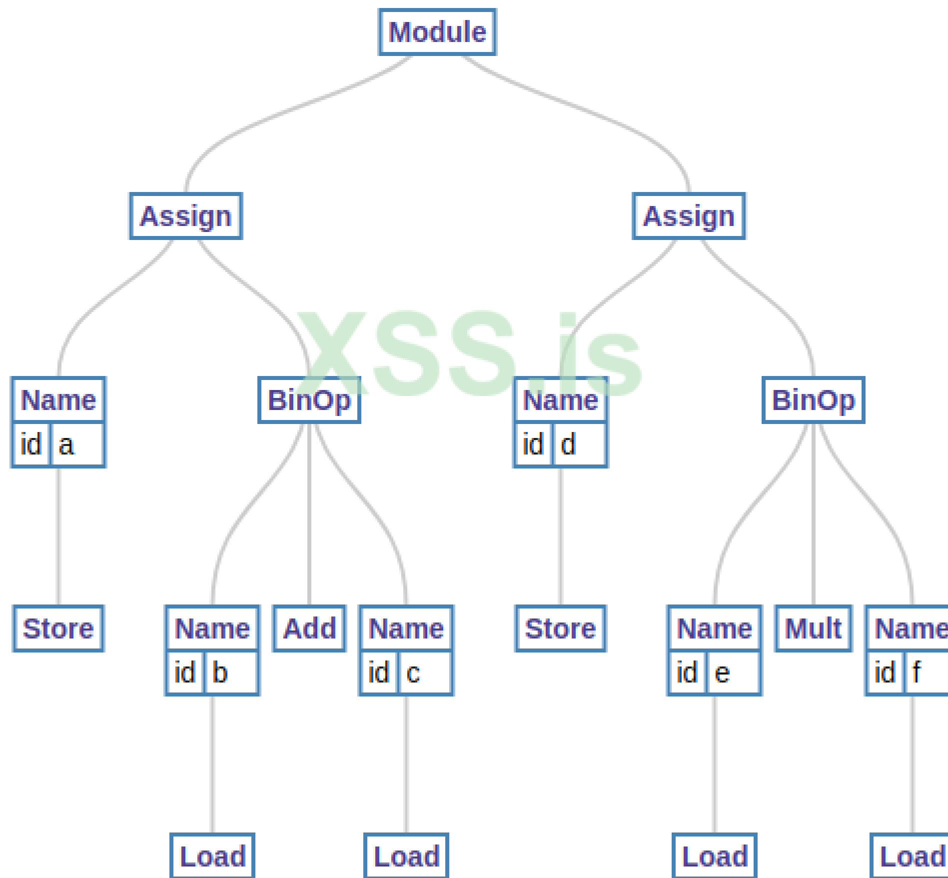
Статья Обфускация C/C++ кода с помощью Python и libclang

Привет, друзья. В нашем уютненьком комьюнити то и дело всплывают вопросы про обфускацию C и/или C++. Почти в каждой такой теме я пишу, мол все просто: берете libclang или pycparser или srcml (в зависимости от ваших нужд) и пишите себе обфускатор. Но в ответ частенько натыкаюсь на возражения, мол «как просто»? «Ничего же не понятно». В плоть до обвинений меня в том, что я только трепать языком на форуме могу. Ну что ж, мне остается только одно — показать вам, дорогие мои кулхацкеры, как писать обфускатор для C и/или C++ с использованием либшланг (ну libclang, ну вы поняли, такая очень смешная шутка). Для этого мы будем использовать самый любимый язык всех кулхацкеров мира — Python, а обфусцировать мы будем на уровне абстрактного синтаксического дерева.

Для начала немножечко дисклеймеров. Во-первых, задача этой статьи показать вам, что и как делается, а не забросать вас элитными алгоритмами и готовыми тулзами. Так что все алгоритмы достаточно простые, я бы даже сказал, слишком простые для современного обфускатора, но да ладно. Вы же статьи читаете, чтобы учиться, правильно? Во-вторых, мой скилл Питона немножечко заржавел, так что не упускайте возможность потроллить ветерана форумных срачей недостатками его кода. В-третьих, обфускатор в текущей реализации делает несколько предположений о том, как выглядит ваш C/C++ код. Для того, чтобы он работал на всем множестве C/C++ исходников придется его серьезно допиливать, что выходит за рамки одной статьи. Но надеюсь, что с теми базовыми знаниями, которые вы получите в этой статье, проблем с этим у вас не возникнет.

Итак, мы будем обфусцировать код на базе абстрактного синтаксического дерева. Что же это такое? В реализации подавляющего большинства компиляторов (разве что Lisp и Forth может без этого обойтись) для внутреннего представления кода используется древовидная рекурсивная структура, которая называется AST (abstract syntax tree). Давайте рассмотрим следующий пример, основанный на AST языка Питон.
Python:

```
a = b + c
d = e * f
```



Вот такие простые две строчки питоновского кода были переведены в такое громоздкое на первый взгляд древовидное представление. Вы можете зайти на сайт https://vpyast.appspot.com/ и сами поэкспериментировать с кодом, рассматривая различные синтаксические деревья, создаваемые для разного кода. С первого взгляда такое представление кажется избыточным и неудобным, однако оно содержит в себе всю необходимую информацию для работы с исходным кодом. Да и в принципе, синтаксис языка программирования в большинстве случаев сам по себе рекурсивная структура (например, выражения могут иметь вложенные выражения), поэтому представления исходного кода в виде дерева, ну… как минимум логично. И да, я пробовал для обфускаторов использовать регулярные выражения и токенизаторы, но ничего особо хорошего из этого не вышло.

Давайте рассмотрим структуру AST на картинке подробнее. Как мы видим в корне дерева находится узел, который олицетворяет питоновский модуль. У модуля есть два потомка — наши две строчки с выражениями присваивания. У каждого выражения присваивания есть потомок, определяющий кому производится присваивание (в данном случае это — идентификаторы «a» и «d» соответственно), а так же потомок, определяющий что именно будет присвоено. В нашем дереве этим потомком является

бинарный оператор, у которого соответсвенно есть левая часть, оператор и правая часть. Ну и так далее. Даже небольшие фрагменты исходного кода могут генерировать большие абстрактные синтаксические деревья. В контексте нашей сегодняшней задачи нам нужно каким-то образом обойти подобную древовидную структуру, выделить интересующие нас узлы дерева и произвести их модификации.

Не то чтобы очень удобным, но точно наиболее часто используемым методом обхода абстрактного синтаксического дерева является паттерн «Посетитель» (он же «Visitor»). Его мы и реализуем для обхода абстрактного синтаксического дерева библиотеки либшланг. Мюсье «Посититель» будет «посещать» узел дерева, а затем рекурсивно вызывать себя же для всех потомков текущего узла. В зависимости от типа текущего узла, мюсье «Посетитель» будет вызывать наши колбеки для обработки. По сути дела, это алгоритм «поиска в глубину» («depth first search»), если вам проще думать об этом в терминах институтского курса «Алгоритмы и структуры данных».

Ну что ж, с вводной теоретической частью мы покончили, давайте плавно переходить к практике. Для разработки нам потребуется установленный интерпретатор языка программирования Python, я буду использовать версию 3.8, потому что люблю подобие статической типизации и некоторые другие фичи современных Питонов. Кроме того, нам понадобится библиотека либшланг с байндингами для нашего Питона. Если вы сидите на Линуксах, то библиотека и байндинги скорее всего будут установлены вместе с компилятором clang. Но если вы на Венде или не хотите ставить весь компилятор, то произвести установку можно с помощью утилиты pip. Нужно запустить комнаду «pip install libclang» и байндинги и библиотека будет установлена. Обратите внимание, что пакет libclang из PyPl содержит только 64-битную версию библиотеки либшланг. То есть вам либо понадобится 64-битный интерпретатор Питона, либо поставить байндинги отдельно «pip install clang» и скачать 32-битную библиотеку либшланг с официального сайта LLVM.

