Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»
Кафедра информатики

Отчет по лабораторной работе №6

Защита ПО от несанкционированного использования

Выполнил:
Брычиков Д.Д.
Проверил:
Чернявский Ю. А.

Минск 2018

Введение

В результате выполнения лабораторной работы №5 были созданы web-приложения, обеспечивающие аутентификацию пользователей, с разделяемым доступом к информационным ресурсам, которые обеспечивают защиту от основных удаленных атак.

Вместе с тем для любых коммерческих приложений не менее актуальным является решение задачи защиты авторских прав разработчика либо пользователя системы.

Одним из основных способов защиты авторских прав разработчика является запутывание (obfuscated) или обфуксация программного кода.

Обфуска́ция (от лат. obfuscare — затенять, затемнять; и англ. obfuscate — делать неочевидным, запутанным, сбивать с толку) или запутывание кода — приведение исходного текста или исполняемого кода программы к виду, сохраняющему её функциональность, но затрудняющему анализ, понимание алгоритмов работы и модификацию при декомпиляции.

Обфуксация производится в следующих целях:

1) Затруднение декомпиляции/отладки и изучения программ с целью обнаружения функциональности.

2) Затруднение декомпиляции пропритарных программ с целью предотвращения обратной разработки или обхода DRM и систем проверки лицензий.

3) Оптимизация программы с целью уменьшения размера работающего кода и (если используется некомпилируемый язык) ускорения работы.

4) Демонстрация неочевидных возможностей языка и квалификации программиста (если производится вручную, а не инструментальными средствами).

«Запутывание» кода может осуществляться на уровне алгоритма, исходного текста и/или ассемблерного текста. Для создания запутанного ассемблерного текста могут использоваться специализированные компиляторы, использующие неочевидные или недокументированные возможности среды исполнения программы. Существуют также специальные программы, производящие обфускацию, называемые обфускаторами (англ. obfuscator).

Описание запутывающих преобразований приведено в приложении.

Постановка задачи

I. Реализовать на выбор 3 метода обфуксации программного кода приложения, разработанного в рамках лабораторных работ 4,5, позволяющие защитить ПО от несанкционированного использования в следующих комбинациях:

* По одному.

* Любые 2 на выбор из трех одновременно.

* Все три одновременно.

II. Протестировать работоспособность приложения с запутанным программным кодом.

III. Проверить и пояснить 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.

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.

Технологии

На уровне исходных текстов
На JavaScript, VBScript и подобных скрипт-языках пользователю доступен исходный текст программы. В этом случае форматированием текста и заменой имён можно сделать текст менее читаемым.

Исходный текст:

```
 int COUNT = 100;
 float TAX_RATE = 0.2;
 for (int i=0; i<COUNT; i++)
 {
   tax[i] = orig_price[i] * TAX_RATE;
   price[i] = orig_price[i] + tax[i];
 }
```

Код после обфускации:

```
 for(int a=0;a<100;a++){b[a]=c[a]*0.2;d[a]=c[a]+b[a];}
```

На уровне машинного кода
Основная статья: Машинный код
Как правило, обфускация на уровне машинного кода уменьшает скорость выполнения и соответственно увеличивает время выполнения программы. Поэтому она применяется в критичных к безопасности, но не критичных к скорости местах программы, таких как проверка регистрационного кода[1].

Простейший способ обфускации машинного кода — вставка в него недействующих конструкций (таких как or ax, ax)

На уровне промежуточного кода
Основная статья: Байт-код
В отличие от обычных языков программирования, таких как C++ или Паскаль, компилирующихся в машинный код, язык Java, NetP и языки платформы .NET компилируют исходный код в промежуточный код (байт-код), который содержит достаточно информации для адекватного восстановления исходного кода. По этой причине для этих языков применяется обфускация промежуточного кода.

Назначение

Усложнение исследования кода
Как было сказано выше, декомпиляция программ Java и .NET достаточно проста. В этом случае обфускатор оказывает неоценимую помощь тем, кто хочет скрыть свой код от посторонних глаз. Зачастую после обфускации декомпилированный код повторно не компилируется.

