методами являются:

1) Группировка инструкций, использующих одни и те же данные (locality grouping), для увеличения количества событий временной локальности.
2) Упаковка данных в памяти (data packing) для увеличения количества событий пространственной локальности.
3) Перестановка процедур, базовых блоков и т.п.

Целью данной работы является исследование вопроса о том, как может быть проведено разделение программы на потоки для увеличения количества событий локальности программы в целом. Для этого предлагается использовать эвристический алгоритм разделения программы на нити, учитывающий в процессе разделения возникающие события локальности и динамически подстраивающий параметры эвристик.

2.1. Алгоритм разбиения программы на нити

В настоящем разделе рассматривается построение промежуточного представления программы, над которым работает алгоритм, а также подробно описывается сам алгоритм разбиения программ на нити. Подробное описание алгоритма можно найти в [3]. Алгоритм состоит из трех частей:

- Построение ценовой модели, отражающей свойства локальности
- Разбиение программы на нити
- Дополнительные оптимизации

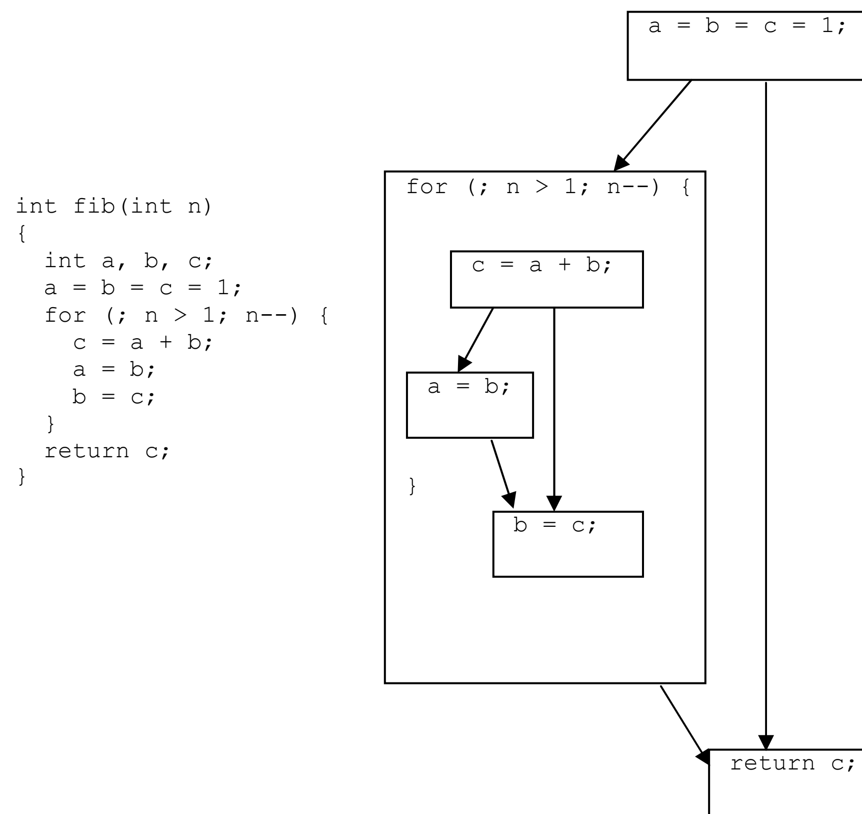


Рис. 1. Пример функции и ее DDG.

2.1.1. Граф зависимостей по данным

При разделении программы на нити прежде всего нужно учитывать зависимости по данным. Поэтому естественно потребовать, чтобы промежуточное представление программы содержало легкодоступную информацию о зависимостях по данным между различными частями программы. В то же время необходимо максимально отразить сведения о «естественном» параллелизме программы, причем на разных уровнях – от отдельных инструкций, до более крупных программных блоков.

Представлением, обладающим всеми необходимыми нам свойствами, является иерархический граф зависимостей по данным, используемый в [9] (data dependence graph, DDG). Узлом такого графа может являться:

- Простой оператор (сложение, умножение, сравнение, присваивание и т.д.)

- Более сложный оператор (условный оператор, оператор цикла и т.д.)
- Граф зависимостей по данным следующего уровня, инкапсулирующий свойства соответствующего программного блока

Дуги графа DDG представляют собой зависимости по данным между узлами. Более формально, пусть u и v – узлы DDG, причем в последовательной программе u предшествует v. Дуга (u, v) входит в граф тогда и только тогда, когда между u и v есть зависимость по данным одного из трех типов:

- «запись-чтение» (в узле v необходимы результаты вычислений узла u),
- «чтение-запись» (в узле v записывается переменная, значение которой считывается в u),
- «запись-запись» (оба узла записывают одну и ту же переменную).

Наличие одной из указанных зависимостей по данным между узлами говорит о том, что при параллельном выполнении программы для получения результатов, совпадающих с последовательной версией, необходимо выполнить u раньше, чем v.

Легко заметить, что граф зависимостей по данным является ориентированным ациклическим графом. Это объясняется тем, что циклы в DDG означают наличие циклических зависимостей по данным, возможных, в свою очередь, только в операторах цикла исходной программы. Но циклы, как и другие сложные операторы, раскрываются на более низком уровне иерархии, обеспечивая разрыв таких зависимостей по данным. Это свойство графа будет использоваться нами в дальнейшем.

Пример функции и ее графа зависимостей по данным приведен на Рис. 1. DDG состоит из трех узлов: двух простых узлов и оператора цикла, раскрывающегося в DDG второго уровня.

Граф зависимостей по данным строится для каждой функции программы. Алгоритм построения состоит из следующих этапов:

- Построение графа потока управления программы.
- Выбор программных блоков, которые будут узлами текущего уровня иерархии DDG.
- Нахождение зависимостей по данным между этими узлами с помощью алгоритма достигающих определений.
- Если необходимо, продвинуться на следующий уровень иерархии и достроить граф.

Для того, чтобы отразить на графе побочные эффекты работы функции, в графе вводится специальный узел EXIT. Все узлы, генерирующие побочные эффекты (например, осуществляющие запись в глобальную переменную), связаны дугой с узлом EXIT.

Все этапы алгоритма разделения на нити, описанные ниже, работают с представлением программы в виде графа зависимостей по данным.

2.1.2. Ценовая модель

Нашей целью является построение разбиения программы на нити, максимально использующего возникающие события локальности. Чтобы иметь возможность судить о степени оптимальности того или иного разбиения, необходимо ввести некоторую ценовую модель. Так как мы оптимизируем время выполнения программы, то естественно ввести веса узлов графа зависимости по данным, равные времени выполнения узла в последовательной программе.

Время выполнения узла может быть найдено с помощью профилирования программы. Для этого необходимо инструментировать исходный код программы, вставляя вызовы функций из библиотеки поддержки, вычисляющих время выполнения инструкций, и выполнить программу на нескольких наборах типичных входных данных. Для получения более точных результатов можно воспользоваться высокоточными аппаратными счетчиками, имеющимися на большинстве современных архитектур (например, инструкцией RDTSC для Pentium III и выше). Эта оценка времени выполнения точно показывает реальное время выполнения программы, но затрудняет эмуляцию кэша на этапе разделения на нити, так как сложно определить, насколько уменьшится время выполнения узла при попадании в кэш (возможно, при профилировании это попадание уже произошло).

Ценовая модель должна также отражать события локальности, происходящие во время выполнения программы. Статических весов для узлов DDG для этой цели недостаточно. Необходима эмуляция кэша в процессе разделения на нити и соответствующая корректировка времени выполнения узла.

2.1.3. Разбиение на нити

На этом шаге производится собственно разбиение графа зависимостей по данным на нити. Количество нитей является параметром алгоритма. Так как целью разбиения является получение выигрыша по времени, возникающего из-за увеличения количества событий локальности в каждой нити, то необходимо привязать каждую нить к одному конкретному процессору или, точнее, к конкретному кэшу. Поэтому количество нитей, на которые производится разбиение, обычно равно количеству процессоров в системе.

