Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 22.Б07-мм

Защита программного кода путём встраивания мёртвого кода и данных

Дугушова Ксения Андреевна

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ст. преподаватель кафедры ИАС К. К. Смирнов

Консультант:
Старший разработчик ПО I категории ООО «Софтком» П. А. Бабанов

Санкт-Петербург
2024

Оглавление

Введение

В современном обществе, где программное обеспечение стало неотъемлемой частью нашей повседневной жизни, обеспечение безопасности программного кода является важной задачей. Защита от внешних угроз, таких как вредоносные атаки и несанкционированный доступ, является предметом постоянного исследования и разработки.

Одним из широко используемых методов защиты ПО является обфускация (англ. obfuscate — делать неочевидным), представляющая собой различные инструменты, которые способны изменить исходный код с целью предотвращения его анализа, при этом сохраняя функциональность.

В настоящее время существует большое количество обфускаторов с открытым кодом, но они не могут противостоять качественному реверс-инжинирингу [8]. Некоторые из закрытых обфускаторов побеждают обратное проектирование, но разбор их принципа работы представляет собой очень сложную, местами невозможную задачу.

ООО «Софтком» разрабатывает обфускатор, написанный на языке Python, который должен соответствовать следующим требованиям: обработка ассемблерного кода, возможность работы с разными компиляторами, наличие расширяемого интерфейса (возможность расширять список поддерживаемых платформ). Исходя из этих условий, вытекает необходимость в разработке обфускатора. Закрытые проекты невозможно расширять самостоятельно, также в большинстве случаев они не привязаны к каким-либо конкретным компиляторам, существующие обфускаторы с открытым кодом не включают в себя все требуемые условия.

На момент начала работы прототип обфускатора ООО «Софтком» обрабатывает программы, написанные на языке C, может работать с компиляторами clang, GCC, bt23h (адаптированный GCC). Реализованы отдельные наборы скриптов под архитектуры, а также адаптированные команды для компиляторов и их версий. Следующий шаг в развитии проекта — добавление возможности обрабатывать програм-

мы, написанные на языке C++, разработка новых методов обфускации таким образом, чтобы они могли стать более общими в контексте используемых компиляторов и архитектур.

Прототип обфускатора «Софтком» состоит из ядра, в нем реализована платформонезависимая часть обфускации, и из различных плагинов, каждый плагин содержит представление, реализующее платформозависимые методы обфускации. Разрабатываемый обфускатор является проектом с закрытым кодом.

Одним из методов, противостоящих реверс-инжинирингу является добавление мёртвого кода и данных таким образом, чтобы нельзя было распознать, что эти части кода не влияют на результат выполнения программы. Встраивание мертвого кода является базовым методом обфускации, во многих проектах реализованы некоторые инструменты по добавлению мёртвого кода, но эти инструменты на основе исследования [1] легко распознаются декомпиляторами.

В обфускаторе компании «Софтком» данный метод отсутствует. В рамках работы предлагается реализовать методы генерации и вставки мертвого кода и внедрить их в исходный прототип обфускатора.

Таким образом, в рамках работы необходимо создать генерацию мёртвого кода и данных, реализовать их встраивание в исходный код, тем самым оказать поддержку в развитии проекта.

1. Постановка задачи

Целью работы является написание алгоритма генерации мертвого кода и данных, их встраивание в исходный код для дальнейшей поддержки развития проекта. Для её выполнения были поставлены следующие задачи:

1. реализовать генерацию и вставку мертвого кода в исходный прототип обфускатора;
2. оценить качество работы обфускатора с помощью известных декомпиляторов;
3. измерить замедление выполнения программы после обфускации.

2. Обзор

В данной главе будет представлен обзор наиболее популярных существующих дизассемблеров и обфускаторов. В том числе будут рассмотрены приемы, используемые последними в качестве методов «запутывания», включая внедрение мёртвого кода и данных.

2.1. Обфускаторы

В данном разделе рассмотрены наиболее качественные и популярные обфускаторы для программ, написанных на языках C/C++. Обфускаторы выбраны на основе исследования [7], в котором провели сравнительный анализ и выделили лучшие обфускаторы. Выборка для анализа в данном исследовании была произведена на основе различных работ и статей.

2.1.1. Obfuscator-LLVM

Obfuscator-LLVM [6] основан на фреймворке LLVM [5] (Low Level Virtual Machine). Обфускация происходит на промежуточном уровне представления, поэтому OLLVM совместим со всеми языками программирования и целевыми платформами. Предусмотрена работа с компилятором clang.

