**Содержание**:

[1. Введение 2](#__RefHeading___Toc779_1887482898)

[2. Краткие теоретические сведения 3](#__RefHeading___Toc781_1887482898)

[3. Блок-схема алгоритма 7](#__RefHeading___Toc783_1887482898)

[4. Демонстрация работы программы 8](#__RefHeading___Toc785_1887482898)

[5. Вывод 12](#__RefHeading___Toc787_1887482898)

[Приложение 1. Исходный код программы 13](#__RefHeading___Toc789_1887482898)

# **Введение**

1). 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:

1.1) По одному.

1.2) Любые 2 на выбор из трех одновременно.

1.3) Все три одновременно.

2). Протестировать работоспособность приложения с запутанным программным кодом.

3). Проверить и пояснить следующие свойства, которым должна удовлетворять запутанная программа:

3.1) Запутывание должно быть замаскированным. То, что к программе были применены запутывающие преобразования, не должно бросаться в глаза.

3.2) Запутывание не должно быть регулярным. Регулярная структура запутанной программы или её фрагмента позволяет человеку отделить запутанные части и даже идентифицировать алгоритм запутывания.

# **Краткие теоретические сведения**

Обфускаация 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.

«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 возможностисреды исполнения программы. Существуют также специальные программы, производящие обфускацию, называемые обфускаторами.

**2.1 Минификация**

Минификация. Этот метод позволяет заметно сократить исходный код программы, заменяя длинные человекочитаемые имена переменных на короткие, удаляет константы, сжимает код программы, удаляя пробелы, отступы и прочие спец символы которые упрощают чтение программ но не влияют на результат исполнения программы.

Данный тип обфускации на самом деле же редко используется чтобы запутать злоумышленника, зачастую так делают с целью уменьшения расходуемой памяти занимаемой самим скриптом. Самый наглядный пример - js скрипты. При использовании современного веб приложения будут в любом случае задействованы js скрипты. Для того чтобы исполнить данный скрипт на стороне клиента, его надо передать веб сервером клиенту, и соответственно получается существенное сокращение расхода трафика, если вместо читаемых кодов передавать минифицированные скрипты.

При использовании такого алгоритма обфускации невозможно восстановить исходный код программы, даже зная алгоритм, но возможно восстановить формат кода (например, известные код форматеры для c++ - uncrustify и clang-format). В лабораторной работе используется минификатор PyMinifier, который может работать в двух режимах - минификация побочных символов и минификация имён переменных. Также на примере были разобраны минифицированные js скрипты для библиотеки bootstrap.

