Анализ запутывающих преобразований программ

Чернов А. В.
E-mail: cher@ispras.ru

Аннотация. *Запутанной* (obfuscated) называется программа, которая на всех допустимых для исходной программы входных данных выдаёт тот же самый результат, что и оригинальная программа, но более трудна для анализа, понимания и модификации. Запутанная программа получается в результате применения к исходной незапутанной программе *запутывающих преобразований* (obfuscating transformations).

Данная работа посвящена анализу запутывающих преобразований графа потока управления программы (функции), опубликованных в открытой печати, с точки зрения их устойчивости к различным видам анализа программы. В данной работе запутывание изучается на уровне языка Си, то есть исходная программа написана на языке Си, и целевая запутанная программа генерируется также на языке Си.

В работе приведена классификация запутывающих преобразований с точки зрения методов анализа программ, которые могут быть использованы для их распутывания. Показано, что для каждого класса рассмотренных запутывающих преобразований существуют методы анализа, которые позволяют эффективно противодействовать таким преобразованиям. Для иллюстрации этого приведены несколько практических примеров распутывания программ, запутанных как вручную, так и автоматическими запутывателями.

1. Введение

В данной работе исследуется проблема анализа запутывающих преобразований графа потока управления функций на языке Си. В работе сделана попытка анализа запутывающих преобразований, опубликованных в открытой печати, с точки зрения их устойчивости к различным видам статического и динамического анализа программ. Запутывание изучается на уровне языка Си, то есть и исходная программа написана на языке Си, и целевая запутанная программа генерируется также на языке Си.

В работе приведена классификация запутывающих преобразований с точки зрения методов анализа программ, которые могут быть использованы для их распутывания. Показано, что для каждого класса рассмотренных запутывающих преобразований существуют методы анализа, которые

позволяют эффективно противодействовать таким преобразованиям. Для иллюстрации этого приведены несколько практических примеров распутывания программ, запутанных как вручную, так и автоматическими запутывателями.

Задачи запутывания и анализа запутанных программ имеют три аспекта: теоретический, включающий в себя разработку новых алгоритмов преобразования графа потока управления или трансформации данных программы, а также теоретическую оценку сложности их анализа и раскрытия. Прикладной аспект включает в себя разработку конкретных методов запутывания (распутывания), то есть наилучших комбинаций алгоритмов, эмпирический сравнительный анализ различных методов, эмпирический анализ устойчивости методов, и т. д.

Третий аспект, психологический пока не поддаётся формализации, но не может игнорироваться. Обратная инженерия (понимание) программ – это процесс, результатом которого является некоторое знание субъекта, изучающего программу, который является неотъемлемой частью процесса понимания [18]. Методы запутывания должны максимально использовать свойства (точнее, слабости) человеческой психики.

Не умаляя ценности теоретических исследований, следует заметить, что теоретические выводы должны подтверждаться результатами практического применения предложенных методов. В данной работе исследуется прикладной аспект задачи запутывания.

В настоящее время широко распространены языки программирования, такие как Java, в которых «исполняемой» формой программы является не машинный код для некоторого типа процессоров, но машинно-нейтральное представление. Задача декомпиляции программы из такого представления обратно в программу на языке Java значительно проще, чем декомпиляция из машинного кода. Существует большое число декомпиляторов для языка Java как распространяемых свободно, так и коммерческих, например [20], что упрощает несанкционированное использование, обратную инженерию и модификацию Java-программ. В качестве одного из способов борьбы с этим рассматривается запутывание программ.

Уже разработано около двух десятков различных запутывателей Java-программ, среди которых есть и коммерческие, например [25]. Простые запутыватели удаляют таблицы символов и отладочную информацию из скомпилированных классов и заменяют исходные имена методов бессмысленными короткими именами. В результате размер файлов уменьшается (до 50%), а скорость выполнения программы значительно возрастает, поэтому такое запутывание может рассматриваться и как один из способов оптимизации программ.

Более развитые запутыватели программ на языке Java, а также запутыватели программ на других языках программирования выполняют преобразования

графа потока управления программы и её структур данных. Методы, используемые в них, как правило, подобраны эмпирически и слабо обоснованы теоретически. Сравнительный анализ запутывателей Java-программ, доступных через Интернет, проведён в работе [14].

Возможны разные уровни постановки задачи запутывания и анализа запутывающих преобразований. Во-первых, запутывание может рассматриваться в рамках языка Java. В этом случае исходная программа написана на языке Java, и запутанная программа также написана на языке Java. Однако язык Java допускает только структурные программы, то есть графы потока управления Java-программ всегда сводимые, что существенно ограничивает диапазон применимых преобразований графа потока управления.

Мы рассматриваем задачу анализа запутывающих преобразований в рамках языка Си. Поскольку Си – язык более низкого уровня, чем Java или даже байт-код Java, задачи запутывания и анализа для этих языков оказываются вложенными в соответствующие задачи для языка Си.

Возможна постановка задачи запутывания на ещё более низком уровне, когда запутывается программа на языке ассемблера или даже объектная программа в машинном коде (в последнем случае она должна генерироваться специальным запутывающим компилятором). В ассемблерных и объектных программах можно использовать специфические особенности работы целевой машины, добившись того, что восстановление программы на Си будет крайне затруднено [17]. Но с другой стороны, методы запутывания, применимые к одной архитектуре, могут оказаться неприменимы к другой архитектуре. Заметим, что проблема низкоуровневого запутывания в настоящее время исследована слабо. Нам не известно каких-либо опубликованных методов низкоуровневого запутывания программ, поэтому проблему низкоуровневого запутывания мы в этой работе рассматривать не будем.

Если программа для анализа представлена в исполняемом или объектном коде, и известно, что к программе не применялись низкоуровневые методы запутывания, задача анализа таких программ может быть разбита на две относительно независимых подзадачи. На первом этапе программа декомпилируется [4] в программу на языке Си, затем программа на языке Си распутывается, то есть применяются алгоритмы анализа программ, которые приводят к её возможной перестройке, выделению в ней циклов, условных операторов и других конструкций высокого уровня. Декомпиляция программ – самостоятельная задача, которая может решаться отдельно.

В данной работе мы ограничим класс запутываемых программ программами пакетной обработки, то есть программами, которые получают все исходные данные в начале работы и выдают результат по ходу работы. Во время работы программа не взаимодействует с пользователем и другими программами. Кроме того, потребуем, чтобы программа не использовала аппарат исключений в работе. Появление исключительной ситуации приводит к завершению работы

программы. Эти ограничения связаны с тем, что все опубликованные методы запутывания применимы только к таким программам.

Запутывание программ – достаточно молодое направление исследований. Обзор (таксономия) запутывающих преобразований, известных на тот момент, был опубликован в работе [5] группы, возглавляемой К. Колбергом и К. Томборсоном. В дальнейших работах [6], [7], [8], [9], [15] этой группы опубликованы результаты исследований конкретных алгоритмов запутывания графа потока управления и данных программы, а также приложения запутывания программ к смежным областям, таким как обеспечение устойчивости программы к несанкционированной модификации (tamper-resistance) или внесение в программу «водяных знаков» (watermarking).

Классификация, введённая в работе [5], широко используется, и получила дальнейшее развитие в работах [10], [13], [22].

В работах [23], [24] был предложен новый подход к запутыванию графа потока управления программы, который заключается в преобразовании графа в «плоскую» форму. Чтобы затруднить статическое определение порядка следования базовых блоков используется преобразование, вводящее в программу *алиасы*. Показывается, что статический анализ запутанной программы с целью восстановления порядка следования базовых блоков является NP-трудной задачей.

В дальнейшем этот подход был развит в работе [3], которая дополнительно предлагает использовать переплетение базовых блоков совместно запутываемых функций и недетерминированный выбор следующего базового блока из множества эквивалентных альтернатив. Доказывается, что статический анализ запутанной программы с целью восстановления порядка следования базовых блоков является PSPACE-трудной задачей.

В работе [2] получен результат, который определяет верхний предел силы запутывающих преобразований. Авторы доказали, что *универсального* запутывателя не существует. Под универсальным понимается такой запутыватель, который для любой программы строит запутанную программу, такую что определение любого свойства программы, легко определимого по исходной программе, неэффективно по запутанной программе.

Тем не менее, можно показать, что если взять некоторое специальное свойство программ, то запутыватель для этого свойства всё же существует. Вопрос о том, для каких классов программ и каких свойств запутыватель существует, остаётся открытым.

