Описание запутывающих преобразований

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:

   a) (эквивалентность преобразования запутывания). Для любой $p \in P$ и $p' \in o(p)$
   $f_p = f_{p'}$,
   $|p'| = poly(|p|)$,
   $\forall x \in Dom_{f_o} \quad time_{p'}(x) = poly(time_p(x))$
   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*.
   b) (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.

1. 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. 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.

1.1. 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.

1.2. 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* . 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 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 $x_1 ... x_k$ 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. 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.

Заключение

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.

Составлено на основании статьи: **Анализ запутывающих преобразований программ** Чернов А. В., Труды Института Системного программирования РАН http://citforum.ru/security/articles/analysis/

Список литературы

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://www.ict.nsc.ru/ws/Lyap2001/2350/
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