Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Лабораторная работа № 6

**Защита ПО от несанкционированного использования**

Выполнил студент гр. 953501

Койко Р.С.

Проверил

Протько М.И.

Минск 2022

# 1. Введение

В данной работе рассмотрены методы обфускации (запутывания) исходного кода программного обеспечения, нацеленные на затруднение анализа исходного кода и его несанкционированного использования/модификации. Также в ходе данной работы были реализованы методы обфускации исходного кода.

# **2.** **Краткие теоретические сведения**

**1. Описание запутывающих преобразований**

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

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

**1.1. Преобразования форматирования**

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

Удаление комментариев и переформатирование программы применимы, когда запутывание выполняется на уровне исходного кода программы. Эти преобразования не требуют только лексического анализа программы. Хотя удаление комментариев - одностороннее преобразование, их отсутствие не затрудняет сильно обратную инженерию программы, так как при обратной инженерии наличие хороших комментариев к коду программы является скорее исключением, чем правилом. При переформатировании программы исходное форматирование теряется безвозвратно, но программа всегда может быть переформатирована с использованием какого-либо инструмента для автоматического форматирования программ (например, indent для программ на Си).

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

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

**1.2. Преобразования потока управления**

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

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

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

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

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

Переменная *v*является *непрозрачной*, если существует свойство http://citforum.ru/security/articles/analysis/pi.gif 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.

**Внесение недостижимого кода** (adding unreachable code). Если в программу внесены непрозрачные предикаты видов *PF* или *PT*, 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 http://citforum.ru/security/articles/analysis/12.gif 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 sin2x+cso2x 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.

**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.

**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.

**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.

# 3. Результаты выполнения программы

/\* Any line comment

another line \*/

function test\_attach\_click (prnt) {

var elements = prnt.get\_elements\_by\_tag\_name("div") // One line comment

for(var i = 0; i < elements.length; i++) {

elements[i].onclick = function() {

alert("click on " + this.number)

}

elements[i].number = ii

}

}

function resize (to\_width, to\_height, save\_proportions, animate) {

// default value

save\_proportions = save\_proportions || true

animate = animate || true

to\_height = to\_height || true

}

function is\_palindrome (my\_str){

var str\_len = my\_str.length,

str\_reverse = my\_str.split('').reverse().join('');

if (str\_reverse == my\_str) {

return "yes";

} else {

return "no";

}

}

function check\_age(age) {

if (age > 18) {

return true;

} else {

return confirm("Success?");

}

}

;function cjLBoNYnpMjM (pENzQCCOwm) {; var elements = pENzQCCOwm.get\_elements\_by\_tag\_name("div") ; for(var UoGed = 0; UoGed < elements.length; UoGed++) {; elements[UoGed].onclick = function() {; alert("click on " + this.dGCrHFyEiqfk); }; elements[UoGed].dGCrHFyEiqfk = tIhMItdbRe; };};function rnHrdz (kInEge, RRQZdXU, pKtNgztilyVvt, DhKRNKuofSbbqjq) {; ; pKtNgztilyVvt = pKtNgztilyVvt || true; DhKRNKuofSbbqjq = DhKRNKuofSbbqjq || true; RRQZdXU = RRQZdXU || true;};function bDHNzpbLTWvSKS (sNghlcKkvvtRq){; var wQPZqsHGQiFL = sNghlcKkvvtRq.length, mlfVmPpsoWeZ = sNghlcKkvvtRq.split('').reverse().join('');; if (mlfVmPpsoWeZ == sNghlcKkvvtRq) {; return "yes";; } else {; return "no";; };};function OYlrNTPlxPy(BRzzdwAPrbYJI) {; if (BRzzdwAPrbYJI > 18) {; return true;; } else {; return confirm("Success?");; };};

# **4. Код программы**

import random

import re

import string

import info

def remove\_comments(tokens):

new\_tokens = []

comment\_view = 0

for token in tokens:

if comment\_view == 1 and token == '\n' or comment\_view == 2 and token == '\*/':

comment\_view = 0

if comment\_view == 0:

if token == '//':

comment\_view = 1

elif token == '/\*':

comment\_view = 2

elif token != '\*/':

new\_tokens.append(token)

tokens[:] = new\_tokens[:]

def remove\_debug\_info(tokens):

new\_tokens = []

is\_console, brackets\_cnt = False, 0

for token in tokens:

if is\_console:

if token == '(':

brackets\_cnt += 1

if token == ')':

brackets\_cnt -= 1

if brackets\_cnt == 0:

is\_console = False

elif token == 'console':

is\_console = True

else:

new\_tokens.append(token)

tokens[:] = new\_tokens[:]

def rename\_variables(tokens):

code\_variables = set()

for token in tokens:

if (

re.match(r'\b[A-Za-z\_]\w\*', token, re.IGNORECASE) and not re.match(r'\".\*\"', token)

and token not in info.keywords

):

code\_variables.add(token)

new\_variables = [''.join([random.choice(string.ascii\_letters) for \_ in range(random.randint(5, 15))])

for \_ in range(len(code\_variables))]

replacing\_dict = dict(zip(code\_variables, new\_variables))

new\_tokens = []

for token in tokens:

if token in replacing\_dict:

new\_tokens.append(replacing\_dict[token])

else:

new\_tokens.append(token)

tokens[:] = new\_tokens[:]

def rm\_consecutive(tokens, c):

"""

Remove consecutive spaces from token list.

Leave only one when sequence in encountered.

"""

new\_tokens = []

space\_removal\_mode = False

for token in tokens:

if token == c and not space\_removal\_mode:

new\_tokens.append(token)

space\_removal\_mode = True

elif token == c and space\_removal\_mode:

continue

else:

if space\_removal\_mode:

space\_removal\_mode = False

new\_tokens.append(token)

tokens[:] = new\_tokens[:]

def despaceify(tokens):

"""

Remove consecutive spaces from token list.

Leave only one when sequence in encountered.

"""

rm\_consecutive(tokens, ' ')

def denewlineify(tokens):

"""

Remove consecutive '\n' leaving only one, then replace

'\n' tokens with ';'.

"""

rm\_consecutive(tokens, '\n')

new\_tokens = []

for token in tokens:

if token == '\n':

token = ';'

new\_tokens.append(token)

tokens[:] = new\_tokens

# 5. Выводы

В данной работе были рассмотрены методы обфускации (запутывания) исходного кода программного обеспечения, нацеленные на затруднение анализа исходного кода и его несанкционированного использования/модификации. Также в ходе данной работы были реализованы методы обфускации исходного кода.