Обфускация HTML помогает спамерам: на почтовом клиенте, который способен отображать HTML, текст читается, но антиспам-фильтр, который имеет дело с исходным HTML-файлом, пропускает нежелательное сообщение, не распознавая в нём запретной строки.

Простейший пример обфусцированного HTML:

Машина
При просмотре пользователь увидит слово «Машина», в то время как в исходном коде оно расчленено и воспринимается как два раздельных слова.

Оптимизация
В интерпретируемых языках обфусцированный код занимает меньше места, чем исходный, и зачастую выполняется быстрее, чем исходный. Современные обфускаторы также заменяют константы числами, оптимизируют код инициализации массивов, и выполняют другую оптимизацию, которую на уровне исходного текста провести проблематично или невозможно.

Проблема уменьшения размера важна, например, при программировании для сотовых телефонов на J2ME, где размер программы серьёзно ограничен. Обфускация JavaScript уменьшает размер HTML-файлов и, соответственно, ускоряет загрузку.

Недостатки

Потеря гибкости кода
Код после обфускации может стать более зависимым от платформы или компилятора.

Трудности отладки
Обфускатор не даёт постороннему выяснить, что делает код, но и не даёт разработчику отлаживать его. При отладке приходится отключать обфускатор.

Недостаточная безопасность
Хотя обфускация помогает сделать распределённую систему более безопасной, не стоит ограничиваться только ею. Обфускация — это безопасность через неясность. Ни один из существующих обфускаторов не гарантирует сложности декомпиляции и не обеспечивает безопасности на уровне современных криптографических схем. Вполне вероятно, что эффективная защита невозможна (по крайней мере в некотором конкретном классе решаемых задач).

Ошибки в обфускаторах
Современный обфускатор — сложный программный комплекс. Зачастую в обфускаторы, несмотря на тщательное проектирование и тестирование, вкрадываются ошибки. Так что есть ненулевая вероятность, что прошедший через обфускатор код вообще не будет работать. И чем сложнее разрабатываемая программа, тем больше эта вероятность.

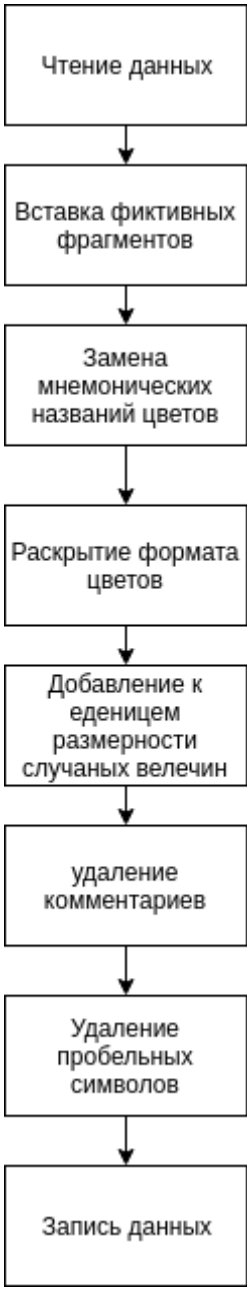
Вызов класса по имени
Большинство языков с промежуточным кодом могут создавать или вызывать объекты по именам их классов. Современные обфускаторы позволяют сохранить указанные классы от переименования, однако подобные ограничения сокращают гибкость программ.

