Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Шитов Егор Александрович

Анализ и генерация байт-кода языка Python

Выпускная квалификационная работа бакалавра

Научный руководитель:
Проф. каф. СП, д.ф.-м.н., проф. А. Н. Терехов

Консультант:
Ст. преп. каф. СП М. В. Баклановский

Рецензент:
Ведущий разработчик ПО, «ДиЭсЭкс ТЕХНОЛОДЖИЗ ЛИМИТЕД» Ф. П. Долголев

Санкт-Петербург
2020

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
System Programming

Egor Shitov

Python Bytecode Analysis and Generation

Graduation Thesis

Scientific supervisor:
Doctor of Physics and Mathematics, Professor A. N. Terekhov

Consultant:
Senior Lecturer M. V. Baklanovsky

Reviewer:
Senior Software Engineer, LLC "DSX Technologies Limited" Ph. P. Dolvolev

Saint-Petersburg
2020

Оглавление

Введение

В настоящее время большое количество ПО пишется на языках с промежуточным представлением. Например, Java, C#, Python.

Безусловно, промежуточное представление делает более удобными процесс разработки и отладки. Однако у такого подхода есть и свои недостатки. Например, у программ, написанных на языках с промежуточным представлением, гораздо легче восстанавливается логика работы, благодаря этому промежуточному представлению [18]. Поэтому сейчас остро стоит вопрос обфускации ПО на уровне промежуточного представления [11], [13], [14], [15].

Исследований в области защиты байт-кода Java гораздо больше, чем в областях защиты других языков, в частности языка Python. Однако Python является достаточно популярным выбором при создании ПО различных уровней сложности, в том числе ПО с требованием защиты от восстановления логики.

Поэтому в рамках данной работы будут рассмотрены существующие подходы обфускации байт-кода различных языков в контексте виртуальной машины и промежуточного представления языка Python.

У качественной обфускации можно выделить три главных аспекта.

1. Обфускация должна сильно усложнять работу по восстановлению логики программы [10].

2. Время работы обфусцированной программы не должно значительно возрастать.

3. Должна отсутствовать программа для автоматического снятия обфускации [17].

Каждый из перечисленных аспектов обфускации требует отдельного исследования. Однако работу над обфускатором следует начинать с второго аспекта, так как изначально нужно выделить приемлемые в плане времени работы программы подходы обфускации. Поэтому в рамках данной работы будет рассмотрен второй аспект обфускации.

1. Постановка задачи

Цель работы — исследовать влияние подходов обфускации на время работы программы в контексте байт-кода виртуальной машины Python. Задачи, поставленные для решения в рамках данной работы:

- сделать обзор виртуальной машины языка Python
- разработать инструментарий для оперирования линейными блоками байт-кода Python
- сконструировать подход для корректного измерения времени работы программы
- провести эксперименты о влиянии подходов обфускации на время работы программы

2. Обзор

2.1. Подходы обфускации

Можно выделить различные способы и подходы обфускации программ на уровне байт-кода [15], [20].

Во-первых, существуют простые и необходимые подходы для усложнения понимания кода: удаление отладочной информации, удаление имени или переименование идентификаторов, переиспользование идентификаторов [12], шифрование строковых констант.

Во-вторых, существуют различные методы запутывание потока выполнения. Например, подстановка (inline) функций, изменение порядка выполнения инструкций и функций [11], вставка дополнительного кода.

В-третьих, существуют подходы, основанные на ограниченной выразительности языка и большей выразительности виртуальной машины [17]. Основная идея заключается в том, чтобы создать на уровне виртуальной машины такие конструкции, которые не могут быть выражены в языке.

В заключение, существуют подходы, основанные на запутывании потока данных. Например, изменение области видимости идентификатора, агрегация данных (объединение двух массивов в один, либо изменение размерности массива), изменение порядка данных в массиве [15].

В рамках данной работы были рассмотрены методы, которые потенциально могут изменить время работы программы: подстановка функций, изменение порядка выполнения инструкций, вставка дополнительного кода. А также, был рассмотрен подход с изменением области видимости идентификаторов.

2.2. Виртуальная машина Python

Сейчас почти повсеместно принято говорить «байт-код языка», особенно, это заметно в сообществе разработчиков на языке Python. Конечно, люди в разговоре понимают, о чем идет речь, но формально

так говорить неправильно. Байт-код — некоторый набор инструкций для некоторой машины. В большинстве случаев имеется в виду виртуальная машина. Язык — множество слов над конечным алфавитом. Исходя из этих определений, словосочетание «байт-код языка» не имеет смысла. Корректно будет говорить «байт-код виртуальной машины Python». Однако, как уже было замечено, словосочетание «виртуальная машина» обычно опускают, но имеют в виду. Поэтому далее тоже будет употребляться конструкция «байт-код языка», подразумевая под собой «байт-код виртуальной машины».

