МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное

учреждение высшего образования

«Омский государственный университет им. Ф.М. Достоевского»

Кафедра информационной безопасности

УТВЕРЖДАЮ

Заведующий кафедрой ИБ,

\_\_\_\_\_\_\_\_\_\_\_ Горев А.И.

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2023 г.

Защита кода от исследования

Выпускная квалификационная работа

по направлению 09.03.01 – Информатика и вычислительная техника

Научный руководитель:

к. ю. н., доцент

\_\_\_\_\_\_\_\_\_\_ Горев А.И.

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2023 г.

Выполнил:

студент группы МИБ-941-ЗИ-01

\_\_\_\_\_\_\_\_\_\_\_ Москалев К.Н.

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2023 г.

Омск

2023

**Содержание**

[**ВВЕДЕНИЕ** 3](#_Toc132893682)

[**ГЛАВА I. ИДЕЯ И КОНЦЕПТ ОБФУСКАТОРА** 4](#_Toc132893683)

[**1.1.** **Исследование предметной области** 4](#_Toc132893684)

[**1.2** **Техническое задание на разработку обфускатора** 5](#_Toc132893685)

[**1.3** **Разбор PE структуры** 6](#_Toc132893686)

[**1.4** **Проектирование обфускатора** 8](#_Toc132893687)

[**ГЛАВА II. АРХИТЕКТУРА РАЗРАБАТЫВАЕМОГО ОБФУСКАТОРА** 10](#_Toc132893688)

[**2.1.** **Разбор Pefile и Keystone-engine** 10](#_Toc132893689)

[**2.2.** **Формат данных CodeBlock** 13](#_Toc132893690)

[**2.3.** **Словарь инструкций** 15](#_Toc132893691)

[**2.4.** **Обфускатор** 21](#_Toc132893692)

[**2.5.** **Кодогенерация** 23](#_Toc132893693)

[**ГЛАВА III. ОТЛАДКА И ИССЛЕДОВАНИЕ** 25](#_Toc132893694)

[**ЗАКЛЮЧЕНИЕ** 28](#_Toc132893695)

[**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ** 29](#_Toc132893696)

[**Приложение А** 31](#_Toc132893697)

# **ВВЕДЕНИЕ**

Обфускация – это способ защиты данных и лежащих в основе программы алгоритмов от злоумышленника, в случае получения доступа к программе или её исходным файлам.

Существует множество техник обфускации, кратко перечислим основные из них:

* Шифрование данных;
* Изменение потока управления исполняемого кода;
* Усложнение доступа к адресам;
* Ловушки для отладчика;
* Полиморфный код;
* Удаление отладочной информации и регулярное обновление ПО со сменой ключа обфускации;
* Создание неисполняемого кода с прыжками на него, которые никогда не будут исполняться;
* Идентичные или обратимые преобразования кода;
* Code flow flattening;
* Неочевидные ложные условия или зависящие от внешних данных.

Несмотря на развитие SaaS (Программное обеспечение как услуга), Обфускация по-прежнему является популярным способом защиты программного обеспечения. А в некоторых случаях - единственно возможным.

Существующие обфускаторы часто выполняют лишь лексические преобразования исходного кода, кроме того, они зависят от используемого языка. Было решено попробовать реализовать более низкоуровневое решение с большим потенциалом для обфускации и доступным для любого компилируемого языка.

# **ГЛАВА I. ИДЕЯ И КОНЦЕПТ ОБФУСКАТОРА**

# **Исследование предметной области**

Целью выпускной квалификационной работы является изучение основных методов защиты кода от исследования. Демонстрация концепта обфускатора, способного выполнять базовые операции, с помощью которых в последствии можно создать рабочий продукт для обфускации кода.

Ранее уже был создан концепт обфускатора, но он часто ломал программу и имел ужасную кодовую базу, его исходный код был переработан, было добавлено множество улучшений и настроек для удобства разработки. Например, логирование или автоматизация выполнения задач.

В процессе исследования предметной области было обнаружено довольно много различных решений для обфускации. Однако все они зависели от используемого языка программирования, были достаточно сложны и объемны, и представляли собой черный ящик, вмешаться в процесс работы которого, было бы не так-то просто. Также очень мало учебных материалов и открытых алгоритмов в открытом доступе по данному направлению.

Данный обфускатор является достаточно небольшим и простым, чтобы иметь академическую ценность. Кроме того, он показывает основы работы с Keystone-engine, поэтому любой разработчик, столкнувшийся с похожей задачей и исследовавший данную работу, сможет разобраться в предметной области быстрее.

# **Техническое задание на разработку обфускатора**

Основные требования предъявляемые к обфускации:

• Код должен быть рабочим, и результат выполнения должен соответствовать программе до обфускации;

• В исследуемой программе граф выполняемого кода должен быть больше оригинала и сложным для понимания изначального алгоритма;

• Граф прыжков исполняемого файла должен изменяться для каждого прогона обфускации;

• Код обфусцируемой программы должен содержать лишние невыполнимые инструкции;

• Код обфускатора должен быть понятным и читаемым;

• Настроенное окружение для выполнения;

• Процесс должен выполняться за приемлемое время (4 секунды);

• Настроенное окружение для выполнения;

Сама программа обфускации должна быть реализована на языке python, чтобы ускорить разработку и тестирование идей для сокрытия кода. В качестве ассемблера и дизассемблера будут выступать библиотеки Keystone и Capstone. Они имеют удобный API для работы с кодом и его байтовым представлением и используются в большинстве проектов, связанных с реверс инжинирингом.

Для чтения структуры файла и работы с секциями, а также записи изменений в файл будет использоваться библиотека pefile, которая также хорошо зарекомендовала себя, в чтении самых разных файлов, поврежденных или зараженных вирусами.

Хранение кода предполагается в структуре похожей на связный список, это позволяет перемещать и вставлять блоки кода в любое место программы, не сдвигая все последующие блоки кода.

Для генерации был использован словарь возможных инструкций, с вероятностным распределением. Должна быть возможность генерировать необходимое количество инструкций.

# **Разбор PE структуры**

Самым первым шагом для создания обфускатора будет чтение PE структуры файла. Нам нужны адреса секции .text отвечающей за расположение кода в файле, а также адрес начальной инструкции entry\_point. Но для начала необходимо прочитать заголовок IMAGE\_OPTIONAL\_HEADER, который содержит необходимую информацию для загрузки файла.

Особенно нас интересуют поля:

* AddressOfEntryPoint: DWORD — RVA адрес точки входа;
* BaseOfCode: DWORD — RVA начала кода программы (секции кода);
* BaseOfData: DWORD — RVA начала кода программы (секции данных);
* SectionAligment: DWORD — размер выравнивания секции при выгрузке в виртуальную память;
* FileAligment: DWORD — размер выравнивания внутри файла;
* SizeOfImage: DWORD — размер файла (в байтах) в памяти, включая все заголовки. Должен быть кратен SectionAligment.

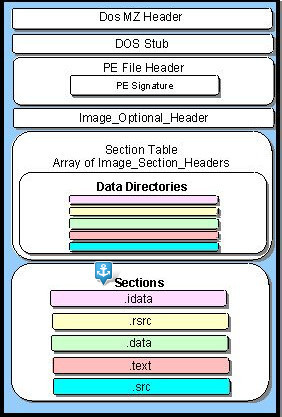


Рис. 1. Структура PE исполняемого файла.

Далее нас интересуют секции IMAGE\_SECTION\_HEADER, расположенные сразу за IMAGE\_DATA\_DIRECTORY.

Секции имеют следующие интересующие нас поля:

* Name: 8-BYTE – секция кода как правило имеет название .text;
* VirtualSize: DWORD – Размер секции в памяти, излишек заполняется нулями;
* VirtualAddress: DWORD – Первый байт секции относительно образа в памяти;
* SizeOfRawData: DWORD – Размер секции в файле. Должен быть кратен FileAligment;
* PointerToRawData: DWORD – Указатель на первую страницу секции для файла.

Адрес в виртуальной памяти называется Virtual address (VA), Предпочитаемый адрес задается в ImageBase. Если мы берем адрес относительно данного адреса, то получим Relative virtual address (RVA).

VA = ImageBase + RVA.

Что бы успешно работать с виртуальными адресами, нужно также уметь вычислять и физические.

RAW = RVA – sectionRVA + rawSection.

Для соотношения между физическим и виртуальным адресом существует выравнивание.

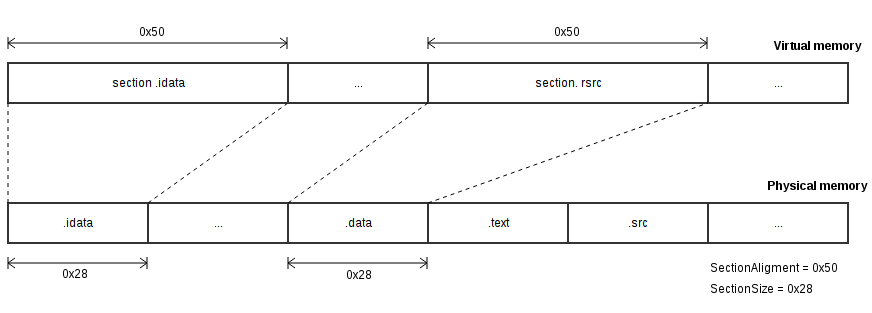


Рис. 2. Выравнивание при выгрузке программы в память

Для того что бы посчитать RAW, необходимо определить секцию, которой принадлежит RVA.

if RVA >= sectionVA and RVA < **align**(sectionVirtualSize, sectionAlignment):

Теперь мы можем обнаружить необходимую секцию. Ее VirtualAddress и будет адресом начала тела секции, который в нашем случае будет началом скомпилированного кода программы.