Давайте начнем писать код с создания общего базового класса для всех обфусцирующих алгоритмов, который будет реализовывать парсинг исходного кода в абстрактное синтаксическое дерево, рекурсивное посещение узлов этого дерева по паттерну «Посетитель», а так же будет запоминать список модификаций, которые мы будет требовать произвести над исходным кодом, производить их и возвращать модифицированный исходный код. Рассмотрим следующий фрагмент кода:
Python:

```
from clang.cindex import CursorKind
from clang.cindex import Cursor
from clang.cindex import Index

#---------------------------------#
# Базовый класс для обфускаторов, #
# реализует паттерн "Посетитель"  #
# для обхода дерева исходников    #
#---------------------------------#
class Visitor:
    #--------------------#
    # Конструктор класса #
    #--------------------#
    def __init__(self):
        self._edit = [] # Список модификаций исходного кода
        self._code = "" # Сохраненная строка исходного кода
```

Сначала мы импортируем необходимые нам классы из библиотеки либшланг, затем создаем класс с конструктором, в котором инициализируем дефолтными значениями поля класса, которые нам потом пригодятся. Теперь реализуем методы, получающие на вход исходный код и возвращающие обфусцированный код (пока сделаем заглушки для приватных методов рекурсивного посещения узлов и применения списка модификаций):
Python:

```
#---------------------------------------#
# Метод загружает и обрабатывает исходный #
# код из указанного в параметрах файла    #
#---------------------------------------#
def process_file(self, path:str) -> str:
    with open(path, 'r', encoding='UTF-8') as fil:
        return self.process_source(fil.read())

#------------------------------------------#
# Метод обрабатывает переданный исходный код #
#------------------------------------------#
def process_source(self, src:str) -> str:
    # Сохраняем исходник
    self._code = src

    # Подготавливаем параметры парсинга
    unsaved = [('__file__.cpp', src)]
    cindex  = Index.create(False)
    args    = []

    # Парсим исходный код либшлангом
    translation_unit = cindex.parse(
        unsaved_files=unsaved,
        path='__file__.cpp',
        args= args
    )

    # Дублируем ссылку на корень дерева,
    # чтобы добавлять всем потомкам
    # ссылки на их родительские узлы
    root = translation_unit.cursor
    root.parent = None

    # Посещаем узлы
    self._visit(root)

    # Применяем модификации
    self._apply_patches()

    # Возвращаем код
    return self._code

#-----------------------------#
# Приватный метод осуществляет  #
# посещение узла и его потомков #
#-----------------------------#
def _visit(self, node:Cursor):
    pass

#-------------------------#
# Приватный метод применяет #
```

```
# все модификации к коду    #
#--------------------------#
def _apply_patches(self):
    pass
```

Метод process_file предельно прост: он считывает исходный код из указанного файла и передает его методу process_source, который принимает исходный код в виде строки. В методе process_source мы в первую очередь сохраняем исходный код в поле класса для того, чтобы в последствии производить над ним изменения. Затем мы создаем индекс библиотеки либшланг, подготавливаем некоторые параметры парсинга исходного кода и дополнительные аргументы. Следующим шагом мы вызываем методы библиотеки либшланг для парсинга исходного кода в абстрактное синтаксическое дерево, которое в контексте библиотеки называется курсором (я не знаю почему, видимо курсором является некая ссылка на текущий элемент дерев и его потомков). Потом мы делаем копию корневого элемента, чтобы иметь возможность добавлять курсору и его потомкам поле parent (ссылка на родительский элемент, об этом чуть позже) и запускаем обход дерева вызовом приватного метода _visit, который мы реализуем чуть позже. После посещения всех узлов дерева и оформления списка модификаций мы вызываем метод _apply_patches, который эти модификации применит к исходному кода, мы тоже реализуем его позже. Ну, и собственно, вернем модифицированный исходный код.

Теперь следует сделать небольшое лирическое отступление. Мы должны понимать, что библиотека либшланг являет собой полноценный компилятор языков C и C++. Мы используем только малую его часть, которая парсит исходный код и представляет его в виде абстрактного синтаксического дерева. C и C++ являются немного (совсем капелечку) старыми языкам и не имеют модульной структуры (ну как минимум до стандарта C++20 не имеют). Своего рода модульность у них достигается с помощью препроцессора. Чтобы библиотека либшланг могла нормально распарсить исходный код ей нужно либо передать уже пред обработанный препроцессором код, либо в дополнительных аргументах (переменная args) передать такие аргументы, чтобы она нашла корректные заголовочные файлы и дефайны режимов сборки. На Линуксах в большинстве случаев библиотека без посторонней помощи подтянет заголовочные файлы и дефайны от GCC. На Венде возможно придется попотеть с этим, мы можем обсудить этот вопрос в комментах и постараться решить все проблемы с этим связанные, но я бы рекомендовал вам просто получить пред обработанные препроцессором исходные коды от вашего компилятора и уже их скормить нашему либшлангу. В MinGW и Clang для этого служит параметр «-E», в компиляторе из состава Visual Studio — параметр «/P».

Ладно, теперь давайте напишем специальный отладочный метод, который будет выводить нам структуру абстрактного синтаксического дерева в более-менее приятном виде и попробуем распарсить наш первый исходный код.
Python:

```
#--------------------------------#
# Базовый класс для обфускаторов, #
# реализует паттерн "Посетитель"  #
# для обхода дерева исходников    #
#--------------------------------#
class Visitor:
    #--------------------#
    # Конструктор класса #
    #--------------------#
    def __init__(self):
        self._edit = [] # Список модификаций исходного кода
        self._code = "" # Сохраненная строка исходного кода

    #----------------------------------------#
    # Метод загружает и обрабатывает исходный #
    # код из указанного в параметрах файла    #
    #----------------------------------------#
    def process_file(self, path:str) -> str:
        with open(path, 'r', encoding='UTF-8') as fil:
            return self.process_source(fil.read())

    #-------------------------------------------#
    # Метод обрабатывает переданный исходный код #
    #-------------------------------------------#
    def process_source(self, src:str) -> str:
        # Сохраняем исходник
        self._code = src

        # Подготавливаем параметры парсинга
        unsaved = [('__file__.cpp', src)]
        cindex  = Index.create(False)
        args    = []

        # Парсим исходный код либшлангом
        translation_unit = cindex.parse(
            unsaved_files=unsaved,
            path='__file__.cpp',
            args= args
        )

        # Дублируем ссылку на корень дерева,
        # чтобы добавлять всем потомкам
        # ссылки на их родительские узлы
        root = translation_unit.cursor
        root.parent = None

        # Выведем дерево в stdout
        self.print_ast(root)

        # Посещаем узлы
        self._visit(root)
```

```
        # Применяем модификации
        self._apply_patches()

        # Возвращаем код
        return self._code

    #------------------------------------------------------#
    # Метод выводит в консоль абстрактное синтаксическое #
    # дерево, начиная с указанного узла вниз по потомкам #
    #------------------------------------------------------#
    def print_ast(self, node:Cursor, indent:int = 0):
        # Получаем информацию об узле
        colm = node.location.column
        line = node.location.line
        text = node.spelling
        kind = node.kind

        # Определяем формат вывода информации
        frm = '{kind} {line}:{colm} {text}'
        frm = '-' * indent + frm

        # Выводим в консоль
        print(frm.format(
            colm=colm,
            line=line,
            text=text,
            kind=kind
        ))

        # Рекурсивно выводим потомков
        for child in node.get_children():
            self.print_ast(child, indent+1)

#--------------------------#
# Юнит тесты на пол шишечки #
#--------------------------#
if __name__ == '__main__':
    Visitor().process_source('''
        extern "C" void puts(const char* str);

        int TestAdd(int a, int b) {
            return a + b;
        }

        void TestPuts() {
            puts("Hello World");
            puts("Goodbye World");
        }
    ''')
```

