Минобрнауки России
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра информационных и управляющих систем

Работа допущена к защите
Зав. кафедрой ИУС
_____И.Г. Черноруцкий
«___» _____ г.

ДИССЕРТАЦИЯ НА СОИСКАНИЕ СТЕПЕНИ МАГИСТРА

Тема: **Метод защиты Java приложений от статического анализа байт-кода**

Направление: 09.04.01 - Информатика и вычислительная техника.
Магистерская программа: 09.04.01_05 - Технология разработки программных систем.

Выполнил студент гр. 63504/10 А.С.Жогов
Руководитель к.т.н., доцент Н.В.Воинов

Санкт-Петербург

2015

Минобрнауки России
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра информационных и управляющих систем

У Т В Е Р Ж Д А Ю

«____»_____ г.

Зав. кафедрой _____

З А Д А Н И Е

к диссертации на соискание степени магистра

студенту группы 63504/10 Жогову Алексею Сергеевичу

1. Тема проекта (работы)

Метод защиты Java приложений от статического анализа байт-кода

(утверждена распоряжением по факультету от

_____ №_____)

2. Срок сдачи студентом оконченного проекта (работы)

_____

3. Исходные данные к проекту (работе)

Спецификация языка Java
Спецификация байт-кода виртуальной машины Java
Спецификация байт-кода виртуальной машины Dalvik
Пакет прикладных программ JBCO
Учебники и публикации по темам «защита Java программ» и «обфускация».

4. Содержание расчетно-пояснительной записки (перечень подлежащих разработке вопросов)

Анализ методов защиты
Анализ запутывающих преобразований
Формулировка требований к разрабатываемому методу
Разработка метода
Описание техник статического анализа, на защиту от которых направлен метод
Описание метода
Практическая реализация метода
Анализ эффективности метода
Описание используемых метрик
Оценка эффективности метода
Анализ полученных результатов

Реферат

С. 78. Рис. 16. Табл. 3.

обфускация, запутывание, Java, байт-код, декомпиляция, статический анализ

Целью работы является создание метода защиты Java приложений от статического анализа исполняемого байт-кода.

В ходе работы были проанализированы основные подходы к решению задачи защиты от статического анализа. На основе собранных данных были сформированы требования к разрабатываемому методу и выбран набор метрик для оценки его эффективности. Был создан новый подход, особенно эффективный для Java приложений, работающих под управлением операционной системы Android. Этот подход был скомбинирован с уже существующими подходами в новый метод защиты. Для метода приведены ограничения на его применение, возможные проблемы, которые могут возникнуть при его использовании, и оценена его эффективность.

Abstract

Pages of text 78, fig. 16, tables 3.

obfuscation, Java, bytecode, decompilation, static analysis

The aim of the work is to provide a protection method from the static analysis of the executable bytecode of Java applications.

During the work general approaches were analyzed. On the basis of the collected data have been formed requirements for developed method and selected set of metrics to evaluate its effectiveness. Established a new transformation approach that is especially effective for Java applications running under the Android operating system. This transformation was combined with the already existing transformations in a new method of protection. For the method were set limits on its use, the possible problems that may arise during its use, and evaluated its effectiveness.

Оглавление

Список иллюстраций

Список таблиц

Определения и сокращения

АС	Автоматизированная система
ОС	Операционная система
ПО	Программное обеспечение
API	Application programming interface
DVM	Dalvik Virtual Machine
JVM	Java Virtual Machine
SDK	Software Development Kit

Введение

В современных реалиях эффективное управление различными системами и организациями немыслимо без их использования информационных технологий. Существует множество компаний, производящих программы, которые могут быть использованы в различных сферах науки и производства. На разработку таких программ зачастую тратится большое количество времени и средств, что обосновывает желание производителей защитить свои продукты от нелицензионного распространения.

Проблеме защиты программного обеспечения от компьютерных пиратов и недобросовестных пользователей уже не один десяток лет. Пиратство наносит большой ущерб индустрии разработки программного обеспечения. Проблему усугубляет быстрое развитие мультимедиа и Интернет технологий, так как количество путей доставки нелицензионного контента и удобство его получения через Интернет растёт с каждым днём.

Разработка наиболее эффективного метода защиты для того или иного программного продукта становится одной из важных задач большинства программистов, которые занимаются разработкой платного программного обеспечения, так как это позволяет им продавать свой интеллектуальный труд, и исключить возможности его нелегального

использования среди потребителей, говоря иными словами, пользователь не сможет использовать оригинальную, лицензионную копию определенной программы предварительно не купив её у издателя или разработчика. Таким образом, затраты производителей на создание эффективного метода защиты их программных продуктов окупаются и компенсируют потенциальный ущерб, наносимый нелегальным копированием и использованием программ. [1]

Особенно уязвимы приложения для платформы Java, так как в их случае компилятор не создает конечный машинный код, а всего лишь его платформенно независимое представление — байт-код. Полученный байт-код содержит очень много осмысленной информации, которая может помочь разобраться взломщику программы в принципе её работы, вплоть до восстановления текста исходной программы, в точности до имён классов, их полей и методов.

Ещё одной причиной для выбора платформы Java является огромное количество программ, написанных для данной платформы. Особенно следует выделить приложения для операционной системы Android, которая занимает доминирующее положение на рынке мобильных устройств (Рисунок 1) [2]. Большая часть этих приложений написана на Java. Естественно желание разработчиков под эту платформу защитить свои разработки от нелегального копирования.[31]

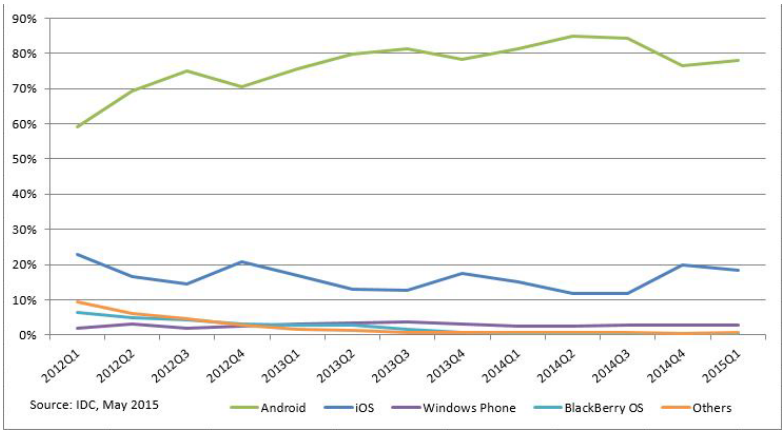


Рисунок 1. Соотношение мобильных операционных систем по количеству пользователей.

Существует множество подходов к решению поставленной проблемы, но ни один не даёт гарантированного результата. Их эффективность оценивается качественно, в зависимости от того, насколько сложно злоумышленнику преодолеть защиту. Перспективным подходом к затруднению нелегального исследования и модификации является запутывание кода (обфускация).

Но запутывание исходного кода является недостаточно эффективным методом для Java приложений, так как их исполняемый код — байт-код достаточно легко преобразовать в исходный код программы на Java. Поэтому в рамках проблемы защиты Java приложений возникает задача защиты исполняемого кода этих приложений. Злоумышленник может

производить анализ исполняемого кода следующими способами:

- Статическими
- Динамическими
- Статистическими

Целью данной работы является разработка метода защиты от статического анализа байт-кода Java приложений. Защита от статического анализа вынуждает злоумышленника перейти к анализу динамическому, значительно более сложному и трудоёмкому процессу, что сильно увеличивает затраты на обратную разработку алгоритмов или обход систем лицензирования копий программного обеспечения.

Создание собственного метода защиты позволяет блокировать именно те векторы атак, которые важны для данного конкретного приложения.

Таким образом, для достижения поставленной цели следует **решить следующие задачи:**

- Проанализировать существующие подходы к защите Java приложений, их слабые и сильные стороны. На их основе сформулировать требования к разрабатываемому методу;
- Разработать новый метод защиты и реализовать его на практике;

- Сформировать метрики, позволяющие судить об эффективности разработанного метода по сравнению с уже существующими;
- Описать ограничения и возможные проблемы, возникающие при применении разработанного метода.

Краткое содержание работы:

- Диссертационная работа состоит из введения, четырех глав, заключения и списка литературы.
- Во введении обосновывается актуальность темы исследования, формулируются цели и задачи работы.
- В первой главе приводится обзор способов защиты программ от статического анализа, выдвигаются требования к разрабатываемому методу.
- Во второй главе рассматривается концепция разрабатываемого метода.
- В третьей главе описывается реализация метода, приводится структура программы-обфускатора.
- В четвертой главе описаны и проанализированы результаты применения разработанного метода.
- В заключении обобщены результаты магистерской диссертации.

Глава 1. Обзор методов защиты.

1.1. Описание проблемы

Приложения, написанные на Java, компилируются компилятором javac. Результатом компиляции является байт-код, то есть машинно-независимый код низкого уровня, исполняемый виртуальной машиной Java. Трансляция в байт-код занимает своеобразное промежуточное положение между компиляцией в машинный код и интерпретацией.