# **Проектирование обфускатора**

Структура взаимодействия обфускатора выглядит следующим образом:

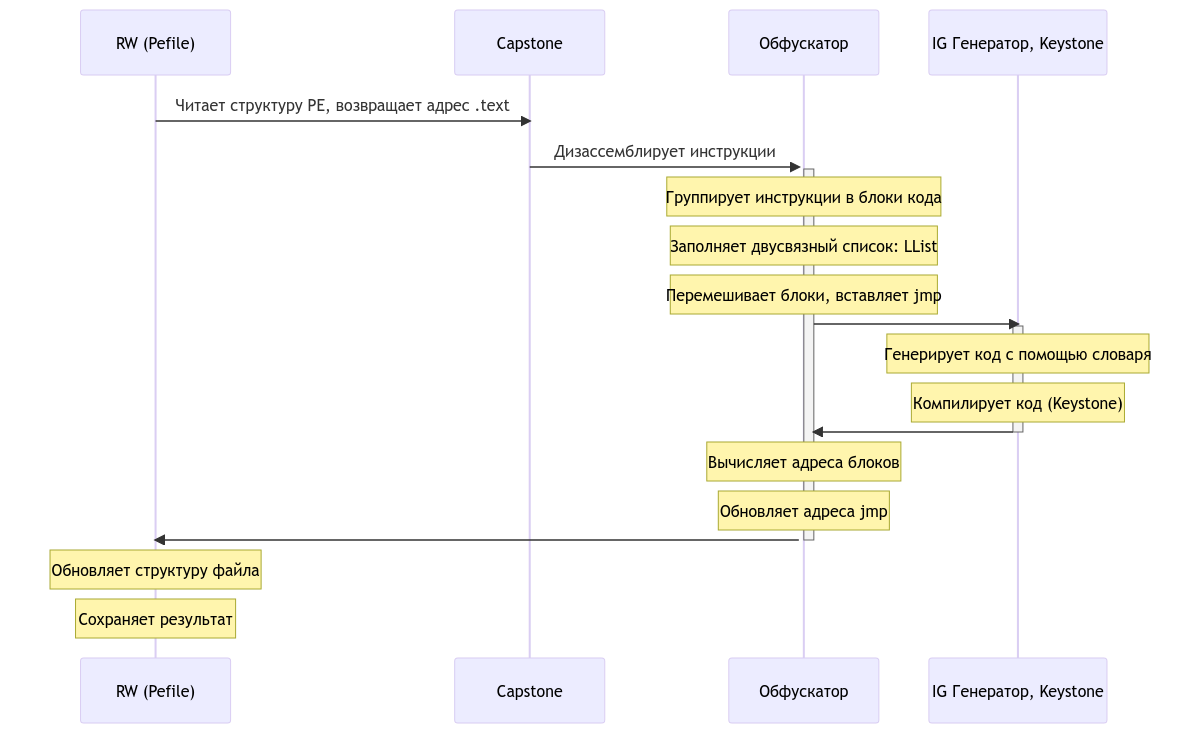


Рис. 3. Последовательность действий обфускатора

RW (Pefile) – Автоматизирует процесс определения секций, и нахождения точки входа в программу. Она возвращает адрес начало тела секции.

Далее в дело вступает Capstone, он распознает инструкции, пока не дойдет до байтов заполненных нулями «00 00», которые обозначают конец фактического кода. Он возвращает много данных касательно кода, но нам интересны прежде всего mnemonic (мнемоника инструкции), op\_str (операнды), bytes (байты скомпилированной инструкции), size (размер в байтах).

Перемещать каждую инструкцию было бы накладно, поэтому их группируем в блоки. Блоки наследуются от Node что бы получить необходимые свойства для использования в LinkedList. Кроме того, они имеют дополнительные свойства, связанные с кодом. Это основной класс для хранения нашего кода на время обфускации.

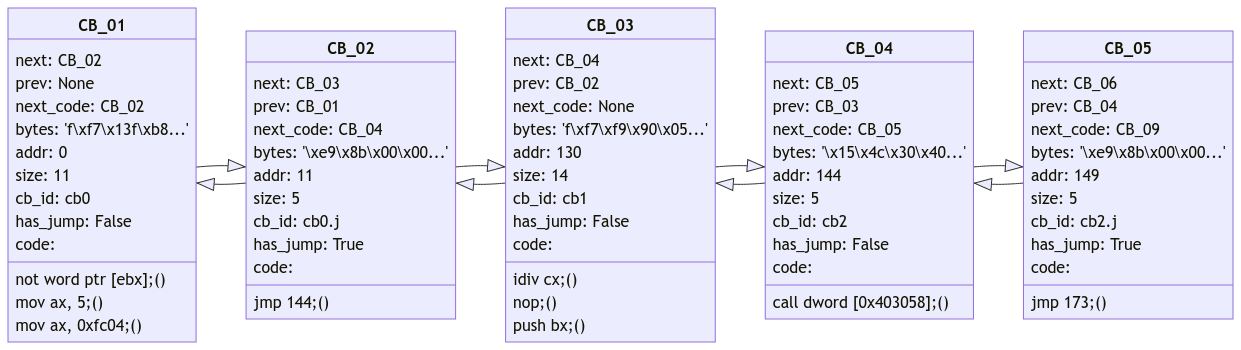


Рис. 4. Так выглядит структура для хранения кода программы

Данная структура обладает преимуществами LinkedList при записи в середину, также из-за частой работы с самими нодами, а не значениями, уходит на нет недостаток взятия произвольного элемента из списка, так как нужный элемент либо следующий по итерации, либо на него есть ссылка. В дальнейшем можно оптимизировать если создавать базовые адреса для группы блоков, тогда вместо изменения всех адресов, достаточно сдвинуть базовые адреса последующих групп, или задать им предел размера для того, чтобы небольшие изменения в коде не приводили к повторному вычислению адресов.

Сам список имеет много различных методов для управления элементами, поддерживая как базовые для связного списка, так и дополнительные, например next\_code для отслеживания потока управления исполняемого кода.

Обфускатор производит над данным списком различные преобразования. Добавляет и перемешивает блоки кода. В целом сама логика преобразований довольно несложная. Для генерации кода используется InstructionGenerator, который выбирает инструкции ориентируясь по словарю. Шансы выбора инструкции основаны на частоте появления инструкций в реальном коде. Он обращается к Keystone каждый раз, когда нужно скомпилировать последовательность байтов.

# **ГЛАВА II. АРХИТЕКТУРА РАЗРАБАТЫВАЕМОГО ОБФУСКАТОРА**

# **2.1. Разбор Pefile и Keystone-engine**

Отвечает за работу со структурой PE:

import **pefile**

import **logger** as **lg**

class **Rw**():

    def **read**(self, file\_path):

        """Парсит структуру файла, и возвращает указатель на .text"""

        self.pe = **pefile**.**PE**(file\_path)

        return self

*# imagebase Адрес базовой загрузки 40\_00\_00h для Win*

*# va Адрес относительно начала виртуальной памяти 1000h*

*# rva место выгрузки программы*

*# va = imagebase + rva*

*# raw = rva - sectionRVA + rawSection*

    def **codedata**(self):

*# AddressOfEntryPoint if guaranteed to be the first byte executed.*

        self.eop = self.pe.OPTIONAL\_HEADER.AddressOfEntryPoint

**print**('entry point is %s'%(self.eop))

        self.code\_section = None

        for section in self.pe.sections:

            if section.contains\_rva(self.eop):

                self.code\_section = section

        data = self.code\_section.get\_data(self.eop, self.code\_section.SizeOfRawData)

        return data

    def **set\_eop**(self, addr\_offset):

        """Устанавливает новую точку входа"""

        self.pe.OPTIONAL\_HEADER.AddressOfEntryPoint = self.eop + addr\_offset

    def **write**(self, file\_path, llist\_code):

        """Записывает изменения в файл"""

        flat\_code = **bytearray**()

        for node in llist\_code.forwardn():

            flat\_code.**extend**(node.bytes)

        self.pe.**set\_bytes\_at\_rva**(self.eop, **bytes**(flat\_code))

**lg**.**info**('result written at:%s'%(file\_path))

        self.pe.**write**(filename=file\_path)

rw = **Rw**()

Листинг 1. rw.py Парсер PE заголовков

Самой важной здесь является функция codedata. Вначале мы берем entryPoint (eop) из заголовка OPTIONAL\_HEADER. Pefile имеет встроенную функцию определения того принадлежит ли адрес этой секции или нет. Функция write выписывает список в массив байтов, и обновляет тело секции.

from **capstone** import **Cs**, CS\_ARCH\_X86, CS\_MODE\_64, CS\_MODE\_32, **CsError**

from **keystone** import **Ks**, KS\_ARCH\_X86, KS\_MODE\_64, KS\_MODE\_32, **KsError**

import **logging**

F32 = 0xffff\_ffff

OFFS = 0x0040\_0000

class **Inst**():

    """Класс хранящий 1 дизассемблированную инструкцию"""

    def **\_\_init\_\_**(self, mnemonic=None, ops=None, bytes=None): *#, size=None, addr=None, id=None*

        self.mnem = mnemonic

        self.ops = ops

        self.bytes = bytes

    def **get\_inst**(self):

        return '\t%s %s\t%s' % ( self.mnem, self.ops, self.bytes)

    def **\_\_str\_\_**(self):

        return '%s %s' % (self.mnem, self.ops)

    def **code**(self):

        return '%s %s\n' % (self.mnem, self.ops)

    i = **property**(**get\_inst**)

    c = **property**(**code**)

class **Compiler**(**object**):