Метод print_ast получает интересующие нас параметры текущего узла, такие как его расположение в файле исходного кода, его тип (kind в терминах библиотеки) и его строковое представление, если такое есть. Затем мы выводит эту информацию в stdout и рекурсивно вызываем метод (увеличивая индентацию на 1, чтобы иметь возможность отличать иерархию узлов родителей и потомков). Так же мы добавили маленький юниттест, чтобы быть уверенным, что парсинг происходит нормально. Давайте запустим код и посмотрим на дерево, которое нам выведет print_ast.
Code:

```
CursorKind.TRANSLATION_UNIT 0:0 __file__.cpp
-CursorKind.UNEXPOSED_DECL 2:16
--CursorKind.FUNCTION_DECL 2:25 puts
---CursorKind.PARM_DECL 2:42 str
-CursorKind.FUNCTION_DECL 4:13 TestAdd
--CursorKind.PARM_DECL 4:25 a
--CursorKind.PARM_DECL 4:32 b
--CursorKind.COMPOUND_STMT 4:35
---CursorKind.RETURN_STMT 5:13
----CursorKind.BINARY_OPERATOR 5:20
-----CursorKind.UNEXPOSED_EXPR 5:20 a
------CursorKind.DECL_REF_EXPR 5:20 a
-----CursorKind.UNEXPOSED_EXPR 5:24 b
------CursorKind.DECL_REF_EXPR 5:24 b
-CursorKind.FUNCTION_DECL 8:14 TestPuts
--CursorKind.COMPOUND_STMT 8:25
---CursorKind.CALL_EXPR 9:13 puts
----CursorKind.UNEXPOSED_EXPR 9:13 puts
-----CursorKind.DECL_REF_EXPR 9:13 puts
----CursorKind.UNEXPOSED_EXPR 9:18
-----CursorKind.STRING_LITERAL 9:18 "Hello World"
---CursorKind.CALL_EXPR 10:13 puts
----CursorKind.UNEXPOSED_EXPR 10:13 puts
-----CursorKind.DECL_REF_EXPR 10:13 puts
----CursorKind.UNEXPOSED_EXPR 10:18
-----CursorKind.STRING_LITERAL 10:18 "Goodbye World"
```

Структура дерева должна быть интуитивно понятной, если вы разобрались с представлениями исходного кода в виде AST в языке Питон в самом начале статьи. В корне дерева находится своего рода модуль (на самом деле в C/C++ и в частности в либшланг это принято называть единицей трансляции). Далее мы видим определение внешней функции puts без тела и определения двух функций. Потомками каждой из них являются отдельные выражения языка, описанные в их телах. Ну и так далее. Метод print_ast будет очень полезным для понимания структуры дерева библиотеки либшланг и для отладки, если вдруг наш обфускатор будет модифицировать код неправильно.

Теперь давайте объявим методы коллбеки, которые будет вызывать наш класс при посещении узла определенного типа и реализуем функции посещения. Обратите внимание, что функции коллбеки (функции вида visit_*) — не что иное как заглушки. Их мы будем реализовывать в дочерних классах (унаследованных от класса visitor), в которых при посещении узла будут производится какие-либо действия. Таким образом в дочерних классах мы сможем переопределить методы посещения только тех узлов, которые нам нужны, а для всех остальных будут вызваны методы базового класса (класса Visitor) — то есть никакого действия не будет произведено.
Python:

```
#-------------------------------------#
# Коллбек для узла строковых литералов #
#-------------------------------------#
def visit_string_literal(self, node:Cursor):
    pass

#----------------------------------------#
# Коллбек для узла объединенных выражений #
#----------------------------------------#
def visit_compound_statement(self, node:Cursor):
    pass

#-------------------------------------#
# Коллбек для узла бинарного оператора #
#-------------------------------------#
def visit_binary_operator(self, node:Cursor):
    pass

#------------------------------------#
# Коллбек для узла унарного оператора #
#------------------------------------#
def visit_unary_operator(self, node:Cursor):
    pass

#---------------------------------------#
# Коллбек для узла определения чего-либо #
#---------------------------------------#
def visit_declaration_statement(self, node:Cursor):
    pass

#-------------------------------------------------#
# Коллбек для узла выражения вызова функции/метода #
#-------------------------------------------------#
def visit_call_expression(self, node:Cursor):
    pass

#----------------------------------#
# Коллбек для узла выражения return #
#----------------------------------#
def visit_return_statement(self, node:Cursor):
    pass

#----------------------------------#
# Коллбек для узла выражения switch #
#----------------------------------#
def visit_switch_statement(self, node:Cursor):
    pass

#---------------------------------#
# Коллбек для узла выражения while #
#---------------------------------#
```

```
def visit_while_statement(self, node:Cursor):
    pass

#-------------------------------#
# Коллбек для узла выражения for #
#-------------------------------#
def visit_for_statement(self, node:Cursor):
    pass

#------------------------------#
# Коллбек для узла выражения if #
#------------------------------#
def visit_if_statement(self, node:Cursor):
    pass

#------------------------------#
# Коллбек для узла выражения do #
#------------------------------#
def visit_do_statement(self, node:Cursor):
    pass

#------------------------------#
# Приватный метод осуществляет  #
# посещение узла и его потомков #
#------------------------------#
def _visit(self, node:Cursor):
    self._visit_any(node)
    for child in node.get_children():
        child.parent = node
        self._visit(child)

#-----------------------------#
# Приватный метод осуществляет #
# непосредственное посещение   #
# указанного узла любого типа  #
#-----------------------------#
def _visit_any(self, node:Cursor):
    jump_table = { # Таблица диспетчиризации вызовов
        CursorKind.BINARY_OPERATOR: self.visit_binary_operator,
        CursorKind.UNARY_OPERATOR:  self.visit_unary_operator,
        CursorKind.STRING_LITERAL:  self.visit_string_literal,
        CursorKind.COMPOUND_STMT:   self.visit_compound_statement,
        CursorKind.RETURN_STMT:     self.visit_return_statement,
        CursorKind.SWITCH_STMT:     self.visit_switch_statement,
        CursorKind.WHILE_STMT:      self.visit_while_statement,
        CursorKind.DECL_STMT:       self.visit_declaration_statement,
        CursorKind.CALL_EXPR:       self.visit_call_expression,
        CursorKind.FOR_STMT:        self.visit_for_statement,
        CursorKind.DO_STMT:         self.visit_do_statement,
        CursorKind.IF_STMT:         self.visit_if_statement,
    }
```

```
    # В зависимости от типа узла
    # вызываем соответствующий
    # метод для его обработки
    if node.kind in jump_table:
        jump_table[node.kind](node)
```