Байт-код называется так, потому что длина каждого кода операции составляет один байт. Инструкция представляет собой однобайтовый код операции (от 0 до 255), за которым могут следовать различные параметры, например адреса памяти или строковые константы. [30]

Байт-код является промежуточным представлением программы, облегчающим и ускоряющим работу интерпретаторов и инструментов компиляции во время исполнения (JIT-компиляции). По сравнению с исходным кодом, удобным для создания и чтения человеком, байт-код является своеобразным компактным представлением программы после семантического анализа, в нём в явном виде закодированы типы, названия полей объектов, области видимости и т.п. Таким образом можно сказать, что байт-код содержит больше информации об исходном коде, нежели

машинный код, получающийся в результате компиляции программ, написанных на таких языках как C и C++.[3]

Байт-код, сгенерированный javac, может быть преобразован обратно в исходный код Java. Для этого требуется особый инструмент — декомпилятор. Декомпилятор — это программа, транслирующая исполняемый модуль (полученный на выходе компилятора) в относительно эквивалентный исходный код на языке программирования высокого уровня.

В свою очередь, декомпиляцией называется процесс восстановления исходного кода декомпилятором. Качество декомпиляции зависит от объема информации, содержащейся в декомпилируемом источнике. Байт-код, используемый большинством виртуальных машин (такими как Java Virtual Machine или .NET Framework Common Language Runtime) содержит много метаданных, делающих декомпиляцию вполне выполнимой, в то время как машинный код более скуден и сложен в декомпиляции. В частности трудночитаемыми представляются вызовы подпрограмм или функций с косвенной адресацией вызовов (в терминах языков программирования высокого уровня — вызовы через указатели на функции). [32]

На рынке программного обеспечения представлено множество различных декомпиляторов для платформы Java. В примере ниже будет использован декомпилятор JAD:

Исходный код:

```
public class HelloWorld
{
String field;
public static void main(String args[])
{
String fooBar = "Hello World!";
System.out.println(fooBar);
}
}
```

Использование JAD для декомпиляции:

```
$> jad HelloWorld.class
Parsing HelloWorld.class... Generating
HelloWorld.java
```

Результат работы JAD:

```
import java.io.PrintStream;
public class HelloWorld
{
String field;
public HelloWorld()
{
}

public static void main(String args[])
{
String s = "Hello World!";
System.out.println(s);
}
}
```

Из представленного примера видно, что исходный код восстановлен очень точно, не сохранились только имена локальных переменных, в то время как имена полей, методов и классов были восстановлены.

Декомпиляторы и другие инструменты, которые могут быть использованы злоумышленниками для обратной разработки алгоритмов и обхода системы контроля лицензий опираются на следующие способы анализа исполняемого кода:

- Статические
- Динамические
- Статистические

Самым простым и наименее затратным способом анализа исполняемого кода является статический анализ. Таким образом, защита от статического анализа вынуждает злоумышленника перейти к анализу динамическому, значительно более сложному и трудоёмкому процессу, что сильно увеличивает затраты на обратную разработку алгоритмов или обход систем лицензирования копий программного обеспечения.

Для защиты от статического анализа может быть применено запутывание кода, называемое обфускацией. Суть процесса обфускации заключается в том, чтобы запутать программный код и устранить большинство логических связей в

нем, то есть трансформировать его так, чтобы он был очень сложен для изучения и изменения, как человеком, так и автоматизированными системами.[4]

Рассматривая процесс обфускации с точки зрения защиты программного продукта и трансформации кода программы без возможности потом вернуться к его первоначальному виду (трансформация "в одну сторону"), можно дать следующее определение:

Определение. Пусть T будет некоторым трансформирующим процессом над программой P1, тогда программа P2 = TR(P1) будет представлять собой трансформированный код программы P1. Процесс трансформации TR будет считаться процессом обфускации, если будут удовлетворены такие требования:

- код программы PR2 в результате трансформации будет существенно отличаться от кода программы PR1, но при этом он будет выполнять те же функции что и код программы PR1, а также будет работоспособным;
- изучение алгоритмов работы, то есть процесс обратной разработки программы PR2 будет более сложным, трудоемким, и будет занимать больше времени, чем программы PR1;

- создание программы, преобразующей программу PR2 в ее наиболее похожий первоначальный вид, будет неэффективно.

Таким образом, обфусцированной называется программа, которая эквивалента исходной, то есть на всех приемлемых для исходной программы входных данных выдаёт тот же самый результат, что и оригинальная программа, но более трудна для анализа, изучения и изменения. Такая обфусцированная программа получается в результате применения к исходной необфусцированной программе запутывающих преобразований.

Задачу обфускации можно разделить следующие части: теоретическую, включающую в себя разработку новых алгоритмов трансформации графа потока управления или трансформации данных программы, а также теоретическую оценку сложности их анализа и раскрытия. Прикладной аспект включает в себя разработку конкретных методов обфускации, то есть наилучших сочетаний алгоритмов, сравнительный анализ различных подходов, эмпирический анализ устойчивости обфусцированных программ, и т. д.[5]

1.2. Стоимость запутывающих преобразований

Обфускация трансформирует исходную программу, затрудняя обратную разработку заложенных в неё алгоритмов.

В результате размер и скорость выполнения обфусцированной программы могут измениться. Стоимость трансформации [6] — это характеристика, которая позволяет оценить, насколько больше требуется ресурсов, таких как оперативная и постоянная память и процессорное время для выполнения обфусцированной программы, чем для выполнения исходной программы. Стоимость определяется по следующей шкале: (бесплатная, дешёвая, умеренная, дорогая)[4].

Стоимость трансформации по размеру получившейся программы позволяет оценить, насколько увеличивается размер программы в результате обфускации. Бесплатная трансформация увеличивает размер программы на O(1), дешевая трансформация увеличивает размер на O(m), где m - размер исходной программы, умеренная по стоимости трансформация увеличивает размер программы на $O(m^p)$, где $p>1$. Дорогая трансформация экспоненциально увеличивает размер обфусцированной программы по сравнению с исходной.

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

Трансформация оценивается как бесплатная, если выполнение преобразованной программы P` требует на O(1) больше ресурсов, чем выполнение исходной программы.

Трансформация оценивается как дешёвая, если выполнение программы P` требует на O(n) ресурсов больше, чем выполнение оригинальной программы. Здесь под n подразумевается размер входных данных. Трансформация оценивается как умеренная по стоимости, если выполнение программы p` требует на $O(n^p)$ больше ресурсов, где $p>1$. Трансформация оценивается как дорогая, если выполнение программы P` требует экспоненциально больше ресурсов, чем выполнение оригинальной программы.

По-видимому, применимыми на практике в автоматическом режиме являются лишь бесплатные и дешёвые методы обфускации. Другие методы следует применять только ограниченно и вручную, оценивая их влияние на обфусцированную программу.

1.3. Описание техник запутывания

В дизайне языка Java имеются определенные пробелы между теми конструкциями, которые представимы в исходном коде Java и представимы в байт-коде. Классическим примером является инструкция GOTO байт-кода, которая не имеет прямого аналога в языке Java.

Многие способы обфускации, рассматриваемые в этом разделе используют данный семантический разрыв. Хороший декомпилятор иногда способен преобразовать обфусцированный байт-код в семантически эквивалентный

участок исходного кода, но этот участок, как правило, оказывается нечитаемым. Часто, однако, эти трансформации могут привести к ситуации, когда декомпиляторы выводят либо неправильный код, либо не производят вообще никакого кода. В отдельных случаях декомпиляторы встречаются с такими конструкциями, что при попытке их обработки происходит аварийное завершение работы декомпиляторов.

Преобразование переходов в инструкции JSR

Инструкция байт-кода JSR (сокращение от «Java subroutine», т.е. «Java подпрограмма») работает почти аналогично команде GOTO: она совершает переход и после него оставляет на вершине стека адрес возврата. Как правило, обратный адрес остаётся на стеке после прыжка посредством JSR, и, когда подпрограмма завершается, байт-код RET использует этот адрес для возврата.

Конструкция JSR-RET сложна для оценки декомпиляторами, так как данная конструкция может быть вызвана из нескольких мест и это затрагивает проблему определения типа возвращаемого значения «подпрограммы». Поэтому декомпиляторам очень важно для каждой инструкции JSR найти соответствующую ей инструкцию RET.

Таким образом, данная трансформация заменяет IF и GOTO инструкцией JSR. Также в месте, куда совершается

прыжок, вставляются инструкции POP, забирающие с вершины стека адрес возврата. Это сделано для того, чтобы следующая инструкция не наткнулась на неожиданное значение на вершине стека.

По стоимости данная техника обфускации может быть классифицирована как бесплатная.

Переименование идентификаторов требует анализа использования идентификаторов. 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. Имена переменных могут быть заменены на длинные, но бессмысленные идентификаторы в расчёте на то, что длинные имена хуже воспринимаются человеком, например, l, l1I, S5$5, __, _____.

Данная трансформация является бесплатной с точки зрения размера программы и потребляемых ресурсов, но требует контроля со стороны человека, так как если в программе используются средства интроспекции (такие как Reflection) и внутренняя логика программы опирается на названия полей классов или следует каким-либо внешним соглашениям,

затрагивающим правила наименования полей и методов, то замена имён сделает программу неработоспособной.

Inlining методов [6] 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.

Даная трансформация является дешёвой с точки зрения размера программы и бесплатной с точки зрения потребляемых ресурсов.

Вынос группы операторов [6]. 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.

Данная трансформация является очень нетривиальной, так как обфускатор должен достаточно точно оценить граф потока управления в данном конкретном месте программы для того, чтобы быть полностью уверенным, что его вмешательство не нарушит работу программы.