OLLVM — проект с открытым исходным кодом, следовательно можно выделить следующие используемые методы обфускации:

- переименование переменных и функций;
- встраивание мёртвого кода и данных;
- изменение потока управления;
- трансформация данных.

Из описания проекта, а также из анализа кода были выделены методы работы OLLVM с мёртвым кодом:

- добавление блока кода в начале основной части программы с дальнейшей реализацией условного перехода к нужным инструкциям;
- основная часть программы случайным образом заполняется ненужными инструкциями. (Реализована генерация ложных условий, бесполезных циклов, глобальных и локальных переменных);
- замена стандартных бинарных операторов (сложение, вычитание и логические операторы) более сложными инструкциями/последовательностями инструкций, результат которых функционально эквивалентен. Реализован случайный выбор замены. Данный метод доступен только для целочисленных операций.

Таким образом, можно заметить, что в OLLVM не реализована некоторая часть методов обфускации, связанных с мёртвым кодом.

2.1.2. Obfuscator Themida

Themida [9] является коммерческим проектом, целью которого является защита кода от нелегального использования, копирования и других видов вмешательств.

Themida действует на уровне машинного кода. В основе данного обфускатора лежат базовые методы обфускации, которые были описаны в обзоре OLLVM. Тем не менее, как заявляют разработчики, в последней версии Themida (Версия 3.1, дата публикации 2 октября 2023г.), реализовано революционное решение (технология защиты SecureEngine) для борьбы с недостатками средств защиты ПО.

Themida является закрытым проектом, поэтому существует ряд работ, в которых проведён анализ Themida. В проекте под названием UnThemida [11] выявлены основные особенности обфускации с помощью Themida, рассмотрим те из них, которые относятся к теме нашей работы:

1. Изменение количества и названий секций.

   Независимо от количества секций в исходном коде, в обфусцированной программе их количество равняется пяти. Название пер-

вого раздела удаляется, а имена двух последних изменяются случайным образом.

2. Искажение адресов функций.

3. Изменение названий переменных и функций.

Как можно заметить из выделенных пунктов выше — о работе Themida известно немного. Также невозможно изменить данный продукт, чтобы он соответствовал заявленным требованиям (так как проект закрытый).

2.1.3. Obfuscator Tigress

Tigress [12] — обфускатор для языка C, сконцентрирован на работе с исходным кодом и с промежуточным представлением, поддерживает различные архитектуры процессора, совместим с компиляторами clang и GCC. Предусмотрена настройка параметров обфускации в соответствии с требованиями пользователя.

Проект является закрытым, разработчики предложили выявлять методы обфускации с помощью многократного применения обфускатора, такой анализ показал, что многие методы похожи на инструменты, используемыми в O-LLVM, также был обнаружен ещё один:

- Дополнительные операции по преобразованию типов, которые не влияют на результат программы, но создают дополнительные шаги при исполнении.

В рассматриваемом обфускаторе мёртвый код может быть встроен в различные места кода:

- Циклы;
- Последовательность инструкций, среди которых точка входа — первая инструкция, выхода — последняя);
- Условные операторы;

- Блоки кода, в которых вставка новых данных не повлияет на результат программы.

2.2. Дизассемблеры

В данном разделе рассмотрены популярные дизассемблеры, которые на основе исследования [10] о сравнении инструментов обратного проектирования программного кода и книги [2], описывающей различные методы обфускации и обратного проектирования, являются ведущими в своей сфере.

Обзор дизассемблеров необходим для дальнейшего тестирования реализованного метода обфускации (тестирование следует проводить с использованием качественного дизассемблере).

2.2.1. IDA

IDA [4] (Interactive Disassembler) — коммерческий продукт, разрабатываемый компанией Hex-Rays. Это инструмент для реверс-инжиниринга, при необходимости разобраться в структуре и логике программы.

Основная функциональность IDA:

- Дизассемблирование — преобразование машинного кода в ассемблерный, что значительно облегчает понимание структуры программы;
- Возможность использования в качестве фронтенда для GDB отладчика (предоставляет удобный графический интерфейс для визуализации, управления выполнением программы и анализа состояния во время отладки);
- Анализ потока управления;
- Декомпиляция — восстановление высокоуровневого исходного кода на основе ассемблерного;
- Анализ данных;

- Создание плагинов и использование скриптов.

Также IDA поддерживает широкий спектр архитектур процессоров и языков программирования.

2.2.2. Ghidra

Ghidra [3] — инструмент для обратного проектирования с открытым кодом, был разработан Агентством национальной безопасности США (NSA), впоследствии предоставлен в открытый доступ.