Следует отметить, что существуют различные реализации виртуальных машин для языка Python. Некоторые взаимозаменяемы на уровне байт-кода с классической реализацией интерпретатора, например micropython [6], некоторые — нет, например Falcon [19]. В рамках данной работы будет рассмотрена стандартная реализация виртуальной машины — CPython [2], так как она является наиболее популярной реализацией.

CPython — стековая виртуальная машина для языка Python. До версии 3.6 инструкции были различной длины — 3 байта для инструкции с аргументами и 1 байт для инструкции без аргументов. Начиная с версии 3.6 все инструкции имеют длину 3 байта. Инструкции без аргументов игнорируют переданные им аргументы. В рамках данной работы использовалась версия Python 3.5.

Инструкции объединяются в объекты кода — CodeObject. В большинстве случаев, один объект кода — одна функция на языке программирования Python. Исключения могут быть при использовании различных языковых конструкций. Например, для конструкции $x = [i + 1\, for\, i\, in\, range(10)]$ выражение в квадратных скобках будет вынесено в отдельный объект кода. Объект кода содержит дополнительную информацию о функции, такую как количество аргументов, количество и имена локальных и глобальных переменных, константы, служебные флаги, отладочную информацию о строках кода, имя файла и имя функции, глубину стека. Объекты кода устроены иерархично — один объект кода может быть вложен в другой объект кода. На прак-

тике это реализуется следующим образом: один объект кода является константой для другого объекта кода.

Следует отметить еще некоторые особенности Python:

- в Python на уровне виртуальной машины и байт-кода поддерживаются различные языковые конструкции, например циклы, итераторы, генераторы, исключения, аннотации
- в стандартной реализации виртуальной машины Python (CPython) нет каких-либо значимых для данной работы оптимизаций байт-кода [3]. Неформально говоря, транслятор почти «дословно» переводит программу, написанную на языке Python, в байт-код виртуальной машины

2.3. Измерительные инструменты

Подходы к измерению программ можно разделить условно на микробенчмаркинг — замер времени работы небольших участков кода и макробенчмаркинг — замер времени работы достаточно больших программ, условно с временем работы более одной секунды. В рамках данной работы нужно использовать подход макробенчмаркинга, так как измеряться будет время работы всей программы, а не отдельные участки кода.

Для измерения времени работы программы на Python существует несколько инструментов. У измерительного инструмента можно выделить такие характеристики как:

- задержка (Latency) — количество времени, нужное инструменту для взятия текущего значения времени,
- точность (Resolution) — минимальное возможное положительное значение разности между двумя последовательными замерами времени.

Все рассмотренные измерительные инструменты фиксируют текущее время с помощью специальных функций в операционной системе.

Следовательно, значения задержки и точности сравнимы с соответствующими значениями соответствующей функции в ОС. В условиях современных операционных систем и современного аппаратного обеспечения данные значения измеряются, в худшем случае, в десятках миллисекунд. Планируется, что программа будет работать несколько секунд. Поэтому все рассмотренные ниже инструменты с учетом их накладных расходов по этим характеристикам удовлетворяют условиям данной работы.

Далее будут рассмотрены существующие способы измерения времени работы программы на Python.

Стандартный модуль Python timeit [9]. Предназначен для микробенчмаркинга. Модуль, по-умолчанию, запускает небольшой участок кода 3 раза с большим количеством итераций и показывает наилучший результат. Как и другие модули Python, может использоваться как в режиме командной строки, так и импортирован в программу на языке Python. Так как данный модуль предназначен для микробенчмаркинга, он не подходит для измерений в рамках данной работы.

Функция time из стандартной библиотеки time. Возвращает количество секунд с начала эпохи. Ввиду своей примитивности является гибким инструментом, так как позволяет параллельно проводить второстепенные измерения, например, частоты процессора. Данный подход может быть использован для измерения времени в рамках данной работы.

Модули профилирования cProfile и profile [1]. Они имеют одинаковый интерфейс для взаимодействия. Различие в том, что cProfile гораздо быстрее profile. Данные модули показывают статистику по каждому объекту кода (CodeObject): количество вызовов, количество рекурсивных вызовов, среднее и накопленное время работы каждого объекта кода. Имеет большую функциональность, чем функция time, однако уступает ей в гибкости. Данный инструментарий предназначен для макробенчмаркинга и, следовательно, может быть использован в рамках данной работы.