Данная трансформация является бесплатной с точки зрения размера программы и потребляемых ресурсов.

Внесение недостижимого кода. 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.

Даная трансформация является дешёвой с точки зрения размера программы и бесплатной с точки зрения потребляемых ресурсов.

Внесение избыточного и бесполезного кода [6]. Избыточный код, в отличие от недостижимого кода выполняется, и результат его выполнения может быть

использован используется в дальнейшем в программе, но такой код можно упростить или совсем удалить. В первом случае вычисляется либо константное значение, либо значение, уже вычисленное ранее. Во втором случае это означает, что вставленный код не имеет никаких побочных эффектов. Для внесения избыточного кода можно использовать алгебраические преобразования выражений исходной программы или введение в программу математических тождеств. Например, можно воспользоваться комбинаторным тождеством и заменить везде в программе использование константы 256 на цикл, который вычисляет сумму биномиальных коэффициентов по приведённой формуле.

Даная трансформация является дешёвой как с точки зрения размера программы, так и с точки зрения потребляемых ресурсов.

Замена ветвления if исключениями try-catch. Проверки, является ли ссылка null, заменяются вызовом методов объекта, на который указывает ссылка, что вызывает исключение. Управление передается в catch-блок, где размещается код, изначально размещавшийся в «действии» блока if.

Хотя данная трансформация является дешёвой как с точки зрения размера программы, так и с точки зрения

потребляемых ресурсов (O(n)), но исключения в Java работают крайне медленно.

Перемещение констант в поля объектов позволяет увеличить количество не-константных переменных в программе и замаскировать использование одной и той же константы в разных частях программы.

Устранение библиотечных вызовов является достаточно действенным шагом, предполагающим замену вызовов библиотек их обфусцированными версиями, либо создание буферных классов к существующим библиотекам. В любом случае размер программы значительно увеличивается.

Использование битовых операций в вычислениях позволяет превратить даже небольшое арифметическое выражение в несколько строк битовых операций, что сильно затрудняет разбор программы человеком. [7]

Развёртка циклов позволяет замаскировать использование цикла путём явной вставки в код всех итераций, перемешанных с бесполезным кодом. Декомпиляторы могут вывести их все без изменений или разбить на несколько меньших циклов, разделённых бесполезными операциями.

Непрозрачный предикат — это такое выражение, значение которого известно во время обфускации, но трудно установить после завершения процесса обфускации [8]. Если

такой предикат является условием перехода, то во время анализа декомпилятор не сможет быть уверенным, какая именно ветвь выполнится. Данный способ является достаточно эффективным, так как обеспечивает хорошую защиту и при этом требует совсем немного ресурсов. [9]

Преобразование сводимого графа потока управления к несводимому [10] возможно в том случае, когда байт-код более выразителен, чем исходный язык, и в нём можно использовать преобразования, "противоречащие" структуре исходного языка. В байт-коде существует команда goto, но в Java нет возможности совершать безусловные переходы. Графы потока управления программ на Java всегда сводимые, но в байт-коде могут быть представлены и несводимые графы.

Клонирование функций позволяет увеличить количество функций, которые выполняют одну и ту же задачу, но для декомпилятора выглядят по-разному. Для этого используются различные имена, списки параметров и лишние операции внутри функций. Метод эффективен, но не поддерживает средства интроспекции. [11]

Переплетение функций основано на создании функций-комбайнов с огромным количеством входных параметров и сложной логикой выбора какие инструкции исполнять при заданном наборе входных аргументов. Метод эффективен, но не поддерживает средства интроспекции.

Шифрование строк основывается на том, что для понимания логики программы человеком очень важны строки, например сообщения об ошибках и истечении срока лицензии. Строки шифруются, а места их использования оборачиваются функцией-расшифровщиком. [12]

1.3. Описание требований к методу

Используя данные прошлого раздела и работ [13], [14] b [15] была получена сводная таблица, содержащая данные о применяемых запутывающих трансформациях для Java программ:

Трансформация	Размер	Производительность	Интроспекция	Android
Замена ветвления if исключениями try-catch	+10-30%	-20-40%	да	да
Внесение избыточного кода	+10-30%	-10-20%	да	нет
Перемещение констант в поля объектов	+10-30%	-10-20%	нет	да
Внесение недостижимого кода	+10-30%	-0-10%	да	да

Устранение библиотечных вызовов	+30-90%	-0-10%	да	да
Использование битовых операций в вычислениях	+10-30%	-10-20%	да	нет
Развёртка циклов	+30-90%	-10-20%	да	да
Непрозрачные предикаты	+0-10%	-10-20%	да	да
Вставка функций	+30-90%	-0-10%	нет	да
Внесение бесполезного кода	+10-30%	-10-20%	да	да
Переименование идентификаторов	+0-10%	-0-10%	нет	да
Вынос группы операторов	+10-30%	-0-10%	нет	да
Преобразование переходов в инструкции JSR	+0-10%	-0-10%	да	нет
Преобразование сводимого графа потока управления к несводимому	+10-30%	-10-20%	да	нет
Клонирование функций	+10-	-10-	нет	да

	30%	20%		
Шифрование строк	+0-10%	-10-20%	да	да
Переплетение функций	+10-30%	-10-20%	нет	да

Таблица 1. Трансформации, применяемые для Java программ.

Здесь колонки «размер» и «ресурсы» показывают стоимость запутывающих преобразований по сравнению с программой, не подвергнувшейся трансформации. Колонки «интроспекция» и «Android» описывают возможность применения средств интроспекции к запутанной программе и применимость трансформации для защиты программ для операционной системы Android соответственно.

Интроспекцией называется возможность определить тип и структуру объекта во время выполнения программы. Критерий интроспекции был выбран в связи с тем, что в последние годы средства разработки и библиотеки, основанные на интроспекции, получили такое широкое распространение, что в сегодняшний реалиях очень сложно написать крупную программу без их интенсивного использования.

В свою очередь приложения для платформы Android являются очень уязвимыми, так как в отличие от большинства

приложений, написанных на Java они устанавливаются напрямую на клиентские устройства [Oracle Corporation, 2013]. Учитывая то, что Android является самой популярной мобильной операционной системой в мире, занимая 78,4% рынка мобильных операционных систем [2015], проблема защиты Android-приложений встаёт ещё более остро.

Учитывая эти факторы и работу [16] можно выдвинуть следующие требования к трансформациям, которые следует использовать в разрабатываемом методе:

- Не должна увеличивать размер программы более, чем на 30%;
- Не должна снижать производительность более, чем на 20%;
- Должна поддерживать средства интроспекции;
- Должна быть применима к программам для операционной системы Android.

Проанализировав данную таблицу, можно заметить, что многие подходы не могут быть использованы для защиты приложений, исполняющихся под управлением операционной системы Android и широко использующих средства интроспекции. В то же время некоторые из оставшихся подходов не могут использоваться во многих проектах в связи с тем, что стоимость их внедрения слишком высока. Другие не

могут быть использованы по причине того, что являются слишком дорогими с точки зрения размера программы и затрат ресурсов во время исполнения.

Трансформация	Размер	Произв-ть	Интро-спекция	Android
Внесение недостижимого кода	+ 10-30%	- 0-10%	да	да
Непрозрачные предикаты	+ 0-10%	- 10-20%	да	да
Внесение бесполезного кода	+ 10-30%	- 10-20%	да	да
Шифрование строк	+ 0-10%	- 10-20%	да	да

Таблица 2. Траснформации, удовлетворяющие требованиям.

Это оставляет очень ограниченное число подходов, пригодных для того, чтобы быть использованными в разрабатываемом методе, поэтому следует разработать ещё одну трансформацию.

Глава 2. Разработка метода.

2.1. Описание техник статического анализа, на защиту от которых направлен метод.

В данном разделе будут описаны способы, применяемые для анализа программ и их исполняемого кода. Задачи этих способов — построение графа потока управления и выявление зависимостей между компонентами программы, что даёт возможность применить какие-либо преобразования программы в обратную запутыванию сторону.

Способы анализа байт-кода программ могут быть разделены на следующие 4 группы:

- **Статические**. К данной группе относятся способы анализа потоков управления и способы, основанные на результатах анализа потоков данных. Статические способы анализа работают с программой, не используя информацию о работе программы на каких-то конкретных начальных данных. [17]
- **Динамические**. Динамические способы анализа программ изучают информацию, получаемую в результате "наблюдения" за различными прогонами программы на конкретных входных данных, то есть профилировании. Однако сами по себе динамические способы очень редко применяются для анализа программ, поскольку, как правило, для их работы требуется информация о поведении

программы на различных наборах входных данных, которая собирается с помощью статистических способов анализа.

- **Статистические**. Статистические способы используют информацию, собранную в результате значительного количества прогонов программы на большом количестве наборов входных данных. Очевидным недостатком такого подхода является необходимость собрать достаточное количество требуемых входных данных и быть уверенными в том, что данный набор полон, то есть включает в себя все входные данные, используемые программой, ведь некоторые данные могут быть получены неявно посредством использования системных вызовов.

Краткая характеристика важнейших для данной работы статических способов анализа программ приведена ниже.