Приватные методы _visit и _visit_any, собственно, и реализуют паттерн «Посетитель». Метод _visit при посещении узла дерева вызывает метод _visit_any, а потом рекурсивно вызывает себя же для всех потомков текущего узла. Метод _visit_any в своем функционале определяет тип текущего узла и вызывает соответствующий метод коллбек. Обратите внимание, что поскольку в Питоне нет конструкции switch, очень часто ее эмулируют с помощью словаря. Это достаточно удобно и зачастую на практике оказывается быстрее, чем писать огромную портянку из if-elif-else конструкций. Так же обратите внимание, что в методе _visit для всех потомков устанавливается ссылка на родителя. Она нам понадобится чуть позже, когда мы будем анализировать, где в дереве находится тот или иной узел. К сожалению, байндинги либшланга не предоставляют такой информации в достоверном виде, поэтому мы просто будем расширять объект курсора либшланга дополнительным полем (Питон же динамический язык в конце то концов). Важно отметить, что типов узлов гораздо больше, чем мы реализовали. Мы реализовали только те узлы, которые нам нужно будет посещать в контексте этой статьи. Теперь давайте реализуем методы для внесения изменений в исходный код и их применения.
Python:

```
#----------------------------------#
# Метод возвращает смещения начала #
# и конца указанного узла дерева   #
#----------------------------------#
def get_node_extent(self, node:Cursor) -> (int, int):
    beg = node.extent.start.offset
    end = node.extent.end.offset
    return (beg, end)

#--------------------------------------#
# Метод заменяет указанный узел на код #
#--------------------------------------#
def replace_node(self, node:Cursor, src:str):
    (beg, end) = self.get_node_extent(node)
    self._edit.append((beg, end, src))

#-------------------------------------------#
# Метод вставляет код перед указанным узлом #
#-------------------------------------------#
def insert_before(self, node:Cursor, src:str):
    (beg, _) = self.get_node_extent(node)
    self._edit.append((beg, beg, src))

#--------------------------------------------#
# Метод вставляет код в начало блока с кодом #
#--------------------------------------------#
def insert_compound(self, node:Cursor, src:str):
    (beg, _) = self.get_node_extent(node)
    self._edit.append((beg+1, beg+1, src))

#--------------------------#
# Приватный метод применяет #
# все модификации к коду    #
#--------------------------#
def _apply_patches(self):
    # Сортируем список модификаций
    # по координате начала вставки
    key = lambda itm: itm[0]
    self._edit.sort(key=key)

    encd = 'UTF-8' # Код -> массив байт
    code = bytearray(self._code, encd)

    pos = 0 # Модифицируем
    for edit in self._edit:
        src = edit[2].encode(encd)
        beg = pos + edit[0]
        end = pos + edit[1]

        code[beg:end] = src
        pos += beg - end + len(src)
```

```
# Массив байт -> обратно в код
self._code = code.decode(encd)
```

Поскольку байндинги либшланга достаточно ограничены в том, что они могут вытворять, в частности нельзя производить модификации узлов дерева, а потом просто сдампить дерево обратно в исходный код, нам придется делать модификации исходного кода самим. Метод get_node_extent получает смещение начала и конца переданного узла дерева. Метод replace_node добавляет в список модификацию исходного кода, которая заменит переданный узел и всех его потомков на фрагмент исходного кода. Метод insert_before добавит в список модификацию, которая вставит фрагмент исходного кода перед переданным узлом. А метод insert_compound (compound в терминах либшланга означает блок с кодом между фигурными скобками) добавляет модификацию, которая вставит фрагмент исходного кода в начало этого блока. Метод _apply_patches применяет все модификации по порядку их следования в буфере с исходным кодом и запоминает, насколько модификации сдвинули символы в исходном коде. Далее быстренько рассмотрим другие вспомогательные методы класса.
Python:

```
#------------------------------------#
# Метод возвращает узел родительского #
# блока кода, если такой существует   #
#------------------------------------#
def get_parent_compound(self, node:Cursor):
    node = node.parent
    while node is not None:
        if node.kind == CursorKind.COMPOUND_STMT:
            return node
        else: node = node.parent

    return None

#----------------------------------------#
# Метод возвращает истину, если родителем #
# для текущего узла является блок кода    #
#----------------------------------------#
def is_parent_compound(self, node:Cursor):
    return self.is_parent(node, CursorKind.COMPOUND_STMT)

#------------------------------------------------------#
# Метод проверяет, является ли родитель узла такого типа #
#------------------------------------------------------#
def is_parent(self, node:Cursor, kind:CursorKind) -> bool:
    return node.parent.kind == kind

#----------------------------------------------#
# Метод возвращает истину, если родителем        #
# текущего блока объединенных выражений является #
# функция или конструктор/деструктор или метод   #
#----------------------------------------------#
def is_top_level_compound(self, node:Cursor):
    parent = node.parent
    if parent.kind == CursorKind.FUNCTION_DECL: return True
    if parent.kind == CursorKind.CONSTRUCTOR:   return True
    if parent.kind == CursorKind.DESTRUCTOR:    return True
    if parent.kind == CursorKind.CXX_METHOD:    return True

    return False

#----------------------------------#
# Метод возвращает исходный код для #
# указанного узла дерева исходников #
#----------------------------------#
def get_source(self, node:Cursor) -> str:
    (beg, end) = self.get_node_extent(node)
    return self._code[beg:end]
```

Метод get_source возвращает исходный код, который ассоциирован с переданным узлом и его потомками, по сути он необходим для отладки, как и метод print_ast.

Остальные методы проверяют родителей текущего узла. Метод get_parent_compound находит родительский блок кода (если такой есть), is_top_level_compound проверяет, является ли текущий блок основным блоком кода функции, метода или конструктора/деструктора и так далее. Эти методы пригодятся нам чуть позже.

Для генерации дешифраторов строк и генерации мусорного кода нам понадобится вспомогательный класс, который будет генерировать нам уникальные имена для определения переменных. Давайте же его реализуем, в этом классе все должно быть интуитивно понятно. Просто статический метод, который генерирует имена с уникальными индексами, увеличивая счетчик.
Python:

```
#--------------------------#
# Генератор уникальных имен #
#--------------------------#
class NameGenerator:
    _dic:dict = {} # Словарь имен и счетчиков

    #-----------------------------#
    # Статический метод генерирует #
    # уникальное имя для различных #
    # переменных, добавляемых в    #
    # исходный код                 #
    #-----------------------------#
    @staticmethod
    def get_name(name:str) -> str:
        # Инициализируем счетчик, если
        # таких имен еще не запрашивали
        if name not in NameGenerator._dic:
            NameGenerator._dic[name] = 0

        # Сохраняем текущее значение
        # счетчика и увеличиваем его
        idx = NameGenerator._dic[name]
        NameGenerator._dic[name] += 1

        # Возвращаем уникальное
        # имя в нашем формате
        fmt = '__{0}_{1:08X}__'
        return fmt.format(name, idx)

#--------------------------#
# Юнит тесты на пол шишечки #
#--------------------------#
if __name__ == '__main__':
    for _ in range(0, 10):
        print(NameGenerator.get_name('one'))
        print(NameGenerator.get_name('two'))
```