Сторонняя библиотека line_profiler [4]. Замеряет время работы каж-

дой строки кода. Требует выставления дополнительных аннотаций в код для тех функций, которые нужно профилировать. Данная библиотека имеет полезную функциональность, однако не может быть использована в рамках данной работы, так как после изменения байт-кода программы метаинформация о соответствии строчек кода и байт-кода становится некорректной. В сгенерированном коде такой метаинформации вообще нет.

Сторонняя библиотека profilehooks [8]. По своей функциональности сильно похожа на cProfile и profile. Требует аннотаций для профилируемых функций. С точки зрения использования имеет те же преимущества и недостатки, что cProfile и profile.

3. Оперирование линейными блоками

Для дальнейшего проведения экспериментов в рамках данной работы был реализован инструментарий для оперирования линейными блоками.

3.1. Задачи инструментария

Были поставлены следующие задачи для данного инструментария.

Во-первых, фиксирование размера линейного блока. Линейные блоки, которые больше заданного размера, делятся на меньшие блоки. Блоки, которые меньше заданного размера, наоборот, дополняются тождественной инструкцией (NOP). Такой прием может быть полезен для обфускации с различных точек зрения. Чем меньше размер минимального объекта, который можно переставлять без нарушения семантики программы, тем сложнее можно запутать поток выполнения. С другой стороны, дополнение маленьких блоков можно сделать не только тождественными инструкциями, а любым произвольным кодом.

Во-вторых, смещение всех линейных блоков относительно начала объекта кода. В рамках данной работы сдвиг осуществляется путем добавления NOP инструкций. В дальнейшем, с целью обфускации, сдвиг можно осуществлять произвольными инструкциями, но с тем учетом, чтобы состояние виртуальной машины не изменилось к началу программы. Такой прием позволит добавлять произвольную логику в начало CodeObject, например с целью инициализации каких-либо runtime процессов, либо глобальных переменных.

В-третьих, изменение порядка линейных блоков. Качественный обфускатор неизбежно сталкивается с задачей запутывания потока выполнения. Изменение порядка линейных блоков — часть этой задачи.

В заключение, вставка произвольного кода в различные места программы. Эта функциональность позволяет вставлять код как для задач обфускации, так и для отладочных задач, и задач с целью профилирования. Например, можно узнать, сколько раз исполнился тот или иной линейный блок, сколько раз исполнился объект кода, узнать частоту пе-

реходов между различными блоками. Потенциально есть возможность создать инструментарий для микробенчмаркинга с целью замера времени выполнения каждого линейного блока.

Также, следует отметить, что, кроме инструментария для оперирования линейными блоками, необходим инструментарий для проведения замеров времени работы программы и инструментарий для удобного представления результатов.

3.2. Архитектура

Для разработки инструментария был выбран язык программирования Python, так как в стандартной библиотеке этого языка есть необходимые модули для работы с программами этого языка: для компилирования, декомпилирования, для работы с байт-кодом. К тому же, этот язык хорошо подходит для быстрого прототипирования программ и для программ, разрабатываемых в исследовательских целях.

Для ускорения разработки инструментария изначально было принято решение применить подход быстрого прототипирования, который удобен для разработки программ в исследовательских целях. К тому же, форма конечной программы на начальных этапах была неизвестна, что также послужило аргументом в пользу выбора быстрого прототипирования.

На первом этапе архитектура инструментария отдельно не прорабатывалась. Основной задачей была задача изучить необходимые библиотеки Python, первоначально выделить линейные блоки в байт-коде, сгенерировать из получившихся блоков работающую программу.

Позже стало понятно, что при добавлении новой функциональности сложность кода многократно увеличивалась. Такое усложнение кода являлось как результатом отсутствия архитектуры, так и как спецификой динамически типизированного языка.

Поэтому было принято решение разбить весь инструментарий на отдельные, как можно меньше зависящие друг от друга, скрипты. Глобально, весь инструментарий можно разделить на три независимых мо-

дуля: оперирование линейными блоками, замер времени работы программы, представление результатов.

Модуль представления результатов является достаточно простым. Он состоит из двух скриптов и нескольких функций: для построения графиков и гистограмм, для анализа результатов, для визуализации графов. Модуль использует, де-факто стандартную, библиотеку для графиков Matplotlib [5] и Networkx [7] для визуализации графов.