Анализ алиасов и указателей [18] возможен в языках, в которых несколько имён или указателей могут быть использованы для обращения к одной и той же области памяти. К примеру, элемент массива a может быть адресован в программе как a[0] (каноническое имя элемента массива), как a[j] (с использованием переменной) или даже как b[-4] (использование другого массива). Результатом процесса анализа указателей является соответствие: 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, использование анализа алиасов и 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 (так называемый "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. Хотя такая схема слишком груба и на самом деле блокирует глубокую трансформацию практически любой программы, но общая задача точного анализа указателей как минимум NP-трудна. В настоящее время существуют способы, работающие за полиномиальное время, для указателей на локальные переменные в случае нерекурсивных функций.

Анализ указателей не может быть непосредственно использован для деобфускации программы, но он является важным для точного анализа свойств исследуемой программы.

Устранение мёртвого кода [19] имеет целью выявить в программе код, который выполняется, но не оказывает никакого влияния на результат работы программы.

Минимизация количества переменных [19] ставит целью уменьшение количества используемых в функции локальных переменных за счёт объединения в одну переменную переменных, времена жизни значений в которых не пересекаются. Самая распространённая техника, используемая для минимизации количества переменных, состоит в построении графа перекрытия областей видимости переменных с помощью анализа потока данных и последующей раскраске вершин этого графа в минимальное или близкое к нему количество цветов.

Продвижение констант и копий переменных [19] заключается в продвижении константных значений как можно дальше по тексту функции. Если выражение использует значения только тех переменных, которые в данной точке программы заведомо содержат известное при анализе программы значение, такое выражение может быть вычислено ещё на этапе анализа программы. Если в выражении используется переменная, про которую заведомо известно, что она является копией какой-то другой переменной, в такое выражение может быть подставлена исходная переменная.

Анализ доменов [20] является расширением алгоритма продвижения констант для множества значений. Он может

определить некое множество значений, которые может принимать данная переменная в данной точке программы, если это множество не слишком велико.

Статический слайсинг [21] — это построение "сокращённой" программы, из которой удалён весь код, не влияющий на вычисление заданной переменной в заданной точке (так называемый 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. Слайсинг может быть использованы при разделении "переплетённых" вычислений, когда одновременно вычисляются две независимые друг от друга величины. Например, если в одном цикле вычисляется скалярное произведение двух векторов и максимальный элемент каждого вектора, то такие циклы могут быть расщеплены на два отдельных с помощью построения слайсов.

Также важен шаг преобразования исходного двоичного файла с байт-кодом в набор инструкций байт-кода. Широко известно два метода преобразования двоичного файла в массив байт-кодов без исполнения самого файла: алгоритмы линейной развёртки и рекурсивного обхода. [22]

Алгоритм линейной развёртки. При использовании алгоритма линейной развертки декомпилятор последовательно проходит по исполняемому файлу. Он начинает с первого байта в секции кода и продвигается вперёд, декодируя каждый байт, в том числе какие-либо промежуточные байты данных, до тех пор, пока не закончится блок. Это делает алгоритм очень простым, но в то же время подверженным многим ошибкам, намеренно оставленным в коде как раз для того, чтобы препятствовать алгоритму линейной развертки. Наиболее известными линейными дизассемблерами являются objdump, WinDbg и SoftICE.

Алгоритм рекурсивного обхода. [23] Этот метод является гораздо менее восприимчивым к ошибкам, чем алгоритм линейной развертки, потому что код разбирается не линейно, а "прыжками". На основном шаге происходит линейная развёртка, но случаях, когда встречается переход (условный или безусловный), алгоритм переходит по нему и снова производит линейную развёртку. Самыми известными рекурсивными дизассемблерами являются OllyDdg и IdaPro.

2.2. Описание метода.

Как было показано в конце раздела 1, требуется разработать трансформацию, которая бы удовлетворяла требованиям:

- Не должна увеличивать размер программы более, чем на 30%;
- Не должна снижать производительность более, чем на 20%;
- Должна поддерживать средства интроспекции;
- Должна быть применима к программам для операционной системы Android.

Традиционные запутывающие трансформации основаны на модификации байт-кода виртуальной машины Java, в который компилируется исходный код программ. Данный подход позволяет обфусцировать приложения, написанные на языках, предназначенных для работы на виртуальной машине Java: Java, Scala, Groovy, Grails.

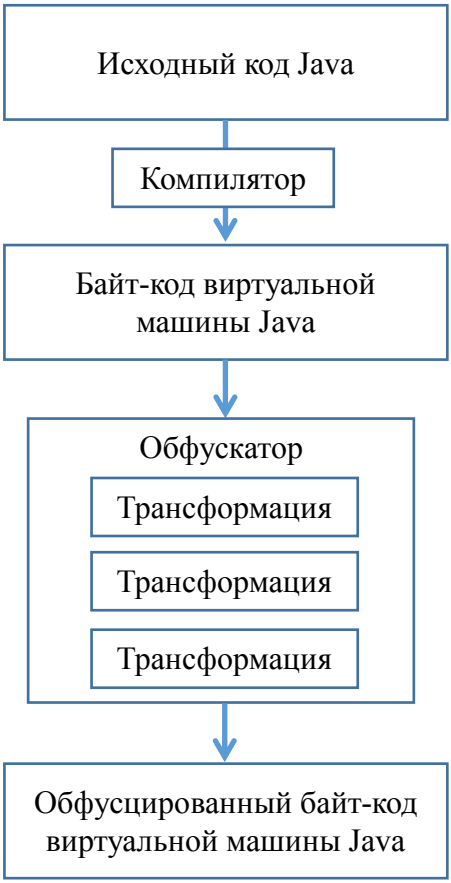


Рисунок 2. Схема традиционных методов запутывания.

В свою очередь, разрабатываемая трансформация основана на схеме компиляции, используемой в программах, исполняющихся под управлением операционной системы Android.

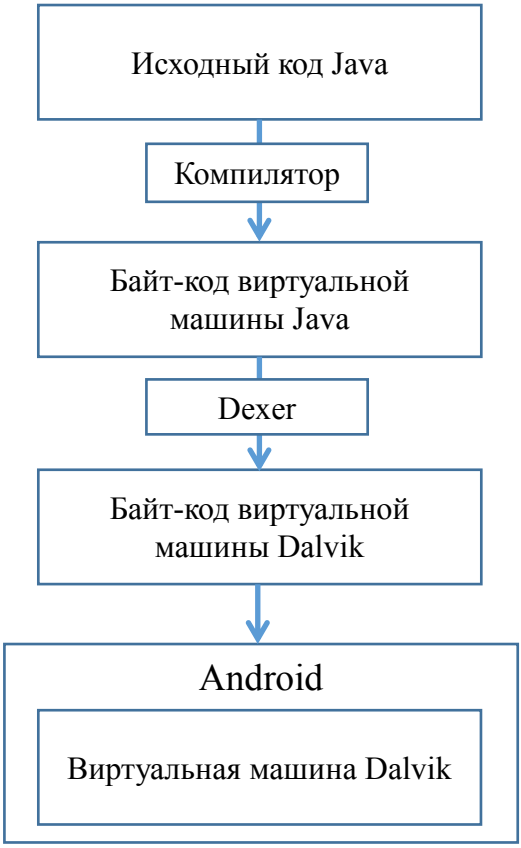


Рисунок 3. Схема компиляции приложений для ОС Android.

Компилятор преобразует исходный код приложения в байт-код, предназначенный для выполнения на виртуальной

машине Java. На следующем этапе специальная утилита под названием Dexer преобразовывает байт-код виртуальной машины Java в байт-код виртуальной машины Dalvik.

Виртуальная машина Dalvik имеет следующие отличия от виртуальной машины Java:

- является регистровой, а не стековой виртуальной машиной
- имеет свой набор команд, больший по размеру
- имеет более слабые правила проверки байт-кода, что позволяет производить трансформации, недоступные в байт-коде Java

Таким образом, из-за большего количества команд и меньшего количества проверок трансформировать байт-код Dalvik можно большим количеством способов, чем байт-код Java.

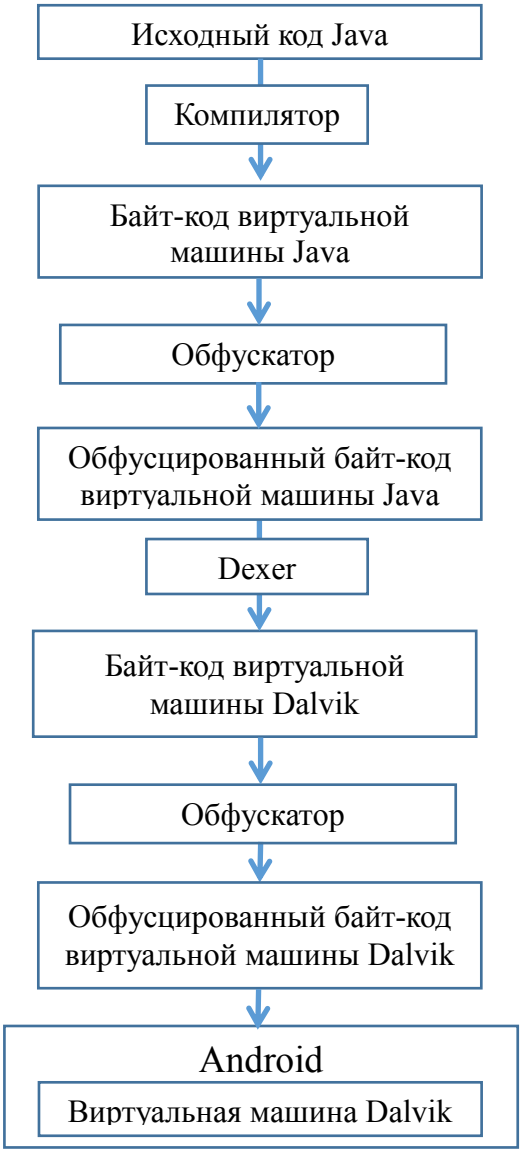


Рисунок 4. Схема применения трансформаций в разрабатываемом методе.