Теперь давайте реализуем несколько алгоритмов шифрования строк. Безусловно они будут весьма примитивны и для современного обфускатора нужно было бы придумать что-то в разы лучше, но они дадут вам понять общую концепцию, при которой разные алгоритмы шифрования абстрагированы друг от друга, но реализуют общий интерфейс. Таким образом, для шифрования каждой отдельной строки можно выбрать любой алгоритм из реализованных или псевдо-случайный из них. Рассмотрим следующий код.
Python:

```
from namegen import NameGenerator

import random
import abc

#----------------------------#
# Абстрактный класс различных #
# алгоритмов шифрования строк #
#----------------------------#
class EncryptorAbstract(abc.ABC):
    #----------------------#
    # Абстрактный метод для #
    # собственно шифрования #
    #----------------------#
    @abc.abstractmethod
    def encrypt(self, stn:str, uni:bool):
        pass

    #--------------------------------#
    # Абстрактное свойство возвращает #
    # фрагмент кода для дешифрования  #
    #--------------------------------#
    @abc.abstractproperty
    def decryptor(self):
        pass

    #--------------------------------#
    # Абстрактное свойство возвращает #
    # имя буффера расшифровки строки  #
    #--------------------------------#
    @abc.abstractproperty
    def replacement(self):
        pass

#------------------------------------------#
# Класс шифрования строк однобайтовым ксором #
#------------------------------------------#
class EncryptorSimpleXor(EncryptorAbstract):
    #------------------#
    # Конструктор класса #
    #------------------#
    def __init__(self):
        self._tpl  = 'unsigned char {enam}[{elen}] = {{ {estr} }};'
        self._tpl += '{btyp} {bnam}[{elen}+1];'
        self._tpl += 'for(int {inam} = 0; {inam} < {elen}; {inam}++) {{'
        self._tpl +=   '{bnam}[{inam}] = ({btyp})({enam}[{inam}] ^ 0x{key:02X});'
        self._tpl += '}}'
        self._tpl += '{bnam}[{elen}] = ({btyp})0;'

    #-----------------------------------#
    # Метод производит шифрование строки #
```

```
    #------------------------------------#
    def encrypt(self, stn:str, uni:bool):
        # Генерация ключа и шифрование
        key = random.randint(1, 0xFF)
        enc = self._encrypt(stn, key)

        # Генерация имен переменных шаблона
        btyp = 'wchar_t' if uni else 'char'
        enam = NameGenerator.get_name('enc')
        bnam = NameGenerator.get_name('buf')
        inam = NameGenerator.get_name('i')

        # Преобразования шифр-данных в строки
        estr = ['0x{0:02X}'.format(x) for x in enc]
        estr = ','.join(estr)

        self._nam = bnam
        self._dec = self._tpl.format(
            enam=enam, bnam=bnam, inam=inam, btyp=btyp,
            estr=estr, elen=len(enc), key=key
        )

    #--------------------------------#
    # Свойство шифратора возвращает  #
    # фрагмент кода для дешифрования #
    #--------------------------------#
    @property
    def decryptor(self):
        return self._dec

    #--------------------------------#
    # Свойство шифратора возвращает  #
    # имя буффера расшифровки строки #
    #--------------------------------#
    @property
    def replacement(self):
        return self._nam

    #------------------------------#
    # Приватная функция шифрования #
    #------------------------------#
    def _encrypt(self, stn:str, key:int):
        res = []
        for chx in stn:
            byt = ord(chx) ^ key
            res.append(byt & 0xFF)

        return res

#--------------------------------------------#
# Класс шифрования строк сложением с ключом #
```

```
#--------------------------------------------#
class EncryptorSimpleAdd(EncryptorAbstract):
    #-------------------#
    # Конструктор класса #
    #-------------------#
    def __init__(self):
        self._tpl  = 'unsigned char {enam}[{elen}] = {{ {estr} }};'
        self._tpl += '{btyp} {bnam}[{elen}+1];'
        self._tpl += 'for(int {inam} = 0; {inam} < {elen}; {inam}++) {{'
        self._tpl +=   'unsigned int {tnam} = {enam}[{inam}] + 0x{key:02X};'
        self._tpl +=   '{bnam}[{inam}] = ({btyp})({tnam} & 0xFF);'
        self._tpl += '}}'
        self._tpl += '{bnam}[{elen}] = ({btyp})0;'

    #-----------------------------------#
    # Метод производит шифрование строки #
    #-----------------------------------#
    def encrypt(self, stn:str, uni:bool):
        # Генерация ключа и шифрование
        key = random.randint(1, 0xFF)
        enc = self._encrypt(stn, key)

        # Генерация имен переменных шаблона
        btyp = 'wchar_t' if uni else 'char'
        enam = NameGenerator.get_name('enc')
        bnam = NameGenerator.get_name('buf')
        tnam = NameGenerator.get_name('tmp')
        inam = NameGenerator.get_name('i')

        # Преобразования шифр-данных в строки
        estr = ['0x{0:02X}'.format(x) for x in enc]
        estr = ','.join(estr)

        self._nam = bnam
        self._dec = self._tpl.format(
            enam=enam, bnam=bnam, inam=inam, btyp=btyp,
            estr=estr, tnam=tnam, elen=len(enc), key=key
        )

    #-------------------------------#
    # Свойство шифратора возвращает  #
    # фрагмент кода для дешифрования #
    #-------------------------------#
    @property
    def decryptor(self):
        return self._dec

    #-------------------------------#
    # Свойство шифратора возвращает  #
    # имя буффера расшифровки строки #
    #-------------------------------#
```

```
    @property
    def replacement(self):
        return self._nam

    #------------------------------#
    # Приватная функция шифрования #
    #------------------------------#
    def _encrypt(self, stn:str, key:int):
        res = []
        for chx in stn:
            byt = ord(chx) - key
            if byt < 0: byt += 256
            res.append(byt & 0xFF)

        return res

#---------------------------#
# Юнит тесты на пол шишечки #
#---------------------------#
if __name__ == '__main__':
    enc = EncryptorSimpleXor()
    enc.encrypt("Hello World", True)
    print(enc.replacement)
    print(enc.decryptor)

    enc = EncryptorSimpleXor()
    enc.encrypt("Hello World", False)
    print(enc.replacement)
    print(enc.decryptor)

    enc = EncryptorSimpleAdd()
    enc.encrypt("Hello World", True)
    print(enc.replacement)
    print(enc.decryptor)

    enc = EncryptorSimpleAdd()
    enc.encrypt("Hello World", False)
    print(enc.replacement)
    print(enc.decryptor)
```

И так мы имеем базовый для всех алгоритмов шифрования строк класс EncryptorAbstract, который с помощью абстрактных методов определяет универсальный интерфейс, через который будут вызываться все реализованные алгоритмы. Затем мы создали два наследуемых от интерфейса класса, которые реализуют конкретные алгоритмы шифрования строк, а именно однобайтовый ксор (EncryptorSimpleXor) и однобайтовое сложение (EncryptorSimpleAdd). У обоих классов в конструкторе реализована подготовка шаблона исходного текста дешифратора, метод encrypt зашифровывает строку и формирует дешифратор. В последствии за

получение кода дешифратора отвечает свойство decryptor, а за получение имени буфера для расшифровки отвечает свойство replacement. Конкретная реализация алгоритмов в контексте статьи не имеет значения, она приведена в качестве примера.