Перед модулем измерения времени работы программы стоят две основные задачи. Во-первых, модуль должен провести серию измерений в рамках одного эксперимента. Во-вторых, модуль должен отслеживать частоту работы процессора во время проведения измерений, так как даже при явно зафиксированной частоте может произойти какой-то сбой.

Так как отслеживать частоту процессора во время измерений, ввиду своей гибкости, позволяет только функция time из библиотеки time, то именно она и была выбрана для реализации данного модуля.

Для измерения времени работы программы модуль запускает указанное количество раз скомпилированную программу на Python, каждый раз фиксируя время работы программы. Частота процессора измеряется в отдельном потоке с квантом времени в 25мс, затем берется среднее значение для каждого запуска программы. Таким образом, для каждого запуска программы модуль предоставляет время работы программы и среднее значение частоты процессора во время работы программы. Полученную статистику модуль экспортирует в указанный файл в формате json.

Модуль оперирования линейными блоками состоит из нескольких по-смыслу разделенных скриптов:

- извлечение линейных блоков из инструкций
- генерирование инструкций из линейных блоков
- внутреннее представление линейного блока
- добавление дополнительного байт-кода в программу
- координация логики

Extractor отвечает за извлечение линейных блоков из инструкций. Скрипт работает в два прохода. В первом проходе происходит преобразование байт-кода в набор инструкций с аргументами. Во втором проходе происходит разделение инструкций на линейные блоки с учетом причины окончания блока (например, условный или безусловный переход), с учетом вложенности циклов и с учетом принудительного дробления блоков (если был передан параметр фиксирования размера блока).

Следует отметить, что на уровне виртуальной машины циклы поддерживаются инструкцией, аргумент которой является размером цикла в байтах, то есть относительным смещением до конца цикла. Ввиду этой особенности, наличие циклов в программе порождает вложенность линейных блоков. Данная особенность усложняет работу по извлечению линейных блоков и генерированию байт-кода.

Generator отвечает за генерирование байт-кода из имеющихся линейных блоков. Скрипт рекурсивно итерируется по вложенным линейным блокам, пересчитывает абсолютные и относительные переходы, вставляет необходимый код как самостоятельно (например, в целях увеличения размера линейного блока), так и по указаниям inject_info.

Inject_info отвечает за предварительную генерацию байт-кода для вставки произвольного дополнительного кода в программу. На данном этапе поддерживается вставка кода в начало CodeObject, в конец CodeObject, в конец каждого линейного блока, вставка кода в программу в виде отдельной функции. С помощью стандартной библиотеки модуль компилирует программу, написанную в отдельном файле на языке Python, в байт-код. Затем пересчитывает необходимые значения для абсолютных и относительных переходов, объединяет и дополняет список констант и переменных, заменяет необходимые инструкции обращения к константам и переменным.

Generator агрегирует в себе inject_info, и при генерации байт-кода, при наличии данных о вставке дополнительного кода Generator запрашивает готовый байт-код у inject_info и вставляет его в указанное место. Ответственность за пересчет параметров перехода лежит на Generator.

Shuffle ответственен за координацию логики и перестановку линейных блоков по заданному правилу. Сначала производится извлечение промежуточного представления из скомпилированной Python программы. Затем, рекурсивно запускаются операции с линейными блоками для вложенных объектов кода. В каждой итерации производится разбиение байт-кода на линейные блоки с помощью Extractor, перестановка линейных блоков по заданному правилу, указание необходимой информации для вставки дополнительного кода с помощью Inject_info, генерация байт-кода с помощью Generator. Получившийся байт-код, новые переменные и константы упаковываются в промежуточное представление и записываются в файл.