Демонстрация работы программы

```
/* сброс стилей браузера */
html, body, div, span, object, iframe, h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code, del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var, b, u, i, center, dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td, article, aside, canvas, details,
embed,
figure, figcaption, footer, header, hgroup, menu, nav, output, ruby, section,
summary,
time, mark, audio, video {
  margin: 0;
  padding: 0;
  border: 0;
  font-size: 100%;
  vertical-align: baseline; }

article, aside, details, figcaption, figure, footer, header, hgroup, menu, nav,
section {
  display: block; }

body {
  line-height: 1.2; }

ol {
  padding-left: 1.4em;
  list-style: decimal; }

ul {
  padding-left: 1.4em;
  list-style: square; }

table {
  border-collapse: collapse;
  border-spacing: 0; }

/* конец кода сброса стилей браузера */
body {
  background: url("../images/background.png") repeat; }

.pageWrapper {
  max-width: 1000px;
  margin: 0 auto;
  background: rgba(255, 255, 255, 0.8);
  border-radius: 50px;
  box-shadow: 0 0 10px 5px rgba(30, 30, 30, 0.8); }

header {
  display: flex;
  color: white;
  width: 100%;
  justify-content: space-around;
  align-items: center;
  background: rgba(50, 50, 50, 0.9);
  position: fixed;
  top: 0;
  left: 0;
  height: 70px; }
  header .logo {
    float: left;
    box-sizing: border-box;
    height: 80%; }
  header h1 {
```

```
html, body, div, span, object, iframe, h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code, del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var, b, u, i, center, dl, dt, dd, ol, ul, li,
fieldset, form, label, legend, table, caption, tbody, tfoot, thead, tr, th, td,
article, aside, canvas, details, embed, figure, figcaption, footer, header, hgroup,
menu, nav, output, ruby, section, summary, time, mark, audio, video { margin: 0;
padding: 0; border: 0; font-size: 99.74977%; vertical-align: baseline; } article,
aside, details, figcaption, figure, footer, header, hgroup, menu, nav, section {
display: block; } body { line-height: 1.2; } ol { padding-left: 1.395164em; list-
style: decimal; } ul { padding-left: 1.395164em; list-style: square; } .register {
width: 19.90772px; height: 19.90772px;} .register a{ text-decoration: none; border-
radius: 4.971596px; color: rgba(225, 32, 0, 0.8);} table { border-collapse:
collapse; border-spacing: 0; } body { background: url("../images/background.png")
repeat; } .pageWrapper { max-width: 997.6924px; margin: 0 auto; background:
rgba(255, 255, 255, 0.8); border-radius: 49.86918px; box-shadow: 0 0 9.949579px
4.971596px rgba(30, 30, 30, 0.8); } header { display: flex; color: #FFFFFF; width:
99.74977%; justify-content: space-around; align-items: center; background: rgba(50,
50, 50, 0.9); position: fixed; top: 0; left: 0; height: 69.48974px; } header .logo {
float: left; box-sizing: border-box; height: 79.58874%; } header h1 { text-
transform: uppercase; font-size: 2.98928em; } header h1 a { color: #FFFFFF; text-
decoration: none; } header .login { float: right; margin: auto 0; font-size:
1.486396em; } header .login span { text-transform: uppercase; margin-right:
-9.933818px; } header .login a { color: #FFFFFF; border: 3.993077px solid
rgba(30,130,170,1); border-radius: 14.86352px; padding: 4.971596px 9.949579px;
margin-left: 14.86352px; text-decoration: none; } header .login a:hover {
background: rgba(30,130,170,1); } main .login { width: 19.90772px; height:
19.90772px; background: rgba(224, 134, 124, 0.5); color: rgba(200, 62, 255, 0.8);}
main { order: 1; padding: 19.90772px 49.86918px; margin-top: 99.16387px; } main .new
{ border-radius: 19.90772px; padding: 0 79.4074px 29.96616px 79.4074px; margin:
49.86918px 19.90772px; box-shadow: 0 0 9.949579px; background: rgba(200, 200, 200,
0.1); } main .new h2 { text-align: left; font-size: 2.478261em; color: rgba(0, 0, 0,
0.7); margin-top: 9.949579px; text-shadow: 1.986934px 1.986934px 2.998638px rgba(0,
0, 0, 0.3); } main .new img { margin: 29.96616px auto; border-radius: 19.90772px;
display: block; box-shadow: 0 0 29.96616px; } main .new .shorttext { font-size:
1.486396em; margin-bottom: 19.90772px; } main .new .time { float: right; font-size:
1.486396em; color: rgba(30,130,170,1); } main .new a { font-size: 1.486396em; } main
.details { padding: 0 49.86918px; } main .details h1 { font-size: 4.989344em; color:
rgba(100, 100, 100, 0.9); text-align: center; } main .details img { margin:
49.86918px auto; border-radius: 19.90772px; display: block; box-shadow: 0 0
29.96616px; } main .details .text { font-size: 1.486396em; margin-bottom:
29.96616px; } main .details .time { float: right; font-size: 1.486396em; color:
rgba(30,130,170,1); } main .details a { font-size: 1.486396em; margin-right:
49.86918px; } main .comments { padding: 0 49.86918px; margin-top: 49.86918px; } main
.comments h2 { font-size: 1.982102em; color: rgba(100, 100, 100, 0.9); text-align:
left; } main .comments .comment { margin: 39.81624px 0; border: 1.986934px solid
rgba(30, 130, 170, 0.8); border-radius: 9.949579px; box-shadow: 0 0 4.971596px
rgba(30, 130, 170, 0.8); } main .comments .comment .name { font-size: 1.486396em;
background: rgba(30, 130, 170, 0.8); border-radius: 4.971596px 4.971596px 0 0;
padding: 9.949579px; font-size: 1.486396em; } main .comments .comment .text { font-
size: 1.486396em; margin: 9.949579px 0; padding: 9.949579px; } main .comments
.comment .time { font-size: 1.29714em; color: rgba(30,130,170,1)
 padding: 9.949579px; } main .comments .comment a { font-size: 1.486396em; padding:
```