    """Обращается к Keystone и Capstone что бы получить код"""

    def **\_\_new\_\_**(cls):

        if not **hasattr**(cls, 'instance'):

            cls.instance = **super**(**Compiler**, cls).**\_\_new\_\_**(cls)

        return cls.instance

    def **\_\_init\_\_**(self):

        self.ks = **Ks**(KS\_ARCH\_X86, KS\_MODE\_32)

        self.cs = **Cs**(CS\_ARCH\_X86, CS\_MODE\_32)

        self.cs.detail = False

    def **asm**(self, op\_str):

        """Компилирует код, что бы получить байты"""

        bytes, size = self.ks.**asm**(op\_str)

        return **bytearray**(bytes), size

    def **asmi**(self, op\_str, mnemonic = None, ops = None):

        """Компилирует код, что бы получить Instruction"""

        bytes, size = self.ks.**asm**(op\_str)

        return **Inst**(mnemonic, ops, bytes) *# size*

    def **asmu**(self, inst):

        """Компилирует изменения в Instruction"""

        bytes, size = self.ks.**asm**(**str**(inst))

        inst.bytes = bytes

*# inst.size = size*

        return inst

    def **disasm**(self, bytes, eoc=b'\00\00', offset=0x0):

        """Декомпилирует код, пока не встретит конец инструкций"""

        instructions = []

        for i in self.cs.**disasm**(bytes, offset):

            if i.bytes == eoc:

                return instructions

            instructions.**append**(**Inst**(i.mnemonic, i.op\_str, i.bytes))

        return instructions

    def **disi**(self, bytes, offset=0x0):

        """Декомпилирует набор байтов, что бы получить Instruction"""

        i = **next**(self.cs.**disasm**(bytes, offset))

        return **Inst**(i.mnemonic, i.op\_str, i.bytes)

    def **asmdis**(self, op\_str, eoc=b'\00\00', offset=0x0):

        """Компилирует код, и сразу же разбирает на Instructions"""

        return self.**disasm**(self.**asm**(op\_str), eoc, offset)

    def **asmdis1**(self, op\_str, eoc=b'\00\00', offset=0x0):

        """Компилирует код, и сразу же создает Instruction"""

        return self.**disi**(self.**asm**(op\_str), eoc, offset)

c = **Compiler**()

Листинг 2. compiler.py ассемблер/дизассемблер кода

Данный класс Compiler представляет собой обертку над Keystone и Capstone. Он возвращает массив Inst при декомпиляции, так как изначально предполагалось, что работа будет идти именно с ними, и из-за удобного обращения к свойствам. В дальнейшем можно воспользоваться массивом объектов самого Keystone. Или возвращать сразу в формате CodeBlock.

Сам проект Keystone-engine состоит из трех модулей:

* Unicorn – Интерактивный отладчик для кода, способен эмулировать архитектуру отличную от текущей.
* Capstone – Дизассемблер инструкций, поддерживает разные форматы исполняемых файлов. Отличительной особенностью является то, что он позволяет дизассемблировать файл постепенно по инструкции за инструкцией.
* Keystone – Ассемблер, поддерживает различные архитектуры и синтаксис. Ему не обязательно давать целый файл, он вполне может быстро скомпилировать произвольную последовательность байтов любой длинны.

Все они имеют простую настройку и инициализацию, с ними удобно работать на ходу. Из минусов, нет доступа к внутренним процессам данного решения, например возможности использовать их словарь инструкций или формат хранения данных, или использовать их внутренние функции для более быстрой и точной модификации кода.

# **2.2. Формат данных CodeBlock**

Данный класс отвечает за хранение данных, наследуется от класса Node. Блоки содержат следующие важные поля:

* next/prev – Ссылки на следующий/предыдущий блок в порядке следования. Необходимы для итерации по списку, и поддержания связности структуры;
* bytes – Байты блока кода;
* addr – Адрес начала блока кода, нужны для определения целевого адреса прыжка;
* has\_jump – Имеет ли блок инструкцию прыжка, нужно ли высчитывать адрес прыжка;
* code – Текстовое представление кода, нужно для обновления инструкций;
* used\_regs – Используемые регистры, нужны в случае создания ложного потока кода, что бы не смешивать используемые регистры;
* next\_code – Следующий для исполнения блок кода, поток управления. В основном необходим для прыжков, чтобы указать смещение.

from **llist** import **Node**

class **CodeBlock**(**Node**):

    """Класс, который отвечаеет за хранение кода в списке"""

    def **\_\_init\_\_**(self, insts, next\_code=None, has\_jump=False, cb\_id = None, \*args, \*\*kw):

*# Ссылка на Codeblock, который должен исполняться следующим*

*# Либо следующий блок должен быть следующим в списке,*

*# либо иметь прыжок, что бы добраться до следующего блока*

*# Иначе нарушится ход исполнения программы*

        self.next\_code = next\_code

*# Скомпилированные байты*

        self.bytes = **bytearray**()

*# Текстовое представление кода*

        self.code = ''

*# Имеет ли данный код инструкцию прыжка, для которой нужно*

*# вычислять адрес*

        self.has\_jump = has\_jump

*# Адрес данного Codeblock. addr = sum(all\_prev\_CB.size)*

        self.addr = 0

        if not (**type**(insts) is **list**):

            insts = [insts]

*#print(insts)*

        for i in insts:

*#print(i)*

            self.code += i.c

            self.bytes.**extend**(i.bytes)

*#print(i.bytes)*

*#print(self.bytes)*

*# Размер данного Codeblock.*

        self.size = **len**(self.bytes)

*# Поскольку наследуется от Node, имеет следующие свойства:*

*# value = bytes, prev и next предыдущий и следующий Codebock в списке*

        self.cb\_id = cb\_id

        self.used\_regs = []

**super**().**\_\_init\_\_**(**len**(self.bytes), \*args, \*\*kw)

**@classmethod**

    def **chain**(that, prev, insts, next\_code=None, has\_jump=False, cb\_id = None, \*args, \*\*kw):

        cb = **CodeBlock**(insts, next\_code=None, has\_jump=False, cb\_id=cb\_id, \*args, \*\*kw)

        prev.next\_code = cb

        return cb

    def **set\_bytes**(self, bytes):

        self.bytes = bytes

        self.size = **len**(bytes)

    b = **property**(fset=**set\_bytes**)

    def **is\_need\_to\_calc\_addr**(self):

        """Определяет необходимость вычисления адреса"""

*# Имеет прыжок, вычислеение адрееса необходимо*

        if self.has\_jump:

            return 1

*# Нет прыжка, и следующий блок тот, который должен исполнятся следующим*

        if self.next\_code == self.next:

            return 0

*# Нет прыжка, и следующий блок не является следуующим по исполнению*

        return -1

    def **insertcf**(self, prev):

        """Вставляет данный блок в поток кода"""

        if prev:

            self.next\_code = prev.next\_code

            prev.next\_code = self

    def **\_\_str\_\_**(self):

        return "cb: " + self.code

Листинг 3. codeblock.py хранит код и доп. информацию

Функция определения необходимости прыжка возвращает 1 в случае наличия инструкции jump у данного кода. 0 в случае отсутствия прыжка, и следующий блок является следующим. -1 в любом другом случае. Отрицательное значение говорит либо об ошибке, либо о том, что данный код не является необходимым для исполнения.

Поскольку LList в целом соответствует общей стандартной спецификации LinkedList в других языках, рассматривать его в работе не будем. Его код будет находится в приложении А.

# **2.3. Словарь инструкций**

Словарь, это список значений, который содержит в себе множество допустимых вариаций.

from **mnemonic** import **mnem**, **get**

from **random** import choice, choices

*# TODO: На самом деле можно создать неесколько значений с 1 мнемоникой*

*# для разного количества операндов. Это бы упростило бы логику*

*# Либо сделать функцию для сборки списка для импорта.*

rm\_rmi = [['r i','r m', 'r r', 'm r', 'm i']]

r\_rm = [['r r', 'r m']]

rmi = [['r', 'm', 'i']]

rm\_rl = [['r r', 'm r', 'r l', 'm l']]

r\_rmi = [['r r', 'r m', 'r i']]

r\_r = [['r r']]

r = [['r']]

r\_l = [['r l', 'm l']]

i = [['i']]

rm\_rm\_i = [['r','m'],['r r', 'r m'],['r r i', 'r m i']]

rm\_rm = [['r m', 'r r', 'm r']]

rmi = [['r', 'm', 'i']]

rm = [['r', 'm']]

ri = [['r', 'm', 'i']]

b = [['b']] *# imm8*

mnemonics = [

*# https://www.strchr.com/x86\_machine\_code\_statistics - анализ вероятности инструкций*

*# [frequency, mnemonic]*

*# -===/ SHIFTS \===-*

    [1, *# Cyclic rotate to the left using CF*

**mnem**('rcl', r\_l)], *#"r CL", "m l", "m CL" for 2*

    [1, *# Cyclir rotate to the right using CF*

**mnem**('rcr', r\_l)],

    [1, *# Logic shift left*

**mnem**('shl', r\_l)],

    [0, *# Double precise shift left*

**mnem**('shld', r\_l)],

    [1, *# Logic shift right*

**mnem**('shr', r\_l)],

    [0, *# Double precise shift right*

**mnem**('shrd', r\_l)],

    [2, *# Cyclic rotate to the left*

**mnem**('rol', r\_l)], *# "r CL", "m l", "m CL"*

    [2, *# Cyclic rotate to the right*

**mnem**('ror', r\_l)], *# for 2*

    [1, *# Arithmetic shift left*

**mnem**('sal', r\_l)],

    [1, *# Arithmetic shift right*

**mnem**('sar', r\_l)],

*# -===/ MATH \===-*

    [2, *# Negates op. CF=0|1 (if qe 0) OF=r SF=r ZF=r AF=r PF=r.*

**mnem**('neg', r)],

    [1, *# AL \* REG = AX with sign.*

**mnem**('imul', rm)],

    [1, *# AL \* REG = AX with sign.*

**mnem**('imul', rm\_rm\_i, (2, 3))],

    [1, *# Increment*

**mnem**('inc', r)],

    [1, *# Decrements op*

**mnem**('dec', r)],

    [1, *# AX / op = AL and mod AH, DX:AX / op = AX and mod DX*

**mnem**('div', r)],

    [1, *# Div with sign.*

**mnem**('idiv', r)],

    [10, *# Add two values.*

**mnem**('add', rm\_rmi)],

    [1, *# Add with carry.*

**mnem**('adc', rm\_rmi)],

    [2, *# Logical and.*

**mnem**('and', rm\_rmi)],

    [1, *# Bit scan forward. 1 op get number, 2 op scan for bits. If 0 only, then ZF = 1*