Одна из задач, которые надо решить перед разработкой трансформации — это определение алгоритмов, используемых декомпиляторами для нахождения инструкций. Данные методы были описаны в разделе 2.1. Наиболее известным методом является рекурсивный обход, но из-за простоты формата байт-кода Dalvik многие инструменты используют линейную развертку, которая отлично подходит для приложений, которые не были обфусцированы. Алгоритм линейной развёртки также используется в Dalvik как часть верификора байт-кода, который проверяет все приложения, прежде чем они могут быть установлены.

Основная проблема с линейной разверткой состоит в том, что алгоритм делает различий между инструкциями, частями сложных инструкций и закодированными данными.

Таким образом, в том случае, когда инструкции в коде перекрываются, вполне вероятна такая ситуация, что алгоритм не сможет "увидеть" как раз ту инструкцию, которая реально выполняется, и, как следствие, её не увидит аналитик.

В байт-коде Dalvik есть целый набор инструкций с переменной длиной, которые можно использовать, чтобы скрыть следующие после них инструкции. Как пример одной из таких инструкций, подходящих для реализации метода, была выбрана инструкция заполнения массива данных: fill-array-data-payload.

Формат инструкции fill-array-data-payload представлен в таблице Таблица 3:

Поле	Формат	Описание
ident	ushort = 0x0300	идентификатор псевдо-инструкции
element_width	ushort	количество байтов в каждом элементе
size	uint	количество элементов в массиве
data	ubyte[]	значения элементов

Таблица 3. Формат инструкции fill-array-data-payload

Инструкция fill-array-data-payload размещается в самом начале метода и операнды element_width и size заполняются такими значениями, чтобы эта инструкция скрывала все следующие инструкциями метода. Чтобы эта инструкция не была выполнена, перед ней размещается безусловный переход, указывающий на первую инструкцию исходного байт-кода. Таким образом, можно гарантировать, что поведение конкретного метода не изменяется.

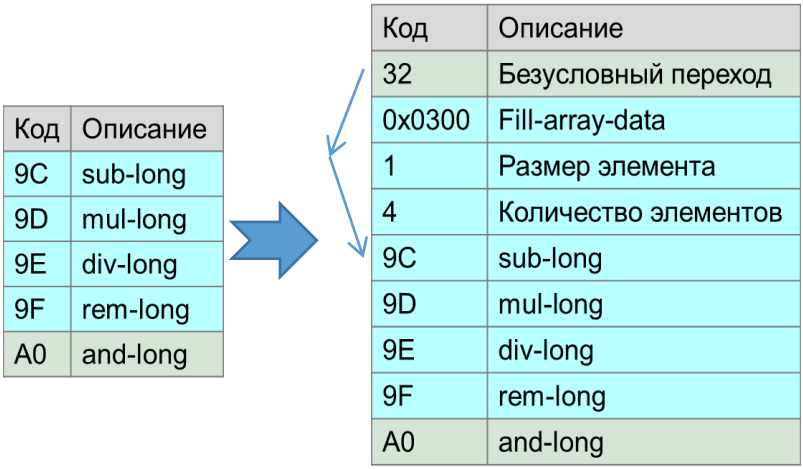


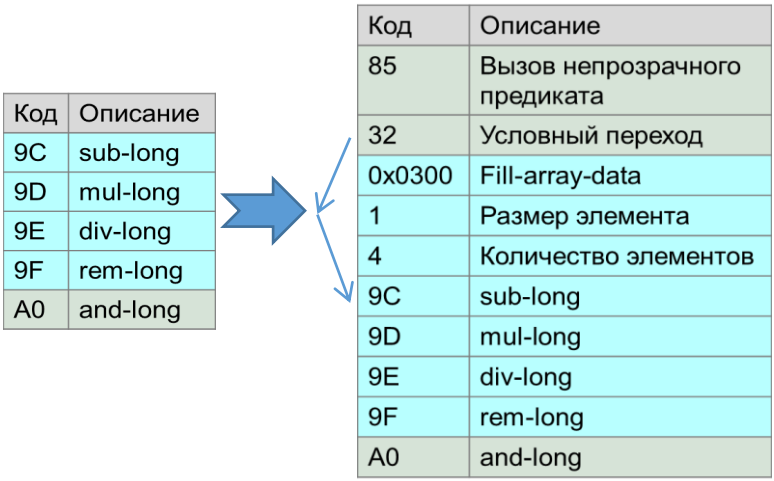
Рисунок 5. Схема трансформации с безусловным переходом.

Дизассемблер, основанный на алгоритме линейной развертки, при обработке данного метода будет наблюдать инструкцию перехода, а за ней инструкцию заполнения массива данных, но не заметит скрытые в ней инструкции. Поэтому оригинальные инструкции будут скрыты от аналитика.

Для дизассемблеров, основанных на алгоритме рекурсивного обхода, данные инструкции не будут скрыты. Дизассемблер перейдёт по безусловному переходу и попадёт на первую команду оригинального байт-кода.

Для того чтобы покрыть дизассемблеры, основанные на алгоритме рекурсивного обхода, следует использовать условный переход, который позволит дизассемблеру обнаружить команду

заполнения массива данных. В качестве условия перехода будет использоваться всегда истинный непрозрачный предикат.



Код	Описание
9C	sub-long
9D	mul-long
9E	div-long
9F	rem-long
A0	and-long

Код	Описание
85	Вызов непрозрачного предиката
32	Условный переход
0x0300	Fill-array-data
1	Размер элемента
4	Количество элементов
9C	sub-long
9D	mul-long
9E	div-long
9F	rem-long
A0	and-long

Рисунок 6. Схема трансформации с условным переходом.

Дальнейшее улучшение может быть сделано путем вставки дополнительных инструкций, использующих массив, созданный инструкцией fill-array-data-payload. Это позволяет симулировать использование вставленных инструкций, что важно для обхода некоторых инструментов, например, androguard.

Таким образом, принятие решения о выборе реально исполняющегося кода ляжет на аналитика, что повышает трудозатраты на обход защиты.

В качестве структуры, на которой основан непрозрачный предикат, выступает массив, начальные значения элементов которого выбираются на этапе обфускации, а затем с помощью циклов изменяются таким образом, чтобы без запуска программы соотношения между этими элементами были трудноустановимы. В работе [24] показано, что задача выявления вставленного мёртвого кода в программе, защищённой выбранным типом непрозрачных предикатов, NP-полна, то есть не существовало полиномиального алгоритма для нахождения их значения.

Ограничения метода состоят в том, что разработанный метод применим только для приложений, не требующих быстрой реакции на внешние воздействия. Потери производительности при использовании всех трансформаций, входящих в метод, могут достигать 70%, и эти потери трудно предсказать.

Также разработанный метод может быть несовместим с будущими реализациями виртуальной машины Dalvik, так как получаемый байт-код не в полной мере удовлетворяет спецификации. Это также может сделать невозможной ahead-of-time (предварительную) компиляцию исполняемого кода, которая становится популярным методом компиляции приложений для операционной системы Android.

Глава 3. Практическая реализация метода.

В первую очередь была разработана архитектура программы.

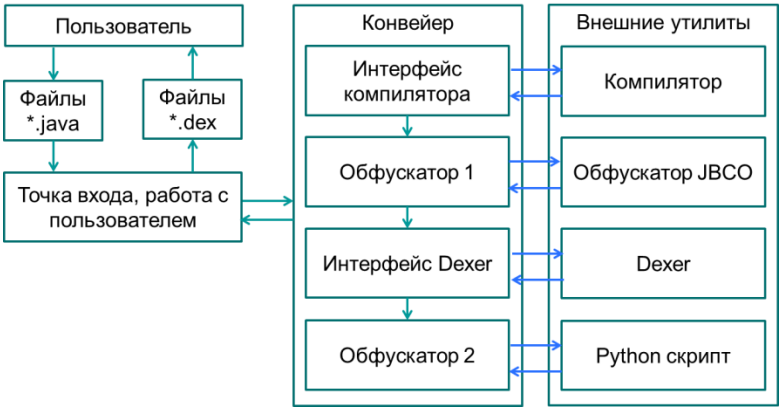


Рисунок 7. Архитектура программы.

Главный класс, отвечающий за запуск программы и обработку входных параметров, взаимодействие с пользователем, он принимает входные параметры и файлы, передаёт их конвейеру.

Управляющий класс — конвейер, реализующий шаблон проектирования «Цепочка обязанностей», (англ. «Chain of Responsibility»). Данный класс отвечает за выполнение действий в правильной последовательности, а так же за передачу промежуточного результата между отдельными «работниками конвейера» — компилятором, обфускатором 1, утилитой Dexer и обфускатором 2.