Схема работы программы



Вывод

В ходе данной работы были рассмотрены основные методы защиты исходного кода от несанкционированного использования. Было реализовано программное средство, выполняющее запутывание кода.

Запутывание кода проводиться в несколько этапов. Существует несколько видов преобразования кода, которые ухудшают его читаемость и усложняют его несанкционированное использование.

Преобразование форматирования позволяют значительно снизить читаемость кода, однако данный вид запутывания можно обратить.

Изменения формата данных позволяют защитить данные, которое использует приложение. Обычно данные приводят к максимально непрезентабельному виду.

Преобразования потока управления являются одним из самых сложны методов преобразования. Хорошие преобразования потока нужно проводить вручную. Однако, при грамотном использовании, данный метод позволяет сильно сбить с толку злоумышленника.

На практике применяется комплексное решение, реализующее несколько методов запутывания.

Исходный код

```
import re

s = open('out.css').read()

s = re.sub('/\*.*?\*/', '', s)
s = re.sub('\s+', ' ', s)

open('out.css', 'w').write(s)
import re

s = open('styles.css').read()

s = s.replace('\n\n', '\n', 4)

s = s.replace('\n\n',
"""\n
.register {
  width: 20px;
  height: 20px;}
  .register a{
    text-decoration: none;
    border-radius: 5px;
    color: rgba(225, 32, 0, 0.8);}\n"""
  , 1
)

s = s.replace('\n\n', '\n', 4)


s = s.replace('\n\n',
"""\n
main .login {
    width: 20px;
    height: 20px;
    background: rgba(224, 134, 124, 0.5);
    color: rgba(200, 62, 255, 0.8);}"""
    , 1
)

open('out.css', 'w').write(s)
import re
import random

s = open('out.css').read()

s = s.replace('white', '#FFFFFF')
s = s.replace('black', '#000000')
s = s.replace('red', '#FF0000')
s = s.replace('green', '#00FF00')
s = s.replace('blue', '#0000FF')

for st in re.findall('#[0-9a-f]{6}', s):
    r = int(st[1:3], 16)
    g = int(st[3:5], 16)
    b = int(st[5:7], 16)
    s = re.sub(st, 'rgba({},{},{},1)'.format(r,g,b), s)


for st in re.findall(r'([^h.0-9-])(-?\d+)(\.\d+)?(.)', s):
    ss = st[1] + st[2]
    a = float(ss) #* (0.99 + random.random() / 100)
    aa = float(ss) * (0.99 + random.random() / 100)
    if (st[3] == 'e' or st[3] == 'p' or st[3] == '%'):
        repl = '{}{:.7}{}'.format(st[0], aa, st[3])
        s = s.replace(st[0]+ss+st[3], repl)


open('out.css', 'w').write(s)
```