В данной работе рассматриваются только запутывающие преобразования графа потока управления программы. Мы сознательно оставляем в стороне преобразования данных программы, а также так называемые *превентивные* преобразования, которые нацелены против определённых методик декомпиляции программы, реализованных в определённых декомпиляторах.

Мы не пытаемся подтвердить или опровергнуть тезис работы [2] о невозможности запутывания программ, но мы делаем попытку показать, что для всех опубликованных в открытой печати методов запутывания программ существуют достаточно эффективные практически, хотя, возможно, пока не совсем обоснованные теоретически способы противодействия.

Данная работа имеет следующую структуру. В разделе **2** даётся формальное определение понятия запутывания, приводится классификация запутывающих преобразований, описываются запутывающие преобразования графа потока управления. В разделе **3** описываются наиболее важные методы, которые применяются на различных стадиях работы компилятора и могут быть использованы для получения информации о запутанной программе, а также специальные методы анализа запутанных программ. В разделе **4** методы запутывания программ сопоставляются с методами их анализа, вводится классификация методов запутывания по уровню необходимых преобразований распутывания. В разделе **5** приводятся примеры применения некоторых методов анализа программы для распутывания программ. Наконец, в разделе **5.3** подводятся итоги и указываются направления дальнейшей работы.

2. Запутывание

В этом разделе мы дадим формальное определение запутывателя свойства π-класса программ *P*. Мы введём некоторые показатели качества запутывающих преобразований и перечислим разнообразные запутывающие преобразования, каждое из которых по отдельности усложняет граф потока управления программы. Опубликованные методы запутывания, которые в настоящее время применяются в программных инструментах, как свободно-распространяемых, так и коммерческих, являются комбинацией нескольких перечисленных запутывающих преобразований. Некоторые из методов запутывания будут описаны в конце раздела.

В дальнейшем мы будем рассматривать программы из класса П неинтерактивных, нереактивных программ. Программе поступают на вход данные характерного размера *n*. При работе программа не возбуждает и не использует исключительных ситуаций.

2.1. Определение

Приводимое здесь определение сформулировано В. А. Захаровым.

Эффективное вычисление – это вычисление, требующее полиномиального от длины входа времени и полиномиальной от длины входа памяти. Эффективная программа (машина Тьюринга) – программа, работающая полиномиальное от длины входа время и требующая полиномиальную от длины входа рабочую память на всех входах, на которых программа завершается.

1. Пусть П – множество всех программ (машин Тьюринга), удовлетворяющих сформулированным выше ограничениям, и пусть программа $p \in \Pi$

вычисляет функцию
$f_p : Input \to Output$,
подмножество $\pi \subseteq \Pi$ называется *функциональным свойством* если
$\forall p_1, p_2 \in \Pi (f_{p_1} = f_{p_2} \Rightarrow (p_1 \in \pi \Leftrightarrow p_2 \in \pi))$.

2. Пусть π – функциональное свойство, $P \subseteq \Pi$ – класс программ такой, что существует эффективная программа c такая, что для любой программы $p \in P$

$$c(p) = \begin{cases} 1, & \text{если } p \in \pi \\ 0, & \text{если } p \notin \pi \end{cases}$$

Другими словами, для функционального свойства π мы определяем класс программ P таких, что существует эффективная программа-распознаватель c свойства π по программе p из класса P.

3. Вероятностная программа o называется *запутывателем* класса P относительно свойства π, (P, π)-запутывателем, если выполняются условия:

- (эквивалентность преобразования запутывания). Для любой $p \in P$ и
  $p' \in o(p)$
  $f_p = f_{p'}$,
  $|p'| = poly(|p|)$,
  $\forall x \in Dom_{f_p} \quad time_{p'}(x) = poly(time_p(x))$
  Здесь $y = poly(x)$ означает, что y ограничен полиномом некоторой степени от переменной x, $time_p(x)$ – время выполнения программы p на входе x, $|p|$ – размер программы p.

- (трудность определения свойств по запутанной программе). Для любого полинома q и для любой программы (вероятностной машины Тьюринга) a такой, что $a(o(P)) = \{0,1\}$, и для любой $p \in P$ выполняется $time_a(o(p)) = poly(|o(p)|)$, существует программа (вероятностная машина Тьюринга с оракулом) b, и при этом для любой $p \in P$ $|\Pr(a(o(p)) = c(p)) - \Pr(b^p(1^{|p|}) = c(p))| \le \frac{1}{q(|p|)}$.

Другими словами, вероятность определить свойство π по запутанной программе равна вероятности определения свойства π только по входам и выходам функции f_p. То есть, наличие текста запутанной программы ничего не даёт для выявления свойств этой программы.

Универсальный запутыватель – это программа O, которая для любого класса программ P и любого свойства π является (P, π)-запутывателем. Как показано в работе [2], универсального запутывателя не существует. Доказательство заключается в построении специального класса программ P и выборе такого свойства π, что для любого преобразования программы из этого класса свойство π устанавливается легко. Однако вопрос о том, существуют ли запутыватели для отдельных классов свойств программ, и насколько широки и практически значимы эти классы свойств, остаётся открытым.

С практической точки зрения запутывание программы можно рассматривать как такое преобразование программы, которое делает её обратную инженерию экономически невыгодной. Несмотря на слабую теоретическую проработку, уже разработано большое количество инструментов для запутывания программ.

2.2. Стоимость запутывающих преобразований

Запутывание преобразует программу, затрудняя её обратную инженерию. В результате запутанная программа может оказаться больше по размеру и работать медленнее. *Стоимость* (cost) преобразования [5] – это метрика, которая позволяет оценить, насколько больше требуется ресурсов (памяти, процессорного времени) для выполнения запутанной программы, чем для выполнения исходной программы. Стоимость определяется по следующей шкале: ⟨*бесплатная*, *дешёвая*, *умеренная*, *дорогая*⟩.

Стоимость преобразования позволяет оценить, насколько увеличивается размер функции в результате запутывания. *Бесплатное* преобразование увеличивает размер функции на $O(1)$, *дешевое* преобразование увеличивает размер на $O(m)$, где m – размер функции, *умеренное* по стоимости преобразование увеличивает размер функции на $O(m^p)$, где $p > 1$. Наконец, *дорогое* преобразование экспоненциально увеличивает размер запутанной функции по сравнению с исходной.

Стоимость выполнения позволяет оценить, насколько больше требуется ресурсов при выполнении программы. Стоимость оценивается как функция от характерного размера входных данных n.

Преобразование оценивается как *бесплатное*, если выполнение преобразованной программы p' требует на $O(1)$ больше ресурсов, чем выполнение оригинальной программы. Преобразование оценивается как *дешёвое*, если выполнение программы p' требует на $O(n)$ ресурсов больше, чем выполнение исходной программы, где n – размер входных данных. Преобразование оценивается как *умеренное* по стоимости, если выполнение программы p' требует на $O(n^p)$ больше ресурсов, где $p > 1$. Преобразование

оценивается как *дорогое*, если выполнение программы p' требует экспоненциально больше ресурсов, чем выполнение исходной программы. Практически применимыми являются, по-видимому, только бесплатные и дешёвые методы запутывания.

2.3. Описание запутывающих преобразований

Запутывающие преобразования можно разделить на несколько групп в зависимости от того, на трансформацию какой из компонент программы они нацелены.

- Преобразования форматирования, которые изменяют только внешний вид программы. К этой группе относятся преобразования, удаляющие комментарии, отступы в тексте программы или переименовывающие идентификаторы.
- Преобразования структур данных, изменяющие структуры данных, с которыми работает программа. К этой группе относятся, например, преобразование, изменяющее иерархию наследования классов в программе, или преобразование, объединяющее скалярные переменные одного типа в массив. В данной работе мы не будем рассматривать запутывающие преобразования этого типа.
- Преобразования потока управления программы, которые изменяют структуру её графа потока управления, такие как развёртка циклов, выделение фрагментов кода в процедуры, и другие. Данная статья посвящена анализу именно этого класса запутывающих преобразований.
- Превентивные преобразования, нацеленные против определённых методов декомпиляции программ или использующие ошибки в определённых инструментальных средствах декомпиляции.

2.3.1. Преобразования форматирования

К преобразованиям форматирования относятся удаление комментариев, переформатирование программы, удаление отладочной информации, изменение имён идентификаторов.