Части конвейера вызывают внешние программы, упаковывают и распаковывают исполняемые файлы, например компилятор, для этого разработаны специальные классы-утилиты:

- Распаковщик jar-файлов
- Упаковщик jar-файлов
- Класс, позволяющий абстрагироваться от платформо-зависимых путей к исполняемым файлам JBCO
- Класс, позволяющий абстрагироваться от платформо-зависимых путей к исполняемым файлам компилятора
- Класс, позволяющий абстрагироваться от платформо-зависимых путей к исполняемым файлам Dexer
- Класс для запуска Python скриптов, позволяющий абстрагироваться от платформо-зависимых путей к исполняемым файлам Python

Интерфейс компилятора используется для того, чтобы предоставить операционной системе возможность подставить правильный путь к используемому по умолчанию компилятору, такому как javac или ajc. На шаге компиляции программа, написанная на языке высокого уровня (Java) преобразуется в платформо-независимое представление — байт-код.

Обфускатор 1 совершает трансформации с байт-кодом виртуальной машины Java, полученным на этапе компиляции. Ниже перечислены выбранные для реализации в методе и подходящие под требования трансформации из раздела 1.3:

- Внесение недостижимого кода
- Использование непрозрачных предикатов
- Внесение бесполезного кода
- Шифрование строк

Данные трансформации реализованы в пакете JBCO, Java ByteCode Obfuscator, разработанном Sable Research Group в рамках проекта Soot framework, фреймворка, предназначенного для проведения манипуляций с исполняемым байт-кодом виртуальной машины Java.

Обфускатор 1 запускает перечисленные трансформации в определённом порядке, используя интерфейс командной строки пакета JBCO. Например, для применения трансформации шифрования строк команда Soot выглядит следующим образом:

java -cp /path-to-soot/soot.jar soot.jbco.Main -cp ".:/path-to-jdk7/jre/lib/rt.jar:/path-to-jdk7/jre/lib/jce.jar" -t:9:saen.j ptss -debug -main-class com.examlpe.Program -app com.examlpe.Program

Вывод, описывающий элементраные шаги, совершаемые фреймворком:

Whole Program Jimple Transformations:

wjtp.mhp default

wjtp.tn default

wjtp.jbco_f_r JBCO

wjtp.jbco_c_c JBCO

wjtp.jbco_c_r JBCO

wjtp.jbco_m_r JBCO

wjtp.jbco_b_apibm JBCO

wjtp.jbco_b_lbc JBCO

Jimple Method Body Transformations:

jtp.jbco_g_ia JBCO

jtp.jbco_a_dss JBCO

jtp.jbco_c_ae2bo JBCO

jtp.jbco_j_l JBCO

Baf Method Body Transformations:

bb.jbco_j_2bl JBCO

bb.jbco_c_b2ji JBCO

bb.jbco_d_cc JBCO

bb.jbco_r_ds JBCO

bb.jbco_r_iitcb JBCO

bb.jbco_i_ii JBCO

bb.jbco_p_lvb JBCO

bb.jbco_r_laii JBCO

bb.jbco_c_tbcb JBCO

bb.jbco_e_cvf JBCO

bb.jbco_p_tss JBCO

bb.jbco_f_ul JBCO

bb.l_so default

bb.p_ho default

bb.u_le default

bb.l_p default

bb.j_bco_rrps JBCO

Обфусцированный файл попадает в выходную директорию и отправляется дальше по конвейеру

Интерфейс утилиты Dexer предназначен для того, чтобы предоставить операционной системе возможность подставить правильный путь к используемому по умолчанию

Android SDK — набору утилит, предназначенных для разработки приложений для операционной системы Android.

Данная утилита преобразует байт-код формата виртуальной машины Java в байт-код виртуальной машины Dalvik, предназначенный для запуска на устройствах, работающих под управлением операционной системы Android.

Обфускатор 2 производит трансформацию байт-кода виртуальной машины Java, полученного на этапе преобразования утилитой Dexer.

Принцип работы этого обфускатора описан в разделе 2.2. Он реализован на языке Python в виде скрипта, так как в момент написания только для Python имелась стабильная библиотека для обработки файлов, содержащих байт-код виртуальной машины Dalvik. Он реализует составную трансформацию:

- Вставка команд, модифицирующих структуру в памяти
- Определение блока защищаемых команд
- Вставка команд переменной длины , скрывающих защищаемые команды
- Вставка команд, использующих результат выполнения команд переменной длины. Данный этап требуется для придания большей

правдоподобности использования и важности результата выполнения вставленных командах

- Вставка условий перехода по непрозрачному предикату

В качестве структуры, на которой основан непрозрачный предикат, выступает массив, начальные значения элементов которого выбираются на этапе обфускации, а затем с помощью циклов изменяются таким образом, чтобы без запуска программы соотношения между этими элементами были трудноустановимы, то есть не существовало полиномиального алгоритма для их нахождения.

Глава 4. Анализ эффективности метода.

4.1. Описание используемых метрик.

Для оценки эффективности разработанного метода защиты следует сравнить его с изначально доступным методом. Сравнения производится по ряду метрик, которые описывают сложность программного обеспечения для понимания человеком. Метрики, приведённые в данном разделе, были выбраны после анализа работ [24], [25] и [33], в которых рассматривались способы оценки обфускаторов и декомпиляторов для программ, написанных на Java.

Первой рассматриваемой метрикой для определения эффективности обфускации является частота встречаемости «сложных» конструкций языка в коде. Конечно, необходимо определить, какие конструкции являются убедительным свидетельством сложности. После рассмотрения работ [24] и [26] были выбраны следующие четыре категории:

- if и if-else
- "Непредвиденная" передача потока управления: операторы break и continue
- Метки переходов.
- Локальные переменные.

Условные переходы показывают количество случаев, когда в программе происходит "принятие решения". Более

сложная программа будет более разветвлённой и, следовательно, будет иметь больше операторов if и if-else.

Операторы "непредвиденной" передачи управления ещё более показательны, если речь заходит о сложности программы. Существует устойчивое мнение, что они сильно запутывают программу и делают поток граф выполнения более сложным для понимания. [27]

Считается, что эти средства передачи управления являются одними из самых сложных для понимания, как человеком, так и декомпилятором, так как они нарушают последовательную структуру потока выполнения. Чем больше таких операторов встречается в программе, тем меньше кода можно прочитать последовательно. Это делает задачу декомпиляции более сложной, так как сильно увеличивает количество последовательных участков на графе потока выполнения, при этом сильно уменьшая их размер, разбивая их на базовые блоки, которые очень тесно переиспользуются, что повышает связность между различными, изначально независимыми участками исходного кода.

Помеченные блоки — это блоки, состоящие из нескольких операторов, переход на которые может быть осуществлён с помощью операторов break и continue, что в большинстве программ будет означать, что данные блоки имеют важное значения для управления потоком выполнения

программы. Так же как и обработка исключений, данные конструкции являются одними из самых сложных для разбора декомпиляторами конструкциями в Java. [28]

Количество локальных переменных также может указывать на сложность программы. Ведь чем больше информации требуется прочитать для понимания отдельного участка программного кода, тем сложнее его понять. Поэтому программисты обычно избегают создавать большое количество избыточных идентификаторов, в то время как обфускаторы используют данную практику очень широко.

Сложность условий. Логические выражения, которые определяют передачу потока управления в программе, играют важнейшую роль при анализе декомпиляторами. Помимо логических констант, таких как истина или ложь), простейшие условные выражения состоят из одной логической переменной. Назначим такому выражению, состоящему всего из одной логической переменной, вес 1. Но также часты случаи, когда условные выражения представляют собой вложенные конструкции или агрегацию нескольких условий. Логическая переменная может быть инвертирована оператором !, может участвовать в сравнении с использованием операторов <, >,<=, >=, ==.Хотя эти операторы сложнее обычной логической переменной, их роль всё равно достаточно легко понять, поэтому им присваивается вес 0,5. Агрегация выражений с

помощью операторов && или || требует от аналитика оценки значения двух подвыражения, а затем объединения результата, что является непростой задачей, поэтому каждому такому оператору агрегации установлен в соответствие вес 1.

Таким образом, сложность каждого логического выражения может быть рассчитана как сумма весов составных частей выражения, описанных выше.

Например, сложность выражения a<b && !c может быть рассчитана следующим образом:

- Выражение a<b содержит две переменные, каждая весом 1, и условный оператор весом 0,5, что даёт в сумме 2,5;
- Выражение !c содержит логическую переменную весом 1 и оператор весом 0,5;
- Оператор агрегации && добавляет ещё 1, таким образом, общий вес выражения становится равным 5.

Среднюю сложность условий можно определить как среднее значение сложностей всех булевых выражений в программе.

Эффективность работы декомпиляторов является в некотором роде субъективной метрикой, так как показывает эффективность работы декомпиляторов в данный конкретный

момент против нового, пока что неизвестного производителям декомпиляторов метода. Но, тем не менее, следует провести сравнение и по этому критерию, так как всё равно важно оценить, сможет ли созданный метод «обмануть» хотя бы существующие декомпиляторы, ведь в тех случаях, когда вопрос защиты интеллектуальной собственности стоит особенно остро, разработчик может прибегнуть к созданию нового, доселе неизвестного метода защиты. Этот новый метод может задержать злоумышленника на некоторое время, ведь ему придётся сначала изучить его работу.

4.2. Оценка эффективности метода.