Алгоритм разбиения состоит в итерировании списка узлов графа, еще не назначенных конкретной нити, и определения нити для какого-либо из узлов (группы таких алгоритмов обычно называются list scheduling). На каждом шаге такой алгоритм делает локально оптимальный выбор. Это значит, что при выборе очередного узла из списка делается попытка присвоить его каждой из имеющихся нитей, после чего выбирается лучшая.

Для того, чтобы иметь возможность оценивать варианты присвоения узла нити, необходимо ввести некоторую оценочную функцию. В нашем случае такая функция содержит время выполнения нити, а также среднеквадратичное отклонение времен выполнения всех нитей. Это следует из того соображения, что в оптимальном разбиении времена выполнения нитей должны быть достаточно близки друг к другу. Возможно включение и других параметров.

При включении узла в какую-либо нить необходимо провести пересчет времени выполнения этой нити. Алгоритм пересчета состоит из следующих шагов:

- Учет времени, необходимого на синхронизацию с другими нитями, если она требуется.
- Учет возникающих событий локальности.

Рассмотрим более подробно каждый из этих шагов.

2.1.3.1. Учет времени на синхронизацию

Обрабатываемый на текущем этапе узел может зависеть по данным от некоторых других. В этом случае необходимо дождаться выполнения каждой нити, которые содержит такие узлы. Порядок обхода узлов в списке должен быть таков, чтобы гарантировалось, что все такие узлы уже были распределены на нити. Для этого можно обходить узлы в естественном порядке, то есть так, как они расположены в последовательной программе, либо выполнить тополо-гическую сортировку графа зависимостей по данным. Еще раз подчеркнем, что иерархичность графа обеспечивает то, что он является ациклическим.

Таким образом, к моменту обработки некоторого узла все узлы, от которых он зависит по данным, уже распределены на нити. Если какие-либо из таких узлов находятся в других нитях, то перед выполнением текущего узла необходимо провести синхронизацию со всеми такими нитями. Для того, чтобы осуществить такую синхронизацию, нужно завести по одной общей переменной для каждой нити. Присваивание значения `i` такой переменной для некоторой нити `j` означает, что эта нить выполнила узел `i`. Нить, ждущая результатов вычисления узла `i`, должна ждать, пока соответствующая общая переменная не примет нужного значения. Пример такой синхронизации показан на Рис. 2.

Времена выполнения каждой из нитей, проводящих синхронизацию, должны быть увеличены соответствующим образом. Нить, пишущая в общую переменную о результатах выполнения узла, дополнительно работает время `t1`. Нить, ждущая данных от нескольких узлов, ожидает последний из выполняющихся узлов, после чего тратит время `t2`. Эти времена являются параметрами алгоритма.

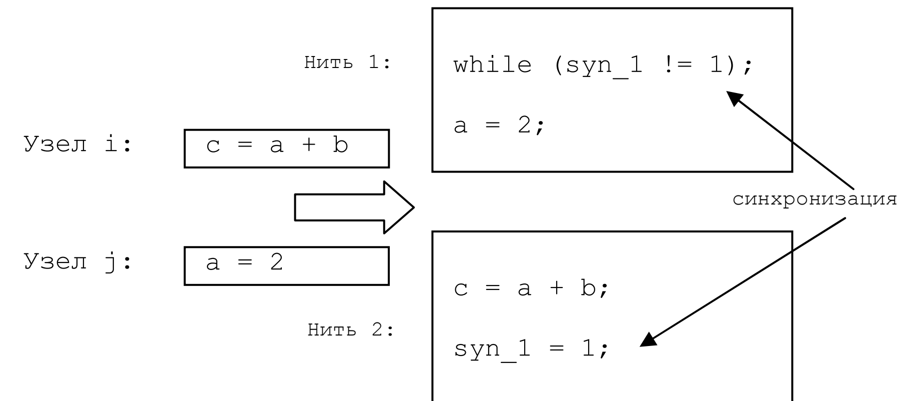


Рис. 2. Пример синхронизации между нитями.

2.1.3.2. Учет возникающих событий локальности

Для учета событий локальности для каждой нити необходимо эмулировать кэш процессора, на котором она выполняется. При распределении текущего узла на какую-либо нить необходимо проверить все переменные, которые читаются либо пишутся узлом, на попадание в кэш. Если попадание произошло, то время выполнения узла должно быть уменьшено на интервал времени `t3`, также являющийся параметром алгоритма.

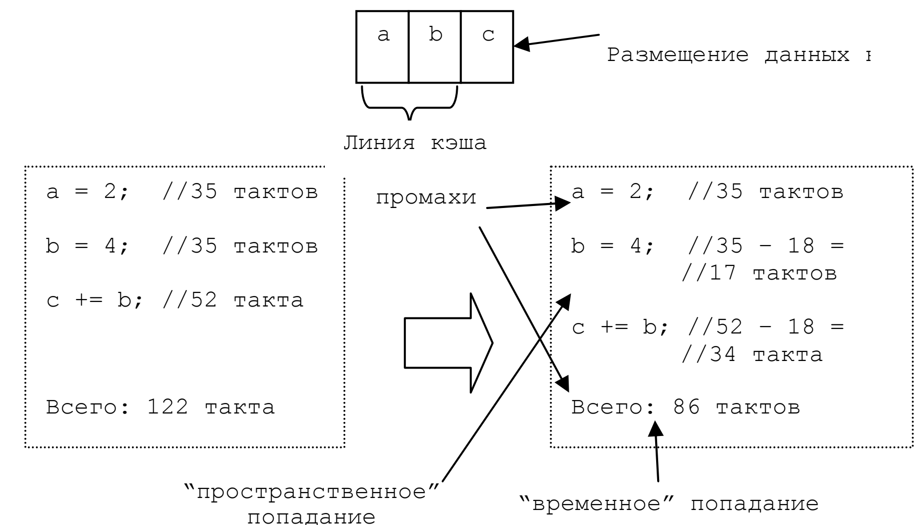


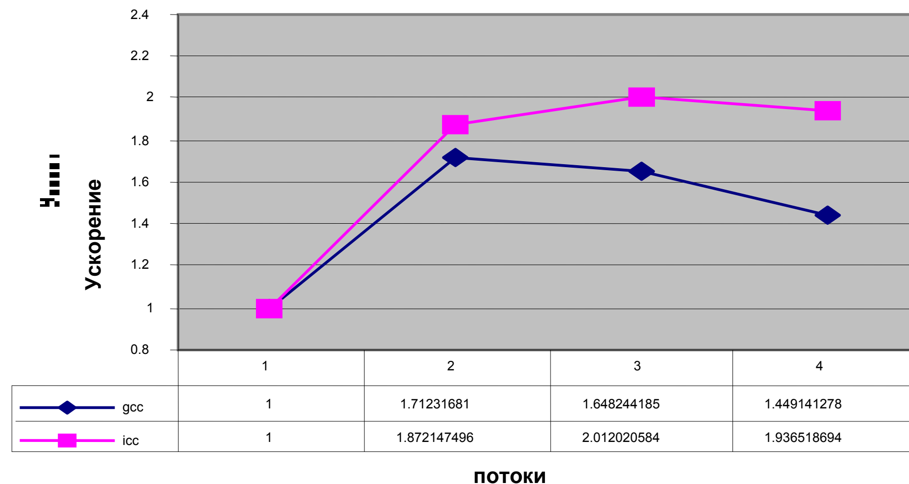
Рис. 3. Пример эмулирования кэша

Для учета событий как временной, так и пространственной локальности необходимо моделирование линий кэша, т.е. помещение в кэш не одной переменной, а некоторого блока памяти, окружающего нужную переменную. Моделирование различных типов кэшей приведет к разным результатам при разделении на нити.

Пример моделирования событий локальности изображен на Рис. 3.