Удаление комментариев и переформатирование программы применимы, когда запутывание выполняется на уровне исходного кода программы. Эти преобразования не требуют только лексического анализа программы. Хотя удаление комментариев – одностороннее преобразование, их отсутствие не затрудняет сильно обратную инженерию программы, так как при обратной инженерии наличие хороших комментариев к коду программы является скорее исключением, чем правилом. При переформатировании программы исходное форматирование теряется безвозвратно, но программа всегда может быть

переформатирована с использованием какого-либо инструмента для автоматического форматирования программ (например, indent для программ на Си).

Удаление отладочной информации применимо, когда запутывание выполняется на уровне объектной программы. Удаление отладочной информации приводит к тому, что имена локальных переменных становятся невосстановимы.

Изменение имён локальных переменных требует семантического анализа (привязки имён) в пределах одной функции. Изменение имён всех переменных и функций программы помимо полной привязки имён в каждой единице компиляции требует анализа межмодульных связей. Имена, определённые в программе и не используемые во внешних библиотеках, могут быть изменены произвольным, но согласованным во всех единицах компиляции образом, в то время как имена библиотечных переменных и функций меняться не могут. Данное преобразование может заменять имена на короткие автоматически генерируемые имена (например, все переменные программы получат имя v<номер> в соответствии с их некоторым порядковым номером). С другой стороны, имена переменных могут быть заменены на длинные, но бессмысленные (случайные) идентификаторы в расчёте на то, что длинные имена хуже воспринимаются человеком.

2.3.2. Преобразования потока управления

Преобразования потока управления изменяют граф потока управления одной функции. Они могут приводить к созданию в программе новых функций. Краткая характеристика методов приведена ниже.

Открытая вставка функций (function inlining) [5], п. 6.3.1 заключается в том, что тело функции подставляется в точку вызова функции. Данное преобразование является стандартным для оптимизирующих компиляторов. Это преобразование одностороннее, то есть по преобразованной программе автоматически восстановить вставленные функции невозможно. В рамках данной статьи мы не будем рассматривать подробно прямую вставку функций и её эффект на запутывание и распутывание программ.

Вынос группы операторов (function outlining) [5], п. 6.3.1. Данное преобразование является обратным к предыдущему и хорошо дополняет его. Некоторая группа операторов исходной программы выделяется в отдельную функцию. При необходимости создаются формальные параметры. Преобразование может быть легко обращено компилятором, который (как было сказано выше) может подставлять тела функций в точки их вызова.

Отметим, что выделение операторов в отдельную функцию является сложным для запутывателя преобразованием. Запутыватель должен провести глубокий анализ графа потока управления и потока данных с учётом указателей, чтобы быть уверенным, что преобразование не нарушит работу программы.

Непрозрачные предикаты (opaque predicates) [5], п. 6.1. Основной проблемой при проектировании запутывающих преобразований графа потока управления является то, как сделать их не только дешёвыми, но и устойчивыми. Для обеспечения устойчивости многие преобразования основываются на введении *непрозрачных* переменных и предикатов. Сила таких преобразований зависит от сложности анализа непрозрачных предикатов и переменных.

Переменная v является *непрозрачной*, если существует свойство π относительно этой переменной, которое априори известно в момент запутывания программы, но трудноустанавливаемо после того, как запутывание завершено. Аналогично, предикат P называется *непрозрачным*, если его значение известно в момент запутывания программы, но трудноустанавливаемо после того, как запутывание завершено.

Непрозрачные предикаты могут быть трёх видов: P^F – предикат, который всегда имеет значение «ложь», P^T – предикат, который всегда имеет значение «истина», и $P^?$ – предикат, который может принимать оба значения, и в момент запутывания текущее значение предиката известно.

В работах [3], [7], [23] разработаны методы построения непрозрачных предикатов и переменных, основанные на «встраивании» в программу вычислительно сложных задач, например, задачи *3-выполнимости*[1]. Некоторые возможные способы введения непрозрачных предикатов и непрозрачных выражений вкратце перечислены ниже.

- Использование разных способов доступа к элементы массива [23]. Например, в программе может быть создан массив (скажем, a), который инициализируется заранее известными значениями, далее в программу добавляются несколько переменных (скажем, i}, j), в которых хранятся индексы элементов этого массива. Теперь непрозрачные предикаты могут иметь вид a[i] == a[j]. Если к тому же переменные i} и j в программе изменяются, существующие сейчас методы статического анализа алиасов позволят только определить, что и i, и j могут указывать на любой элемент массива a.
- Использование указателей на специально создаваемые динамические структуры [7]. В этом подходе в программу добавляются операции по созданию ссылочных структур данных (списков, деревьев), и

[1] Задача формулируется следующим образом: дана булевская формула $f(x_1,\ldots,x_n) = C_1 \wedge C_2 \wedge \ldots \wedge C_m$, где $C_i = x_{i_1}^{\sigma_1} \vee x_{i_2}^{\sigma_2} \vee x_{i_3}^{\sigma_3}$, над множеством переменных $X = \{x_1,\ldots,x_n\}$. Определить, существует ли такой набор значений переменных X, при котором значение f равно 1. Известно, что эта задача NP-полна.

добавляются операции над указателями на эти структуры, подобранные таким образом, чтобы сохранялись некоторые инварианты, которые и используются как непрозрачные предикаты.

- Конструирование булевских выражений специального вида [3].
- Построение сложных булевских выражений с помощью эквивалентных преобразований из формулы *true*. В простейшем случае мы можем взять k произвольных булевских переменных $x_1,\ldots,x_k$ и построить из них тождество $(x_1 \vee \overline{x_1}) \wedge \ldots \wedge (x_k \vee \overline{x_k})$. Далее с помощью эквивалентных алгебраических преобразований часть скобок (или все) раскрываются, и в результате получается искомый непрозрачный предикат.
- Использование комбинаторных тождеств, например $\sum_{i=0}^{n} C_n^i = 2^n$.

Внесение недостижимого кода (adding unreachable code). Если в программу внесены непрозрачные предикаты видов P^F или P^T, ветки условия, соответствующие условию «истина» в первом случае и условию «ложь» во втором случае, никогда не будут выполняться. Фрагмент программы, который никогда не выполняется, называется *недостижимым* кодом. Эти ветки могут быть заполнены произвольными вычислениями, которые могут быть похожи на действительно выполняемый код, например, собраны из фрагментов той же самой функции. Поскольку недостижимый код никогда не выполняется, данное преобразование влияет только на размер запутанной программы, но не на скорость её выполнения. Общая задача обнаружения недостижимого кода, как известно, алгоритмически неразрешима. Это значит, что для выявления недостижимого кода должны применяться различные эвристические методы, например, основанные на статистическом анализе программы.

Внесение мёртвого кода (adding dead code) [5], п. 6.2.1. В отличие от недостижимого кода, *мёртвый* код в программе выполняется, но его выполнение никак не влияет на результат работы программы. При внесении мёртвого кода запутыватель должен быть уверен, что вставляемый фрагмент не может влиять на код, который вычисляет значение функции. Это практически значит, что мёртвый код не может иметь побочного эффекта, даже в виде модификации глобальных переменных, не может изменять окружение работающей программы, не может выполнять никаких операций, которые могут вызвать исключение в работе программы.

Внесение избыточного кода (adding redundant code) [5], п. 6.2.6. Избыточный код, в отличие от мёртвого кода выполняется, и результат его выполнения используется в дальнейшем в программе, но такой код можно упростить или совсем удалить, так как вычисляется либо константное значение, либо

значение, уже вычисленное ранее. Для внесения избыточного кода можно использовать алгебраические преобразования выражений исходной программы или введение в программу математических тождеств. Например, можно воспользоваться комбинаторным тождеством $\sum_{i=0}^{8} C_8^i = 2^8 = 256$ и заменить везде в программе использование константы 256 на цикл, который вычисляет сумму биномиальных коэффициентов по приведённой формуле.

Подобные алгебраические преобразования ограничены целыми значениями, так как при выполнении операций с плавающей точкой возникает проблема накопления ошибки вычислений. Например, выражение $\sin^2 x + \cos^2 x$ при вычислении на машине практически никогда не даст в результате значение 1. С другой стороны, при операциях с целыми значениями возникает проблема переполнения. Например, если использование 32-битной целой переменной `x` заменено на выражение `x*p/q`, где `p` и `q` гарантированно имеют одно и то же значение, при выполнении умножения `x*p` может произойти переполнение разрядной сетки, и после деления на `q` получится результат не равный `p`. В качестве частичного решения задачи можно выполнять умножение в 64-битных целых числах.