В качестве программы-примера использовались листинги трёх программ из [24], представляющих собой программы, «сбалансированные» по количеству используемых шаблонов программирования и содержащие некоторые сложные случаи, как для обфускаторов, так и для декомпиляторов. Эти примеры являются своеобразным «портретом» типичной современной программы, требующей обфускации. На диаграммах за 100% взят обфусцированный код, полученный после обработки методом, состоящим из четырёх трансформаций, выбранных в разделе 1.3 подходящими под требования метода.

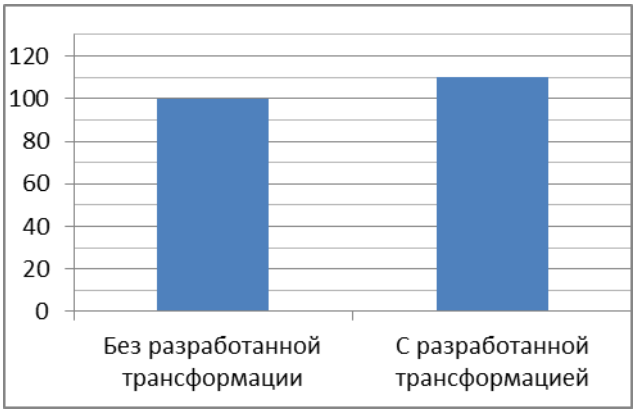


Рисунок 8. Изменение количества условных переходов, %.

Так как разработанная трансформация добавляет новые условные переходы для того, чтобы обойти декомпиляторы, использующие алгоритм рекурсивного обхода, то количество условных переходов увеличивается.

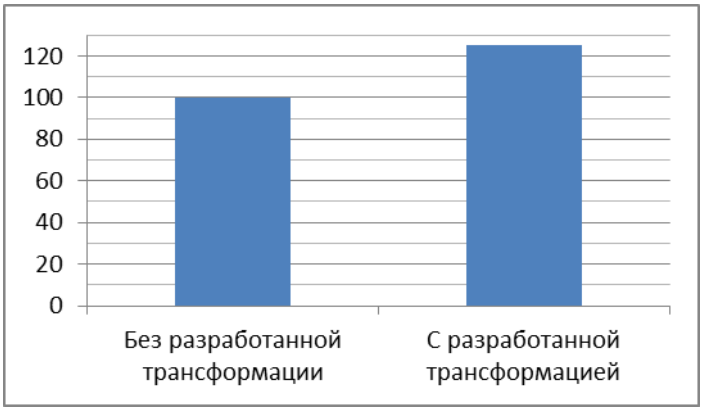


Рисунок 9. Изменение сложности условных переходов, %.

Как уже было описано ранее, сложность условий — это мера сложности условий внутри различных условных операторов языка, таких как if, if-else, while и for. Сложность условий сильно возрастает, когда условия агрегируются с помощью операторов && и ||. Так как разработанная трансформация добавляет новые условные переходы, основанные на непрозрачных предикатах, то сложность условий в трансформированной программе растёт, что отражает приведённая диаграмма.

"Непредвиденная" передача потока управления: операторы break и continue

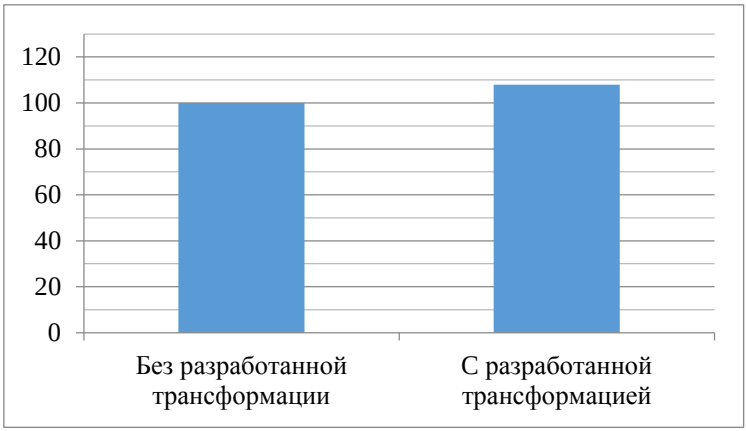


Рисунок 10. Изменение количества переходов типа break и continue, %.

Данная диаграмма показывает незначительное увеличение количества операторов передачи управления, так как они использовались среди других инструкций, использованных для того, чтобы симулировать работу с результатом команд переменной длины.

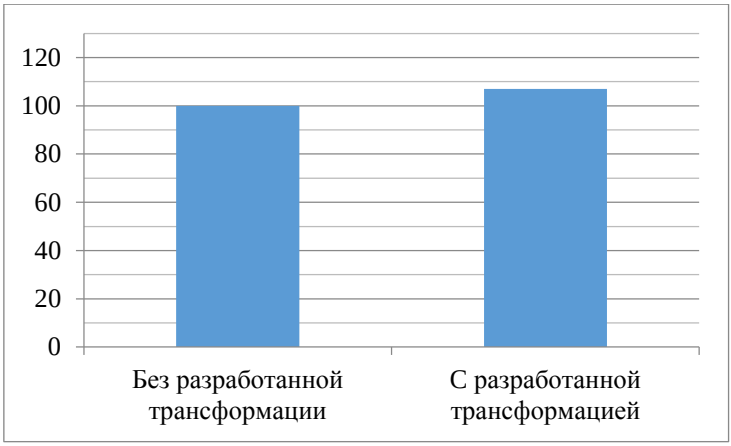


Рисунок 11. Изменение количества помеченных блоков, %.

Данная метрика непосредственно связана с предыдущей метрикой, описывающей количество конструкций, обеспечивающих "внезапную" передачу управления, так как помеченные блоки являются теми местами, куда передается управление с помощью операторов "внезапной" передачи управления. Также меченые блоки могу использоваться вместе с конструкцией GOTO, что ещё больше повышает запутанность графа потока выполнения. Приведённая диаграмма показывает увеличение количества помеченных блоков.

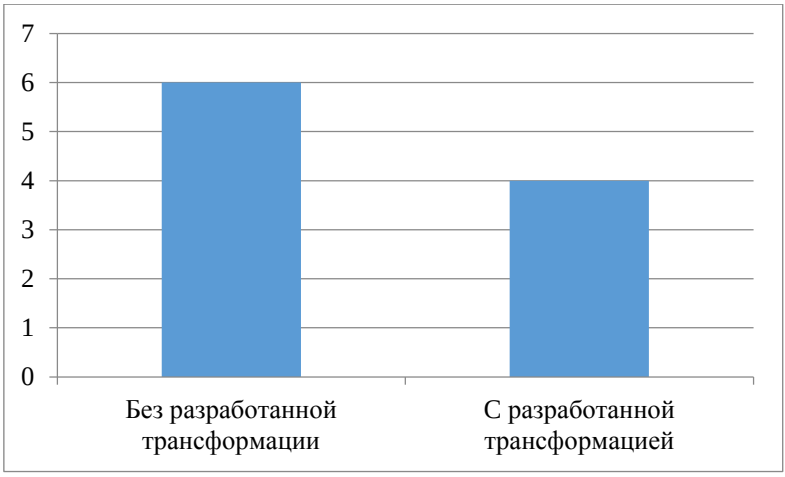


Рисунок 12. Успешно отработавшие декомпиляторы, штук.

Всего использовалось шесть различных декомпиляторов, которые изначально успешно анализировали и генерировали исходный код программы. После применения разработанной трансформации два декомпилятора не смогли справиться с задачей (вывели ошибку выполнения), остальные вывели код, но не смогли избавиться от ветвей, защищённых непрозрачными предикатами, аналитику придётся определять значение предикатов самостоятельно.

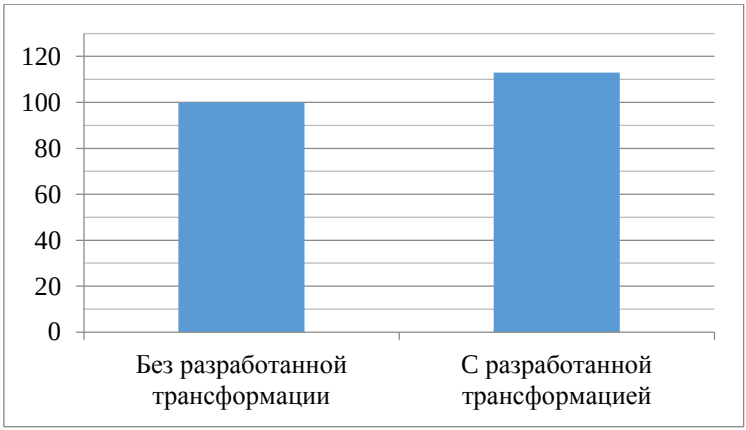


Рисунок 13. Изменение размера программы, %.

Вставка новых конструкций увеличивает размер программы. В программе-примере размер увеличился на 10%, и так как разработанная трансформация является в терминах раздела 1.2 «дешёвой», то можно сказать, что для большинства программ она также будет увеличивать размер на 10%.