2.2. Экспериментальные результаты

Мы применили нашу реализацию алгоритма к тестовой функции, решающей алгебраическое уравнение четвертой степени $x^4 + ax^3 + bx^2 + cx + d = 0$. Функция не содержит циклов и не может быть распараллелена традиционными способами. Полученная многопоточная версия функции была реализована с помощью библиотеки `pthread` под операционной системой Linux. Экспериментальные запуски были проведены на четырехпроцессорном Intel Itanium, на которых установлена ОС RedHat Linux 7; использовались компиляторы GCC 3.3.1 и ICC 8.00b. Программа запускалась 100 раз, время ее выполнения измерялось с помощью высокоточных аппаратных счетчиков. Вычислялось среднее значение времени выполнения μ и среднеквадратичное отклонение σ. Все значения времени выполнения, не укладывающиеся в промежуток $[\mu - 2\sigma, \mu + 2\sigma]$, удалялись из выборки, после чего среднее значение пересчитывалось. Эта величина использовалась для подсчета ускорения. Результаты эксперимента приведены на Рис. 4.



	1	2	3	4
gcc	1	1.71231681	1.648244185	1.449141278
icc	1	1.872147496	2.012020584	1.936518694

Рис. 4. Ускорение, достигнутое на Itanium

3. Маскирующие преобразования программ

Другим направлением, развиваемым в рамках IRE является исследование маскировки (obfuscation) программ. Мы рассматриваем проблему защиты программ от обратной инженерии, проводимой с целью модификации и/или включения фрагментов защищаемой программы во вновь разрабатываемый программный код. Защита в данном случае состоит в том, чтобы затруднить понимание деталей реализации компонент большой программной системы, сделав его настолько дорогим, чтобы дешевле было разработать оригинальное программное обеспечение.

Одним из способов такой защиты является маскировка программ, заключающаяся в применении к исходному тексту программы цепочки маскирующих преобразований, то есть преобразований, сохраняющих реализуемую программой функцию (являющихся функционально эквивалентными), но затрудняющих понимание этой функции.

Целью нашего исследования является новый метод маскировки программ, удовлетворяющего следующим требованиям:

- Исходная и замаскированная программа записаны на языке Си.
- В методе применяются цепочки маскирующих преобразований, элементы которых берутся из некоторого заранее зафиксированного множества параметризованных маскирующих преобра-зований.
- Все цепочки таких преобразований порождаются автоматически поддерживающими метод инструментальными средствами и являются допустимыми по своим характеристикам, то есть увеличивают размер маскируемой программы и/или уменьшают скорость её работы не более чем в фиксированное количество раз.
- Метод маскировки устойчив относительно современных методов статического и полустатического анализа программ, развитых для языка Си.

Предложена новая методология анализа маскирующих преобразований, которая позволяет давать качественную и количественную характеристику преобразований. Количественная характеристика преобразований позволяет дать их классификацию и выявить среди них наиболее подходящие для использования при маскировке программ. Основываясь на проведённом исследовании нами предложен новый метод маскировки программ, который удовлетворяет всем целям, сформулированным выше.

3.1. Методология анализа маскирующих преобразований программ

Для оценки действия маскирующих преобразований на программу мы используем несколько показателей сложности кода программ. Традиционно, подобного рода показатели называются «метрики», в дальнейшем мы будем придерживаться этого термина и в нашей работе. Метрики сложности кода программы ставят в соответствие любой программе некоторое число, которое

тем больше, чем «сложнее» программа. Простейшая *метрика LC размера процедуры* равна количеству инструкций промежуточного представления MIF в записи процедуры. *Метрика YC сложности циклической структуры* равна мощности транзитивного замыкания отношения достижимости в графе потока управления процедуры. *Метрика DC сложности потока данных* определяется как количество дуг в графе зависимостей по данным, строящемся по результатам анализа достигающих определений программы. *Метрика UC количества недостижимого кода* определяется как количество инструкций в программе, которые не выполняются ни при каких наборах входных данных. Известно, что точное вычисление метрик *DC* и *UC* является алгоритмически неразрешимой задачей.

Через *O(p,e)* мы обозначим применение метода маскировки *O* к программе *p*. *e* – это параметр, который позволяет выбрать единственную программу *p′ = O(p,e)* из множества функционально эквивалентных замаскированных программ *O(p)*. В частности, параметр *e* может быть случайным числом. Множество изменения параметра *e* обозначим через *E*. Тогда *LC(p)* – значение метрики *LC* до маскировки, а *LC(O(p,e))* – после маскировки.

Композитная метрика *цены TC преобразования O(p,e)* вычисляется по метрикам *LC* и *UC* следующим образом:

$$TC(p) = \frac{LC(O(p,e))}{LC(p)} + \frac{LC(O(p,e)) - UC(O(p,e))}{LC(p) - UC(p)}.$$

Для конечного множества D программ цена маскирующего преобразования определяется как $TC(D) = \max_{p \in D, e \in E} TC(p)$.

Метод маскировки называется *допустимым* для множества *D*, если $TC(D) \le TC_{MAX}$, где константа TC_{MAX} устанавливается директивно, исходя из эксплуатационных требований к замаскированной программе.

Композитная метрика MC усложнения программы вычисляется по метрикам YC и DC следующим образом:

$$MC(p) = \frac{YC(O(p))}{YC(p)} + \frac{DC(O(p))}{DC(p)}.$$

Для конечного множества D программ усложнение определяется как $MC(D) = \min_{p \in D, e \in E} MC(p)$. Чем больше метрика усложнения программы, тем сложнее для понимания становится замаскированная программа в силу увеличения числа информационных и управляющих связей.

Для оценки сложности самого маскирующего преобразования вводится *оценка OC* сложности *маскирующего преобразования*, которая определяется как требуемая для выполнения данного маскирующего преобразования глубина анализа программы согласно таблице 1.

Требуемая глубина анализа	**Значение *OC***
Лексический анализ	0
Синтаксический анализ	1
Семантический анализ	2
Анализ потока управления	3
Консервативный анализ потока данных	4
Полустатический анализ	5
Точный анализ потока данных	6

Таблица 1. Шкала оценки сложности маскирующих преобразований

Для оценки степени различия текстов программ вводится расстояние $\rho(p_1, p_2)$ между текстами программ p_1 и p_2, которое используется для оценки устойчивости маскирующего преобразования. Введём расстояние ρ_C между графами потока управления двух процедур, равное минимальному количеству операций удаления и добавления рёбер и дуг, переводящих один граф в граф, изоморфный другому. Аналогично введём расстояние ρ_D между графами зависимостей по данным двух процедур. Обозначим через $\rho_{CD}(f_1, f_2)$ сумму $\rho_C(f_1, f_2) + \rho_D(f_1, f_2)$. Тогда расстояние $\rho(p_1, p_2)$ вычисляется по формуле

$$\rho(p_1, p_2) = \min_M \left[\sum_{(f_1, f_2) \in M} \rho_{CD}(f_1, f_2) + \sum_{f_1 \in M_1} \rho_{CD}(f_1, E) + \sum_{f_2 \in M_2} \rho_{CD}(E, f_2) \right].$$ Здесь

минимум берётся по всевозможным множествам *M*, состоящим из пар (f_1, f_2), где f_1 – процедура из программы p_1, f_2 – процедура из программы p_2. Все процедуры f_1 и f_2 встречаются в *M* не более одного раза. Через M_1 обозначено множество процедур программы p_1, отсутствующих в *M*, а через M_2 – множество процедур программы p_2, отсутствующих в *M*. *E* – это пустой граф.

Такие подготовительные определения позволяют нам определить понятие устойчивости метода маскировки программ по отношению к заданному набору алгоритмов их анализа и основанным на них методам демаскировки программ следующим образом.

Пусть Θ – это множество маскирующих преобразований и необходимых для их *выполнения* алгоритмов анализа программ. Тогда метод маскировки – это цепочка $O \in \Theta^+$. Все алгоритмы $\{o_{i_1} \ldots o_{i_m}\} \subset \Theta$ анализа, необходимые для выполнения маскирующего преобразования o_i, находятся в цепочке *O* перед o_i. Пусть $A = \{A_1, \ldots A_m)$ – заданный набор демаскирующих преобразований и необходимых для их выполнения алгоритмов анализа программ. *Метод демас-*

кировки – это цепочка $A \in \mathrm{A}^*$. Все алгоритмы $\{a_{j_1} \ldots a_{j_n}\} \subset \mathrm{A}$ анализа, необходимые для выполнения маскирующего преобразования a_j, находятся в цепочке A перед a_j. Длину $|A|$ строки A назовём *сложностью* метода демаскировки.