IDA и Ghidra имеют схожую функциональность. Тем не менее первый дизассемблер известен качественным анализом низкоуровневого кода и работы с различными типами данных, Ghidra выделяется своими инструментами декомпиляции, рассмотрим те из них, которые относятся к теме работы:

1. Ghidra использует анализ потока данных для определения зависимостей между данными и инструкциями. Соответственно, несвязный код будет определён как мёртвый.

2. Представлен интерактивный граф потока управления. Таким образом, при анализе структуры программы возможно визуально выделить части кода, которые не влияют на результат.

3. Символьное выполнение (Symbolic Execution) позволяет анализировать возможные пути выполнения программы, что может помочь в обнаружении мертвого кода.

4. В Ghidra предусмотрена возможность повторного использования кода, включая функции и сценарии, что позволяет сэкономить время и ресурсы при многократном анализе.

5. Мёртвый код можно сворачивать, поэтому результат декомпиляции более компактный и понятный, что значительно облегчает работу с объёмными проектами.

6. Идентификация констант может помочь выявить код, который не зависит от входных данных и всегда возвращает фиксированный результат.

7. Ghidra определяет бесконечные циклы

8. Возможность работы с программой, скомпилированной в двоичный код, статический анализ этого кода может выявить недостижимые или неиспользуемые инструкции.

2.3. Выводы обзора

Из анализа существующих обфускаторов с открытым кодом можно выделить главные недостатки, которые не позволяют использовать готовые методы в нашей работе:

1. Реализованы базовые методы обфускации, связанные с мёртвым кодом, все они легко определяются деобфускаторами, что значительно снижает надёжность защиты программы.

2. Методы, представленные в открытых обфускаторах, чаще всего реализованы для определенной архитектуры. Поэтому с них можно взять именно идею обфускации, но при этом реализовать их таким образом, чтобы они могли подходить под разные архитектуры.

В случае с коммерческими проектами невозможно до конца разобраться в обфускации. Также практически отсутствует информация о методах добавлении мёртвого кода.

Таким образом, необходимо реализовать метод добавления мёртвого кода, который будет встроен в исходный прототип обфускатора, тем самым позволит развивать разрабатываемый обфускатор.

3. Реализация

В данном разделе описаны методы, которые были реализованы в исходном прототипе обфускатора. Также рассмотрены решения, которые соответствуют поставленным задачам в рамках работы.

3.1. Представление обфускатора

Обфускатор состоит из нескольких частей. Одна их них — ядро, в нем описаны интерфейсы и общеприменительные методы (методы, которые идентичны для всех архитектур). Остальные части представляют собой платформозависимые плагины.

3.2. Решение

Анализ существующих методов обфускации, показал, что мертвый код, используемый на уровне ассемблерного кода, можно разделить на две части. Первая представляет собой недостижимый код, то есть участок кода, который никогда не выполняется в процессе выполнения программы (это могут быть условия, которые всегда оцениваются как ложные или ветвления, которые никогда не будут достигнуты). Вторая часть представляет собой вставку отдельных команд в линейные участки (они могут дублировать уже существующую логику программы или же содержать в себе действия, которые не влияют на результат выполнения).

В открытых проектах обфускаторов по большей части реализован недостижимый код. Скорее всего это связано с более легким встраиванием данного метода в программу, а также с более легкой его поддержкой (при встраивании отдельных команд необходимо учитывать многие особенности архитектуры, в то же время при встраивании недостижимых инструкций данный фактор требует чуть меньшего внимания). Но в коде, который не задействуется при выполнении программы, есть определенный минус: при реверс-инжиниринге он легко распознается.

Исходя из соображений выше, было принято решении, что в качестве мертвого кода будет произведена определенная вставка отдельных команд в линейные участки. Для реализации этого решения нужно было понять: какие методы стоит реализовать в ядре обфускатора, а какие являются платформозависимыми, следовательно, должны находиться в плагинах обфускатора.

Зависимыми элементами в контексте генерации и вставки мертвого кода являются регистры и инструкции. Поэтому в плагине были реализованы следующие методы: метод parse, который в зависимости от архитектуры разбирает линейные участки; метод генерации команд мертвого кода, метод получения используемых регистров, здесь же и реализован список доступных регистров для данной архитектуры; метод, позволяющий установить количество аргументов для каждой вызывающей команды; метод, который в случае наличия свободных мест в вызывающей команде генерирует дополнительные аргументы.

В ядре обфускатора реализованы следующие методы: метод вставки мертвого кода; метод получения свободных регистров и случайный выбор одного из них; метод вставки аргументов в вызовы функций; метод, который помогает установить количество аргументов для каждой вызывающей команды (В ядре реализована общая логика работы с использованием платформозависимых данных, получаемых из плагина). Также в ядре реализован флаг активации вставки мертвого кода при работе обфускатора.