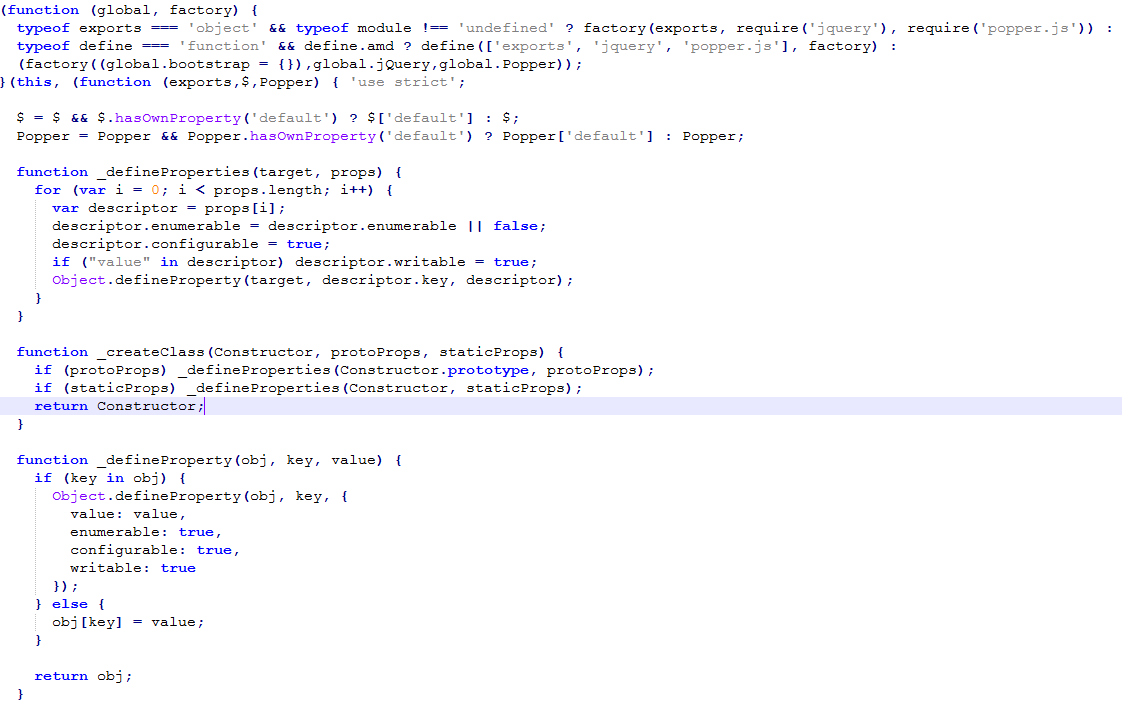


Рисунок 1.1 Исходный вариант фрагмента js библиотеки

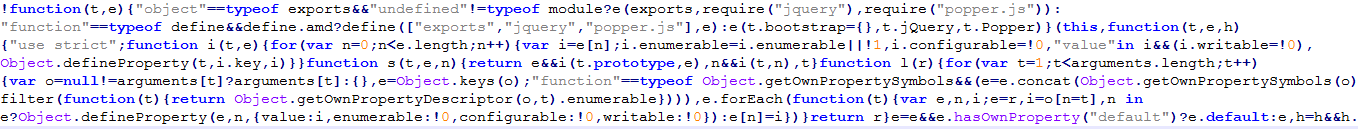


Рисунок 1.2 Минифицированный вариант фрагмента той же библиотеки

**2.2 Обфускация на уровне алгоритма**

Обфускация на уровне алгоритма - также один из популярных методов обфускации. Данный метод уже может активно применяться для запутывания злоумышленника, так как даже при восстановлении формата кода (если был использован предыдущий метод обфускации), код остаётся по прежнему практически не читаемым в некоторых местах. Один из самых простых примеров.

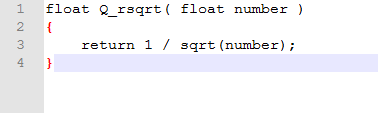
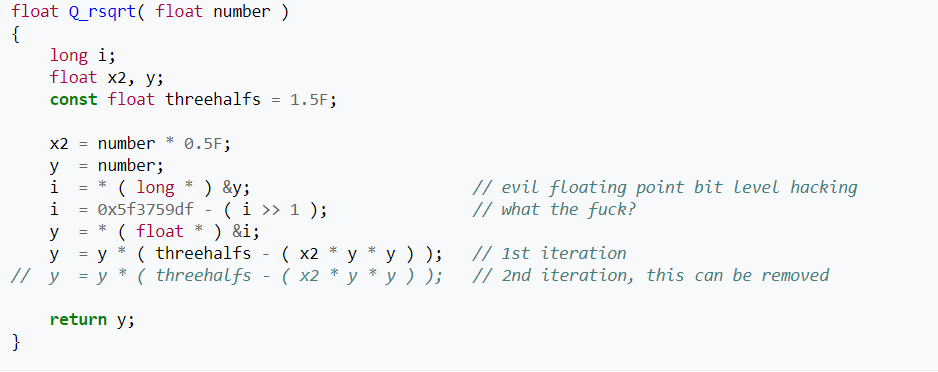


Рисунок 2.1 Обратный корень

  
Рисунок 2.2 Быстрый обратный квадратный корень

Примечание: данный пример достоверно точно нельзя назвать обфускацией, хотя и оба метода выполняют одну и ту же функцию, но учитывая время появления первого алгоритма (1999 год, вероятно разработан для игры Quake 3: Arena для подсчёта нормалей), он работал гораздо быстрее чем второй вариант, так как процессорная функция sqrt стоит сильно дороже чем вышепредставленный аналог.