**mnem**('bsf', r\_rm, (2, 3))],

    [1, *# Bit scan reverse.*

**mnem**('bsr', r\_rm, (2, 3))],

    [1, *# Bit test. 1 op from take the bit, 2 op number of bit. CF have the value of that bit.*

**mnem**('bt', rm\_rl, (2, 3))],   *# "mw l" for 4*

    [1, *# Bit test and complement. Negate tested bit.*

**mnem**('btc', rm\_rl, (2, 3))],

    [1, *# Bit test and reset. Set tested bit to 0.*

**mnem**('btr', rm\_rl, (2, 3))],

    [1, *# Bit test and set. Set tested bit to 1.*

**mnem**('bts', rm\_rl, (2, 3))],

    [1, *# Sub with Borrow*

**mnem**('sbb', rm\_rmi)],

    [3, *# Substraction*

**mnem**('sub', rm\_rmi)],

    [5, *# Logical or*

**mnem**('xor', rm\_rmi)],

    [2, *# AL \* REG = AX*

**mnem**('mul', r)],

    [2, *# Logical inversion.*

**mnem**('not', 'r')],

    [3, *# Logical or*

**mnem**('or', rm\_rmi)],

*# -===/ JUMPS \===-*

    [15,

**mnem**('jmp', rmi, (2, 3))],

    [8,

**mnem**('jmp', b, (1, 1))],

    [10,

**mnem**('je', b, (1, 1))],

    [10,

**mnem**('jne', b, (1, 1))],

    [10,

**mnem**('jl', b, (1, 1))],

    [10,

**mnem**('jle', b, (1, 1))],

    [10,

**mnem**('jg', b, (1, 1))],

    [10,

**mnem**('jge', b, (1, 1))],

    [10,

**mnem**('jb', b, (1, 1))],

    [10,

**mnem**('jbe', b, (1, 1))],

    [10,

**mnem**('ja', b, (1, 1))],

    [10,

**mnem**('jae', b, (1, 1))],

    [10,

**mnem**('jz', b, (1, 1))],

    [10,

**mnem**('js', b, (1, 1))],

    [10,

**mnem**('jc', b, (1, 1))],

    [10,

**mnem**('jo', b, (1, 1))],

    [10,

**mnem**('jp', b, (1, 1))],

    [10,

**mnem**('jnz', b, (1, 1))],

    [10,

**mnem**('jns', b, (1, 1))],

    [10,

**mnem**('jnc', b, (1, 1))],

    [10,

**mnem**('jno', b, (1, 1))],

    [10,

**mnem**('jnp', b, (1, 1))],

    [10,

**mnem**('jcxz', b, (1, 1))],

    [10,

**mnem**('jecxz', b, (1, 1))],

*# -===/ STACK \===-*

    [10,*# Pop from stack to op*

**mnem**('pop', r, (2,3))],

    [0, *# Pop DI, SI, ВР, SP, BX, DX, СХ, АХ*

**mnem**('popa')],

    [0, *# Pop EDI, ESI, ЕВР, ESP, EBX, EDX, ЕСХ, ЕАХ*

**mnem**('popad')],

    [1, *# Pop FLAGS*

**mnem**('popf')],

    [0,  *# Pop EFLAGS*

**mnem**('popfd')],

    [17, *# Push to stack 17*

**mnem**('push', r, (2,3))],

    [0, *# Pop DI, SI, ВР, SP, BX, DX, СХ, АХ*

**mnem**('pusha')],

    [0, *# Pop EDI, ESI, ЕВР, ESP, EBX, EDX, ЕСХ, ЕАХ*

**mnem**('pushad')],

    [1,  *# Pop FLAGS*

**mnem**('pushf')],

    [0,  *# Pop EFLAGS*

**mnem**('pushfd')],

*# -===/ CYCLE, FUNCS \===-*

    [1, *# Return from procedure*

**mnem**('ret')],

    [0, *# Return far*

**mnem**('retn')],

    [1, *# Retunr far + privilegies*

**mnem**('retf')],

    [1, *# Dec ECX then continue or Jump to label if CX != 0 !imp*

**mnem**('loop', b)],

    [0, *# Dec ECX then continue or Jump to label if CX !=0 and ZF = 0 !imp*

**mnem**('loopnz', b)],

    [0, *# Dec ECX then continue or Jump to label if CX !=0 and ZF = 1 !imp*

**mnem**('loopz', b)],

    [1, *# Synonyms*

**mnem**('loopne', b)],

    [1,

**mnem**('loope', b)],

*# -===/ FLAGS \===-*

    [1, *# Call function and save IP and CS (far only). !imp*

**mnem**('call', rmi, (2,3))],

    [1, *# AH -> FLAGS*

**mnem**('sahf')],

    [1, *# Set FLAGS -> AH (SF:ZF:0:AF:0:PF:1:CF)*

**mnem**('lahf')],

    [1, *# CF=1*

**mnem**('stc')],

    [1, *# DF=1*

**mnem**('std')],

    [1, *# IF = 1*

**mnem**('sti')],

    [1, *# CF = 0*

**mnem**('clc')],

    [1, *# DF = 0*

**mnem**('cld')],

    [1, *# IF = 0*

**mnem**('cli')],

    [1, *# CF = r*

**mnem**('cmc')],

*# -===/ OTHER \===-*

    [1, *# Fill ah to 1 or 0 depends on elder bit of al*

**mnem**('cbw')],

    [1, *# Word -> Double word AX -> DX*

**mnem**('cwd')],

    [1, *# DW -> QW EAX -> EDX*

**mnem**('cdq')],

    [9, *# Compares two ops. eq -> ZF = 1 gr -> SF=OF ls -> SF!=OF*

**mnem**('cmp', rm\_rmi)],

    [0, *# Word -> Double word AX -> EAX*

**mnem**('cwde')],

    [0, *#*

**mnem**('enter', i, (1,2))],

    [1, *# Copy BP -> SP. stack-> BP.*

**mnem**('leave')],

    [50,*# move value of op 2 in op 1. !imp*

**mnem**('mov', rm\_rmi)],    *# 'm i'*

    [1, *# mov r16, r8 with sign fill.*

**mnem**('movsx', r\_r, (1,2), (1,0))],

    [1, *# mov r16, r8 with zero fill*

**mnem**('movzx', r\_r, (1,2), (1,0))],

    [1, *# Logical compare*

**mnem**('test', rm\_rmi)],

    [1, *# Swap two ops*

**mnem**('xchg', rm\_rm)],

    [1, *# Do notthing exept inc EIP.*

**mnem**('nop')],

]

ptr\_sizes = [

    'byte',

    'word',

    'dword',

    'qword',

    'oword',

    'yword',

    'tword'

]

aviable\_regs = [

    ["AL", "BL", "CL", "DL", "AH", "BH", "CH", "DH"],

    ["AX", "BX", "CX", "DX"], *# "SI", "DI", "BP", "SP", "CS", "DS", "SS", "ES", "IP"*

    ["EAX", "EBX", "ECX", "EDX"], *# "ESI", "EDI", "EBP", "ESP", "CS", "DS", "SS", "ES", "EIP"*

]

def **choice\_reg**(size):

    return choice(aviable\_regs[size-1])

def **gen\_probs**():

    """Считает шанс появления инструкций"""

    probalities = []

    inc\_instructions = []

    value = 0

    for m in mnemonics:

*# Если вероятность генерации инструкции выше 0*

        if m[0] != 0:

            probalities.**append**(value + m[0])

            inc\_instructions.**append**(m)

            value += m[0]

    return probalities, inc\_instructions

def **choose\_inst**(mnems, probs):

    """Выбирает случайную инструкцию"""

    m = choices(mnems, cum\_weights=probs, k=1)[0][1]

    return  **get**(m)

def **ptr**(size):

    return ptr\_sizes[size-1]

Листинг 4. mnemData.py словарь инструкций

Поскольку часто различные мнемоники имеют разные наборы операндов, было решено записать варианты операндов в массив. Некоторые мнемоники имеют операнды разных размеров, в таком случае можно указать разницу в размере операндов. Если операнд имеет очень много различных вариаций, то можно записать его в словарь несколько раз. Также нужна вероятность встречи данной инструкции в коде.

У языка python для этого есть встроенная функция random.choices() которая принимает список, и список содержащий шансы, лучше всего передавать cumulative\_weights, так как для выбора, в любом случае придется посчитать сумму предыдущих элементов для определения вероятности выбора. Стоит иметь в виду что эта функция всегда возвращает массив. Для лучшей производительности предварительно сделаем это сами. В наивных реализациях выбор случайного элемента с вероятностью выпадения происходит в среднем за O(n). В случае реальных реализаций используется бинарный поиск, но требуется также сортировка списка, работающий за O(log n).