Ну и наконец-то мы дошли до самого интересного, а именно — до реализации обфускатора строк. Как и говорилось ранее класс текущего обфускатора унаследован от класса, в котором мы реализовали паттерн «Посетитель» чуть ранее (класс Visitor). Он переопределяет единственный метод для посещения узлов — посещение строковых литералов. Рассмотрим следующий код.
Python:

```
from clang.cindex import Cursor

from encrgen import EncryptorSimpleXor
from encrgen import EncryptorSimpleAdd
from encrgen import EncryptorAbstract
from visitor import Visitor

import random

#-----------------------#
# Класс шифрования строк #
#-----------------------#
class Encryptor(Visitor):
    #-------------------------------------#
    # Коллбек для узла строковых литералов #
    #-------------------------------------#
    def visit_string_literal(self, node:Cursor):
        # Получаем блок кода, внутри которого
        # объявлен строковый литерал, если такого
        # нет (строка в глобальном скоупе), то
        # принципиально не обрабатываем ее
        block = self.get_parent_compound(node)
        if block is None: return

        # Получаем значение и параметры литерала
        (value, isuni) = self._extract_value(node)

        # Выбираем алгоритм и шифруем
        algo = self._get_algorithm()
        algo.encrypt(value, isuni)

        # Вставляем дешифратор и заменяем литерал
        self.insert_compound(block, algo.decryptor)
        self.replace_node(node, algo.replacement)

    #------------------------------------------#
    # Приватный метод выбирает псевдо-случайный #
    # алгоритм для шифрования текущей строки    #
    #------------------------------------------#
    def _get_algorithm(self) -> EncryptorAbstract:
        return random.choice([
            EncryptorSimpleAdd,
            EncryptorSimpleXor
        ])()

    #-------------------------------------------#
    # Приватный метод получает значение литерала #
    #-------------------------------------------#
    def _extract_value(self, node:Cursor) -> (str, bool):
        # Хак-получение значения
        value = node.spelling
```

```
        # Определяем, является ли
        # строка wchar_t и обрезаем
        # префикс и кавычки строки
        isuni = value.startswith('L')
        if isuni: value = value[2:-1]
        else: value = value[1:-1]

        # Обрабатываем escape-символы
        value = value.encode('utf-8')
        value = value.decode('unicode_escape')
        return (value, isuni)

#---------------------------#
# Юнит тесты на пол шишечки #
#---------------------------#
if __name__ == '__main__':
    print(Encryptor().process_source('''
        static auto globalw = L"GLOBAL STRING";
        static auto globala = "GLOBAL STRING";

        class TestClass {
            TestClass() {
                auto constructorw = L"CONSTRUCTOR STRING";
                auto constructora = "CONSTRUCTOR STRING";
            }

            ~TestClass() {
                auto destructorw = L"DESTRUCTOR STRING";
                auto destructora = "DESTRUCTOR STRING";
            }

            void TestMethod() {
                auto methodw = L"METHOD STRING";
                auto methoda = "METHOD STRING";
            }
        };

        void TestFunction() {
            auto functionw = L"FUNCTION STRING";
            auto functiona = "FUNCTION STRING";
        }
    '''))
```

При посещении узла строкового литерала обфускатор пытается найти его родительский блок кода (в него мы будем вставлять код для расшифровки строки). Если такого блока кода нет, то скорее всего строка объявлена в виде глобальной переменной. В контексте статьи эту ситуацию мы рассматривать не будем и просто пропустим эти строки, но в вкратце могу сказать, что в этой ситуации можно

обработать глобальные строки, но код для расшифровки вставить в глобальные конструкторы (__attribute__ ((constructor)) в GCC/MinGW, в студии не знаю, как они объявляются).

Как только мы нашли блок, мы должны получить значение строкового литерала. И тут мы опять сталкиваемся с ограничениями байндингов либшланга, по крайней мере мы с гуглом не нашли более красивого метода это сделать. Поэтому мы берем текстовое представление (spelling) строкового литерала, определяем является ли литерал char или wchar_t строкой, обрезаем кавычки и заменяем все escape-символы в строке. Далее мы псевдо-случайным образом выбираем алгоритм шифрования, шифруем строку, добавляем дешифратор в начало родительского блока кода, а строковый литерал заменяем на имя буфера с расшифрованной строкой.

Обратите внимание, что как я и говорил в начале статьи, текущая реализация достаточно наивна в отношении кода, который она должна обрабатывать. В частности, не обрабатываются глобальные строки, инициализация строк в объявлении класса и в заголовках конструкторов так же либо не будут обработаны, либо будут обработаны ошибочно. Поэтому для вашей кодовой базы могут потребоваться более детальный анализ того, где находится строковый литерал.

Теперь давайте рассмотрим генерацию мусорного кода для нашего обфускатора. Опять же, «умные» генераторы мусорного кода тянут на несколько отдельных статей. Поэтому я покажу вам достаточно неплохую архитектуру для реализации генератора мусорного кода, но генерировать мы будет только один тип выражения. Рассмотрим следующий код.
Python:

```
from namegen import NameGenerator

import random
import abc

#----------------------------#
# Абстрактный класс отдельной #
# переменной мусорного кода   #
#----------------------------#
class VariableAbstract(abc.ABC):
    #--------------------#
    # Конструктор класса #
    #--------------------#
    def __init__(self):
        self._name = NameGenerator.get_name('tvar')
        self._init = self.value

    #------------------------#
    # Свойство возвращает имя #
    # текущей переменной      #
    #------------------------#
    @property
    def name(self) -> str:
        return self._name

    #-----------------------------------#
    # Свойство возвращает объявление для #
    # текущей переменной мусорного кода  #
    #-----------------------------------#
    @property
    def declaration(self) -> str:
        fmt = 'static {typ} {nam} = {ini};'
        return fmt.format(
            typ=self.type_name,
            nam=self._name,
            ini=self._init
        )

    #-------------------------------------#
    # Свойство возвращает псевдо-случайный #
    # оператор, применимый для переменной  #
    #-------------------------------------#
    @property
    def operator(self) -> str:
        ops = ['+', '-', '*', '&', '|', '^']
        return random.choice(ops)

    #---------------------------------#
    # Абстрактное свойство возвращает #
    # псевдо-случайное значение для   #
    # типа мусорной переменной        #
```

```
    #--------------------------------#
    @abc.abstractproperty
    def value(self) -> str:
        pass

    #---------------------------------#
    # Абстрактное свойство возвращает  #
    # название типа текущей переменной #
    #---------------------------------#
    @abc.abstractproperty
    def type_name(self) -> str:
        pass

#----------------------------#
# 64-битная беззнаковая целая #
# переменная мусорного кода   #
#----------------------------#
class VariableUint64(VariableAbstract):
    #------------------------#
    # Свойство возвращает тип #
    # мусорной переменной     #
    #------------------------#
    @property
    def type_name(self) -> str:
        return 'unsigned long long'

    #------------------------------------#
    # Свойство возвращает псевдо-случайное #
    # значение переменной текущего типа    #
    #------------------------------------#
    @property
    def value(self) -> str:
        val = random.randint(0, 0xFFFFFFFFFFFFFFFF)
        return '0x{0:016X}'.format(val)

#----------------------------#
# 32-битная беззнаковая целая #
# переменная мусорного кода   #
#----------------------------#
class VariableUint32(VariableAbstract):
    #------------------------#
    # Свойство возвращает тип #
    # мусорной переменной     #
    #------------------------#
    @property
    def type_name(self) -> str:
        return 'unsigned int'

    #------------------------------------#
    # Свойство возвращает псевдо-случайное #
    # значение переменной текущего типа    #
```

```
    #-------------------------------------#
    @property
    def value(self) -> str:
        val = random.randint(0, 0xFFFFFFFF)
        return '0x{0:08X}'.format(val)

#----------------------------#
# 16-битная беззнаковая целая #
# переменная мусорного кода   #
#----------------------------#
class VariableUint16(VariableAbstract):
    #------------------------#
    # Свойство возвращает тип #
    # мусорной переменной     #
    #------------------------#
    @property
    def type_name(self) -> str:
        return 'unsigned short'

    #-------------------------------------#
    # Свойство возвращает псевдо-случайное #
    # значение переменной текущего типа    #
    #-------------------------------------#
    @property
    def value(self) -> str:
        val = random.randint(0, 0xFFFF)
        return '0x{0:04X}'.format(val)

#---------------------------#
# 8-битная беззнаковая целая #
# переменная мусорного кода  #
#---------------------------#
class VariableUint8(VariableAbstract):
    #------------------------#
    # Свойство возвращает тип #
    # мусорной переменной     #
    #------------------------#
    @property
    def type_name(self) -> str:
        return 'unsigned char'

    #-------------------------------------#
    # Свойство возвращает псевдо-случайное #
    # значение переменной текущего типа    #
    #-------------------------------------#
    @property
    def value(self) -> str:
        val = random.randint(0, 0xFF)
        return '0x{0:02X}'.format(val)

#-------------------------#
```

```
# Генератор мусорного кода #
#-------------------------#
class TrashGenerator:
    #--------------------#
    # Конструктор класса #
    #--------------------#
    def __init__(self, mn = 2, mx = 8):
        nvar = random.randint(mn, mx)
        lvar = [
            VariableUint64,
            VariableUint32,
            VariableUint16,
            VariableUint8
        ]

        self._vars = []
        for _ in range(nvar):
            var = random.choice(lvar)
            self._vars.append(var())

    #--------------------------------#
    # Свойство возвращает определения #
    # для переменных мусорного кода   #
    #--------------------------------#
    @property
    def declaration(self):
        decl = [x.declaration for x in self._vars]
        return ' '.join(decl)

    #---------------------------------#
    # Свойство генерирует мусорный код #
    #---------------------------------#
    @property
    def trash(self):
        # Генерируем псевдо-случайные значения
        op1 = random.choice(self._vars).name
        var = random.choice(self._vars)
        typ = var.type_name
        opr = var.operator
        op2 = var.value
        res = var.name

        # Форматирует новую строку мусорного кода
        return '{res} = ({typ}){op1} {opr} {op2};'.format(
            res=res, op1=op1, typ=typ, opr=opr, op2=op2
        )

#--------------------------#
# Юнит тесты на пол шишечки #
#--------------------------#
if __name__ == '__main__':
```

```
tgen = TrashGenerator()
print(tgen.declaration)

for _ in range(0, 10):
    print(tgen.trash)
```