Как было сказано выше в качестве мертвого кода в данной работе выступают отдельные команды в линейных участках ассемблерного кода. Для этого потребовалось доделать парсер, который работает следующим методом: ассемблерный файл разделяется на секции, внутри каждой секции выделяются объекты, объекты типа «функция» разделяются на линейные участки. После получения линейных участков формируется список свободных регистров.

3.3. Методы в плагине x86-64

Для запуска обфсукатора в паре с ядром необходимо использовать плагин, отвечающий за конкретную архитектуру. Плагин x86 - 64 был полностью реализован и протестирован, было принято решение совершить полный цикл работы обфускатора для архитектуры x86 - 64.

Так как обфускация происходит на уровне ассемблерного кода, нужно учитывать многие факторы. Один из них это то, что регистры в соответствии с соглашением о вызовах делятся на две группы: volatile-регистры и nonvolatile (изменяемые и неизменяемые регистры). Первые могут изменять свои значения, поэтому функции не нужно сохранять значения этих регистров при вызове другой функции (за их сохранность отвечает вызывающая функция). Значения вторых должна сохранять перед использованием вызываемая функция.

Таким образом, для вставки команд в линейные участки необходимо знать: какие регистры являются свободными, то есть относятся к volatile-регистрам и не были задействованы с момента вызова функции. в противном случае использование такого регистра может привести к потери данных, с которыми работает функция.

Необходимо реализовать метод, который позволит получать использованные регистры, а также выделить список регистров, которые можно использовать в последующем при генерации команд.

Тут следует учитывать следующие моменты: внутри функции можно использовать только те регистры, которые сохраняет вызывающая функция; через некоторые регистры передаются аргументы, и так как на данный момент нет реализации, позволяющей получать реальное количество аргументов, передающихся функции, от использования упомянутых регистров пришлось отказаться.

Method 1. Метод получения используемых регистров

```
class IntelArch(ArchetectureInterface):
    arch_name = "x86_64"

    @staticmethod
    def get_initial_registers() -> list[str]:
        return ["RAX", "R10","R11"]


class IntelCommand(Command):
    def get_affected_registers(self) -> list[str]:
        if len(self.args) > 0:
            if self.mnemonic in ['push', 'pushq']:
                return []
            elif self.mnemonic in ['call', 'callq']:
                return ["RAX"]
            elif len(self.args) == 2:
                return[self._prepare_reg_name(self.args[1])]
            else:
                return[self._prepare_reg_name(self.args[0])]
        return []

    def _prepare_reg_name(self, str) -> str:
        if str.startswith("%"):
            return str[1:].upper()
        return str
```

Метод получения команды, которая будет использована в качестве мертвого кода, также реализован в плагине, так как зависит от конкретной платформы. Метод реализован только для трех команд, так как они являются более простыми в использовании, при их вставке нет необходимости следить за значениями, которые могут меняться при их использовании.

Method 2. Метод получения команды для мертвого кода

```
class IntelArch(ArchetectureInterface):
    arch_name = "x86_64"

    @staticmethod
    def get_command_for_dead_code(free_registers) -> Command:
        commands = ["movq", "addq", "subq"]
        random_command = random.choice(commands)
        if len(free_registers) > 0:
            random_register = random.choice(free_registers)
            random_number = f"{random.randint(-99999, 99999)}"
            return Command(random_command, [random_number, f" %
                {random_register.lower()}"])
```

Для дополнительной генерации команд, имитирующих использование дополнительных аргументов было принято решение реализовать метод получения количества аргументов с определенными регистрами в коде. Если полученное значение меньше шести, то генерируются дополнительные команды (на данный момент ограничились шестью аргументами, чтобы не использовать стек).

Method 3.Метод получения количества аргументов

```
class IntelParser(ParserInterface):
    def get_args_count(self, commands: list[IntelCommand]) -> int:
        mov_commands = ["movl", "movb", "mov", "leaq"]
        registers = ["RDI", "RSI", "RDX", "RCX", "R8", "R9", "EDI", "ESI",
            "EDX", "ECX"]
        push_command = "push"
        args_count = 0
        for command in reversed(commands):
            if command.mnemonic in mov_commands and
               command.get_affected_registers()[0] in registers:
                args_count +=1
                continue
            if command.mnemonic.startswith(push_command):
                args_count +=1
                continue
            if command.mnemonic in mov_commands and
               command.get_affected_registers()[0] in ["AL", "AX", "EAX",
               "RAX"] and args_count == 0:
                continue
            break
        return args_count
```