Метод маскировки *O называется устойчивым* для множества программ *D* по отношению к множеству демаскирующих преобразований Α с порогом устойчивости Δ, если выполняются следующие условия:

- Метод маскировки *O* является допустимым, то есть $TC(D) \le TC_{MAX}$.
- Для любой программы p', полученной из *p*, любой метод демаскировки *A* сложности не более *C* получает программу p'' отстоящую от р более чем на Δ, то есть:

$$\forall p \notin D, \forall e \in E, p' = O(p,e), \forall A \in \mathrm{A}^*, |A| \le C, p'' = A(p') : \rho(p'', p) > \Delta .$$

Константа Δ называется *порогом устойчивости*.

Параметры C и Δ подбираются эмпирически. Параметр *C* – это оценка вычислительных ресурсов используемых для атаки на замаскированную программу. Чем больше *C*, тем более сложные методы демаскировки могут применяться для атаки. Параметр Δ зависит от ценности защищаемой программы и уровня экспертизы, ожидаемого при атаке на замаскированную программу. Чем больше ценность маскируемой программы и чем выше ожидаемый уровень подготовки лиц, выполняющих атаку на замаскированную программу, тем больше должен быть порог устойчивости Δ.

Наш *анализ* методов маскировки программ исходит из предположения, что множество всех возможных алгоритмов анализа и преобразования программ, которые могут использоваться для демаскировки программ, фиксировано. Для нашего анализа мы выбрали представительное множество методов, составленное следующим образом. Большинство рассматриваемых демаскирующих преобразований являются оптимизирующими преобразованиями, используемыми в оптимизирующих компиляторах. К таким преобразованиям относятся, например, устранение общих подвыражений и устранение мёртвого кода. Кроме них описываются алгоритмы полустатического анализа такие, как построение и анализ покрытия дуг и базовых блоков, и основанные на них разработанные в рамках данной работы демаскирующие преобразования.

В рамках нашего исследования нам удалось получить качественную и количественную характеристику опубликованных другими исследователями и разработчиками систем маскировки маскирующих преобразований. Был проведён сравнительный анализ их цены *TC*, усложнения маскируемой программы *MC* и оценки сложности *OC*. На примере программы, вычисляющей функцию Фибоначчи, проводится ранжирование маскирующих преобразо-ваний по соотношению усложнение/цена.

3.2. Анализ маскирующих преобразований

Все маскирующие преобразования делятся на текстуальные преобразования, преобразования управляющей структуры и преобразования структур данных. Преобразования управляющей структуры в свою очередь делятся на две группы: маскирующие *преобразования* реструктуризации всей программы и маскирующие преобразования над одной процедурой. Преобразования структур данных в рамках данной работы не рассматривались.

В группе **текстуальных маскирующих преобразований** рассматриваются преобразования удаления комментариев, переформатирования текста программы и *изменения* идентификаторов в тексте программы. В группе **маскирующих преобразований управляющей структуры**, воздействующих на программу в целом, рассматриваются преобразования открытой вставки процедур, выделения процедур, переплетения процедур, клонирования процедур, устранения библиотечных вызовов. В группе маскирующих преобразований маскировки над одной процедурой рассмотрены преобразования внесения непрозрачных предикатов и переменных, внесения недостижимого кода, внесения мёртвого кода, внесения дублирующего кода, внесения тождеств, преобразования сводимого графа потока управления к несводимому, клонирования базовых блоков, развёртки циклов, разложения циклов, переплетения циклов, диспетчеризации потока управления, локализации переменных, расширения области действия переменных, повторного использования переменных, повышения косвенности.

```
Int fib(int n)
{
  int a, b, c;
  a = 1;
  b = 1;
  if (n <= 1) return 1;
  for (; n > 1; n--) {
    c = a + b;
    a = b;
    b = c;
  }
  return c;
}
```

(a) исходная программа

```
int fib(int n)
{
  int a, b, c, i;
  long long t;
  a = 1;
  b = 1;
  if (n <= 1) return 1;
  while (1) {
    t = n * n % 65537;
    for (i = 0; i < 15; i++)
      t = t * t % 65537;
    if (n * t <= 1) break;
    c = a + b;
    a = b;
    b = c;
    n--;
  }
  return c;
}
```

(b) замаскированная программа

Рис. 5. Пример применения маскирующего преобразования внесения тождеств

На Рис. 5 показан пример применения маскирующего преобразования внесения тождеств к программе, вычисляющей функцию Фибоначчи. Преобразование основано на малой теореме Ферма ($a^{p-1} \equiv 1 \pmod p$) для любого целого *a*, такого, что $a \bmod p \neq 0$, и простого числа *p*). В таблице 2 приведена цена применения маскирующего преобразования и полученное усложнение программы. Для этого примера цена равна 3.83, а усложнение программы – 4.85. Расстояние между исходной и замаскированной программой равно 21.

	LC	UC	YC	DC
Исх. процедура `fib`	12	0	0.1111	14
Замаскированная `fib`	23	0	0.3086	29

Таблица 2. Оценка влияния маскирующего преобразования внесения тождеств

Другое маскирующее преобразование – введение непрозрачных предикатов – является ключевым для повышения устойчивости других маскирующих преобразований, например, внесения недостижимого кода. *Непрозрачным предикатом* называется предикат, всегда принимающий единственное значение *true* или *false*. При маскировке программы предикат строится таким образом, что его значение известно, но установить по тексту замаскированной программы, является ли некоторый предикат непрозрачным, трудно. В работе рассматриваются методы построения непрозрачных предикатов, использующие динамические структуры данных и булевские формулы.

На основании определения устойчивости маскирующих преобразований, данного выше, становится возможным провести анализ всех опубликованных маскирующих преобразований для выявления их устойчивости по отношению к нашему множеству демаскирующих преобразований и алгоритмов анализа. Мы можем ввести количественную классификацию маскирующих преобразований и выявить наиболее устойчивые маскирующие преобразования. Для этого вводятся – эвристическая *оценка CL сложности анализа*, которая устанавливает глубину анализа замаскированной программы, необходимого для выполнения демаскирующего преобразования, и эвристическая *оценка SL степени поддержки* демаскировки, устанавливающая необходимую степень участия человека в процессе демаскировки. Для оценки *CL* используется шкала, приведённая в таблице 1. Для оценки *SL* используется шкала: «автоматический анализ» (0 баллов), «полуавтоматический анализ» (1 балл), «ручной анализ с развитой инструментальной поддержкой» (2 балла), «только ручной анализ» (3 балла). Итоговая *оценка DL трудоёмкости анализа* равна $DL = CL + SL$.

В нашей работе показано, что для каждого маскирующего преобразования можно предложить автоматический или полуавтоматический метод демаскировки, позволяющий приблизить демаскированную программу p'' к исходной

программе *p*. При использовании полустатических алгоритмов анализа программ мы исходим из предположения, что для замаскированной программы существует достаточное количество тестовых наборов, обеспечивающее требуемый уровень доверия.

Таким образом можно получить количественную классификацию маскирующих преобразований. Для каждого маскирующего преобразования приводится оценка сложности маскировки и оценка трудоёмкости демаскировки. Значение, получаемое как разность оценки трудоёмкости демаскировки и оценки сложности маскировки, позволяет оценить насколько демаскировка данного маскирующего преобразования сложнее, чем маскировка. Исходя из этого определяются маскирующие преобразования, применение которых неоправдано, например, переформатирование программы, разложение циклов, локализация переменных; методы маскировки, которые следует применять только в комплексе с другими методами, например, изменение идентификаторов, внесение дублирующего кода и методы маскировки, применение которых наиболее оправдано, например, внесение тождеств, переплетение процедур, построение диспетчера, повышение косвенности. Сравнение двух маскирующих преобразований приведено в таблице 3. Через *D* обозначена разность $OC - DL$, через ρ_1 обозначено расстояние между текстами замаскированной и исходной программ fib, а через ρ_2 – расстояние между текстами демаскированной и исходной программ. Из таблицы следует, что маскирующее преобразование построения диспетчера предпочтительнее, так как, при равных с методом внесения тождеств трудозатратах на демаскировку, обеспечивает лучшее соотношение усложнения программы к цене преобразования.