Мы объявили абстрактный класс VariableAbstract, который будет базовым для всех переменных мусорного кода, которые мы будем вводить. Для простоты мы введем только 4 типа мусорных переменных — целочисленные беззнаковые переменный размером 64, 32, 16 и 8 бит. Каждая из переменных знает свой тип, знает операторы, которые можно к ней применить, и знает, как генерировать псевдо-случайное число такого типа. В «умном» генераторе мусорного кода к этому добавится еще возможность хранения состояния и эмуляция операторов (для генерации непрозрачных предикатов), но об этом, наверное, в другой раз (если, конечно, у комьюнити есть интерес к таким темам). Генератор мусорного кода создает себе пул таких переменных псевдо-случайного размера, затем может генерировать псевдо-случайные математические выражение (для примера только одного типа) над двумя выбранными опять же псевдо-случайно переменными.

Теперь, когда мы научили наш генератор мусорного кода генерировать мусорный код (ммм, маслецо масленное), нам нужно подумать в какие места исходного кода мы можем его безопасно вставлять. Для начала нам нужно вставить объявление мусорных переменных в блок кода, который является потомком функции, метода или конструктора/деструктора. Потом в начало любого compound блока, кроме того, родителем которого является switch, мы можем вставить мусорный код. Перед многими выражениями, такими как if, do, while, for, switch мы можем уверенно вставить мусорный код. Перед выражениями присваивания и объявления переменных, которые находятся внутри функций/методов, тоже можем вставить мусорный код. Ну и так далее. Рассмотрим исходный код подобного класса.
Python:

```python
from clang.cindex import CursorKind
from clang.cindex import Cursor

from trashgen import TrashGenerator
from visitor  import Visitor

import random

class Trasher(Visitor):
    #--------------------#
    # Конструктор класса #
    #--------------------#
    def __init__(self):
        super().__init__() # Вызов конструктора родителя
        self._tgen = None  # Генератор мусорного кода

    #----------------------------------------#
    # Коллбек для узла объединенных выражений #
    #----------------------------------------#
    def visit_compound_statement(self, node:Cursor):
        if self.is_parent(node, CursorKind.SWITCH_STMT):
            return

        if self.is_top_level_compound(node):
            self._tgen = TrashGenerator()
            edit = self._tgen.declaration
        else: edit = self.generate_trash()

        self.insert_compound(node, edit)

    #---------------------------------------#
    # Коллбек для узла определения чего-либо #
    #---------------------------------------#
    def visit_declaration_statement(self, node:Cursor):
        if self.is_parent_compound(node):
            self.insert_trash_before(node)

    #-------------------------------------#
    # Коллбек для узла бинарного оператора #
    #-------------------------------------#
    def visit_binary_operator(self, node:Cursor):
        if self.is_parent_compound(node):
            self.insert_trash_before(node)

    #------------------------------------#
    # Коллбек для узла унарного оператора #
    #------------------------------------#
    def visit_unary_operator(self, node:Cursor):
        if self.is_parent_compound(node):
            self.insert_trash_before(node)
```

```
#-----------------------------------#
# Коллбек для узла выражения return #
#-----------------------------------#
def visit_return_statement(self, node:Cursor):
    self.insert_trash_before(node)

#--------------------------------------------------#
# Коллбек для узла выражения вызова функции/метода #
#--------------------------------------------------#
def visit_call_expression(self, node:Cursor):
    if self.is_parent_compound(node):
        self.insert_trash_before(node)

#-----------------------------------#
# Коллбек для узла выражения switch #
#-----------------------------------#
def visit_switch_statement(self, node:Cursor):
    self.insert_trash_before(node)

#----------------------------------#
# Коллбек для узла выражения while #
#----------------------------------#
def visit_while_statement(self, node:Cursor):
    self.insert_trash_before(node)

#--------------------------------#
# Коллбек для узла выражения for #
#--------------------------------#
def visit_for_statement(self, node:Cursor):
    self.insert_trash_before(node)

#-------------------------------#
# Коллбек для узла выражения if #
#-------------------------------#
def visit_if_statement(self, node:Cursor):
    self.insert_trash_before(node)

#-------------------------------#
# Коллбек для узла выражения do #
#-------------------------------#
def visit_do_statement(self, node:Cursor):
    self.insert_trash_before(node)

#------------------------------------------#
# Метод вставляет мусорный код перед узлом #
#------------------------------------------#
def insert_trash_before(self, node:Cursor):
    edit = self.generate_trash()
    self.insert_before(node, edit)

#-----------------------------------#
```

```
    # Метод генерирует псевдо-случайное #
    # количество строк мусорного кода   #
    #-----------------------------------#
    def generate_trash(self, mn = 1, mx = 3):
        num = random.randint(mn, mx)
        res = [self._tgen.trash for _ in range(num)]
        return ''.join(res)

#--------------------------#
# Юнит тесты на пол шишечки #
#--------------------------#
if __name__ == '__main__':
    print(Trasher().process_source('''
        static auto TestGlobal = "GLOBAL";

        int TestFunc1() {
            auto v1 = 100;
            int v2 = 200;
            short v3 = 300;
            unsigned char v4[] = { 0, 1, 2, 3 };

            if(v1 == v2) {
                v3 = 150;
            }

            while(v1 < 200) {
                v2 = v1 + 1;
                v1++;
            }

            for(int i = 0; i < 4; i++) {
                v4[i] = v3 ^ 0x42;
            }

            return 0;
        }

        int TestFunc2() {
            int v1 = 0;

            do {
                v1++;
            } while(v1 < 10);

            switch(v1) {
                case 0: { v1++; break; }
                case 1: { v1++; break; }
            }

            TestFunc1();
```

```
            return TestFunc1();
        }
    '''))
```

Предельно простая реализация заключается в том, что класс с помощью метода generate_trash генерирует мусорный код, а с помощью метода insert_trash_before вставляет его в те места, какие мы посчитали безопасными для вставки мусорного кода. Обратите внимание, что эта реализация опять же рассчитана на определенное подмножество языка С/С++, и для вашей кодовой базы ее, возможно, придется допилить. Но, вооружившись новыми знаниями из этой статьи, у вас не должно возникнуть с этим проблем. Как минимум их можно обсудить в комментариях под этой статьей, постараемся помочь, если что.