Данный метод обфускации будет реализован вручную, для наглядной демонстрации данного типа на конкретном примере.

**2.3 Обфускация на уровне байт-кода**

Обфускация на уровне байт-кода. Также один из популярных методов обфускации, который позволяет менять программу уже на самом низком уровне. В случае С# и Java можно менять промежуточный код на соответствующих платформах, в случае с компилируемыми языками можно использовать специальные компиляторы которые будут вставлять лишние и запутывающие инструкции прямо в наш скомпилированный код. В случае питона можно скомпилированные в рантайме объекты (co\_objects) обфусцировать, запутывая инструкции и подставляя также лишние или более тяжело читаемые инструкции вместо старых.

Для такого метода используется библиотека Pyarmor, которая умеет:

- обфусцировать скомпилированные объекты

- подчищать локальные переменные на каждом кадре (frame) виртуальной машины

- обфусцировать код всей программы, шифруя байт-код и проверяя его на рантайме. Таким образом, исходный код python скрипта полностью шифруется и не доступен злоумышленнику без ключа.

# **Б****лок-схема алгоритма**

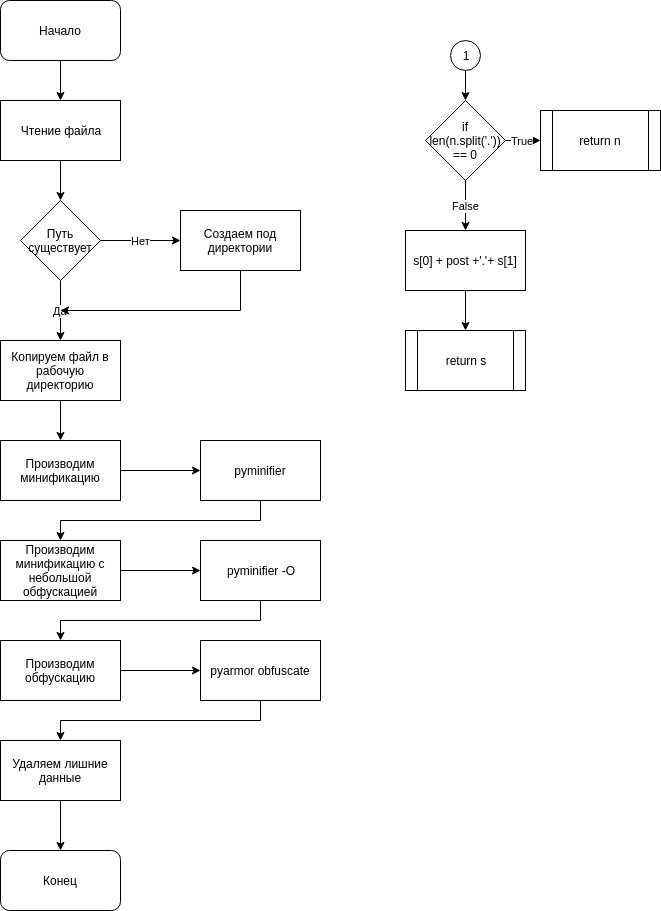


Рисунок 3.1. Блок-схема алгоритма программы

# **Демонстрация работы программы**

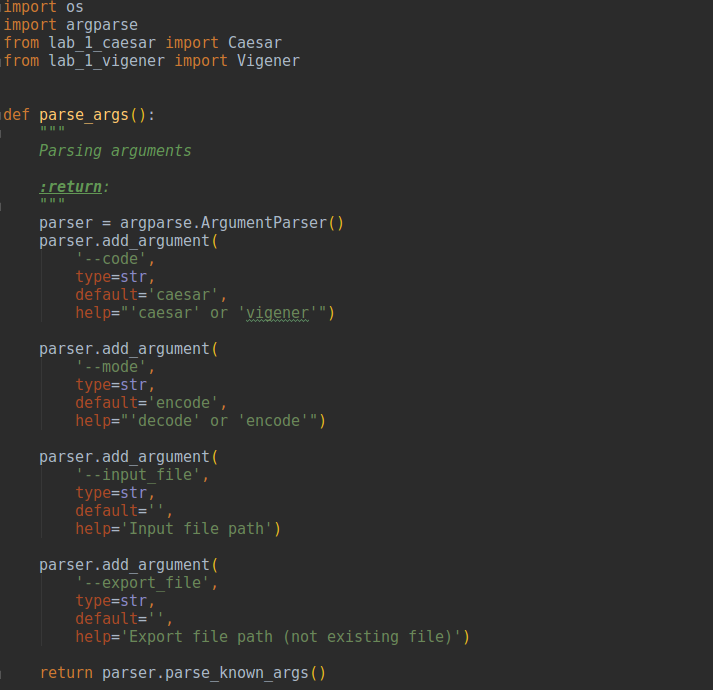


Рисунок 4.1. Неизмененный код.

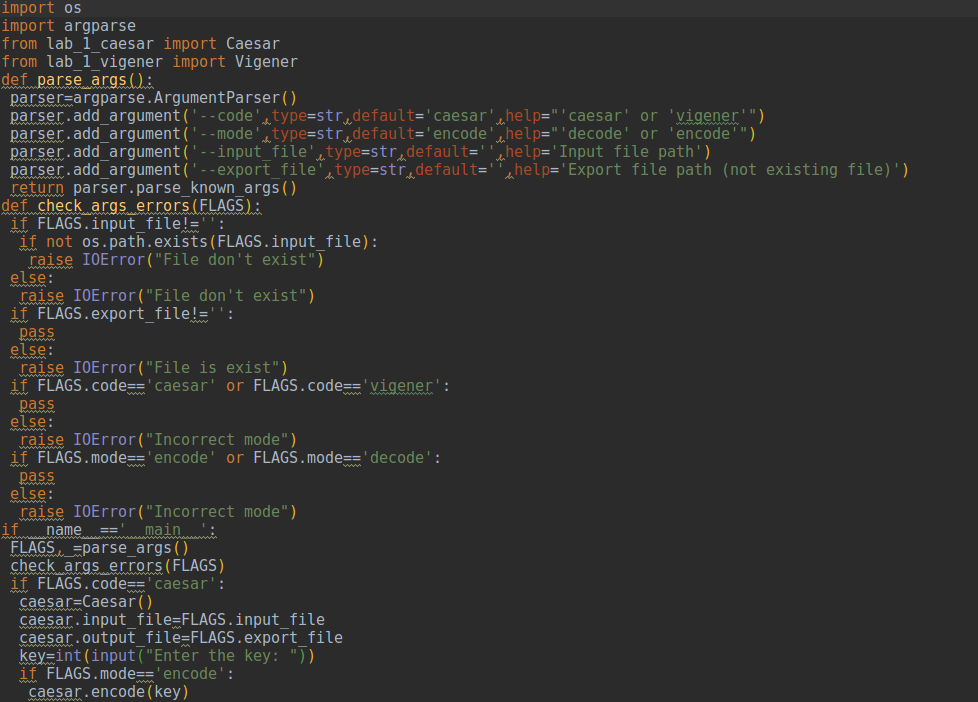


Рисунок 4.2. Минимизированный код.