Преобразование	*OC*	*TC*	*MC*	ρ_1	*CL*	*SL*	*DL*	ρ_2	*D*	$\frac{MC(fib)}{TC(fib)}$
Внесение тождеств	2	3.83	4.85	21	4 – 5	2	7	0	5	1.27
Построение диспетчера	2	3.83	6.14	39	5	2	7	2	5	1.60

Таблица 3. Сравнение методов маскировки

3.3. Новый метод маскировки программ

Новый метод маскировки программ мы далее обозначим аббревиатурой «ММ». Его более подробное описание можно найти в [2]. Метод ММ применяется к функциям маскируемой программы по отдельности, при этом структура маскируемой программы в целом не изменяется. Для изменения структуры маскируемой программы могут применяться стандартные методы открытой вставки и выноса функции, которые, однако, не являются частью предлагаемого метода маскировки.

При маскировке каждой функции ММ использует, наряду с локальными несущественными переменными, глобальные несущественные переменные, которые формируют *глобальный несущественный контекст*. В маскируемую программу вносятся несущественные зависимости по данным между существенным и несущественным контекстом функции. Наличие глобального несущественного контекста, совместно используемого всеми замаскированными функциями, приводит к появлению в замаскированной программе зависимостей по данным между всеми функциями и глобальными переменными.

Метод ММ состоит главным образом из преобразований графа потока управления. В результате граф потока управления замаскированной программы значительно отличается от графа потока управления исходной программы. Метод не затрагивает структур данных исходной программы, но вносит в за-маскированную программу большое количество несущественных зависимостей по данным. В результате, замаскированная программа значительно сложнее исходной как по управлению, так и по данным.

Мы предполагаем, что перед маскировкой были выполнены все стандартные шаги анализа программы: лексический, синтаксический, семантический, анализ потока управления (построение графа потока управления и деревьев доминирования и постдоминирования) и консервативный глобальный анализ потоков данных (достигающие определения и доступные выражения с учётом возможных алиасов). Дополнительно может быть выполнено профилирование дуг, результаты которого учитываются в преобразованиях клонирования дуг и развёртки циклов.

Общая идея метода может быть охарактеризована следующим образом.

- Во-первых, значительно увеличить сложность графа потока управления, но так, чтобы все дуги графа потока управления, внесённые при маскировке, проходились при выполнении программы. Это позволяет преодолеть основную слабость «непрозрачных» предикатов: насколько бы не были они сложны для статического анализа программы, полустатический анализ позволяет выявить такие предикаты (точнее, порождённые ими несущественные дуги графа потока управления) с большой долей уверенности.
- Во-вторых, увеличить сложность потоков данных маскируемой функции, «наложив» на неё программу, которая заведомо не влияет на окружение маскируемой функции и, как следствие, не изменяет работы программы. «Холостая» функция строится как из фрагментов маски-руемой функции, семантические свойства которых заведомо известны, так и из фрагментов, взятых из библиотеки маскирующего трансля-тора. Чтобы затруднить задачу выявления холостой части замаскиро-ванной функции используются как языковые конструкции, трудно поддающиеся анализу (указатели), так и математические тождества.

Метод маскировки можно разбить на несколько этапов:

1. Увеличение размера графа потока управления функции без нарушения его структурности. На этом этапе выполняются различные преобразования перестройки циклов, которые изменяют структуру циклов в теле функции, клонирование базовых блоков. Цель этого этапа — существенно увеличить размер графа потока управления функции.
2. Разрушение структурности графа потока управления функции. На этом этапе в граф потока управления вносится значительное количество новых дуг. При этом существовавшие базовые блоки могут оказаться разбитыми на несколько меньших базовых блоков. В графе потока управления могут появиться пока пустые базовые блоки. Цель этого этапа — подготовить место, на которое в дальнейшем будет внесён несущественный код.
3. Генерация несущественного кода. На этом этапе граф потока управления заполняется инструкциями, которые не оказывают никакого влияния на результат, вырабатываемый маскируемой программой. Несущественная, «холостая» часть пока никак не соприкасается с основной, функциональной частью программы.
4. «Зацепление» холостой и основной программы. Для этого используются как трудноанализируемые свойства программ (например, указатели), так и разнообразные математические тождества и неравенства.

В качестве примера применения предложенного метода маскировки мы выбрали небольшую программу, которая решает задачу о 8 ферзях. Для маскировки мы выберем основную функцию queens этой программы.

Метрика	`queens`	*MM*(`queens`)	*CM*(`queens`)
LC	49	711	4171
YC	0.595	0.8119	0.2402
UC	0	0	0
DC	82	8964	143807

Таблица 4. Изменение метрик для замаскированной процедуры queens

Преобразование	*OC*	*TC*	*MC*	*MC/TC*	*CL*	*SL*	*DL*	*D*
MM(`queens`)	4	29.02	110.68	3.81	5	2	7	3
CM(`queens`)	?	170.24	1754.14	10.30	5	2	7	?

Таблица 5. Сравнение методов маскировки для функции queens

В таблице 4 в столбце queens приведены базовые метрики сложности кода для исходной процедуры queens; в столбце *MM*(queens) – для процедуры, замаскированной с помощью предложенного метода маскировки; в столбце CM(queens) – для процедуры, замаскированной с помощью коммерческого маскировщика рассмотренного выше.

В таблице 5 приведены метрики цены применения маскирующего преобразования, усложнения программы требуемой глубины анализа для предложенного метода маскировки *MM* и для коммерческого маскировщика *CM*. Для коммерческого маскировщика сложность алгоритма маскировки неизвестна, поэтому в соответствующих ячейках таблицы стоит знак «?». Из таблицы видно, что новый метод маскировки *MM* существенно дешевле, чем реализованный в коммерческом маскировщике.

4. Автоматическое выявление уязвимостей защиты 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 заметить, что данное 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.

Рассмотрим, например, последний случай "взлома" Интернет-сервера проекта Debian Linux. Программа-сервер синхронизации файлов по сети rsync

содержала уязвимость в защите, которая позволяла, подключившись к серверу rsync и подав на ему на вход специально подготовленные входные данные, принудить процессор исполнить не исполняемый код программы rsync, а исполняемый код, переданный в этих входных данных. Сама по себе программа rsync не является привилегированной, но таким образом был получен доступ к компьютеру с возможностью запускать произвольные программы (доступ shell account). Естественно, такой способ получения доступа к компьютеру обходит все нормальные средства аутентификации, такие как ввод регистрационного имени и пароля, ввод однократного ключа и т. д.

Получив возможность выполнения произвольных программ на сервере, злоумышленник использовал другую уязвимость в защите, теперь непосредственно ядра Linux, которая была связана с недостаточной проверкой параметра, передаваемого системному вызову sbrk. Передавая этому системному вызову отрицательные значения можно было добиться открытия доступа к критически важным страницам памяти, после чего можно было добиться выполнения произвольной программы уже с правами суперпользователя (root). Обычно такая программа – это интерпретатор командной строки /bin/sh. Таким образом, неизвестный злоумышленник в два этапа получил полный контроль над машиной, на которой он раньше даже не имел shell account.

Уязвимости в защите, которые могут быть использованы просто подключением к уязвимой программе без какой-либо авторизации называются удалённо-эксплуатируемыми (remotely exploitable). Уязвимости в защите, которые требуют наличия локального доступа типа shell account обычно называются локально-эксплуатируемыми (locally-exploitable). Наиболее опасен первый тип уязвимостей, так как он позволяет вообще произвольному (неизвестному) лицу получить возможность запуска произвольных программ.

Следует заметить, что данный пример отнюдь не единичен. Уязвимости разной степени опасности обнаруживаются в программах систематически несколько раз в месяц. В связи со столь неблагоприятной ситуацией, в США была разработана процедура сертификации программного обеспечения (Common Criteria). ПО, не прошедшее сертификацию по этой процедуре не может работать на критически важных серверах государственного значения.