Так же здесь есть функции выбора значений из словаря, что абстрагирует от реальной работы со словарем. При внимательном изучении инструкций можно заметить, что варианты операторов как правило имеют несколько общих паттернов, поэтому что бы упростить редактирование и избежать опечаток, лучше создать наборы паттернов и указывать в списке ссылку на них, также это будет дешевле в плане памяти, поскольку хоть строки являются иммутабельными, id у них вполне может различаться.

from **random** import randint, choice, randrange

*#i - IMMediate - constant value*

*#b - BITs (length of previous operand in bits)*

*#r - REGister*

*#m - MEMory*

global\_size = (1,3)

def **bound**(value, borders):

    """Ограничивает число на отрезке"""

    return **max**(**min**(value, borders[1]),borders[0])

def **add\_tuple**(a, b):

    return **tuple**(**map**(**sum**, **zip**(a, b)))

def **intersect**(borders1, borders2):

    """Находит пересечение двух отрезков"""

    return (**max**(borders1[0], borders2[0]), **min**(borders1[1], borders2[1]))

def **mnem**(mnemonic, operands = [], size=(1,3), ops\_size=None):

    """Создает мнемонику, список комбинаций операндов сгрупированы в списки по их количеству"""

    return {'mnem': mnemonic, 'ops\_list': operands, 'borders':**intersect**(global\_size, size), 'ops\_size':ops\_size}

def **get**(mnem, ops\_index=None, cur\_size=None):

    """Возвращает название, размер, количество операндов, типы операндов, и

    возможные комабинации для данного количества операндов"""

    if cur\_size is None:

        cur\_size = randint(\*mnem['borders'])

    elif **type**(cur\_size) is **tuple**:

        cur\_size = randint(\***intersect**(cur\_size, mnem['borders']))

    else:

        cur\_size = **bound**(cur\_size, mnem['borders'])

    ops = []

    cur\_ops = ''

    count = 0

    if mnem['ops\_list']:

        if ops\_index is None:

            ops\_index = randrange(0, **len**(mnem['ops\_list']))

        ops = mnem['ops\_list'][ops\_index]

        cur\_ops = choice(ops)

        count = **len**(cur\_ops.split(' '))

    return {'mnem':mnem['mnem'], 'size':cur\_size, 'ops\_count':count, 'ops':cur\_ops, 'combs':ops, 'ops\_size':mnem['ops\_size']}

def **random\_bit**(size):

    """Случайный бит для регистра указанной размерности"""

    max\_bit = **pow**(2, 3 + size)

    return randrange(max\_bit)

def **random\_value**(size):

    """Случайное число указанной размерности"""

    max\_value = **pow**(256,size)-1

    return randrange(max\_value)

Листинг 5. mnemonic.py несмотря на название, не имеет класса

Сам файл мнемоник, содержит набор функций по их составлению и генерации на основе выбранной инструкции. Здесь нет сложной логики, разве что часто необходимо брать пересечения между двумя кортежами из двух чисел, если несколько правил накладывают ограничений на возможный размер инструкций, то выбирать нужно всегда их пересечение. В дальнейшем можно реализовать сохранения результата их составления в файл. Что бы уменьшить время загрузки обфускатора.

# **2.4. Обфускатор**

Данный класс управляет работой обфускации, предоставляя удобный API для модификации кода, когда каждое действие это функция, что позволяет легко создавать нужный результат.

from **llist** import **LList**

from **codeblock** import **CodeBlock**

from **mnemonics** import ig

from **numx** import \*

import **random**

from **logger** import **is\_debug**, **debug**

class **Obfuscator**:

    def **split\_insts\_on\_inst\_groups**(self, code\_list, min=3, max=6):

        """Объединяет инструкции в небольшие группы кода"""

        code\_groups = []

        index = 0

        while index < **len**(code\_list):

            length = **random**.randrange(**min**, **max**)

            code\_group = code\_list[index:index+length]

            code\_groups.**append**(code\_group)

            index += length

        return code\_groups

    def **make\_llist\_of\_codeblocks**(self, inst\_groups):

        """Создаем список представляющий код программы"""

        llist\_codeblocks = **LList**()

        prev\_cb = **CodeBlock**(inst\_groups[0], cb\_id="0")

        llist\_codeblocks.**addn**(prev\_cb)

**id** = 1

        for inst\_group in inst\_groups[1:]:

            cb = **CodeBlock**.**chain**(prev = prev\_cb, insts = inst\_group, cb\_id=**str**(**id**))

            llist\_codeblocks.**addn**(cb)

            prev\_cb = cb

**id**+=1

*# Устанавливает свойство entry\_point, которое всегда указывает на*

*# первый исполняющийся блок, независимо от его расположения в списке.*

        llist\_codeblocks.entry\_point = llist\_codeblocks.head

        return llist\_codeblocks

    def **\_\_init\_\_**(self, code\_list) -> None:

        cg = self.**split\_insts\_on\_inst\_groups**(code\_list)

        self.llist\_cb = self.**make\_llist\_of\_codeblocks**(cg)

    def **add\_jumps**(self):

        """Добавляет прыжки в код"""

        for cb in self.llist\_cb.**forwardn**():

            if cb.next\_code and not cb.has\_jump:

                insts = ig.**gen\_jmp**()

                jump\_cb = **CodeBlock**(insts)

                jump\_cb.next\_code = cb.next\_code

                jump\_cb.has\_jump = True

                jump\_cb.cb\_id = cb.cb\_id + ".0j"

                cb.next\_code = jump\_cb

                self.llist\_cb.**ins**(cb, jump\_cb)

    def **add\_garbage**(self):

        """Добавляет мусор в код"""

        for cb in self.llist\_cb.**backwardn**():

            if cb.has\_jump:

                garbage\_cb = ig.**genb**(10)

                self.llist\_cb.**ins**(cb, garbage\_cb)

    def **print\_cb**(self):

        """Выводим Codeblocks в консоль"""

        for cb in self.llist\_cb.**forwardn**():

            next\_addr = cb.next\_code.addr if cb.next\_code else 'empty'

**print**("%s has addr:%s and size:%s and next:%s, has\_jump:%s, bytes:%s"% (cb.cb\_id, cb.addr, cb.size, next\_addr, cb.is\_need\_to\_calc\_addr(), cb.bytes))

            if cb.is\_need\_to\_calc\_addr():

**print**(cb.code)

                assert cb.code != 'jmp <pos>\n', 'JMP POS'

    def **print\_cbcf**(self):

        """Выводим указатели"""

        for cb in self.llist\_cb.**forwardn**():

            next\_cb\_id = cb.next\_code.cb\_id if cb.next\_code else 'none'

            to = ''

            if cb.is\_need\_to\_calc\_addr():

                next\_ca = cb.next\_code.addr if cb.next\_code else 'none'

                diff = **n\_s**(value=**int**(next\_ca) - cb.addr, size=4) if next\_ca != 'none' else 'nan'

                lb = cb.bytes[-1]

                to = "lb:%s, ta:%s, ca:%s, ta-ca:%s"%(**n\_s**(lb, 'ui'), next\_ca, cb.addr, diff)

**print**("cb:%s -> cb:%s, %s"%(cb.cb\_id, next\_cb\_id, to))

    def **calculate\_addr\_of\_blocks**(self):

        """Подсчет адресов"""

        prev\_cb\_addr = 0

        for cb in self.llist\_cb.**forwardn**():

            cb.addr = prev\_cb\_addr

            prev\_cb\_addr += cb.size

    def **calculate\_jump\_addr**(self):

        """вычисление адреса прыжков"""

        for cb in self.llist\_cb.**forwardn**():

*#print('called')*

*#print(cb.is\_need\_to\_calc\_addr())*

            if cb.is\_need\_to\_calc\_addr() > 0:

                if (tar\_addr := cb.next\_code.addr) is not None:

                    ig.**set\_addr**(cb, tar\_addr, cb.addr)

                    lb = **b\_i**(cb.bytes)

                    tc = tar\_addr - cb.addr

                for\_u = lb + 2 == tc

                for\_s = lb - 2 == tc

*# assert for\_u or for\_s, 'not eq: lb+2 is %s, when ta-ca is %s'%(lb, tc)*

*# assert cb.code != 'jmp <pos>\n', 'JMP POS'*

    def **shuffle**(self):

        self.llist\_cb.**shuffle**()

        pass

    def **get\_ep**(self):

        """Получения начального адреса"""

        return self.llist\_cb.entry\_point.addr

Листинг 6. obfuscator.py методы обфускатора

Первые две функции служат для инициализации списка, сборки отдельных инструкций в большие куски кода, а также их упаковка в список. Функция прыжков добавляет прыжки после каждой инструкции. После чего используем сгенерированные последовательности кода для вставки между блоками кода.