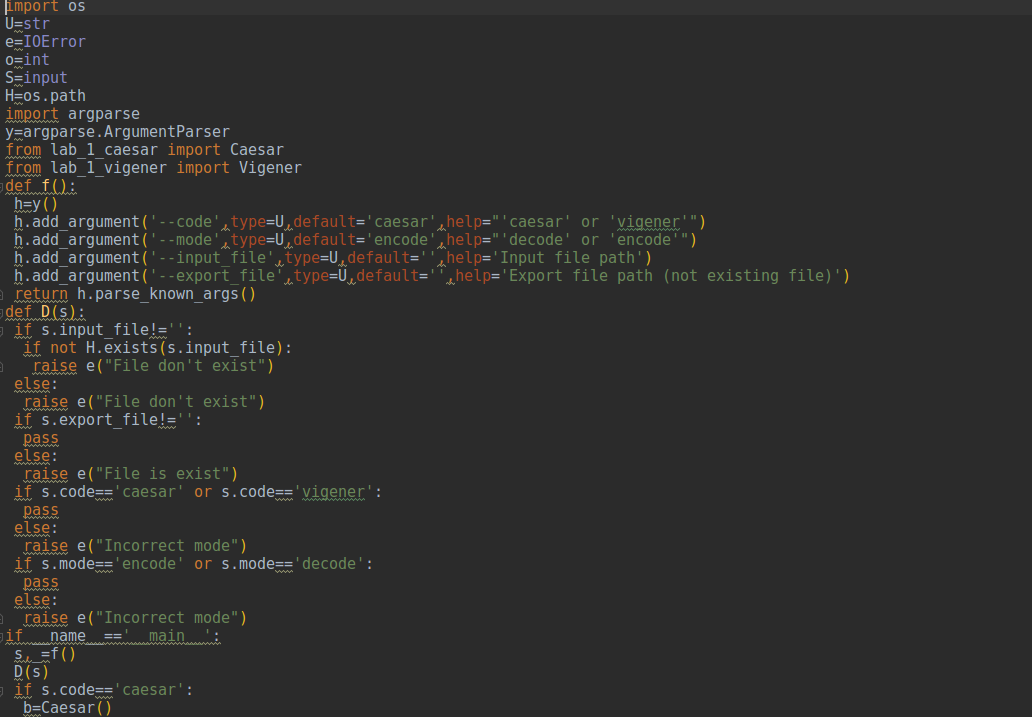


Рисунок 4.3. Минимизированный код с слабой обфускацией

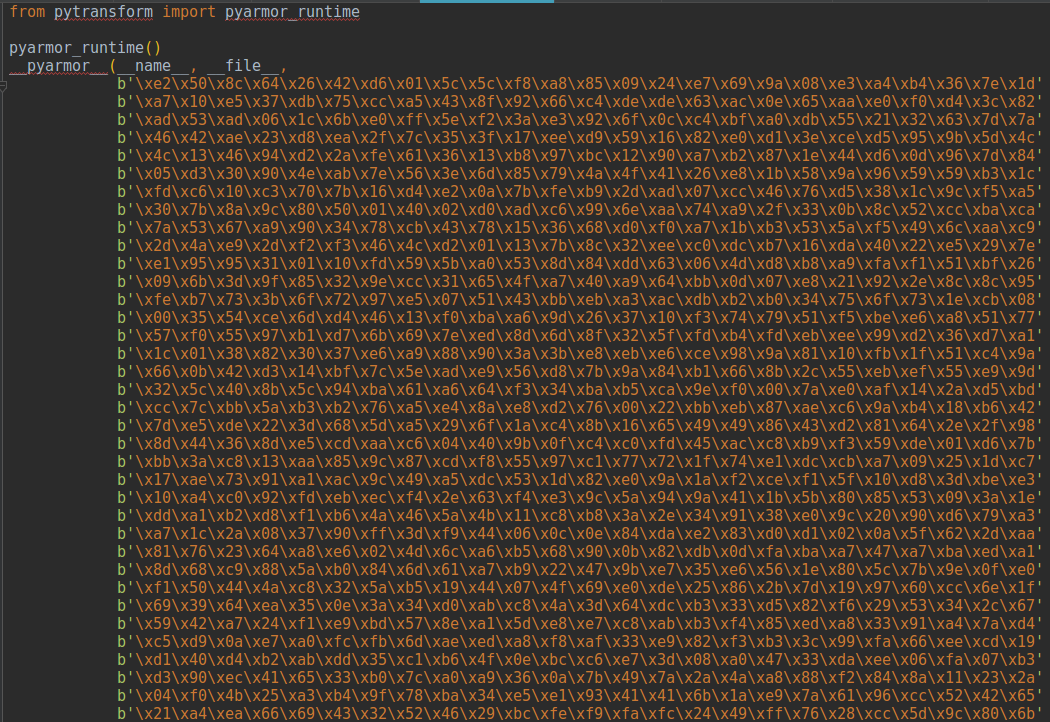


Рисунок 4.4. Сильно обфусцированный код

# **Вывод**

В ходе написания лабораторной работы были изучены методы обфускации кода а также применены на практике. Были протестированы три метода обфускации в различных комбинациях, которые могут защитить продукт в серьёзной промышленной разработке.

# **Приложение 1. Исходный код программы**

import os

import argparse

from lab\_1\_caesar import Caesar

from lab\_1\_vigener import Vigener

def parse\_args*()*:

*"""*

*Parsing arguments*

***:return****:*

*"""*

parser = argparse.ArgumentParser*()*

parser.add\_argument*(*

'--code',

type=str,

default='caesar',

help="'caesar' or 'vigener'"*)*

parser.add\_argument*(*

'--mode',

type=str,

default='encode',

help="'decode' or 'encode'"*)*

parser.add\_argument*(*

'--input\_file',

type=str,

default='',

help='Input file path'*)*

parser.add\_argument*(*

'--export\_file',

type=str,

default='',

help='Export file path (not existing file)'*)*

return parser.parse\_known\_args*()*

def check\_args\_errors*(*FLAGS*)*:

*"""*

*Checking errors in arguments*