Method 4. Метод генерирования дополнительных команд

```
class IntelParser(ParserInterface):
    def generate_additional_commands(self, args_count: int) -> list[Command]:
        mov_command = "movq"
        registers = ["RDI", "RSI", "RDX", "RCX", "R8", "R9"]
        additional_commands = []
        if args_count <= 6:
            for i in range(args_count, 6):
                register = registers[i]
                value = random.randint(-655655, 655254)
                command_str = f"{mov_command}\t{value},\t%{register.lower()}"
                command = IntelArch.get_command(command_str)
                command.set_free_registers([])
                additional_commands.append(command)
        return additional_commands
```

3.4. Методы в ядре

Рассмотрим методы, которые реализованы в ядре обфускатора, следовательно, они подходят для всех платформ.

Исходя из логики программы, был реализован метод, который позволяет получать список свободных регистров (для данного метода потребовалось реализовать метод с получением используемых регистров в плагине), а также метод, позволяющий случайным образом выбрать свободный регистр, из которого будет состоять образованная команда.

Method 5. Метод получения списка свободных регистров

```
class Command(SourceLine):
   def set_free_registers(self, free_registers: list, architecture=None):
        self.free_registers = free_registers
        self.architecture = architecture

    def get_free_registers(self) -> list:
        if self.free_registers is None:
            raise ValueError("free registers not set")
        return self.free_registers

    def get_random_free_register(self, free_registers: list) -> str:
        random_register = random.choice(free_registers)
```

Также потребовался метод определения количества аргументов для команд вызова.

Method 6. Метод получения количества аргументов

```
class ParserInterface(ABC):
    def help_to_get_arg_count(self, lin_section: LinearSection):
            commands = lin_section.get_elements()
            previous_call_command = None
            for idx in range(len(commands)):
                if isinstance(commands[idx], CallCommand):
                    start = 0 if previous_call_command is None else
                       previous_call_command
                    commands[idx].set_args_count(self.get_args_count
                     (commands[start:idx]))
                    previous_call_command = idx
```

Чтобы метод генерации команды для мертвого кода стала активной в ядре, его нужно «запросить» у плагина. Метод в ядре получает список свободных регистров, затем «запрашивает» у плагина команду для мертвого кода, выбирает регистр и возвращает сгенерированную команду.

Method 7. Метод генерирования команды

```
def request_dead_code(self, architecture, free_registers: list[str]) ->
   Command:
        dead_code_cmd = architecture.get_command_for_dead_code()
        random_register = Command.get_random_free_register(free_registers)
        dead_code_cmd.set_free_registers([random_register], architecture)
        return dead_code_cmd
```

Чтобы пользователь мог использовать или не использовать добавление мертвого кода (по своему смотрению), стало необходимым реализовать флаг вызова и передать ему нужный метод.

Method 8. Флаг активации метода вставки мертвого кода

```
    def get_argparser(self):
        parser.add_argument("-dc", action="store_true",
           dest="dead_code_flag", help="Activate dead code inserttion")

    def run(self):
        args = self.get_argparser()
        target_arch, target_parser, target_printer = self.get_arch(
            self.archs[args.arch]()
        )
        with open(args.input_file, encoding="utf-8") as f:
            current_section: Section = None
            source: Source = target_arch.get_source()
            dirs = []
            if args.dead_code_flag:
                self.insert_dead_code(source)
```

Метод вставки мертвого кода работает следующим образом: найдя очередной вызов функции, он получает количество аргументов и запрашивает у плагина генерацию команд для дополнительных аргументов.

Method 9. Метод вставки мертвого кода

```
class ObfuscatorApp:
    def insert_dead_code(self, source: Source, archs, parser:
        ParserInterface):
         for section in source.get_sections():
             objects = section.get_objects()
             for name in objects:
                 if isinstance(objects[name], Function):
                     for linear_section in objects[name].get_lin_sections():
                         for idx, command in
                             enumerate(linear_section.get_elements()):
                              if isinstance(command, CallCommand):
                                  if(command.get_fake_args_count() > 0):
                                      continue
                                  arg_count = command.get_args_count()
                                  additional_args =
                                     parser.generate_additional_commands
                                      (arg_count)
                                  for additional_command in additional_args:
                                      linear_section.insert_element
                                          (additional_command, idx)
                                  command.set_fake_args_count
                                      (len(additional_args))
```

4. Эксперимент

Для проверки работы мертвого кода в обфускаторе, и оценки того, насколько он запутывает код, было принято решение: передать обфусцированный код в известные дизассемблеры, выбор которых описан в обзоре данной работы.