# **2.5. Кодогенерация**

Класс для генерации инструкций обращается к словарю и собирает мнемонику с выбранными операндами. Также умеет собирать и обновлять адрес инструкций прыжков.

from **compiler** import c, **Inst**, OFFS, **KsError**

from **codeblock** import **CodeBlock**

from **mnemData** import **gen\_probs**, **choice\_reg**, **choose\_inst**, **ptr**

from **mnemonic** import **random\_bit**, **random\_value**

import **random**

import **re**

import **logger** as **lg**

def **randbool**():

    return **bool**(**random**.getrandbits(1))

def **random\_val\_or\_reg**(size):

    if **randbool**():

        return **random\_value**(size)

    return **choice\_reg**(size)

class **InstructionGenerator**:

    def **\_\_init\_\_**(self):

        self.c = c

        self.probs, self.insts = **gen\_probs**()

    def **build\_op**(self, optype, size) -> **str** | None:

        """Создает операнд и адресацию к нему"""

        match optype:

                case 'r': *# Регистр*

                    return **choice\_reg**(size)

                case 'i': *# Число*

                    return **str**(**random\_value**(size))

                case 'm': *# Адрес в памяти*

                    return '%s ptr [%s]'%(**ptr**(size),**str**(**random\_val\_or\_reg**(size))) *# 32bit*

                case 'l': *# Номер бита*

                    return **str**(**random\_bit**(size))

                case 'b': *# 1 Байт*

                    return **str**(**random\_value**(1))

                case 'a': *# 32bit Адрес*

                    return '%s ptr [%s]'%(**ptr**(**min**(size, 2)),**str**(**random\_val\_or\_reg**(size)))

                case \_:

**lg**.**error**('Non existent operand <%s>' % optype)

                    return None

    def **make\_args**(self, ops, size, ops\_size = None) -> **str**:

        """Определяет аргументы для инструкции"""

        if **len**(ops) == 0:

            return ''

        result = []

        if ops\_size:

            for op, op\_size in **zip**(ops.split(' '), ops\_size):

                result.**append**(self.**build\_op**(op, size+op\_size))

        else:

            for op in ops.split(' '):

**print**('%s:%s'%(op, size))

                result.**append**(self.**build\_op**(op, size))

**print**(result[-1])

        return ", ".**join**(result)

    def **make\_inst**(self) -> **Inst** | None:

        """Создает инструкцию"""

        mnem\_data = **choose\_inst**(self.insts, self.probs)

        ops = self.**make\_args**(mnem\_data['ops'], mnem\_data['size'], mnem\_data['ops\_size'])

        instruction = **Inst**(mnem\_data['mnem'], ops)

        try:

            instruction\_updated = self.c.**asmu**(instruction)

        except **KsError** as ex:

**lg**.**error**('KsError:%s'%(ex), exc\_info=True)

            return None

        return instruction\_updated

    def **genb**(self, size\_cap):

        """Создает код длиной не больше size\_cap байтов"""

        cur\_size = 0

        insts = []

        while cur\_size < size\_cap:

            inst = self.**make\_inst**()

            insts.**append**(inst)

            cur\_size += **len**(inst.bytes)

        return **CodeBlock**(insts)

    def **genc**(self, count):

        """Создает код длиной count инструкций"""

        insts = []

        for i in **range**(count):

            inst = self.**make\_inst**()

            insts.**append**(inst)

        return **CodeBlock**(insts)

    def **gen\_jmp**(self, jump\_type = 'jmp', where='<pos>', condition='none', op1=None, op2=None):

        """Создает инструкции прыжка"""

        insts = []

        match(condition):

            case 'cmp':

                inst = **Inst**('cmp', '%s, %s \n'%(op1, op2))

                inst = self.c.**asmu**(inst)

                insts.**append**(inst)

            case 'test':

                inst = **Inst**('test', '%s, %s \n'%(op1, op2))

                inst = self.c.**asmu**(inst)

                insts.**append**(inst)

        inst = **Inst**(jump\_type, 4)

        inst = self.c.**asmu**(inst)

        inst.ops = where

        insts.**append**(inst)

        return insts

    def **set\_addr**(self, cb, tar\_addr, cur\_addr = 0, def\_offset = OFFS):

        """Обновляет адрес прыжка"""

        pattern = **re**.**compile**(r"<.{3}>")

        match = **re**.**search**(pattern, cb.code)[0]

        address = def\_offset

        match(match):

            case '<pos>':

                address = **str**(tar\_addr-cur\_addr)

                new\_op\_str = **re**.**sub**(match, address, cb.code)

                assert cb.code != new\_op\_str

                bytes, size = c.**asm**(new\_op\_str)

                cb.bytes = bytes

                cb.size = **len**(bytes)

                cb.code = new\_op\_str

**lg**.**debug**('ta:%s ca:%s ta-ca:%s op\_str:%s cb.b:%s'%(tar\_addr, cur\_addr, address, new\_op\_str, cb.bytes[-1]))

ig = **InstructionGenerator**()

Листинг 7. instGenerator.py позволяет создавать код

Данный класс можно дополнить созданием изоморфных и обратимых блоков кода. Также можно сделать словарь для вставки ловушек от отладчика. Или вставку функций расшифровки строк, например. Такая необходимость действительно возникает довольно часто. Знание, где находится строка и кто к ней обращается дает информацию для отладки.

# **ГЛАВА III. ОТЛАДКА И ИССЛЕДОВАНИЕ**

После рассмотрения теоретических основ и архитектуры обфускации необходимо рассмотреть сам результат произведенных преобразований. Для начала рассмотрим заголовок файла:



Рис. 5. Реальный вид PE структуры

Первые 64 байта это IMAGE\_DOS\_HEADER, далее идет заглушка на 64 байта DOS\_STUB. Ниже находится IMAGE\_NT\_HEADER вплоть до «.text». Он содержит в себе IMAGE\_OPTIONAL\_HEADER начиная с 98h (152) байта. В нем выделен красным адрес точки входа в программу. Необходимо помнить, что числа в файле записываются в little-endian порядке от младшего к старшему байту. Такой формат записи удобен при обращении процессора к памяти для вычислений, так как при увеличении размера операнда адрес не смещается. Числа в заголовке занимают как правило 4 байта, то есть двойное слово.

Далее идут BaseOfCode, BaseOfData и ImageBase. Стоит заметить, что в ОС Windows ImageBase как правило равняется 0x00400000. Далее идут SectionAligment и FileAligment. Дальше по списку нас интересуют секции. Секции начинаются с названия. Длинной оно в двойное машинное слово. Код как правило обозначается секцией .text. Начиная с 12 байта идут VirtualAddress, SizeOfRaw, PointerToRaw.

Далее рассмотрим сам код. Также изменился адрес точки входа. Было 0x00100000 (начало секции .text), а стало 0x40100000.

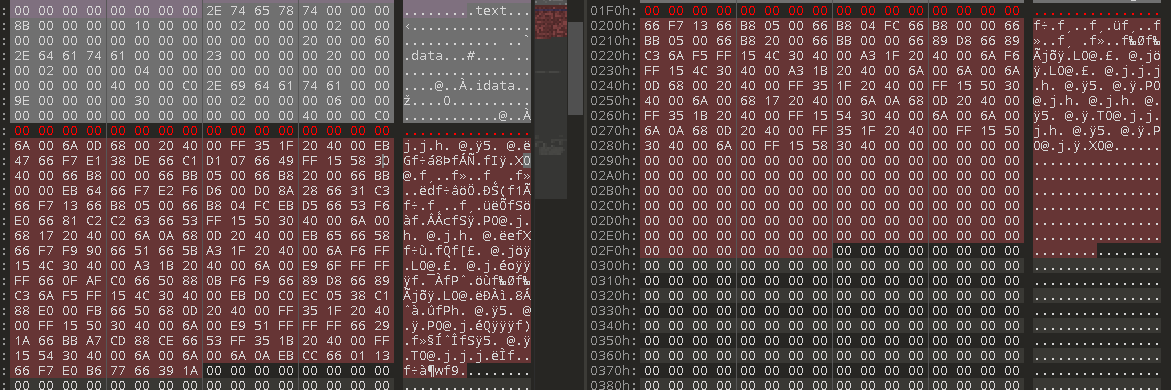


Рис. 6. так выглядит код в hex редакторе

Как видно из-за добавленных прыжков и ложного кода файл стал в 2 раза длиннее. Откроем его в отладчике и посмотрим на результат.

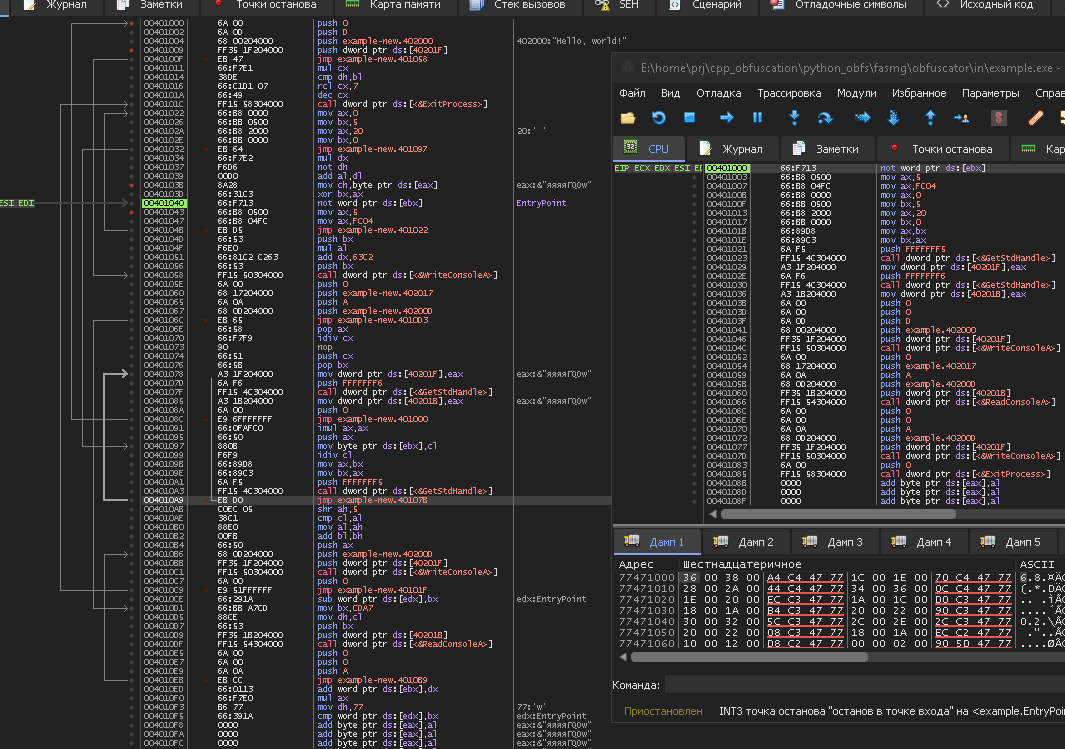


Рис. 7. разница в виде кода при отладке программы



Рис. 8. результат работы обеих программ

При визуальном осмотре кода программы после изменения (слева) прыжков на неправильные адреса обнаружено не было. Код стал сложнее, трудно определить с первого взгляда что данная программа должна делать. Программа по-прежнему выполняет свою работу.