***:param*** *FLAGS:*

***:return****:*

*"""*

if FLAGS.input\_file != '':

if not os.path.exists*(*FLAGS.input\_file*)*:

raise IOError*(*"File don't exist"*)*

else:

raise IOError*(*"File don't exist"*)*

if FLAGS.export\_file != '':

# if os.path.exists(FLAGS.export\_file):

# raise IOError("File is exist")

pass

else:

raise IOError*(*"File is exist"*)*

if FLAGS.code == 'caesar' or FLAGS.code == 'vigener':

pass

else:

raise IOError*(*"Incorrect mode"*)*

if FLAGS.mode == 'encode' or FLAGS.mode == 'decode':

pass

else:

raise IOError*(*"Incorrect mode"*)*

if \_\_name\_\_ == '\_\_main\_\_':

FLAGS, \_ = parse\_args*()*

check\_args\_errors*(*FLAGS*)*

if FLAGS.code == 'caesar':

caesar = Caesar*()*

caesar.input\_file = FLAGS.input\_file

caesar.output\_file = FLAGS.export\_file

key = int*(*input*(*"Enter the key: "*))*

if FLAGS.mode == 'encode':

caesar.encode*(*key*)*

else:

caesar.decode*(*key*)*

elif FLAGS.code == 'vigener':

vigener = Vigener*()*

vigener.input\_file = FLAGS.input\_file

vigener.output\_file = FLAGS.export\_file

key = input*(*"Enter the key: "*)*

if FLAGS.mode == 'encode':

vigener.encode*(*key*)*

else:

vigener.decode*(*key*)*