Преобразование сводимого графа потока управления к несводимому (transforming reducible to non-reducible flow graph) [5], п. 6.2.3. Когда целевой язык (байт-код или машинный язык) более выразителен, чем исходный, можно использовать преобразования, «противоречащие» структуре исходного языка. В результате таких преобразований получаются последовательности инструкций целевого языка, не соответствующие ни одной из конструкций исходного языка.

Например, байт-код виртуальной машины Java содержит инструкцию `goto`, в то время как в языке Java оператор `goto` отсутствует. Графы потока управления программ на языке Java оказываются всегда *сводимыми*, в то время как в байт-коде могут быть представлены и *несводимые* графы.

Можно предложить запутывающее преобразование, которое трансформирует сводимые графы потока управления функций в байт-коде, получаемых в результате компиляции Java-программ, в несводимые графы. Например, такое преобразование может заключаться в трансформации структурного цикла в цикл с множественными заголовками с использованием непрозрачных предикатов.

С одной стороны, декомпилятор может попытаться выполнить обратное преобразование, устраняя несводимые области в графе, дублируя вершины или вводя новые булевские переменные. С другой строны, распутыватель может с помощью статических или статистических методов анализа определить значение непрозрачных предикатов, использованных при запутывании, и

устранить никогда не выполняющиеся переходы. Однако, если догадка о значении предиката окажется неверна, в результате получится неправильная программа.

Устранение библиотечных вызовов (eliminating library calls) [5], п. 6.2.4. Большинство программ на языке Java существенно используют стандартные библиотеки. Поскольку семантика библиотечных функций хорошо известна, такие вызовы могут дать полезную информацию при обратной инженерии программ. Проблема усугубляется ещё и тем, что ссылки на классы библиотеки Java всегда являются именами, и эти имена не могут быть искажены.

Во многих случаях можно обойти это обстоятельство, просто используя в программе собственные версии стандартных библиотек. Такое преобразование не изменит существенно время выполнения программы, зато значительно увеличит её размер и может сделать её непереносимой.

Для программ на традиционных языках эта проблема стоит менее остро, так как стандартные библиотеки, как правило, могут быть скомпонованы статически вместе с самой программой. В данном случае программа не содержит никаких имён функций из стандартной библиотеки.

Переплетение функции (function interleaving) [5], п. 6.3.2. Идея этого запутывающего преобразования в том, что две или более функций объединяются в одну функцию. Списки параметров исходных функций объединяются, и к ним добавляется ещё один параметр, который позволяет определить, какая функция в действительности выполняется.

Клонирование функций (function cloning) [5], п. 6.3.3. При обратной инженерии функций в первую очередь изучается сигнатура функции, а также то, как эта функция используется, в каких местах программы, с какими параметрами и в каком окружении вызывается. Анализ контекста использования функции можно затруднить, если каждый вызов некоторой функции будет выглядеть как вызов какой-то другой, каждый раз новой функции. Может быть создано несколько клонов функции, и к каждому из клонов будет применён разный набор запутывающих преобразований.

Развёртка циклов (loop unrolling) [5], п. 6.3.4. *Развёртка циклов* применяется в оптимизирующих компиляторах для ускорения работы циклов или их распараллеливания. Развёрка циклов заключается в том, что тело цикла размножается два или более раз, условие выхода из цикла и оператор приращения счётчика соответствующим образом модифицируются. Если количество повторений цикла известно в момент компиляции, цикл может быть развёрнут полностью.

Разложение **циклов** (loop fission) [5], п. 6.3.4. *Разложение циклов* состоит в том, что цикл с сложным телом разбивается на несколько отдельных циклов с простыми телами и с тем же пространством итерирования.

Реструктуризация графа потока управления [24]. Структура графа потока управления, наличие в графе потока управления характерных шаблонов для циклов, условных операторов и т. д. даёт ценную информацию при анализе программы. Например, по повторяющимся конструкциям графа потока управления можно легко установить, что над функцией было выполнено преобразование развёртки циклов, а далее можно запустить специальные инструменты, которые проанализируют развёрнутые итерации цикла для выделения индуктивных переменных и свёртки цикла. В качестве меры противодействия может быть применено такое преобразование графа потока управления, которое приводит граф к однородному («плоскому») виду. Операторы передачи управления на следующие за ними базовые блоки, расположенные на концах базовых блоков, заменяются на операторы передачи управления на специально созданный базовый блок *диспетчера*, который по предыдущему базовому блоку и управляющим переменным вычисляет следующий блок и передаёт на него управление. Технически это может быть сделано перенумерованием всех базовых блоков и введением новой переменной, например `state`, которая содержит номер текущего исполняемого базового блока. Запутанная функция вместо операторов `if`, `for` и т.д. будет содержать оператор `switch`, расположенный внутри бесконечного цикла.

Локализация переменных в базовом блоке [3]. Это преобразование локализует использование переменных одним базовым блоком. Для каждого запутываемого базового блока функции создаётся свой набор переменных. Все использования локальных и глобальных переменных в исходном базовом блоке заменяются на использование соответствующих новых переменных. Чтобы обеспечить правильную работу программы между базовыми блоками вставляются так называемые *связующие* (connective) базовые блоки, задача которых скопировать выходные переменные предыдущего базового блока в входные переменные следующего базового блока.

Применение такого запутывающего преобразование приводит к появлению в функции большого числа новых переменных, которые, однако, используются только в одном-двух базовых блоках, что запутывает человека, анализирующего программу.

При реализации этого запутывающего преобразования возникает необходимость точного анализа указателей и контекстно-зависимого межпроцедурного анализа. В противном случае нельзя гарантировать, что запись по какому-либо указателю или вызов функции не модифицируют настоящую переменную, а не текущую рабочую копию.

Расширение области действия переменных [5], п. 7.1.2. Данное преобразование по смыслу обратно предыдущему. Это преобразование пытается увеличить время жизни переменных настолько, насколько можно. Например, вынося блочную переменную на уровень функции или вынося

локальную переменную на статический уровень, расширяется область действия переменной и усложняется анализ программы.

Здесь используется то, что глобальные методы анализа (то есть, методы, работающие над одной функцией в целом) хорошо обрабатывают локальные переменные, но для работы со статическими переменными требуются более сложные методы межпроцедурного анализа.

Для дальнейшего запутывания можно объединить несколько таких статических переменных в одну переменную, если точно известно, что переменные не могут использоваться одновременно. Очевидно, что преобразование может применяться только к функциям, которые никогда не вызывают друг друга непосредственно или через цепочку других вызовов.

2.4. Применение запутывающих преобразований

Существующие методы запутывания и инструменты для запутывания программ используют не единственное запутывающее преобразование, а некоторую их комбинацию. В данном разделе мы рассмотрим некоторые используемые на практике методы запутывания.

В работах Ч. Ванг [23], [24] предлагается метод запутывания, и описывается его реализация в инструменте для запутывания программ на языке Си. Предложенный метод запутывания использует преобразование введения «диспетчера» в запутываемую функцию. Номер следующего базового блока вычисляется непосредственно в самом выполняющемся базовом блоке прямым присваиванием переменной, которая хранит номер текущего базового блока. Для того чтобы затруднить статический анализ, номера базовых блоков помещаются в массив, каждый элемент которого индексируется несколькими разными способами. Таким образом, для статического прослеживания порядка выполнения базовых блоков необходимо провести анализ указателей.

В работе [3] предлагается метод запутывания, основанный на следущих запутывающих преобразованиях: каждый базовый блок запутываемой функции разбивается на более мелкие части (т.н. piece) и клонируется один или несколько раз. В каждом фрагменте базового блока переменные локализуются, и для связывания базовых блоков создаются специальные связующие базовые блоки. Далее в каждый фрагмент вводится мёртвый код. Источником мёртвого кода может быть, например, фрагмент другого базового блока той же самой функции или фрагмент базового блока другой функции. Поскольку каждый фрагмент использует свой набор переменных, объединяться они могут безболезненно (при условии отсутствия в программе указателей и вызовов функций с побочным эффектом). Далее из таких комбинированных фрагментов собирается новая функция, в которой для переключения между базовыми блоками используется диспетчер. Диспетчер принимает в качестве параметров номер предыдущего базового блока и набор булевских переменных, которые используются в базовых блоках для вычисления условий перехода, и вычисляет номер следующего блока. При этом следующий блок

может выбираться из нескольких эквивалентных блоков, полученных в результате клонирования. Выражая функцию перехода в виде булевской формулы, можно добиться того, что задача статического анализа диспетчера будет PSPACE-полна. Работа [3] описывает алгоритм запутывания, но не указывает, реализован ли этот алгоритм в какой-либо системе.