Запустим обфускатор еще раз, чтобы убедится, что при каждом запуске, ветвление кода будет отличаться от предыдущего.

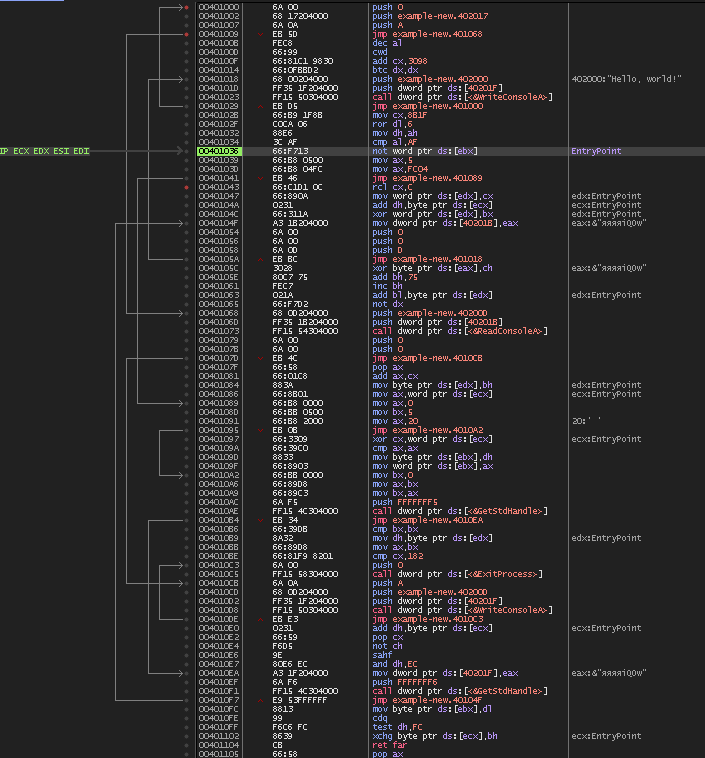


Рис. 9. повторный запуск обфускатора

При повторном запуске мы убедились, что каждый раз ветвление программы отличается от предыдущего. Обфусцированная программа по-прежнему сохранила работоспособность.

Таким образом, обфускатор выполняет свою работу, цели и задачи выпускной квалификационной работы были достигнуты.

# **ЗАКЛЮЧЕНИЕ**

В результате был разработан концепт обфускатора, который показывает обязательные ключевые шаги для любого обфускатора бинарных файлов. В дальнейшем используя полученные наработки, можно существенно ускорить проверку идей для обфускации.

Также был исследован формат PE, который является необходимой для любого исполняемого файла под управлением ОС Windows. Были рассмотрены способы адресации как в файле, так и в виртуальном пространстве при работе приложения.

Был найден удобный вариант структуры для хранения инструкций в процессе модификации кода.

Ссылка на github проекта: <https://github.com/shin0kaze/exeObfuscaor>.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

Список использованной литературы:

1. Электронный источник: [https://habr.com/ru](https://habr.com/ru/articles/266831/). PE (Portable Executable): На странных берегах.
2. Электронный источник: [https://habr.com.ru](https://habr.com/ru/articles/533954/). Обфускация как метод защиты программного обеспечения.
3. Электронный источник: [https://felixcloutier.com](https://www.felixcloutier.com/x86/index.html). Справочник по x86 инструкциям ассемблера.
4. Электронный источник: [https://rpis.ec](https://rpis.ec/blog/dissection-llvm-obfuscator-p1/). Dissecting LLVM Obfuscator Part 1.
5. Электронный источник: [https://medium.com](https://polarply.medium.com/build-your-first-llvm-obfuscator-80d16583392b). Build your first LLVM Obfuscator.
6. Электронный источник: [https://github.com/](https://github.com/ssyuqixe/obfCoder). obfCoder simple c++ obfuscator.
7. Электронный источник: [https://www.codeproject.com](https://www.codeproject.com/Articles/856846/Binary-Obfuscation). Binary Obfuscation.
8. Электронный источник: [https://www.codeproject.com](https://www.codeproject.com/Articles/502283/Strings-Obfuscation-System). Strings Obfuscation System.
9. Электронный источник: [https://www.codeproject.com](https://www.codeproject.com/Articles/1210398/TinyObfuscate-A-Tiny-String-Obfuscator-for-C-Cplus). TinyObfuscate - A Tiny String Obfuscator for C / C++.
10. Электронный источник: [https://fasmworld.ru](https://fasmworld.ru/uchebnik/). Учебный курс программирования на ассемблере FASM.
11. Электронный источник: [https://www.cyberforum.ru](https://www.cyberforum.ru/fasm/thread1240590.html). Уроки Iczelion'a на FASM.
12. Электронный источник: [https://www.cyberforum.ru](https://www.cyberforum.ru/fasm/thread1516255.html). Программирование на языке Assembler в FASM.
13. Электронный источник: [https://www.cyberforum.ru](https://www.cyberforum.ru/fasm/thread1253291.html). Руководство по препроцессору FASM
14. Электронный источник: [https://dotzero.blog](https://dotzero.blog/weighted-random-simple/). Простой алгоритм случайной выборки с учетом веса.
15. Электронный источник: [https://www.strchr.com/](https://www.strchr.com/x86_machine_code_statistics). Статистика инструкций x86
16. Крис Касперски Техника и философия хакерских атак. - М.: Солон - Р, 1999. 315с.
17. Бабаш, А.В. Информационная безопасность. Лабораторный практикум: Учебное пособие М.: КноРус, 2016. 136 c.
18. Гафнер, В.В. Информационная безопасность: Учебное пособие РД: Феникс, 2017. 324 c.
19. Громов, Ю.Ю. Информационная безопасность и защита информации: Учебное пособие Ю.Ю. Громов, В.О. Драчев, Ст. Оскол: ТНТ, 2017. 384 c.
20. Малюк, А.А. Информационная безопасность: концептуальные и методологические основы защиты информации А.А. Малюк. — М.: ГЛТ, 2016. 280 c.
21. Партыка, Т.Л. Информационная безопасность: Учебное пособие / Т.Л. Партыка, И.И. Попов. М.: Форум, 2016. 432 c.
22. Семененко, В.А. Информационная безопасность: Учебное пособие / В.А. Семененко. - М.: МГИУ, 2017. 277 c.
23. Шаньгин, В.Ф. Информационная безопасность и защита информации / В.Ф. Шаньгин. — М.: ДМК, 2017. 702 c.
24. Ярочкин, В.И. Информационная безопасность: Учебник для вузов / В.И. Ярочкин. — М.: Акад. Проект, 2018. 544 c.

# **Приложение А**

Код двусвязного списка LList:

from **random** import shuffle

class **Node**:

    def **\_\_init\_\_**(self, value, prev=None, next=None):

        self.value = value

        self.prev = prev

        self.next = **next**

    def **nextv**(self):

        """Следующее значение, если оно есть"""

        return self.next.value if self.next else None

    def **prevv**(self):

        """Предыдущее значение, если оно есть"""

        return self.prev.value if self.prev else None

    def **set\_pn**(self, prev, next):

        """Устанавливает prev и next"""

        self.prev = prev

        self.next = **next**

    def **set\_pno**(self):

        """Устанавливает prev и next у соседей"""

        if self.prev:

            self.prev.next = self

        if self.next:

            self.next.prev = self

        return **bool**(self.prev), **bool**(self.next)

    def **set\_pnlinks**(self, prev, next):

        """Устанавливает соседние ссылки"""

        self.**set\_pn**(prev, **next**)

        return self.**set\_pno**()

    def **set\_plink**(self, prev):

        """устанавливат предыдущую Node"""

        self.prev = prev

        if prev:

            self.prev.next = self

        return **bool**(prev)

    def **set\_nlink**(self, next):

        """устанавливат следующую Node"""

        self.next = **next**

        if **next**:

            self.next.prev = self

        return **bool**(**next**)

    def **pnlinks**(self):

        """Возвращает все ссылки как tuple"""

        p = None

        pn = None

        n = None

        np = None

        if self.prev:

            p = self.prev

            pn = self.prev.next

        if self.next:

            p = self.next

            pn = self.next.prev

        return (p, pn, n, np)

    def **pnlinks\_str**(self, node=False):

        """Возвращает строковое представление ссылок"""

        prev = 'node.prev -> '

        if self.prev:

            prev += '%s.value: %s, prev -> %s.value: %s ' % (self.prev if node else 'n', self.prev.value, self if node else 'n', self.value)

        else:

            prev += 'None '

**next** = 'node.next -> '

        if self.next:

**next** += '%s.value: %s, next -> %s.value: %s' % (self.next if node else 'n', self.next.value, self if node else 'n', self.value)

        else:

**next** += 'None'

        return prev + **next**

    def **forward**(self):

        """Итератор вперед"""

        yield self

        if self.next:

            for node in self.next:

                yield node

    def **backward**(self):

        """Итератор назад"""

        yield self

        if self.prev:

            for node in self.prev:

                yield node

    def **\_\_iter\_\_**(self):

        """Итератор по умолчанию"""

        return self.**forward**()

    def **\_\_str\_\_**(self) -> **str**:

        return **str**(self.value)

class **LList**:

    def **\_\_init\_\_**(self):

        self.**clr**()

    def **\_\_iter\_\_**(self):

        return self.**forwardn**()

    def **empty**(self):