Это показывает, почему ведущие производители телекоммуникационного оборудования и программного обеспечения привлекают большие ресурсы для аудита существующего массива программного обеспечения для выявления и устранения в них уязвимостей защиты. К сожалению, в настоящий момент процесс аудита программного обеспечения с целью выявления уязвимостей защиты совершенно неудовлетворительно поддерживается инструментальными средствами. Как будет показано далее, основная проблема существующих инструментальных средств – высокий процент ложных срабатываний, когда фрагмент программы, не содержащий ошибок, приводящих к уязвимостям защиты, отмечается как опасный. Высокий процент ложных срабатываний требует большого количества ручного труда для отсеивания ложных сообщений от сообщений, действительно выявляющих ошибки.

В настоящее время в рамках контракта с Nortel Networks в отделе компиляторных технологий ведётся разработка инструментального средства для автоматического выявления уязвимостей защиты некоторых типов. Дальнейшие разделы настоящей работы посвящены описанию разрабатываемого прототипа инструментального средства.

4.1. Виды уязвимостей защиты

В настоящее время сложилась некоторая классификация уязвимостей защиты в зависимости от типа программных ошибок, которые могут приводить к появлению уязвимости в программе. В рамках данной работы мы рассмотрим лишь некоторые виды уязвимостей.

Переполнение буфера (buffer overflow). Данная уязвимость возникает как следствие отсутствия контроля или недостаточного контроля за выходом за пределы массива в памяти. Языки Си/Си++, чаще всего используемые для разработки программного обеспечения системного уровня, не реализуют автоматического контроля выхода за пределы массива во время выполнения программы. Это самый старый из известных типов уязвимостей (знаменитый червь Морриса использовал, среди прочих, уязвимости переполнения буфера в программах sendmail и fingerd), уязвимости такого типа наиболее просто использовать.

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. Так, упомянутая выше программа rsync содержала уязвимость буфера в куче. Рассмотрим для примера более детально уязвимость переполнения буфера в стеке как наиболее простую на примере следующей простой программы:

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

Предположим, что стек процесса растёт в направлении уменьшения адресов памяти. В таком случае непосредственно перед выполнением функции gets стек будет иметь следующую структуру:

SP+96	Аргументы командной строки, переменные окружения и т. д.
SP+88	Аргументы функции main (argc, argv)
SP+84	Адрес возврата из main в инициализационный код
SP+80	Адрес предыдущего стекового фрейма
SP+80	Сохранённые регистры (если есть), локальные переменные (если есть)
SP	Буфер (char buf[80])

Как известно, функция gets не позволяет ограничивать длину вводимой со стандартного потока ввода строки. 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, таким образом 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. Теперь, чтобы проэксплуатировать такое переполнение буфера, необходимо подать на вход программе специальным образом подготовленную строку, которая будет содержать небольшую программу, выполняющую нужные злоумышленнику действия (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.

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 Си. 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 Си 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 “$”.

Уязвимости “испорченного ввода” (tainted input vulnerability). Это широкий класс уязвимостей защиты, в качестве подкласса включающий в себя уязвимости форматных строк. Уязвимости испорченного ввода могут возникать в случаях, когда вводимые пользователем данные без достаточного контроля передаются интерпретатору некоторого внешнего языка (обычно это язык 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. В рамках данной работы мы остановимся лишь на трёх перечисленных выше типах.

4.2. Инструментальные средства для обнаружения уязвимостей защиты

В настоящее время разработано большое количество инструментальных средств, предназначенных для автоматизации поиска уязвимостей защиты программ на языках Си и Си++. В данном разделе мы рассмотрим наиболее распространённые инструментальные средства.

По виду использования инструментальные средства можно разделить на два типа: инструменты, добавляющие дополнительные динамические проверки в программу и инструменты только статического анализа программ. В нашей работе мы рассмотрим инструменты, которые выявляют уязвимости защиты с помощью статического анализа программ.

- **CodeSurfer**. CodeSurfer – это инструмент анализа программ, который не предназначается непосредственно для поиска ошибок уязвимости защиты. Его основными достоинствами являются:
  - Анализ указателей
  - Различные анализы потока данных (использование и определение переменных, зависимость данных, построение графа вызовов)
  - Скриптовый язык.

CodeSurfer может быть использован для поиска ошибок в исходном коде, для улучшения понимания исходного кода, и для реинженерии программ. В рамках среды CodeSurfer велась разработка прототипа инструментального средства для обнаружения уязвимостей защиты, однако разработанное инструментальное средство используется только внутри организации разработчиков.

- **Flawfinder**, **ITS4**, **RATS**, **PScan.** Все эти программы разработаны для поиска ошибок переполнения буфера и ошибок, связанных с использованием форматных строк. Данные инструменты во многом схожи. Они все используют возможности только лексического и простейшего синтаксического анализа, поэтому в данном случае не приходится говорить о сколь-нибудь эффективном нахождении ошибок при помощи этих программ – результаты, выданные ими, могут содержать до 100% ложных сообщений.

Основные свойства этих программ:

- База данных потенциально опасных функций (ITS4, RATS)
- Подробное аннотирование исходного кода (Flawfinder, ITS4)
- Возможность поиска функций, принимающих внешний ввод. (Flawfinder – с опцией *–inputs*, RATS)

- **UNO.** UNO – простой анализатор исходного кода. Он был разработан для нахождения таких ошибок, как неинициализированные переменные, нулевые указатели и выход за пределы массива. UNO позволяет выполнять несложный анализ потока управления и потоков данных, осуществлять как внутри- так и меж-процедурный анализ, специфицировать свойства пользователя. Однако, к сожалению, данный инструмент не доработан для анализа реальных приложений, не поддерживает многие стандартные библиотеки, и, на данном этапе, разработки не позволяет анализировать сколь-нибудь серьёзные программы.
- **FlexeLint, Splint.** Это наиболее мощные инструменты из всех, рассмотренных в данной работе. Они предназначены для анализа исходного кода с целью выявления различных ошибок.

  Обе программы производят семантический анализ исходного кода, анализ потоков данных и управления. В конце работы выдаются сообщения нескольких основных типов:
  - Возможен нулевой указатель.
  - Проблемы с выделением памяти (например, нет free() после malloc()).
  - Проблемный поток управления (например, недостижимый код).
  - Возможно переполнение буфера, арифметическое переполнение.

Количество таких сообщений при работе над кодом большой программы может быть очень значительным. Поэтому оба инструмента поддерживают опции, позволяющие настраивать сообщения. Опции могут задаваться также и в исходном коде, например, в виде аннотаций.

FlexeLint и Splint не разрабатывались с целью поиска ошибок уязвимости защиты. Однако обе программы позволяют находить некоторые, наиболее простые случаи потенциального переполнения буферов. При этом FlexeLint не подходит для нахождения ошибок с форматными строками, в то время как Splint выдаёт предупреждения такого типа.

Как видно из приведённого краткого обзора, существующие инструменты не задействуют все современные методы статического анализа программ, ограничиваясь лишь контекстным анализом. Как было отмечено в [12], применение глубокого статического анализа программ может позволить существенно снизить количество ложных срабатываний и повысить точность обнаружения уязвимостей защиты.

4.3. Использование методов анализа потоков данных для решения задачи обнаружения уязвимостей

В отделе компиляторных технологий по контракту с фирмой Nortel Networks разрабатывается прототип инструментального средства для автоматического обнаружения уязвимостей. В прототипе нами реализовано автоматическое выявление уязвимостей переполнения буфера, форматных строк, испорченного ввода. В дополнение к ним реализовано обнаружение ошибок утечки динамической памяти.

Для разрабатываемого инструментального средства нами реализован новый алгоритм выявления уязвимостей защиты, основанный на глубоком межпроцедурном анализе указателей в программе. Алгоритм состоит из следующих основных компонент:

- Внутрипроцедурный анализ указателей, основанный на понятии “абстрактной ячейки памяти” (abstract memory location).
- Анализ диапазонов для переменных целых типов.
- Контекстно-зависимый (context-sensitive) и потоково-зависимый (flow-sensitive) межпроцедурный анализ.
- Специальная поддержка основных функций стандартной библиотеки языка Си и возможность спецификации семантики функции с помощью аннотаций.