Запутыватели для языка Java, например Zelix KlassMaster [25], как правило, используют следующее сочетание преобразований: из `.class`-файлов удаляется вся отладочная информация, включая имена локальных переменных; классы и методы переименовываются в короткие и семантически неосмысленные имена; граф потока управления запутываемой функции преобразовывается к несводимому графу, чтобы затруднить декомпиляцию обратно в язык Java.

Побочным эффектом такого запутывания является существенное ускорение работы программы. Во-первых, удаление отладочной информации делает `.class`-файлы существенно меньше, и соответственно их загрузка (особенно по медленным каналам связи) ускоряется. Во-вторых, резкое уменьшение длины имён методов приводит к тому, что ускоряется поиск метода по имени, выполняемый каждый раз при вызове. Как следствие, запутывание Java-программ часто рассматривается как один из способов их оптимизации.

3. Методы анализа программ

В данном разделе мы рассмотрим методы, которые применяются при анализе программ в компиляторах. Цель таких методов – выявление зависимостей между компонентами программы, что даёт возможность применить определённые оптимизационные преобразования, или накладывает ограничения на проводимые оптимизационные преобразования.

Методы анализа программ могут быть разделены на 4 группы:

- **Синтаксические.** К этой группе относятся методы, основанные только на результатах лексического, синтаксического и семантического анализа программы.
- **Статические**. К этой группе относятся методы анализа потоков управления и данных и методы, основанные на результатах анализа потоков управления и данных. Статические методы анализа работают с программой, не используя информацию о работе программы на конкретных начальных данных.
- **Динамические**. Динамические методы анализа программ используют информацию, полученную в результате «наблюдения» за работой программы на конкретных входных данных. Заметим, что сами по себе динамические методы редко применяются для анализа программ, поскольку, как правило, необходима информация о поведении программы

на разных наборах входных данных, которая собирается с помощью статистических методов анализа.

- **Статистические**. Статистические методы используют информацию, собранную в результате значительного количества запусков программы на большом количестве наборов входных данных.

Краткая характеристика важнейших для нас методов анализа программ приведена ниже.

3.1. Методы статического анализа

Статический анализ алиасов (alias analysis) [11] необходим в языках, в которых несколько имён могут быть использованы для доступа к одной и той же области памяти. Например, некоторый элемент массива `a` может быть адресован в программе как `a[0]` (каноническое имя элемента массива), как `a[j]` или вообще как `b[-4]`. В результате анализа алиасов каждому оператору, выполняющему косвенную запись в память или косвенное чтение из памяти, ставится в соответствие множество имён переменных, которые могут затрагиваться данной операцией.

Если язык допускает алиасы, проведение в той или иной мере анализа указателей необходимо для корректного анализа потоков данных и для преобразования программ. В случае доступа к элементам массивов и полям структур мы можем в простейшем случае предполагать, что считывается или модифицируется сразу весь массив или вся структура. Для указателей или ссылок в простейшем случае («консервативный» анализ) мы можем исходить из предположения о том, что косвенное чтение из памяти затрагивает все локальные и глобальные переменные, а косвенная запись в память может все их модифицировать. Такая схема слишком груба и в действительности блокирует глубокую трансформацию практически любой программы.

Известно, что общая задача точного анализа указателей как минимум NP-трудна. В настоящее время существуют методы, работающие за полиномиальное время, для указателей на локальные переменные в случае нерекурсивных функций.

Анализ указателей не может быть непосредственно использован для запутывания или распутывания программы, но он является ключевым для точного анализа свойств программы.

Статическое устранение мёртвого кода (dead-code elimination) [19], разд. 18.10. имеет целью выявить в программе код, который выполняется, но не оказывает влияние на результат работы программы.

Статическая минимизация количества переменных (variable minimization) [19], разд. 16.3. имеет целью уменьшить количество используемых в функции локальных переменных за счёт объединения переменных, времена жизни значений в которых не пересекаются, в одну переменную. Стандартная техника, которая используется для минимизации количества переменных,

состоит в построении *графа перекрытия* переменных с помощью итерационного решения уравнения потока данных и последующей раскраске вершин этого графа в минимальное или близкое к минимальному количество цветов.

Статическое продвижение констант и копий (constant and copy propagation) [19], разд. 12.5, 12.6. заключается в продвижении константных выражений как можно дальше по тексту функции. Если выражение использует только значения переменных, которые в данной точке программы заведомо содержат одно известное при анализе программы значение, такое выражение может быть вычислено на этапе анализа программы. Если в выражении используется переменная, которая в данной точке программы заведомо является копией какой-то другой переменной, в выражение может быть подставлена исходная переменная.

Статический анализ доменов (domain analysis) является расширением алгоритма продвижения констант. Он позволяет определить множество значений, которые может принимать данная переменная в данной точке программы, если это множество не велико.

Статический слайсинг (slicing) [21] – это построение «сокращённой» программы, из которой удалён весь код, не влияющий на вычисление заданной переменной в заданной точке (обратный слайс), но при этом программа остаётся синтаксически и семантически корректной и может быть выполнена. Кроме описанного выше *обратного* слайсинга разработаны алгоритмы *прямого* слайсинга. Прямой слайсинг оставляет в программе только те операторы, которые зависят от значения переменной, вычисленного в данной точке программы. Методы слайсинга могут быть полезны при разделении «переплетённых» вычислений, когда одновременно вычисляются две независимые друг от друга величины. Например, в одном цикле может вычисляться скалярное произведение двух векторов, а также минимальный и максимальный элемент каждого вектора, и такие циклы могут быть расщеплены с помощью построения слайсов.

3.2. Методы динамического и статистического анализа

Статистический анализ покрытия базовых блоков программы позволяет установить, выполнялся ли когда-либо при выполнении программы на заданном множестве наборов входных данных заданный базовый блок.

Статистическое сравнение трасс позволяет выявить, одинаковы ли трассы программы, полученные при разных запусках на одном и том же наборе входных данных.

Статистическое построение графа потока управления строит граф потока управления на основании информации о порядке следования базовых блоков на одном наборе или на множестве наборов входных данных.

Динамическое продвижение копий вдоль трасс необходимо для точного межпроцедурного анализа зависимостей по данным на основе трассы выполнения программы. Поскольку трасса выполнения программы, по сути, является одним большим базовым блоком, продвижение копий – несложная задача.

Динамическое выделение мёртвого кода позволяет выявить инструкции программы, которые выполнялись при данном запуске программы, но не оказали никакого влияния на результат работы программы. Если анализируется совокупность запусков программы на множестве наборов входных данных, можно говорить о статистическом выделении мёртвого кода.

Динамический слайсинг оставляет в трассе программы только те инструкции, которые повлияли на вычисление данного значения в данной точке программы (прямой динамический слайсинг), или только те инструкции, на которые повлияло присваивание значения данной переменной в данной точке программы.

Заметим, что о точности динамических методов анализа можно говорить, только если известно полное тестовое покрытие программы (построение полного тестового покрытия – алгоритмически неразрешимая задача). В противном случае статистическое выявление свойств программы не позволяет нам утверждать, что данное свойство справедливо на всех допустимых наборах входных данных. Например, условие `if (leap_year(current_data))` всегда будет равно значению «истина», если текущий год високосный, и значению «ложь» в противном случае, однако удаление этого ператора из программы приведёт к её неправильной работе.

Поэтому описанные выше динамические методы не могут применяться в автоматическом инструменте анализа программ. Роль этих методов в том, чтобы привлечь внимание пользователя инструмента анализа программ к особенностям работы программы. В дальнейшем пользователь может изучить «подозрительный» фрагмент кода более детально с применением других инструментов, чтобы подтвердить или опровергнуть выдвинутую гипотезу.

Если непрозрачные предикаты и недостижимый код устраняются только на основании статистического анализа, всегда остаётся возможность, что предикат был существенным (как в примере выше). Чтобы всё же упростить программу, можно, например, вынести предположительно недостижимый код из общего графа потока управления функции в обработчик специального исключения, которое возбуждается каждый раз, когда предикат примет значение, отличное от обычного. С одной стороны, граф потока управления и потока данных основной программы в результате упростится, а с другой стороны, программа сохранит свою функциональность.