        """Проверяет пустоту листа"""

        return self.count == 0

    def **len**(self, node=None):

        """Длина списка начиная с Node"""

        if not node: node = self.head

        return **sum**(1 for e in **list**(node.**forward**()))

*#region iterators*

    def **forward**(self):

        """Итерация вперед"""

        current = self.head

        while current:

            yield current.value if current else None

            current = current.next

    def **backward**(self):

        """Итерация назад"""

        current = self.tail

        while current:

            yield current.value if current else None

            current = current.prev

    def **forwardn**(self):

        """Итерация вперед, возращая Node"""

        current = self.head

        while current:

            yield current

            current = current.next

    def **backwardn**(self):

        """Итерация назад, возращая Node"""

        current = self.tail

        while current:

            yield current

            current = current.prev

*#endregion*

*#region adding elements*

    def **addn**(self, node):

        """Добавить Node в конец"""

        self.count+=1

        node.prev = self.tail

        node.next = None

        if self.head is None:

            self.head = node

            self.tail = node

        else:

            self.tail.next = node

            self.tail = node

        return node

    def **extendn**(self, lst):

        """Добавить список Node в конец"""

        for el in lst:

            self.**addn**(el)

    def **add**(self, el):

        """Создать и добавить Node в конец"""

        self.count+=1

        new\_node = **Node**(el, self.tail)

        if self.head is None:

            self.head = new\_node

            self.tail = new\_node

        else:

            self.tail.next = new\_node

            self.tail = new\_node

        return new\_node

    def **extend**(self, lst):

        """Создать и добавить список Node в конец"""

        for el in lst:

            self.**add**(el)

    def **addns**(self, node):

        """Добавить Node в начало"""

        self.count+=1

        node.next = self.head

        if not self.head:

            self.head = node

            self.tail = node

        else:

            self.head.prev = node

            self.head = node

        return node

    def **extendns**(self, lst):

        """Добавить список Node в начало"""

        for el in lst:

            self.**addns**(el)

    def **adds**(self, el):

        """Создать и добавить Node в начало"""

        self.count+=1

        new\_node = **Node**(el, next = self.head)

        if not self.head:

            self.head = new\_node

            self.tail = new\_node

        else:

            self.head.prev = new\_node

            self.head = new\_node

        return new\_node

    def **extends**(self, lst):

        """Создать и добавить список Node в начало"""

        for el in lst:

            self.**adds**(el)

*#endregion*

*#region find elements*

    def **get**(self, value):

        """Находит элемент по значению"""

        current = self.head

        while(current.next and current.value != value):

            current = current.next

        if current.value == value:

            return current

        return None

    def **find**(self, value, from\_node = None, until\_node = None):

        """Находит элемент в указанном списке"""

        current = from\_node if from\_node else self.head

        count = 0

        while(current.next and current.value != value):

            current = current.next

            count += 1

            if current.value == value:

                return current, count

            if current == until\_node:

                break

        return None, -1

    def **inl**(self, value, from\_node = None, until\_node = None):

        """Проверяет наличие элемента в указанном списке"""

        return not (self.**find**(value, from\_node, until\_node)[0] is None)

    def **ind**(self, node, from\_node = None, until\_node = None):

        """Находит индекс элемента в указанном списке"""

        current = from\_node if from\_node else self.head

        count = 0

        while(current.next and current is not node):

            current = current.next

            count += 1

            if current is node:

                return count

            if current == until\_node:

                break

        return -1

    def **seek**(self, node, count):

        """Пропускает count элементов"""

        current = node

        for i in **range**(count):

            if not (current := current.next):

                return self.tail

        return current

    def **seek\_n\_of**(self, node, count):

        """Пропускает count элементов и возвращает остаток count"""

        current = node

        for i in **range**(count):

            if not (current := current.next):

                return (self.tail, count - i)

        return (current, 0)

    def **at**(self, index):

        """Берет элемент по индексу"""

        if index >= self.count or index < -self.count:

            return None

        current = None

        if index >= 0:

            current = self.head

            for i in **range**(0, index):

                current = current.next

        else:

            current = self.tail

            for i in **range**(-1, index, -1):

                current = current.prev

        return current

    def **v**(self, index):

        """Берет значение элемента по индексу"""

        if current := self.**at**(index):

            return current.value

    def **midn**(self):

        """Возвращает элемент по середине списка"""

        return self.**at**(self.count//2)

    def **mid**(self):

        """Возвращает значение по середине списка"""

        return self.**midn**().value

*#endregion*

    def **ins**(self, node, new\_node):

        """Вставляет элемент после текущего"""

        next\_node = node.next

        has\_pn = new\_node.set\_pnlinks(node, next\_node)

*# если нет следующего элемента*

        if not has\_pn[1]:

            self.tail = new\_node

        self.count+=1

    def **putn**(self, node, el):

        """Создает и вставляет элемент после текущего"""

        next\_node = node.next

        new\_node = **Node**(el, node, next\_node)

        node.next = new\_node

        if next\_node:

            next\_node.prev = new\_node

        else:

            self.tail = new\_node

        self.count+=1

        return new\_node

    def **braze**(self, els, ele, node, count=None):

        """Вставляет список от els до ele после указанного элемента"""

        if not count:

            count = self.**ind**(ele, els) + 1

        next\_node = node.next

        node.next = els

        els.prev = node

        if next\_node:

            next\_node.prev = ele

            ele.next = next\_node

        else:

            self.tail = ele

        self.count += count

    def **d**(self, node):

        """Удаляет указанный элемент из списка"""

        prev\_node = node.prev

        next\_node = node.next

        if prev\_node:

            prev\_node.next = next\_node

        else:

            self.head = next\_node

        if next\_node:

            next\_node.prev = prev\_node

        else:

            self.tail = prev\_node

        node.prev = None

        node.next = None

        self.count-=1

    def **rem**(self, index):

        """Удаляет указанный элемент из списка по индексу"""

        if current := self.**at**(index):

            self.**d**(current)

            return current.value

    def **neib**(self, el1, el2):

        """Определяет 2 элемента на соседство"""

        if el1 and el2:

            if el1.next == el2: return 1

            if el2.next == el1: return 2

        return 0

    def **dl**(self, els = None, ele = None, count = None):

        """Удаляет элементы от els до ele"""

        if els == None and ele == None:

            all\_nodes = self.**get\_all**()

            self.**clr**()

            return all\_nodes

        if els is ele or self.**neib**(els, ele):

            return (None, None, 0)

        nels = els.next if els else self.head

        nele = ele.prev if ele else self.tail

        nels.prev = None

        nele.next = None

        if els: els.next = ele

        else: self.head = ele

        if ele: ele.prev = els

        else: self.tail = els

        count = count if count else self.**len**(nels)

        self.count -= count

        return (nels, nele, count)

    def **remn**(self, node, count):

        """Удаляет N элементов из списка, начиная с текущего"""

        if count < 1:

            return (None, None, 0)

        next\_node, overflow = self.**seek\_n\_of**(node, count-1)

        return self.**dl**(node.prev, next\_node.next, count - overflow)

    def **reml**(self, index, count):

        """Удаляет N элементов из списка, начиная с текущего индекса"""

        return self.**remn**(self.**at**(index), count)

    def **r**(self, index):

        """Удаление элемента по индексу, возвращает ссылку на себя"""

        self.**rem**(index)

        return self

    def **put**(self, index, el):

        """Вставляет элемент по индексу"""

        if current := self.**at**(index):

            return self.**putn**(current, el)

    def **putl**(self, index, lst):

        """Вставляет список по индексу"""

        current = self.**at**(index)

        for el in lst:

            new\_node = self.**putn**(current, el)

            current = new\_node

    def **u**(self, index, el):

        """Вставка элемента по индексу, возвращает ссылку на себя"""

        self.**put**(index, el)

        return self

    def **pop**(self):

        """Достает элемент с конца"""

        if tail\_node := self.tail:

            self.tail = tail\_node.prev

            if self.tail: self.tail.next = None

            else: self.head = None

            self.count-=1

        return tail\_node

    def **pops**(self):

        """Достает первый элемент"""

        if head\_node := self.head:

            self.head = head\_node.next

            if self.head: self.head.prev = None

            else: self.tail = None

            self.count-=1

        return head\_node

    def **cut**(self, els, ele, new\_els, new\_ele, count=None, count\_new=None):

        """Заменяет часть списка на другую"""

        old\_nodes = self.**dl**(els, ele, count)

        self.**braze**(new\_els, new\_ele,els, count\_new)

        return old\_nodes

    def **o**(self, \*nodes):

        """Чинит ссылки"""

        for node in nodes:

            if node.next == None:

                self.tail = node

            else:

                node.next.prev = node

            if node.prev == None:

                self.head = node

            else:

                node.prev.next = node

    def **xch**(self, n1, n2):

        """Обмен двух элементов"""

        n1.prev, n2.prev = n2.prev, n1.prev

        n1.next, n2.next = n2.next, n1.next

        self.**o**(n1, n2)

    def **tol**(self):

        """Возвращает список как массив"""

        lst = []

        for node in self.**forwardn**():

            lst.**append**(node)

        return lst

    def **shuffle**(self):

        """Перемешивает список"""

        lst = self.**tol**()

        shuffle(lst)

        self.**clr**()

        self.**extendn**(lst)

    def **get\_all**(self):

        """Дает все данные списка"""

        return (self.head, self.tail, self.count)

    def **clr**(self):

        """Очищает список"""

        self.head = None

        self.tail = None

        self.count = 0

    def **\_\_str\_\_**(self):

        """Строковое представление списка"""

        nodes\_concat = 'llist: ['

        nodes\_concat += ', '.**join**(**str**(el) for el in self)

        nodes\_concat += ']'

        return nodes\_concat