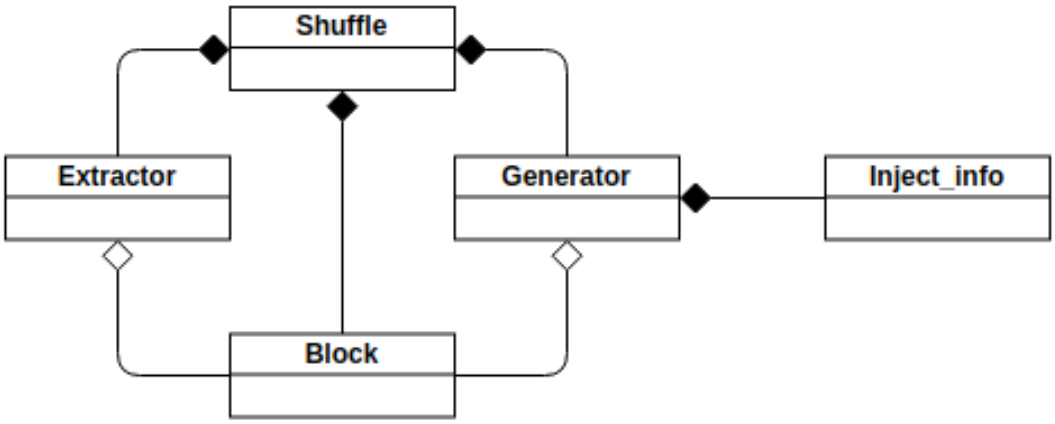

Рис. 1: Диаграмма классов UML

Таким образом, получившаяся архитектура довольно сильно похожа на pipes and filters. К тому же, ввиду наличия промежуточного представления для блоков и инструкций, инструментарий получился довольно гибкий. Например, можно, не изменяя основную логику, внести поддержку равнодлинных инструкций для другой версии Python.

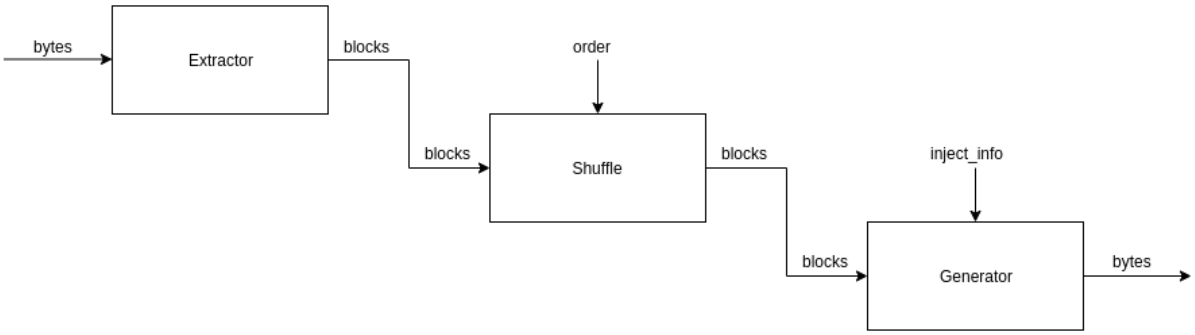

Рис. 2: Диаграмма IDEF0

4. Измерение времени работы программы

4.1. Окружение проведения измерений

Все измерения проводились на ноутбуке Lenovo Ideapad 720s 14. Технические характеристики:

- процессор — Intel(R) Core(TM) i7-8550U
- ОС — Linux: Elementary OS 0.4.1 Loki, версия ядра 4.15.0-88-generic. Windows 10
- RAM — 16 Гб

Использовался стандартный интерпретатор CPython версии 3.5.2 в рамках ОС Linux и CPython версии 3.5.9 в рамках ОС Windows.

Так же следует отметить, что все посторонние процессы, в меру возможностей, были отключены, на Windows отключены процессы обновления. В ходе проведения измерений отсутствовало какое-либо сетевое подключение. Запуск программ производился в сборке release.

В современных операционных системах и в современном аппаратном обеспечении поддерживается функция динамического регулирования частоты процессора. Такой подход позволяет гибко управлять производительностью процессора и расходом батареи ноутбука. Однако он абсолютно неприемлем при измерении времени работы программы, так как чем больше частота процессора, тем быстрее работает программа. Сравнение результатов измерений с динамической частотой не отражают реальных результатов. Следовательно, для корректности измерений была экспериментально найдена частота, которую ноутбук может стабильно поддерживать на протяжении долгого времени в рамках обеих операционных систем (Linux, Windows) — 1500 МГц. Таким образом, все замеры проводились с фиксированной частотой в 1500 МГц.

Кроме динамического регулирования частоты процессора, в современных системах есть несколько уровней различных кешей. Итоговое время работы программы может значительно меняться в зависимости

от того, есть ли нужные данные в каком-то кеше или нет. Такое поведение дает результат, который некорректно сравнивать и анализировать в рамках данной работы. Поэтому перед началом серии экспериментов проводятся вспомогательные замеры — прогрев кешей.

Также, в угоду чистоты эксперимента было исключено время запуска процесса в ОС, время запуска виртуальной машины Python, время IO операций.

4.2. Программы для тестирования

Для программ, на которых будут проводиться измерения, были выдвинуты следующие требования.