4. Анализ запутанных программ

В данном разделе мы сопоставим методы запутывания, описанные в разделе **2**, и методы анализа программ, рассмотренные в разделе **3**. На основании этого сопоставления вводится новая мера устойчивости запутывающих преобразований, а именно:

Метод запутывания	**Метод распутывания**
Искажение имён переменных	Переименование переменных
Использование специфических языковых конструкций	Упрощение специфических языковых конструкций
Развёртка цикла	Визуализация графа потока управления функции для выделения потенциальных кандидатов на свертку в цикл; выявление индуктивной переменной, что требует (интерактивного) сравнения базовых блоков и (интерактивных) эквивалентных преобразований выражений, причём при таких преобразованиях выражение может даже (незначительно) усложняться; свёртка цикла
Использование уникальных переменных в базовых блоках	Продвижение копий (статическое и статистическое), минимизация количества используемых переменных
Введение детерминированного диспетчера	Статистическое восстановление графа потока управления
Введение недетерминированного диспетчера	Сравнение трасс, полученных на одном и том же наборе входных данных
Переплетение кода нескольких базовых блоков в один запутанный базовый блок	Статистическое устранение мёртвого кода
Введение непрозрачных предикатов	Статистический анализ покрытия для выявления потенциальных непрозрачных предикатов, поиск по образцам известных непрозрачных предикатов, алгебраическое упрощение, доказательство теорем
все методы	Свёртка констант, продвижение констант, продвижение копий, статическое устранение мёртвого кода – могут выполняться после каждого шага распутывающих преобразований

Таблица 1. *Методы запутывания программ и методы, которые могут применятся для их анализа.*

запутывающее преобразование называется *устойчивым относительно некоторого класса методов* анализа программ, если методы этого класса не позволяют надёжно раскрыть данное запутывающее преобразование Перечисление некоторых методов запутывания программ с точки зрения методов их возможного распутывания дано в таблице 1.

Преобразование	**Сложность распутывания (необходимый тип анализа)**	**Автоматизируемость (тип распутывателя)**
Удаление комментариев	Одностороннее преобразование	
Переформатирование программы	Синтаксический	автомат.
Удаление отладочной информации	Одностороннее преобразование	
Изменение имён идентификаторов	Синтаксический	полуавтомат.
Языково-специфические преобразования	Синтаксический	автомат.
Открытая вставка функций	Одностороннее преобразование	поддерж.
Вынос группы операторов	Синтаксический	автомат.
Непрозрачные предикаты и выражения	Синтаксический – статистический (зависит от вида предиката)	автомат. – поддерж.
Внесение недостижимого кода	Зависит от стойкости непрозрачных предикатов	автомат., полуавтомат.
Внесение мёртвого кода	Синтаксический – статистический	автомат., полуавтомат.
Внесение избыточного кода	Синтаксический – статистический	автомат. – поддерж.
Внесение несводимости в граф	Статический, но зависит от стойкости непрозрачных предикатов	автомат., полуавтомат.
Устранение библиотечных вызовов	Одностороннее преобразование	поддерж.
Переплетение функций	Статический – статистический	автомат. – поддерж.
Клонирование функций или базовых блоков	Статический	автомат. – поддерж.
Развёртка циклов	Одностороннее преобразование	поддерж.
Разложение циклов	Статический	автомат. – поддерж.
Введение диспетчера	Статический – статистический	автомат., полуавтомат.
Локализация переменных в базовом блоке	Статический – статистический	автомат., полуавтомат.
Расширение области действия переменных	Статический – статистический	автомат., полуавтомат.

Таблица 2. *Классификация запутывающих преобразований*

В таблице 2 приведена классификация методов запутывания по отношению к требуемым методам анализа программ. В третьем столбце таблицы указана степень, в которой возможно автоматическое распутывание. Степень автоматизма оценивается по следующей шкале: *автоматический* – поиск в программе запутанных фрагментов и их распутывание возможны полностью автоматически; *полуавтоматический* – поиск в программе подозрительных фрагментов и их распутывание по отдельности выполняются автоматически, но пользователь должен подтвердить применение распутывающего преобразования; *поддерживаемый* – поиск в программе запутанных фрагментов и применение распутывающих преобразований требуют существенного участия человека, но процесс может быть поддержан специальными инструментальными средствами; *неавтоматизируемый* – автоматизация выполнения распутывающего преобразования принципиально затруднена.

Для некоторых видов запутывающих преобразований требуемые инструменты (синтаксические, статические, статистические) зависят от того, каким способом было реализовано преобразование. Например, непрозрачные предикаты могут быть самого разного вида, от простейших `if(0)`, до очень сложных. Для анализа и устранения простейших непрозрачных предикатов достаточно инструментов уровня синтаксического анализа, которые работают автоматически, а для устранения сложных непрозрачных предикатов требуется статистический анализ, либо сложные инструменты, такие как полуавтоматический доказатель теорем. Поэтому некоторые ячейки таблицы содержат несколько необходимых видов анализа или несколько оценок автоматизируемости.

5. Практическое использование

В данном разделе мы приведём примеры использования арсенала анализирующих преобразований для анализа запутанных программ.

5.1. Устранение диспетчера на основе динамического анализа

Рассмотрим в качестве простейшего примера запутывание функции `fib`, которая принимает один параметр *n* и вычисляет *n*-е число Фибоначчи. Преобразуем её граф потока управления введением диспетчера. Текст исходной функции и текст функции с введённым диспетчером приведены на рис. 1.

```
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;
}
```

```
static int transtable_fib[12] =
{
  1, 1, 2, 3, 0, 0, 4, 5, 3, 3, 0, 0
};
int fib(int L3)
{
  int _r1, L29, L30, L5, L6, L7;

  L29 = 0; L30 = 0;
  L35: L29 = transtable_fib[L29 * 2 +
L30];
  switch (L29) {
  case 1: goto L26; case 2: goto L27;
  case 3: goto L22; case 4: goto L28;
  case 5: goto L23;
  }
 L47: goto L9;
 L26: L5 = 1; L6 = 1; L30 = (L3 > 1);
      goto L35;
 L27: _r1 = 1; goto L35;
 L22: L30 = (L3 <= 1); goto L35;
 L28: L7 = (L5 + L6); L5 = L6; L6 =
L7;
      L3--; goto L35;
 L23: _r1 = L7; goto L35;
 L9:  return _r1;
}
```

Рис. 1. *Исходный текст функции* `fib` *и её запутанный вариант*

Для анализа запутанной программы были проделаны следующие действия:

- Запутанная программа была проинструментирована для сбора трасс. В начало каждого базового блока был добавлен вызов специальной функции, которая записывала в файл трассы номер базового блока.
- По собранным трассам был построен граф потока управления, вид которого совпадал с графом потока управления запутанной функции, поскольку собранные трассы обеспечивали полное покрытие.
- Поскольку граф потока управления, построенный по трассам, имел характерную регулярную структуру, указывающую на использование диспетчера, блок диспетчера был в трассах проигнорирован, что позволило вскрыть изначальный порядок следования базовых блоков в функции.

Сравнение графа потока управления исходной программы и графа потока управления, полученного в результате анализа трасс, приведено на рис. 3.