В качестве заключения хочу сказать следующее. Либшланг и, в частности, ее питоновсике байндинги являются не самым удобным средством для обфускации кода. Например, работать с модулем ast для парсинга языка Питон куда приятнее (кстати да, в будущем можем с вами и Питон пообфусцировать, если будет интерес у комьюнити). Однако на практике именно либшланг понимает стандарты языков С/С++ наиболее полно. Да и как мы увидели в этой статье немножечко смекалки может победить большинство проблем. Давайте посмотрим, как работает наш обфускатор, обработав сначала классом Encryptor, а потом классом Trasher простой исходник на языке С (исходный код, дизассемблерный листинг без обфускации, и две страницы дизассемблерного листинга после обфускации приведены ниже).
С++:

```
#include <stdio.h>

int main(int argc, char** argv) {
    puts("Hello World!");
    puts("Goodbye World!");
    return 0;
}
```

```
29: int main (int argc, char **argv, char **envp);
0x00001040      push    rax         ; [13] -r-x section size 405 named .text
0x00001041      lea     rdi, str.Hello_World ; 0x2004 ; const char *s
0x00001048      call    puts        ; sym.imp.puts ; int puts(const char *s)
0x0000104d      lea     rdi, str.Goodbye_World ; 0x2011 ; const char *s
0x00001054      call    puts        ; sym.imp.puts ; int puts(const char *s)
0x00001059      xor     eax, eax
0x0000105b      pop     rdx
0x0000105c      ret
```

XSS.is

```
371: int main (int argc, char **argv, char **envp);
; var int64_t var_2h @ rsp+0x2
; var int64_t var_ah @ rsp+0xa
; var int64_t var_eh @ rsp+0xe
; var int64_t var_1ah @ rsp+0x1a
; var int64_t var_1bh @ rsp+0x1b
; var int64_t var_29h @ rsp+0x29
; var int64_t var_37h @ rsp+0x37
; var int64_t canary @ rsp+0x38
0x00001050      movabs  r9, 0xa3cec108860fbdd ; [13] -r-x section size 757 named .text
0x0000105a      push    rbp
0x0000105b      xor     edx, edx
0x0000105d      sub     rsp, 0x40
0x00001061      mov     rcx, qword [main::__tvar_00000002__] ; 0x4038
0x00001068      mov     rax, qword fs:[0x28]
0x00001071      mov     qword [canary], rax
0x00001076      movabs  rax, 0x1a02cb1a171710f3
0x00001080      mov     dword [var_ah], 0xcc0f171d
0x00001088      lea     r8, [var_eh]
0x0000108d      lea     rdi, [var_2h]
0x00001092      mov     qword [var_2h], rax
0x00001097      movzx   esi, byte [rdx + rdi]
0x0000109b      mov     eax, ecx
0x0000109d      inc     rdx
0x000010a0      and     eax, 0x92ec9cfc
0x000010a5      add     esi, 0x55
0x000010a8      mov     ecx, eax
0x000010aa      mov     byte [rdx + r8 - 1], sil
0x000010af      xor     rcx, r9
0x000010b2      cmp     rdx, 0xc
0x000010b6      jne     0x1097
0x000010b8      imul    eax, eax, 0x2c6b6d61
0x000010be      lea     rdi, [var_1bh]
0x000010c3      mov     ecx, 0xe
0x000010c8      xor     edx, edx
0x000010ca      lea     rsi, [0x00002004]
0x000010d1      mov     byte [var_1ah], 0
0x000010d6      lea     r9, [var_1bh]
0x000010db      movabs  r10, 0x2517dbdd94451a5a
0x000010e5      rep     movsb byte [rdi], byte ptr [rsi]
0x000010e7      and     eax, 0xf0640c28
0x000010ec      xor     eax, 0xc720a958
0x000010f1      add     eax, 0x537cbe32
0x000010f6      mov     dword [main::__tvar_00000001__], eax ; 0x4040
0x000010fc      and     eax, 0xddd362a2
0x00001101      mov     dil, byte [r9 + rdx]
0x00001105      mov     esi, eax
0x00001107      or      eax, 0x39ae3d62
0x0000110c      lea     rbp, [var_29h]
0x00001111      mov     ecx, eax
```

XSS.is

```
0x00001111      mov     ecx, eax
0x00001113      xor     edi, 0xffffffbb ; 4294967227
0x00001116      mov     rax, rcx
0x00001119      mov     byte [rsp + rdx + 0x29], dil
0x0000111e      inc     rdx
0x00001121      imul    rax, r10
0x00001125      cmp     rdx, 0xe
0x00001129      jne     0x1101
0x0000112b      mov     eax, esi
0x0000112d      mov     rdi, r8     ; const char *s
0x00001130      mov     byte [var_37h], 0
0x00001135      or      eax, 0xfbaf3def
0x0000113a      mov     dword [main::__tvar_00000000__], eax ; 0x4044
0x00001140      movabs  rax, 0xe21c23f8ec5a101e
0x0000114a      imul    rax, rcx
0x0000114e      mov     qword [main::__tvar_00000002__], rax ; 0x4038
0x00001155      call    puts        ; sym.imp.puts ; int puts(const char *s)
0x0000115a      mov     rdi, rbp    ; const char *s
0x0000115d      movabs  rax, 0x39e6a512e52c2e55
0x00001167      imul    rax, qword [main::__tvar_00000002__]
0x0000116f      mov     qword [main::__tvar_00000002__], rax ; 0x4038
0x00001176      call    puts        ; sym.imp.puts ; int puts(const char *s)
0x0000117b      mov     eax, dword [main::__tvar_00000001__] ; 0x4040
0x00001181      movabs  rdx, 0x748e84954a777e2d
0x0000118b      add     rax, rdx
0x0000118e      mov     qword [main::__tvar_00000002__], rax ; 0x4038
0x00001195      mov     eax, dword [main::__tvar_00000000__] ; 0x4044
0x0000119b      xor     eax, 0x14bff6d0
0x000011a0      mov     dword [main::__tvar_00000001__], eax ; 0x4040
0x000011a6      mov     rax, qword [canary]
0x000011ab      sub     rax, qword fs:[0x28]
0x000011b4      je      0x11bb
0x000011b6      call    __stack_chk_fail ; sym.imp.__stack_chk_fail ; void __stack_chk_fail(void)
0x000011bb      add     rsp, 0x40
0x000011bf      xor     eax, eax
0x000011c1      pop     rbp
0x000011c2      ret
```

XSS.is

Очень надеюсь, что вам понравилась данная статья. Если комьюнити интересны такие темы, то пишите, подкидывайте идеи для новых статей на следующие конкурсы и на статьи вне конкурса. Вы же знаете я всегда рад потереть за обфускацию и обменяться идеями, это моя, наверное, самая любимая тема в программировании. Всего вам хорошего!