Во-первых, программа должна быть написана полностью на Python, без использования библиотек, которые большую часть времени выполняют код на языках C/C++, так как при использовании последних перестановка линейных блоков не будет значительно влиять на скорость работы программы. Аналогично, должно использоваться минимальное количество библиотек, написанных на Python, так как в рамках данной работы не стоит задача переставлять линейные блоки в библиотеках.

Во-вторых, программы должны быть разного размера: как с небольшим количеством инструкций, линейных блоков и переменных, так и с большим.

Таким образом, в качестве программ для тестирования были выбраны сортировка пузырьком, быстрая сортировка и модуль Extractor, который ответственен за извлечение линейных блоков из байт-кода.

Размер входных данных был подобран так, чтобы программы работали несколько секунд.

5. Результаты

Перед представлением результатов стоит отметить, что каждое численное значение было получено путем проведения серии измерений из 50 запусков и взятия среднего значения.

Во всех экспериментах среднее отклонение составило 0.1-0.2 секунды для программ сортировок и 0.3-0.4 секунды для модуля Extractor. Таким образом, будем считать, что времена работы программ сортировок существенно отличаются, если разность их средних значений по абсолютной величине больше 0.04 секунды, а для модуля Extractor — больше 0.08 секунд.

5.1. Тождественное преобразование

С целью выяснения, как разработанный инструментарий влияет на время работы программы, был проведен следующий эксперимент. Рассмотрены две серии измерений. Первая — время работы программы до каких-либо преобразований. Вторая — время работы программы после тождественного преобразования порядка линейных блоков. То есть, данный эксперимент покажет, какие накладные расходы появляются в связи с неизбежными изменениями байт-кода инструментарием.

Результаты представлены таблицами 1 и 2.

ОС	пуз.	быстр.	extr.
Linux	4.21	3.6	3.74
Windows	5.67	4.95	5.55

Таблица 1: Время работы программы до преобразования, сек

ОС	пуз.	быстр.	extr.
Linux	4.27	3.69	3.84
Windows	5.98	5.21	5.98

Таблица 2: Время работы программы после преобразования, сек

Из результатов видно, что в процессе преобразования программ существуют накладные расходы. Однако такое замедление является незначительным и, с практической точки зрения, его можно считать приемлемым в контексте обфускации.

В дальнейшем, все сравнения будут происходить с временем работы программы после тождественного преобразования.

5.2. Фиксирование размера линейного блока

Далее, был проведен эксперимент с фиксированием размера линейного блока. Фиксирование осуществлялось от 16 байт до 32 байт.

Результаты представлены таблицей 3.

Размер	пуз.	быстр.	extr.
16	4.36 (6.18)	3.77 (5.42)	3.92 (6.22)
17	4.35 (6.18)	3.76 (5.56)	3.90 (6.07)
18	4.35 (6.17)	3.77 (5.54)	3.96 (6.06)
19	4.34 (6.23)	3.77 (5.52)	3.96 (6.11)
20	4.34 (6.18)	3.73 (5.32)	3.91 (6.00)
21	4.35 (6.30)	3.73 (5.34)	3.90 (6.06)
22	4.32 (6.11)	3.73 (5.34)	3.91 (6.11)
23	4.32 (6.11)	3.72 (5.28)	3.90 (6.08)
24	4.32 (6.11)	3.71 (5.29)	3.90 (6.02)
25	4.33 (6.23)	3.71 (5.37)	3.89 (5.99)
26	4.34 (6.23)	3.71 (5.28)	3.90 (6.02)
27	4.33 (6.23)	3.71 (5.27)	3.89 (6.05)
28	4.29 (6.01)	3.72 (5.27)	3.89 (5.95)
29	4.28 (6.01)	3.72 (5.27)	3.87 (5.92)
30	4.29 (6.01)	3.72 (5.27)	3.88 (5.97)
31	4.29 (6.01)	3.71 (5.28)	3.88 (5.96)

Таблица 3: Фиксирование размера блока, Linux (Windows), сек

Видно, что прослеживается определенная тенденция. Чем меньше размер линейного блока, тем дольше работает программа. Однако, с практической точки зрения, замедления являются несущественными, но увеличение количества линейных блоков в два раза существенно увеличит количество возможных перестановок линейных блоков, что,

теоретически, позволит сильнее запутать поток выполнения. Следовательно, можно сделать вывод, что уменьшать размер линейного блока с целью обфускации целесообразно.