Для оценки скорости работы программы до обфускации и после компанию предоставила тесты производительности linpack, на которых с помощью многократного запуска был вычислен процент замедления программы после применения обфускатора.

4.1. Первый эксперимент

4.1.1. Тестовый стенд

Исходя из акцента заказчика на использование компилятора Clang и языка C, тесты linpack соответствовали требованиям упомянутым выше.

В качестве дизассемблеров в данном эксперименте выступили Ghidra и IDA.

4.1.2. Оценка качества обфускации

Для проведения оценки тесты linpack, с помощью компилятора Clang были представленны в виде ассемблерного кода, затем переданы в обфускатор, в качестве метода обфускации использовался только мертвый код. После обфускации полученный код был загружен в Ghidra и IDA.

Хорошим показателем для обфускации выступает фактор, что дизассемблеры не могут разобрать исходный код программы.

Рассмотрим частично main одного из тестов. Ниже представлен оригинальный и обфусцированный ассемблерный код.

```
main: # @main
  .cfi_startproc
# %bb.0:
  pushq %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset %rbp, -16
  movq %rsp, %rbp
  .cfi_def_cfa_register %rbp
  subq $64, %rsp
  movl $0, -4(%rbp)
  leaq .L.str(%rip), %rdi
  callq getenv@PLT
  movq %rax, -16(%rbp)
  cmpq $0, -16(%rbp)
  jne .LBB0_2
# %bb.1:
  movl $100, -20(%rbp)
  jmp .LBB0_3
.LBB0_2:
            ...
.LBB0_11:
  movq mempool(%rip), %rdi
  callq free@PLT
  leaq .L.str.11(%rip), %rdi
  movb $0, %al
  callq printf@PLT
 .LBB0_12:
            ...
```

Оригинальный код

```
main:
  pushq %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset %rbp,-16
  movq %rsp,%rbp
  .cfi_def_cfa_register %rbp
  subq $64,%rsp
  movl $0,-4(%rbp)
  leaq .L.str(%rip),%rdi
  movq $-335027,%r9
  movq $23938,%r8
  movq $401271,%rcx
  movq $-470475,%rdx
  callq getenv@PLT
  movq %rax,-16(%rbp)
  cmpq $0,-16(%rbp)
  jne .LBB0_2
.LBL1_0:
  movl $100,-20(%rbp)
  jmp .LBB0_3
.LBB0_2:
            ...
 .LBB0_11:
  movq mempool(%rip),%rdi
  movq $58966,%r9
  movq $296046,%r8
  movq $-236809,%rcx
  movq $-430707,%rdx
  movq $-642297,%rsi
  callq free@PLT
  leaq .L.str.11(%rip),%rdi
  movb $0,%al
  movq $480603,%r9
  movq $273652,%r8
  movq $430373,%rcx
  movq $237,%rdx
  movq $355715,%rsi
  callq printf@PLT
 .LBB0_12:
            ...
```

Обфусцированный код

Как заметно из кода, обфускатор выполнил свою работу, вставил необходимые аргументы. Добавлены только movq, так как следую соглашению о вызовах только эту команду использовать в вызывающих функциях достаточно легко — остальные требуют дополнительной логики во всем обфускаторе. В примере не добавлены иные команды вне вызывающих функциях, так как тестирование показало, что с подобным типом мертвого кода дизассемблеры справляются достаточно легко (данные участки кода были закомментированы для более четкого представления рабочего мертвого кода).

Теперь рассмотрим результаты анализа Ghidra оригинального и обфусцированного кода.

Как видно из Рис. 1 Ghidra полностью распознала функцию main в оригинальном ассемблерном коде linpack.