Анализ указателей (alias analysis) [10] позволяет для каждой переменной указательного типа в каждой точке программы построить множество объектов, на которые он может указывать. В языке Си анализ указателей является необходимым шагом для выполнения практически любого оптимизирующего преобразования. Кроме того, анализ указателей для языка Си осложняется неразличимостью указателей и массивов, а также присутствием указательной арифметики.

Для анализа указателей мы выбрали подход, основанный на моделировании операций с указателями. Каждому объекту, который может существовать при работе программы, то есть статическим переменным, переменным, создаваемым и уничтожаемым на стеке, переменным в динамической памяти ставится в соответствие абстрактная ячейка памяти. При анализе программы абстрактные ячейки памяти создаются, когда в работающей программе выделяется память под соответствующую переменную. Абстрактная ячейка памяти хранит следующую информацию:

Size	Размер данной абстрактной ячейки. Для функций динамического выделения памяти размер ячейки определяется по параметру функции.
overlap	Множество абстрактных ячеек памяти, которые накладываются на данную абстрактную ячейку. Этот атрибут используется для переменных агрегатных, потому что в таких случаях создаются отдельные абстрактные ячейки памяти и для структуры целиком, и для каждого составляющего структуру поля.

Атрибуты абстрактной ячейки памяти, описанные выше, являются статическими, то есть не изменяются всё время жизни абстрактной ячейки памяти.

В каждой точке программы с абстрактной ячейкой памяти могут быть связаны динамические атрибуты, которые описываются ниже.

Len	Текущая длина строки для строковых переменных.
value	Значение переменной.
input	Указывает, зависит ли данная абстрактная ячейка памяти в данной точке программы прямо или косвенно от внешних источников данных.

Поле value содержит текущее значение переменных. Для хранения текущего значения переменных используются мета-типы, являющиеся расширением соответствующих типов языка. Например, значения переменных целых типов при анализе представляются типом M_Integer, который может принимать следующие значения:

Undef	Неопределённое значение, изначально возникает, когда переменная неинициализирована и как результат операций, если один из аргументов – Undef.
Any	Переопределённое значение. Возникает когда статического анализа недостаточно для определения значения переменной (например, при чтении значения переменной из внешнего источника), либо как результат операции, если один из аргументов имеет значение Any, либо как результат операции при соответствующих операндах.
[a,b]	Любое значение в интервале от a до b.

При выполнении операций над интервалами, в особенности операций слияния значений в точках слияния потока управления, может возникнуть ситуация, когда результирующее множество целых значений состоит из нескольких непересекающихся интервалов, например [1,2]+[6,15]. В этом случае берётся минимальный интервал, включающий в себя все непересекающиеся интервалы, то есть в данном случае результатом будет интервал [1,15].

Значения указательных типов представляются множествами пар (AML, offset), где AML – абстрактная ячейка памяти, а offset – смещение от начала ячейки, которое имеет мета-тип M_Integer, то есть представляет собой диапазон смещений указателя внутри данного объекта. Кроме того, поддерживается значение Undef для неопределённых (неинициализированных) переменных, значение Any(type), если указательное выражение может принимать

произвольное значение типа type, и значение Any, если указательное выражение может принимать произвольное значение. Значения Any могут возникать как результат слияния значений переменных в точках слияния потока управления, вследствие ограниченной глубины анализа объектов в динамической памяти, и когда указательное значение возвращается функцией, для которой не существует исходного кода и не специфицированы аннотации.

Внутрипроцедурный анализ указателей реализован по стандартной схеме итеративного прямого анализа потоков данных процедуры [11]. Для каждой инструкции процедуры на основании входящих динамических атрибутов абстрактных ячеек памяти вычисляются выходящие атрибуты, которые затем подаются на вход следующей инструкции. В точке слияния потока управления выполняется операция слияния динамических атрибутов (join). Анализ выполняется итеративно до тех пор, пока множества динамических атрибутов не перестанут изменяться.

Одновременно с анализом указателей по аналогичной схеме выполняется анализ целочисленных интервалов. Эти два вида анализа переплетаются друг с другом, поскольку от интервалов целочисленных переменных могут зависеть указательные выражения и наоборот.

При выполнении внутрипроцедурного анализа основную сложность представляют циклы. Для циклов реализуются специальный алгоритм вычисления диапазонов индуктивных переменных и зависящих от них выражений, состоящий из двух шагов. На первом шаге делается расширение диапазона индуктивной переменной до максимального, на втором шаге выполняется ограничение диапазона с учётом условий в теле цикла.

При анализе практически любой программы возникает необходимость в межпроцедурном анализе. По степени точности анализа можно выделить контекстно-чувствительные и контекстно-нечувствительные методы анализа. Во втором типе методов анализа процедуры рассматриваются независимо от контекста, в котором они вызываются. Множество динамических атрибутов на входе в процедуру получается как слияние множеств динамических атрибутов во всех точках вызова данной процедуры. Множество динамических атрибутов на выходе процедуры переносится во все точки возврата из процедуры в вызывающую её процедуру. Как и в случае внутрипроцедурного анализа межпроцедурный анализ ведётся итеративно до тех пор, пока множества на входах и выходах не перестанут изменяться. В контекстно-чувствительных методах анализа, анализ каждой процедуры проводится независимо для каждого вызова процедуры и учитывает контекст вызова этой процедуры. Как следствие, контекстно-чувствительный анализ требует больше ресурсов для своего выполнения.

В нашем инструментальном средстве мы используем гибридный подход. В точке вызова каждой процедуры учитывается контекст вызова, но контекст возврата из процедуры берётся как объединение всех контекстов выхода для

всех вызовов данной процедуры в программе. Это позволяет повысить точность анализа, несильно увеличивая его сложность.

Дополнительной сложностью, возникающей при межпроцедурном анализе, является необходимость анализа указателей при вызовах процедур через указатель. Хотя множество возможных значений указательного выражения нарабатывается в процессе межпроцедурного анализа, это может потребовать перестройки графа вызовов процедур на ходу. В текущем прототипе нашего инструментального средства мы используем консервативный подход, предполагая, что каждый раз по указателю могут быть вызваны все процедуры, списки формальных параметров которых соответствуют фактическим параметрам вызова.

Для выявления уязвимостей защиты программ, работающих в некотором операционном окружении необходимо знание семантики работы этого окружения. Прототипная версия инструментального средства разработана в расчёте на операционное окружение, предоставляемое Linux. Непосредственно в алгоритм анализа встроена поддержка основных функций стандартной библиотеки языка Си (в особенности функций работы со строками и с памятью, в том числе динамической), основных примитивов POSIX работы с файлами, файловой системой, процессами и т. д., а также некоторых специфичных расширений Linux, в частности, интерфейса модулей ядра.

В настоящее время разрабатывается язык аннотаций, чтобы дать возможность пользователю нашего инструментального средства самому специфицировать семантику процедур, отсутствующих в исходном коде или для которых автоматический анализ недостаточно точен. При разработке языка аннотаций необходимо учитывать противоречивые требования: во-первых, язык должен быть достаточно мощным, чтобы позволять специфицировать семантику с требуемой для анализа степенью точности, но, с другой стороны, он должен быть достаточно простым, чтобы не требовать от пользователей специфических знаний формальных методов, и т. д.

Язык аннотаций строится на расширении синтаксиса языка Си, уже применяющемся в широко распространённом компиляторе GCC. Так, для спецификации того, что возвращаемое значение некоторой функции foo находится в интервале [0,5] используется следующая конструкция:

```
int foo(int x) __attribute__ ((post(foo >= 0 && foo <= 5)));
```

Здесь `__attribute__((...))` - это синтаксическое расширение GNU C, поддерживаемое нашим инструментальным средством, `post` – специальный атрибут, позволяющий определять постусловие для функции, а имя функции foo используется в постусловии для обозначения значения, возвращаемого этой функцией. Кроме того, реализуется специальная поддержка для стандартного макроса assert.