5.3. Изменение порядка линейных блоков

Далее, был проведен эксперимент с перестановкой линейных блоков. Так как произвольных перестановок для n блоков n!, было принято решение выделить и проверить гипотезу, которая теоретически могла бы уменьшить время работы программы. Основная идея — уменьшение длины переходов между блоками. Таким образом, в данном эксперименте будет исследовано две различные группировки блоков. Идеи для обеих группировок были позаимствованы из работы [16]. В ней проводились аналогичные группировки для case-блоков в switch-case операторе интерпретатора. Для каждой группировки была заранее собрана необходимая информация с помощью инструментария вставки дополнительного кода.

Первая группировка — сортировка блоков по уменьшению количества вхождений потока управления в блок. С применением данной группировки все часто используемые блоки находятся в одном месте. Следовательно, общая длина переходов между блоками сократится.

Для второй группировки рассмотрим взвешенный граф, вершинами которого являются блоки, ребрами — переходы потока управления между блоками, весами — количество переходов потока управления. Далее, необходимо построить путь, который выходит из вершины с ребром наибольшего веса, и каждый переход производится так же по ребру наибольшего веса. Таким образом, блоки, между которыми происходят частые переходы, будут расположены рядом. Следовательно, общая длина переходов между блоками сократится.

Результаты представлены таблицей 4.

Прог., группировка	Linux	Windows
Пузырек, станд.	4.27	5.98
Пузырек, груп. 1	4.27	6.09
Пузырек, груп. 2	4.28	5.97
Быстрая, станд.	3.69	5.21
Быстрая, груп. 1	3.69	5.26
Быстрая, груп. 2	3.69	5.26
Extr., станд.	3.84	5.98
Extr., груп. 1	3.85	5.95
Extr., груп. 2	3.84	5.95

Таблица 4: Перестановка линейных блоков, сек

Из результатов видно, что группировка с целью уменьшения общей длины переходов не влияет на скорость работы программы в рамках ОС Linux. В рамках ОС Windows наблюдается незначительное замедление у быстрой сортировки.

5.4. Вынесение идентификаторов и подстановка функций

В заключение, был проведен эксперимент с вынесением локальных переменных в глобальную область видимости и эксперимент с подстановкой функций (объектов кода). В качестве программ рассматривались быстрая сортировка и модуль Extractor. Сортировка пузырьком не рассматривалась, так как в ней невозможно применить ни вынесение локальных переменных, ни подстановку функций.

В модуле Extractor рассматривалось только вынесение локальных переменных в глобальную область видимости. Подстановка функций для модуля Extractor не имеет смысла.

В быстрой сортировке логика разделения частей изначально была вынесена в отдельную функцию. Следовательно, каждый рекурсивный вызов быстрой сортировки вызывает функцию разделения частей. Таким образом, для быстрой сортировки рассматривался как подход с

вынесением локальных переменных, так и подход с подстановкой функций.

Нумерация подходов следующая: 1 — без изменений, 2 — с вынесением локальных переменных, 3 — с вынесением локальных переменных и с подстановкой функций, 4 — с подстановкой функций.

Результаты представлены таблицами 5 и 6.

Подход	Linux	Windows
1	3.69	5.21
2	4.35	6.00
3	4.30	5.92
4	3.65	5.17

Таблица 5: Вынесение идентификаторов и подстановка функций, быстрая сортировка, сек

Подход	Linux	Windows
1	3.84	5.98
2	6.34	8.73

Таблица 6: Вынесение идентификаторов, модуль Extractor, сек

Из результатов видно, что вынесение локальных переменных негативно влияет на скорость работы программы. Причем, чем больше программа и чем больше переменных, тем сильнее выражено замедление. С практической точки зрения, данный подход нужно осторожно использовать в рамках обфускации, обращая внимание на замедление программы в каждом конкретном случае.

Подход с подстановкой функций показал несущественное уменьшение времени работы программы.

5.5. Выводы

Для того, чтобы понять получившиеся результаты, программы были запущены с использованием утилиты perf stat, которая показывает

некоторые счетчики процессора. В данном случае были использованы такие счетчики как количество исполненных инструкций, количество инструкций ветвления и промахи предсказателя переходов, количество обращений в кеш и промахи кеша.

В ходе сравнения этих показателей стало видно, что тождественное преобразование вызывает замедление ввиду добавления дополнительного кода в логику программы, что приводит к увеличению числа исполненных процессорных инструкций (в пределах 3% для всех рассмотренных программ). Показатели кешей значительно не изменяются.