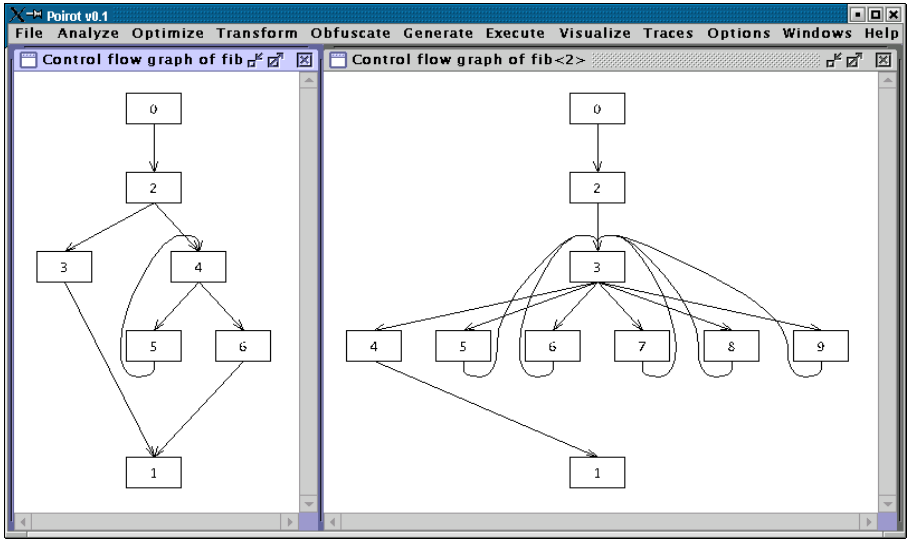

Рис. 2. *Граф потока управления исходной и запутанной функции* `fib`

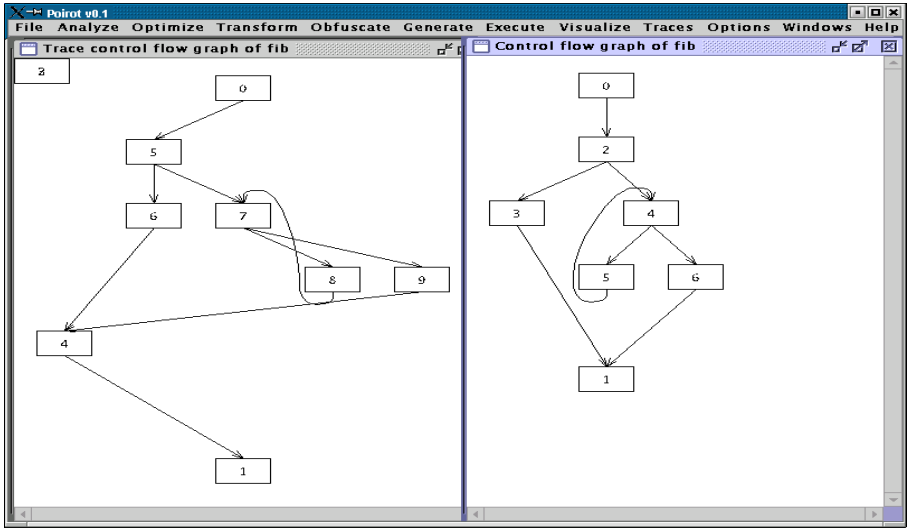

Рис. 3. *Исходный граф потока управления функции и восстановленный по трассам граф потока управления функции.*

5.2. Анализ специфических языковых конструкций и алгебраические преобразования

В качестве второго примера мы выбрали запутанную программу, которая существенно использует специфические языковые конструкции. Эта программа – победитель 1998 года *международного конкурса запутанного кода на Си* [12]. В конкурсе участвуют программы на языке Си, запутанные вручную. Победители выявляются в таких номинациях, как «самый предательский код на Си», «лучшее анти-использование ANSI C».

```
#include<stdio.h>
#include<string.h>
main()
{
  char*O,l[999]="'`acgo\177~|xp .-\0R^8)NJ6%K4O+A2M(*0ID57$3G1FBL";
  while(O=fgets(l+45,954,stdin)){
    *l=O[strlen(O)[O-1]=0,strspn(O,l+11)];
    while(*O)switch((*l&&isalnum(*O))-!*l){
      case-1:{char*I=(O+=strspn(O,l+12)+1)-2,O=34;
      while(*I&3&&(O=(O-16<<1)+*I---'-')<80);
      putchar(O&93?*I&8||!(I=memchr(l,O,44))?'?':I-l+47:32);
      break; case 1:;}*l=(*O&31)[l-15+(*O>61)*32];
      while(putchar(45+*l%2),(*l=*l+32>>1)>35);
    case 0:  putchar((++O,32));}
    putchar(10);}}
```

Рис. 4. *Конвертер ASCII/код Морзе (автор Frans van Dorsselaer).*

Программа выполняет перекодирование символов, считываемых со стандартного потока, из кода Морзе и обратно. Она имеет запутанную структуру графа потока управления, её внутренние таблицы закодированы. Для распутывания программы были проделаны следующие шаги:

1. Программа была оттранслирована во внутреннее представление.
2. Умышленно трудновоспринимаемые идентификаторы (I, l) были переименованы в более простые a, b, c и т. д.
3. Над программой во внутреннем представлении были выполнены некоторые эквивалентные алгебраические преобразования, например (a-1)[strlen(a)] было заменено на a[strlen(a)-1].
4. Анализ алиасов позволил установить, что только элемент a[0] изменяется в процессе работы программы, следовательно в вызовах функций strspn, memchr вместо массива a могут использоваться строковые литералы.
5. Анализ доменов позволил определить, что выражения, записанные в условном операторе, могут принимать всего два или три значения. Решая Диофантовы уравнения, можно значительно упростить условия.
6. В анализируемой программе далее были проведены анализ избыточных выражений (partial redundancy elimination), анализ нулевых выражений

(zero-value analysis). Все это позволило разделить один внутренний цикл на два независимых цикла.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
int main()
{
  signed char *_r53, _r112, _r65, *a, d[999], *c, b;

  memcpy(d,
         "'`acgo\177~|xp .-\0R^8)NJ6%K4O+A2M(*0ID57$3G1FBL",
         45);
  while ((a = fgets(d+45,954,stdin))) {
    a[strlen(a) - 1] = 0;
    *d = a[strspn(a, " .-")];
    if (!*d) {
      while (*a) {
        a = a + strspn(a, ".-") + 1;
        c = a - 2;
        b = 34;
        while (1) {
          if ((*c & 3) == 0) break;
          _r53 = c;
          c--;
          b = (b - 16 << 1) + *_r53 - 45;
          if (b >= 80) break;
        }

        if ((b & 93) == 0) {
          _r65 = 32;
        } else {
          if ((*c & 8) != 0) {
            _r65 = '?';
          } else {
            c = memchr(d,b,44);
            if (c != 0) _r65 = (c - d) + 47;
            else _r65 = '?';
          }
        }
        putchar(_r65);
      }
    } else {
      while (*a) {
        if (isalnum(*a)) {
          *d = d[(*a & 31) - 15 + (*a > 61) * 32];
          do {
            putchar(45 + *d % 2);
```

```
            _r112 = *d + 32 >> 1;
            *d = _r112;
          } while (_r112 > 35);
        }
        a++;
        putchar(32);
      }
    }
    putchar(10);
  }
  return 0;
}
```

Рис. 5. *Конвертер ASCII/MORSE после преобразований.*

7 Программа из внутреннего представления была переведена обратно в Си, при этом был проведён структурный анализ графа потока управления для выявления циклов и представления их в высокоуровневой форме.

На рис. 5 приведён текст программы после всех описанных преобразований. Несмотря на то, что он стал больше, чем текст исходной программы, его структура стала намного проще и понятнее. После этого не составляет труда установить правила перекодирования кода Морзе в код ASCII и обратно.

```
Short up_[16], down_[16], x[8], row[8];

int queens(int R)
{
  int i, k1, k2;
  for (i=0;i<8;i++) {
    if (row[i]!=0) {
      if ((i + R) % 2 == 0) {
          k1 = 6+i-R;
          k2 = R+i;
          if (up_[k1+1] != 0) {
            if (down_[k2] != 0) {
              up_[k1 + 1] = 0;
              down_[k2] = 0;
              row[i]=0;
              x[R]=i;
              if (R==7)
                print();
              else
                queens(R+1);
              row[i]=1;
              up_[k1 + 1] = 1;
              down_[k2] = 1;
            } /* if (down_) */
          } /* if (up_) */
      } else {
          k1 = 7+i-R;
```