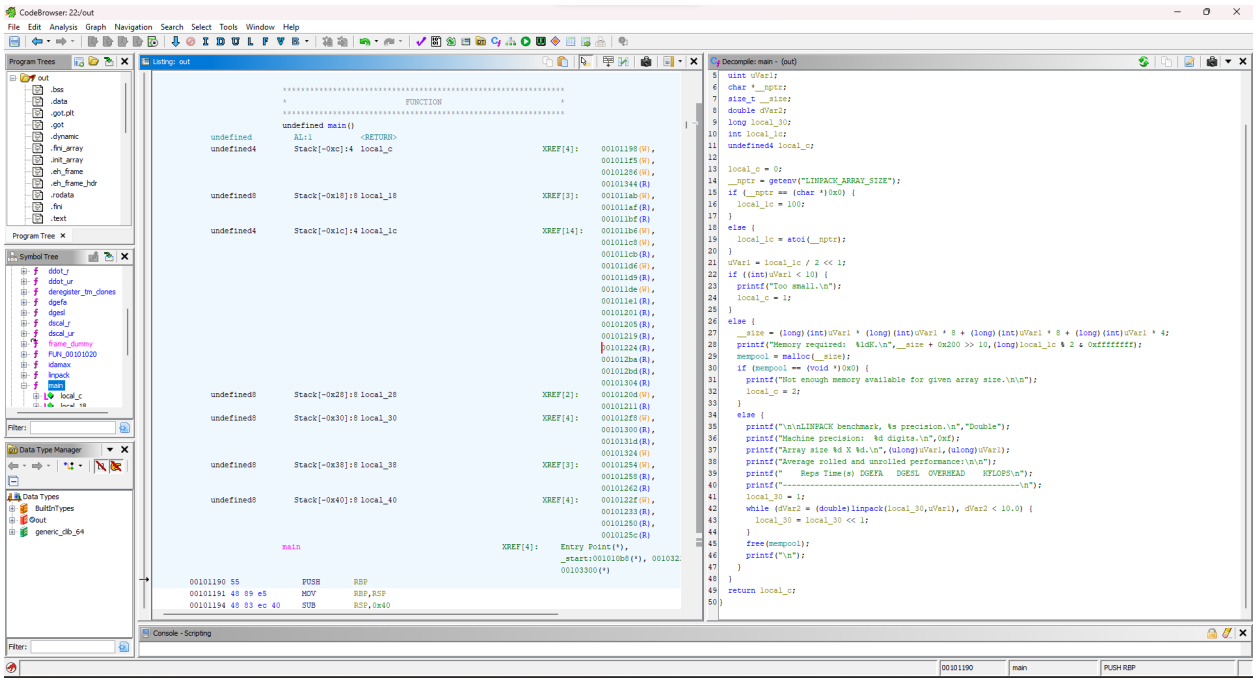

Рис. 1: Результат анализа Ghidra оригинального кода

Как видно из Рис. 2 Ghidra распознала большее количество параметров в printf, чем есть на самом деле.

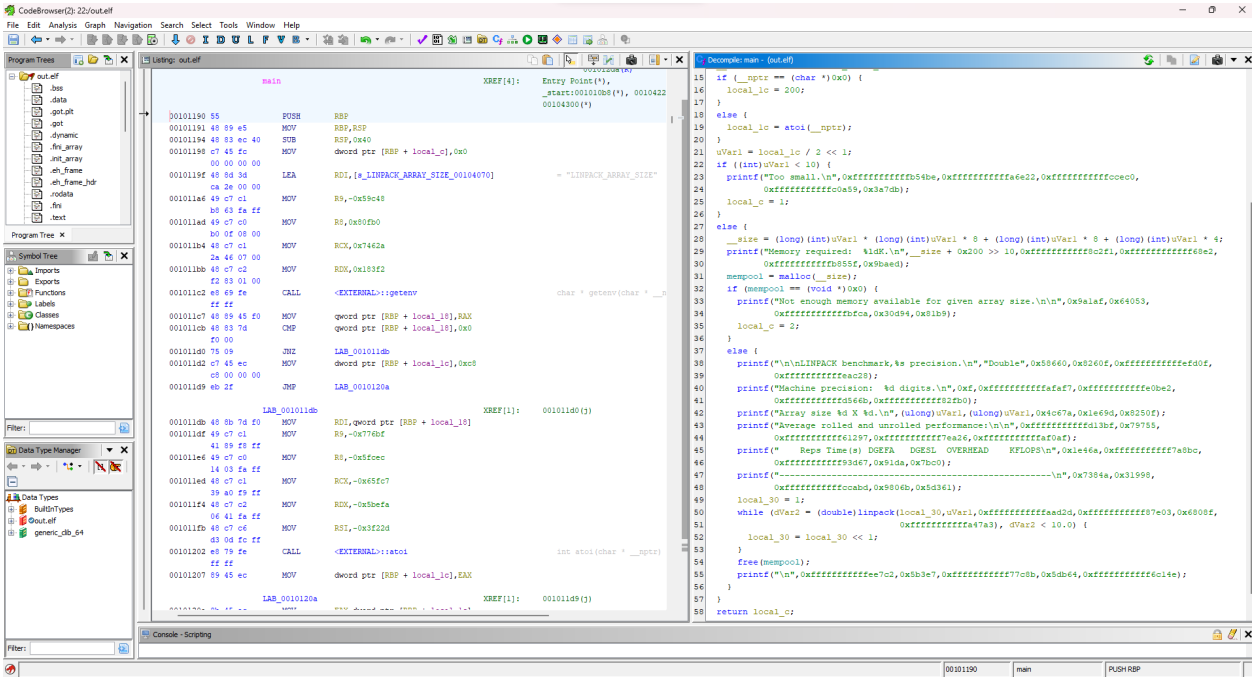

Рис. 2: Результат анализа Ghidra обфусцированного кода

IDA справилась со своей работой на тесте linpack. Ниже приведен пример, в котором в качестве тестовой программы взяли элементарный код – на вход поступает два аргумента, затем выводится их сумма.