Уменьшение размера линейного блока имеет такой же результат: число исполненных процессорных инструкций увеличивается (в пределах 2.5% для всех рассмотренных программ), показатели кешей значительно не изменяются.

Перестановки линейных блоков не влияют значимым образом ни на количество исполненных инструкций, ни на показатели кешей.

Вынесение локальных переменных в глобальную область видимости значительно увеличивает количество исполненных процессорных инструкций (17% для быстрой сортировки и 66% для модуля Extractor) и количество обращений в кеш (на 20% для быстрой сортировки и в 9 раз для модуля Extractor). Также, в случае модуля Extractor, значительно увеличивается количество промахов L3 кеша (с 2% до 27%).

Подстановка функций приводит к незначительному уменьшению всех параметров (менее 1% для быстрой сортировки).

Таким образом, можно выделить подход для обфускации, который незначительно замедлит программу, но значительно увеличит сложность работы по восстановлению логики. Для этого нужно максимально подставить все возможные функции, чтобы внутри одного объекта кода было как можно больше инструкций. Затем, нужно зафиксировать достаточно маленький размер линейного блока, например, 16 байт. В каждом объекте кода получится большое количество мелких линейных блоков. Следовательно, будет большое количество возможных перестановок таких блоков. Следовательно, есть возможность запутать поток выполнения максимально сильно.

Ввиду того что подстановка функций и перестановка линейных блоков не приводят к значительным изменениям во времени работы программы, можно предположить, что такой подход обфускации тоже не приведет к значительному замедлению.

6. Заключение

В ходе данной работы исследовано влияние подходов обфускации на время работы программы в контексте байт-кода виртуальной машины Python, в частности:

- сделан обзор виртуальной машины языка Python,
- разработан инструментарий для оперирования линейными блоками байт-кода Python,
- разработан подход для измерения времени работы программы,
- проведены эксперименты о влиянии подходов обфускации на время работы программы.

Список литературы

[1] cprofile & profile. https://docs.python.org/3/library/profile.html. Online; accessed 20.05.2020.

[2] Cpython. https://github.com/python/cpython. Online; accessed 20.05.2020.

[3] The cpython bytecode compiler is dumb. https://nullprogram.com/blog/2019/02/24/. Online; accessed 20.05.2020.

[4] Line profiler. https://github.com/rkern/line_profiler. Online; accessed 20.05.2020.

[5] Matplotlib. https://matplotlib.org/. Online; accessed 20.05.2020.

[6] Micropython. https://micropython.org/. Online; accessed 20.05.2020.

[7] Networkx. https://networkx.github.io/. Online; accessed 20.05.2020.

[8] Profilehooks. https://github.com/mgedmin/profilehooks. Online; accessed 20.05.2020.

[9] timeit. https://docs.python.org/3/library/timeit.html. Online; accessed 20.05.2020.

[10] Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. Program obfuscation: A quantitative approach. In *Proceedings of the 2007 ACM Workshop on Quality of Protection*, QoP '07, page 15–20, New York, NY, USA, 2007. Association for Computing Machinery.

[11] V. Balachandran, N. W. Keong, and S. Emmanuel. Function level control flow obfuscation for software security. In *2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems*, pages 133–140, 2014.

[12] Jien-Tsai Chan and Wuu Yang. Advanced obfuscation techniques for java bytecode. *Journal of Systems and Software*, 71(1):1 – 10, 2004.

[13] Zhao B. et al. Dexpro: A bytecode level code protection system for android applications. *Lecture Notes in Computer Science*, 10581, 2017.

[14] Hunt J. Byte code protection. *Java for Practitioners*, 1999.

[15] Douglas Low. Protecting java code via code obfuscation. *XRDS*, 4(3):21–23, April 1998.

[16] Jason McCandless and David Gregg. Optimizing interpreters by tuning opcode orderings on virtual machines for modern architectures: Or: How i learned to stop worrying and love hill climbing. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, page 161–170, New York, NY, USA, 2011. Association for Computing Machinery.

[17] J. M. Memon, Shams-ul-arfeen, A. Mughal, and F. Memon. Preventing reverse engineering threat in java using byte code obfuscation techniques. In *2006 International Conference on Emerging Technologies*, pages 689–694, 2006.

[18] Aaron Portnoy and Ali Rizvi-Santiago. Reverse engineering python applications.

[19] Russell Power and Alex Rubinsteyn. How fast can we make interpreted python? *CoRR*, abs/1306.6047, 2013.

[20] Dănuţ Rusu. Protection methods of java bytecode.