```
          k2 = R+i-1;
          if (up_[k1] != 0) {
            if (down_[k2+1] != 0) {
              up_[k1] = 0;
              down_[k2 + 1] = 0;
              row[i]=0;
              x[R]=i;
              if (R==7)
                print();
              else
                queens(R+1);
              row[i]=1;
              up_[k1] = 1;
              down_[k2 + 1] = 1;
            } /* if (down_) */
          } /* if (up_) */
      }
    }
  }
  return 0;
}
```

Рис. 6. *Распутанная программа решения задачи «8 ферзей»*

5.3. Устранение мёртвого кода и свёртка циклов

В качестве третьего примера мы опишем анализ программы, запутанной одним из коммерческих запутывателей программ на Си.

Распутывание программы состояло из следующих шагов.

- Был построен граф потока управления запутанной программы. Вид графа показал, что запутанная программа имеет две ветки условного оператора, не отличающиеся структурой потока управления друг от друга. Каждая из веток состояла из повторяющегося 8 раз фрагмента, что позволило предположить, что при запутывании была выполнена развёртка цикла.
- Поскольку запутанная программа была представлена на языке Си, и содержала некоторые конструкции, использованные специально для увеличения размера и затруднения восприятия программы, все избыточные конструкции (ключевое слово register, ключевое слово signed в типе signed int, ненужные приведения типов и ненужные скобки) были удалены. В результате размер программы сократился на 20 Кб.

- Далее было проведено статическое устранение мёртвого кода, которое позволило уменьшить размер программы до 64 Кб.
- Далее была выполнена минимизация количества локальных переменных. Запутанная программа определяла в начале функции около 800 локальных переменных, время жизни которых в функции было невелико. Был применён алгоритм минимизации количества переменных, который выявил всего 11 необходимых переменных. Лишние определения переменных были удалены. Новые переменные получили имена вида r_1, r_2 вместо использовавшихся в запутанной программе имён вида r_dddddd, где d – десятичная цифра. В результате всех преобразований размер программы сократился до 26 Кб.
- Далее в программы были выполнены преобразования по свёртке цикла. Были идентифицированы точные границы итерации цикла; все тела итерации цикла были сопоставлены друг с другом, в результате определились места, зависящие от номера итерации; выражения, зависящие от номера итерации, были переписаны в виде, позволившем ввести переменную цикла. Данные шаги были проведены полуавтоматически с использованием простейших средств сравнения базовых блоков (на текстуальном уровне). Это преобразование привело к тому, что размер программы достиг 3.5 Кб.
- На заключительном шаге, применив эквивалентные алгебраические преобразования и выявив принципы кодирования данных, программа была ещё более упрощена. Размер программы составил примерно 1.5 Кб, структура потока управления и структура данных, с которыми манипулировала программа, стали полностью очевидны. Текст распутанной программы приведен на рис. 6.

6. Заключение

В данной работе мы рассмотрели запутывание программ с точки зрения устойчивости запутывающих преобразований к различным методам статического и динамического анализа программ. Мы сделали попытку определить, какие методы анализа программ могут использоваться для противодействия большинству из описанных методов запутывания программ. Для иллюстрации нашего подхода мы привели примеры распутывания программ, запутанных как вручную, так и с помощью специальных запутывателей.

Поскольку теория запутывания программ сейчас находится в стадии активного формирования, кроме того, разрабатываются всё новые и новые эмпирические методы запутывания, существует потребность в среде как наборе инструментов, которая выступает в роли «испытательного стенда» для

проверки как теоретических положений, так и новых методов запутывания. Для исследования различных вопросов, связанных с запутыванием программ, нами разрабатывается интегрированная среда (ИС) для изучения запутывания программ Poirot [1].

Цель разработки – получить удобную и мощную ИС для исследования методов и алгоритмов преобразования программ. С её помощью можно быстро получить прототип нового запутывающего преобразования программ, проверить его устойчивость против разнообразных известных методов анализа, либо проверять устойчивость уже существующих алгоритмов запутывания программ. ИС Poirot уже содержит многие инструменты синтаксического, статического, динамического и статического анализа, с помощью которых можно анализировать методы запутывания программ и сами запутанные программы. Заметим, что такая ИС полезна и потому, что реализация методов анализа программ, описанных выше, намного более трудоёмка, чем их использование, а польза этих методов несомненна.

Как было уже упомянуто выше, методы запутывания должны учитывать свойства человеческой психики и пытаться ставить в тупик человека, который управляет системой анализа программ. Можно отметить следующие свойства, которым должна удовлетворять запутанная программа:

- Запутывание должно быть замаскированным. То, что к программе были применены запутывающие преобразования, не должно бросаться в глаза.
- Запутывание не должно быть регулярным. Регулярная структура запутанной программы или её фрагмента позволяет человеку отделить запутанные части и даже идентифицировать алгоритм запутывания.
- Применение стандартных синтаксических и статических методов анализа программ на начальном этапе её анализа не должно давать существенных результатов, так как быстрый результат может воодушевлять человека, а его отсутствие, наоборот, угнетать.

Нам не известно ни одного метода запутывания, который бы удовлетворял всем этим признакам. Разработка таких методов является одним из направлений дальнейших исследований.

Другим направлением является разработка методов запутывания и методов анализа запутанных программ на уровне языка ассемблера (машинного кода). Хотя методы запутывания низкого уровня могут оказаться непереносимыми на другие архитектуры, в низкоуровневом запутывании скрыт большой потенциал.

Литература

1. А. В. Чернов. Интегрированная среда для исследования «обфускации» программ. Доклад на конференции, посвящённой 90-летию со дня рождения А.~А.~Ляпунова. Россия, Новосибирск, 8---11 октября 2001 года. http://res.tsu.ru/Lyapunov-2001/abstracts/2350.htm
2. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang. On the (Im)possibility of Obfuscating Programs. LNCS, 2001, **2139**, pp. 1–18.
3. S. Chow, Y. Gu, H. Johnson, V. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. LNCS, 2001, **2200**, pp. 144–155.
4. C. Cifuentes, K. J. Gough. Decompilation of Binary Programs. Technical report FIT-TR-1994-03. Queensland University of Technology, 1994. http://www.fit.qut.edu.au/TR/techreports/FIT-TR-94-03.ps
5. C. Collberg, C. Thomborson, D. Low. A Taxonomy of Obfuscating Transformations. Department of Computer Science, The University of Auckland, 1997. http://www.cs.arizona.edu/~collberg/Research/Publications/`CollbergThomborsonLow97a`
6. C. Collberg, C. Thomborson, D. Low. Breaking Abstractions and Unstructuring Data Structures. In *IEEE International Conference on Computer Languages*, ICCL'98, Chicago, IL, May 1998.
7. C. Collberg, C. Thomborson, D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Principles of Programming Languages 1998*, POPL'98, San Diego, CA, January 1998.
8. C. Collberg, C. Thomborson. On the Limits of Software Watermarking. Technical Report #164. Department of Computer Science, The University of Auckland, 1998. http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborson98e
9. C. Collberg, C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection. Technical Report 2000–03. Department of Computer Science, University of Arizona, 2000. http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborson2000a
10. G. Hachez, C. Vasserot. State of the Art in Software Protection. Project FILIGRANE (Flexible IPR for Software Agent Reliance) deliverable/V2. http://www.dice.ucl.ac.be/crypto/filigrane/External/d21.pdf
11. M. Hind, A. Pioli. Which pointer analysis should I use? In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 113–123, August 2000.
12. The International Obfuscated C Code Contest. http://www.ioccc.org
13. M. Jalali, G. Hachez, C. Vasserot. FILIGRANE (Flexible IPR for Software Agent Reliance). A security framework for trading of mobile code in Internet. In *Autonomous Agent 2000 Workshop: Agents in Industry*, 2000.
14. H. Lai. A comparative survey of Java obfuscators available on the Internet. http://www.cs.auckland.ac.nz/~cthombor/Students/hlai
15. D. Low. Java Control Flow Obfuscation. MSc Thesis. University of Auckland, 1998. http://www.cs.arizona.edu/~collberg/Research/Students/DouglasLow/thesis.ps
16. J. MacDonald. On Program Security and Obfuscation. 1998. http://www.xcf.berkeley.edu/~jmacd/cs261.pdf
17. M. Mambo, T. Murayama, E. Okamoto. A Tentative Approach to Constructing Tamper-Resistant Software. In *ACM New Security Paradigms Workshop*, Langdale, Cumbria UK, 1998.
18. A. von Mayrhauser, A. M. Vans. Program Understanding: Models and Experiments. In *M. Yovits, M. Zelkowitz (eds.), Advances in Computers*, Vol. 40, 1995. San Diego: Academic Press, pp. 1–38.
19. S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997.
20. SourceAgain Java decompiler. http://www.ahpah.com
21. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3): 121–189, September 1995.

22. E. Walle. Methodology and Applications of Program Code Obfuscation. Faculty of Computer and Electrical Engineering, University of Waterloo, 2001. http://walle.dyndns.org/morass/misc/wtr3b.doc
23. C. Wang. A Security Architecture for Survivability Mechanisms. PhD Thesis. Department of Computer Science, University of Virginia, 2000. http://www.cs.virginia.edu/~survive/pub/wangthesis.pdf
24. C. Wang, J. Davidson, J. Hill, J. Knight. Protection of Software-based Survivability Mechanisms. Department of Computer Science, University of Virginia, 2001. http://www.cs.virginia.edu/~jck/publications/dsn_distribute.pdf
25. Zelix KlassMaster Java Code Obfuscator and Obfuscation. http://www.zelix.com