4.4. Результаты экспериментов

Текущий прототип инструментального средства был проверен на нескольких тестовых примерах, как широко распространённых (bftpd), так и являющихся частью приложений, разрабатываемых в фирме Nortel для своего оборудования. Были получены следующие результаты.

Application	Total number of warnings	Number of “true positives”	Number of “possible errors”	Number of “false positives”	“false positives” %	FlexeLint
Bigfoot	20	4	3	13	65%	0 (0%)
Log_api	4	1	3	0	0%	0 (0%)
Bftpd	57	2	32	23	40%	0 (0%)
Config_api	11	9	0	2	18%	1 (11%)

Таблица 6. Результаты запуска инструментального средства

В этой таблице, в столбце «Total» приведено общее количество сообщений об обнаруженных ошибках, выведенных нашим инструментальным средством. В столбце «True» дано количество сообщений, указывающих на действительные проблемы в коде. В столбце «Possible» дано количество сообщений, истинность или ложность которых мы не смогли подтвердить из-за недостатка информации о программе. В столбце «False» дано количество ложных срабатываний. Наконец, в столбце «FlexeLint» дано количество истинных сообщений об ошибке, выявленных инструментальным средством FlexeLint.

Как видно из проведённых испытаний, текущий прототип системы для обнаружения уязвимостей защиты демонстрирует очень хорошие результаты. Процент ложных срабатываний оказался намного ниже, чем у других аналогичных инструментальных средств.

5. Интегрированная среда

Все направления исследований, описанные в настоящей статье реализуются на базе единой интегрированной среды для изучения алгоритмов анализа и оптимизации программ [1]. IRE является системой с открытыми исходными кодами и распространяется на условиях Общей публичной лицензии GNU (GNU General Public License). Система доступна для загрузки из сети Интернет [4].

Интегрированная среда построена как набор связанных друг с другом инструментов, работающих над общим промежуточным представлением программ MIF. Для управления инструментами ИС предоставляется графический интерфейс пользователя.

В настоящее время в качестве исходного и целевого языка программирования используется язык Си, но внутреннее представление разработано таким

образом, чтобы поддерживать широкий класс процедурных и объектно-ориентированных языков программирования. Все компоненты ИС реализованы на языке Java, за исключением транслятора из Си в MIF, который реализован на языке Си.

5.1. Состав среды

Программа на языке Си транслируется в промежуточное представление с помощью компонента "Анализатор Си в MIF". В настоящее время поддерживается стандарт ISO C90 и некоторые расширения GNU. Чтобы обеспечить независимость интегрированной среды от деталей реализации стандартной библиотеки Си для каждой конкретной платформы и обеспечить возможность корректной генерации программы на Си по её внутреннему представлению, анализатор использует собственный набор стандартных заголовочных файлов (`stdio.h` и т. д.). На уровне стандартной библиотеки полностью поддерживается стандарт ISO C90 и некоторые заголовочные файлы POSIX. Внутреннее представление программы находится в памяти инте-грированной среды, но возможно сохранение внутреннего представления в файле.

Компонент "Генератор MIF ->C" позволяет по программе во внутреннем представлении получить программу на языке Си. При генерации программы корректно генерируются директивы `#include` для всех использованных в исходной программе системных заголовочных файлов. Для проведения полустатического анализа программ генератор поддерживает несколько типов инструментирования программы. Инструментирование программы заключается во внесении в ее текст специальных операторов, собирающих информацию о ходе выполнения программы. В настоящее время генератор поддерживает инструментализацию программы для сбора полных трасс выполнения программы, профилирование базовых блоков и дуг, профилирование значений. Собранные в результате выполнения инструментированной программы профили выполнения могут впоследствии использоваться для анализа и преобразования программ.

Компоненты "Анализаторы" реализуют различные методы статического и полустатического анализа программ. При этом сама программа не трансформируется, а во внутреннее представление программы добавляется полученная в результате выполнения анализа информация. В интегрированной среде эти компоненты доступны посредством пункта меню Analyze. Например, алгоритм разбиения программы на базовые блоки, доступный через пункт меню Mark basic blocks, строит граф потока управления программы, создаёт соотвествующие структуры данных в памяти системы и добавляет ссылки на построенный граф в структуры данных внутреннего представления программы.

Компоненты "Трансформаторы" реализуют различные преобразования программ. При этом результатом работы компонента трансформации является

новая программа во внутреннем представлении, для которой в интегрированной среде создаётся новое окно. Исходная программа сохраняется неизменной. Трансформационные компоненты доступны в интегрированной среде посредством пунктов меню Optimize, Transform и Obfuscate в зависимости от класса преобразования.

Компоненты "Визуализаторы" реализуют различные алгоритмы визуализации информации о программе. Эти компоненты доступны посредством пункта меню Vizualize интегрированной среды.

5.2. Промежуточное представление

Промежуточное представление MIF используется всеми инструментами интегрированной среды. Оно является представлением среднего уровня и спроектировано таким образом, чтобы представлять программы, написанные на широком спектре процедурных и объектно-ориентированных языков программирования.

Программа в представлении MIF представляет собой последовательность четвёрок, которые используются для представления как декларативной, так и императивной информации о программе. Текстуальное представление MIF используется как интерфейс между анализатором языка и интегрированной средой, а также для хранения анализируемых программ.

6. Заключение

В данной работе мы рассмотрели несколько направлений исследований, которые ведутся в отделе компиляторных технологий Института системного программирования РАН. Эти исследования используют интегрированную среду исследования алгоритмов анализа и трансформации программ, разрабатываемую в ИСП РАН и на факультете ВМиК МГУ. Открытость и расширяемость интегрированной среды позволяет достаточно легко накапливать прототипные реализации алгоритмов анализа и трансформации программ, которые разрабатываются в рамках проводимых исследований.

Накопление библиотеки алгоритмов позволяет с успехом применять интегрированную среду в учебном процессе факультетов ВМиК МГУ и ФПМЭ МФТИ. Студенты, выполняя курсовые и дипломные работы, получают в своё распоряжение развитый инструментарий методов анализа и оптимизации, на основе которых они могут реализовывать новые методы анализа и оптимизации. После включения в интегрированную среду результаты работы станут доступны для дальнейшего использования. Другое учебное применение интегрированной среды заключается в использовании её в качестве пособия для изучающих курсы по методам анализа и оптимизации программ.

В дальнейшем мы планируем развивать все три рассмотренных в данной работе направления работ, а также усовершенствовать интегрированную среду для облегчения её использования. Для этого, в частности, планируется

реализация объектной библиотеки и объектно-ориентированного интерфейса ко внутреннему представлению.

Литература

[1] А. В. Чернов. Интегрированная инструментальная среда Poirot для изучения методов маскировки программ. *Препринт Института системного программирования РАН*. М.: ИСП РАН, 2003.

[2] A. Chernov. A New Program Obfuscation Method. In *Proceedings of the Adrei Ershov Fifth International Conference "Perspectives of Systems Informatics". International Workshop on Program Understanding*, Novosibirsk, July 14-16, 2003.

[3] A. Chernov, A. Belevantsev, O. Malikov. A Thread Partitioning Algorithm for Data Locality Improvement. To appear in *Proceedings of Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM 2003)*, Czestochowa, Poland, September 7-10, 2003.

[4] The IRE Home Page. http://www.ispras.ru/groups/ctt/ire.html

[5] J. E. Moreira. On the implementation and effectiveness of autoscheduling for shared-memory multiprocessors. Ph.D. thesis, Department of Electrical and Computer Engineering, Univ. of Illinois at Urbana-Champaign, 1995.

[6] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving Value Communication for Thread-Level Speculation. Computer Science Department Carnegie Mellon University. 2002.

[7] SUIF Compiler System Group (http://suif.stanford.edu)

[8] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Totonto, CA, June 1991.

[9] X. Tang, J. Wang, K. Theobald, and G. R. Gao. Thread partitioning and scheduling based on cost model. ACAPS Tech. Memo 106, School. of Computer Science, McGill University, Montreal, Quebec. Apr. 1997.

[10] R. P. Wilson, M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1-12, June 1995.

[11] S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997.

[12] D. Wagner, J. S. Foster, E. A. Brewer, A. Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium*, pages 3-17, February 2000.