Производительность программы

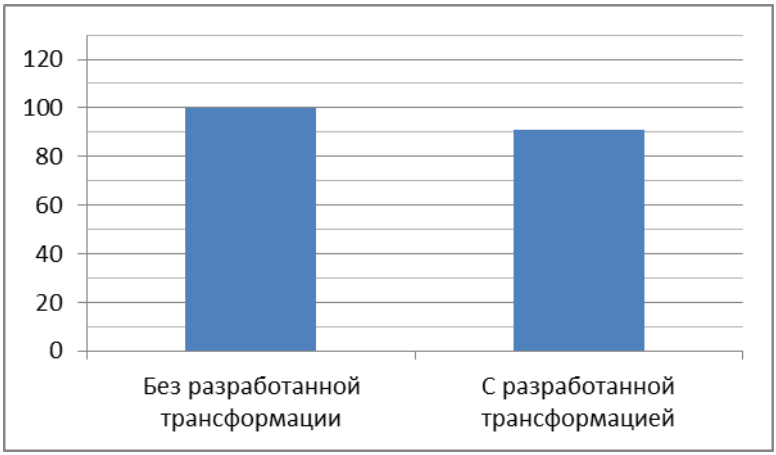


Рисунок 14. Изменение производительности программы, %.

Вставка неявных предикатов увеличивает количество кода, который надо выполнить. В программе-примере производительность упала на 10%, а так как разработанная трансформация является в терминах раздела 1.2 «дешёвой», то можно сказать, что для большинства программ она также будет уменьшать производительность на 10%.

Эффективность в целом

Основываясь на [24], для того чтобы обеспечить одну универсальную метрику эффективности, были использованы различные значения весов, описывающие соотношения между различными метриками:

Метрика	Весовой коэффициент
Количество условий	0,8
Сложность условий	1
«Непредвиденная» передача управления	2
Помеченные блоки	1,5
Размер программы, байт	-0,5
Падение производительности	-0,5

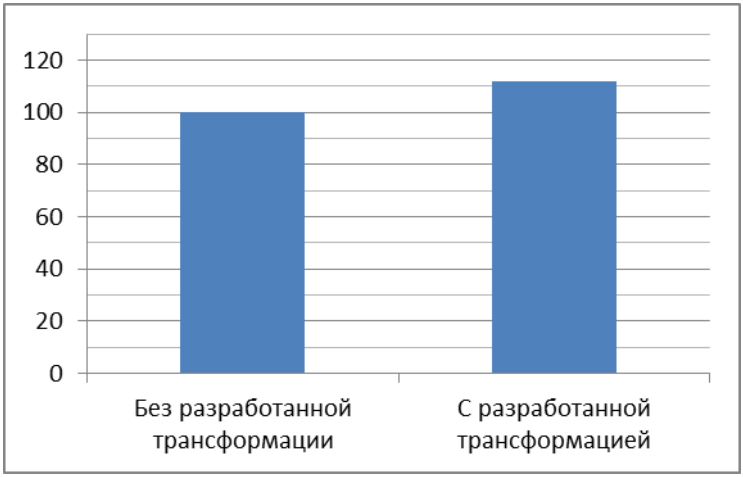


Рисунок 15. Сравнительная эффективность, %.

Диаграмма показывает увеличение эффективности обфускации программы в результате включения в метод разработанной трансформации и об эффективности метода в целом.

Разработанная трансформация также удовлетворяет требованиям, выдвинутым в разделе 1.3:

- Увеличивает размер программы не более, чем на 30%. Отмечено увеличение размера программы на 10%.
- Снижает производительность не более, чем на 20%. Отмечено снижение производительности программы на 10%.
- Поддерживает средства интроспекции;
- Применима к программам для операционной системы Android.

Заключение

Цель настоящей работы заключается в создании метода защиты Java приложений от статического анализа байт-кода.

Для достижения указанной цели перед работой были поставлены следующие задачи:

- Проанализировать существующие подходы к защите Java приложений, их слабые и сильные стороны;
- Разработать новый метод защиты и реализовать его на практике;
- Сформировать метрики, позволяющие судить об эффективности разработанного метода по сравнению с уже существующими;
- Описать ограничения и возможные проблемы, возникающие при применении разработанного метода.

При решении первой задачи в работе были освещены основные подходы к решению задачи защиты от статического анализа, выявлены их достоинства и недостатки. Также были сформулированы требования к разрабатываемому методу:

- низкие затраты ресурсов, таких как память и процессорное время;
- возможность использования интроспекции в обфусцированной программе;
- соблюдение соглашений об именованиях, принятых в Java;

- применимость для приложений, написанных для операционной системы Android.

Для создания метода, удовлетворяющего заявленным требованиям, была описана и реализована на практике новая трансформация. Эта трансформация и некоторый набор уже существующих трансформаций вошли в новый метод. Также были представлены некоторые ограничения на применение метода и описаны возможные проблемы, с которые могут возникнуть при его использовании.

Следующим шагом стала разработка метрик для оценки эффективности. Применение данных метрик показало, что представленный метод является более эффективным, чем метод, не использующий разработанную трансформацию.

Таким образом, поставленные задачи решены в полном объеме, цель работы достигнута.

Возможные задачи на будущее можно сформулировать следующим образом:

- протестировать разработанный метод;
- интегрировать разработанный метод в одну из систем автоматической сборки для обеспечения удобства использования даже неподготовленным пользователем;
- разработать дополнительную функциональность, которая позволила бы пользователю в интерактивном режиме

выбирать используемые трансформации и создавать несколько версий обфусцированной программы для того, чтобы сравнить их производительность, размер и устойчивость к декомпиляции.

Список использованных источников

1. Обфускация и защита программных продуктов. CIT Forum. 2005. http://citforum.ru/security/articles/obfus/.

2. Smartphone OS Market Share, Q1 2015. 2015. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

3. А. Джок. Компиляторы, интерпретаторы и байт-код. Computerworld Россия. 2001. http://www.osp.ru/cw/2001/06/9339/.

4. А.В. Чернов. Анализ запутывающих преобразований программ. Труды Института Системного программирования РАН. 2003.

5. A. Mayrhauser, A. M. Vans. Program Understanding: Models and Experiments. Advances in Computers. 1995.

6. A Taxonomy of Obfuscating Transformations. C. Collberg, C. Thomborson и D. Low. Department of Computer Science, The University of Auckland, 1997.

7. S. Chow, Y. Gu, H. Johnson, V. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. LNCS 2200, 2001.

8. Manufctoring cheap, Resilent and Stealthy Opaque Constructs. Collberg, Christian, Thomborson, Clark и Low, Douglas. Department of Computer Science, The University of Auckland, 1998.

9. Majumdar, Anirban и Thomborson, Clark. Securing Mobile Agents Control Flow. Department of Computer Science, The University of Auckland, 2005.

10. Goldstein, Boaz, Hizkiau, Guy и Kadouri, Or. Code Obfuscation Final Report. 2011. http://webcourse.cs.technion.ac.il/236349/Spring2015/ho/WCFiles/2010-01-2.report.pdf.
11. C. Collberg, C. Thomborson. Obfuscation Tools for Software Protection. : Department of Computer Science, University of Arizona, 2000.
12. Cracking String Encryption in Java Obfuscated Bytecode. Exploit database. 2006. https://www.exploit-db.com/docs/117.pdf.
13. E.A.V Nava. Application Obfuscation. Syngress, 2010.
14. C. Collberg. Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley, 2009.
15. A. Kalinovsky. Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering. SAMS Publishing, 2004.
16. G. Nolan. Decompiling Android. Нью-Йорк Apress, 2012.
17. А.И.Аветисян. Технологии статического и динамического анализа уязвимостей программного обеспечения. Вопросы кибербезопасности. 2014 г., 3.
18. M. Hind, A. Pioli. Which pointer analysis should I use? ACM SIGSOFT International Symposium on Software Testing and Analysis, 2000 г.
19. S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997 г.

20. И.Н.Ледовских, М.Г.Бакулин. Подход к восстановлению потока управления запутанной программы. Труды Института системного программирования РАН. 2012 г., Т. 22.
21. F. Tip. A survey of program slicing techniques. Journal of Programming Languages, 1995 г.
22. E. Eilam. Reversing: Secrets of reverse engineering. : Wiley, 2005.
23. R.Giacobazzi. Code obfuscation. Defence strategies.ASP, Верона, 2009.
24. А.В.Чернов. Об одном методе маскировки программ. 2004.
25. N. Naeem, M. Batchelder, L. Hendren. Metrics for Measuring the Effectiveness of Decompilers and Obfuscators. McGill University, School of Computer Science, Sable Research Group. 2006 г. www.sable.mcgill.ca/publications.
26. N. Naeem, L. Hendren. Programmer-friendly Decompiled Java. McGill University, School of Computer Science, Sable Research Group. 2006 г. www.sable.mcgill.ca/publications.
27. G. Nolan. Decompiling Java.Apress, Нью-Йорк, 2004.
28. Д. Кнут. Искусство программирования. Москва : Вильямс, 2014. Т. 1.
29. M. Karnick. A qualitative analysis of Java obfuscation. Rowan University, Глассборо, 2006.
30. T. Lindholm. The Java Virtual Machine Specification. Java SE 7 Edition. Oracle Corporation, 2013.

31. Statistics and facts about Android. Statista. 2015. http://www.statista.com/topics/876/android/.
32. Oracle Corporation. Java Magazine. 2013.
33. M. Batchelder, L. Hendren. Obfuscating Java: the most pain for the least gain. McGill University, School of Computer Science, Sable Research Group. 2006. www.sable.mcgill.ca/publications.