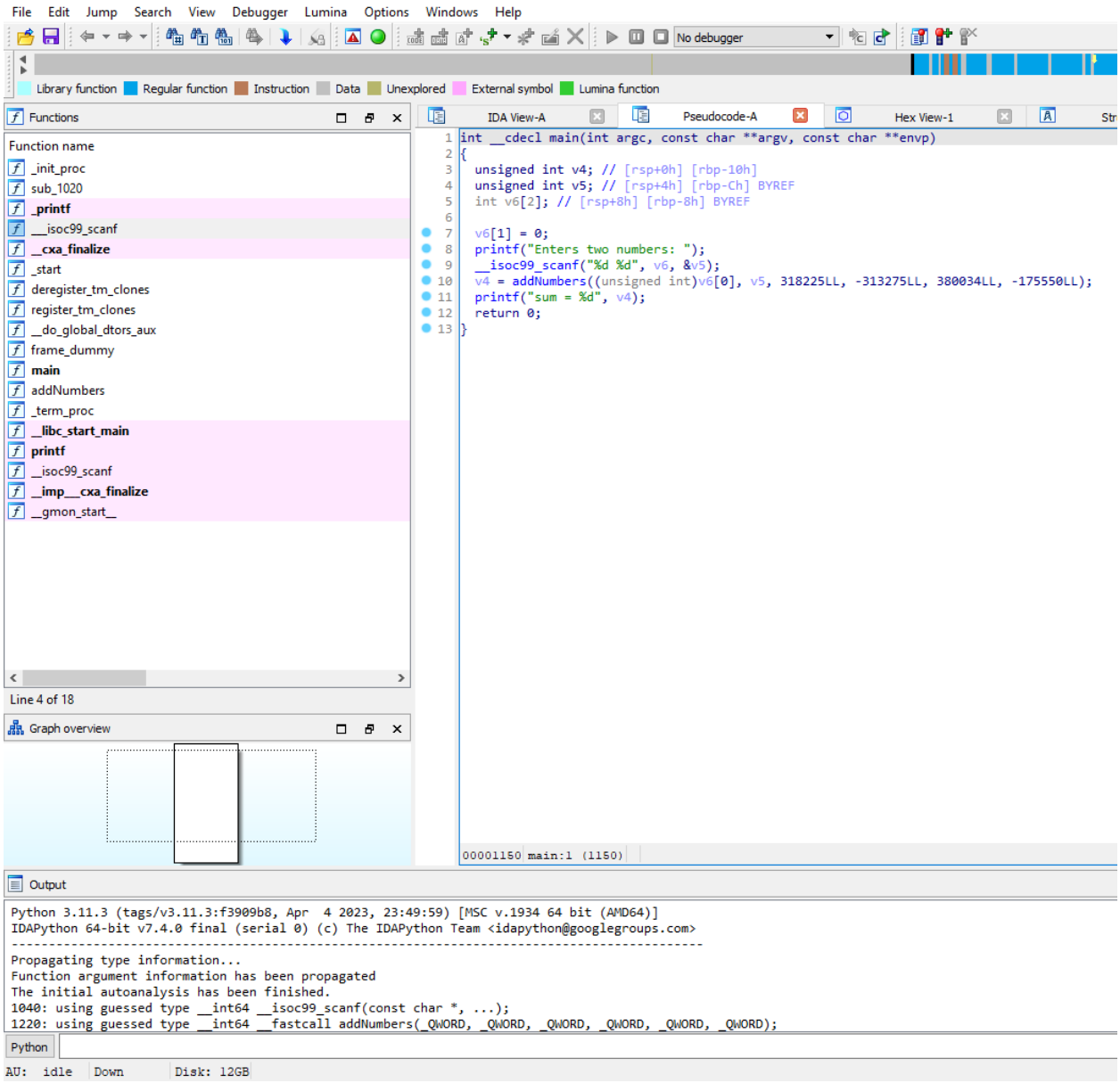

Рис. 3: Результат анализа IDA (main 1 просмотр)

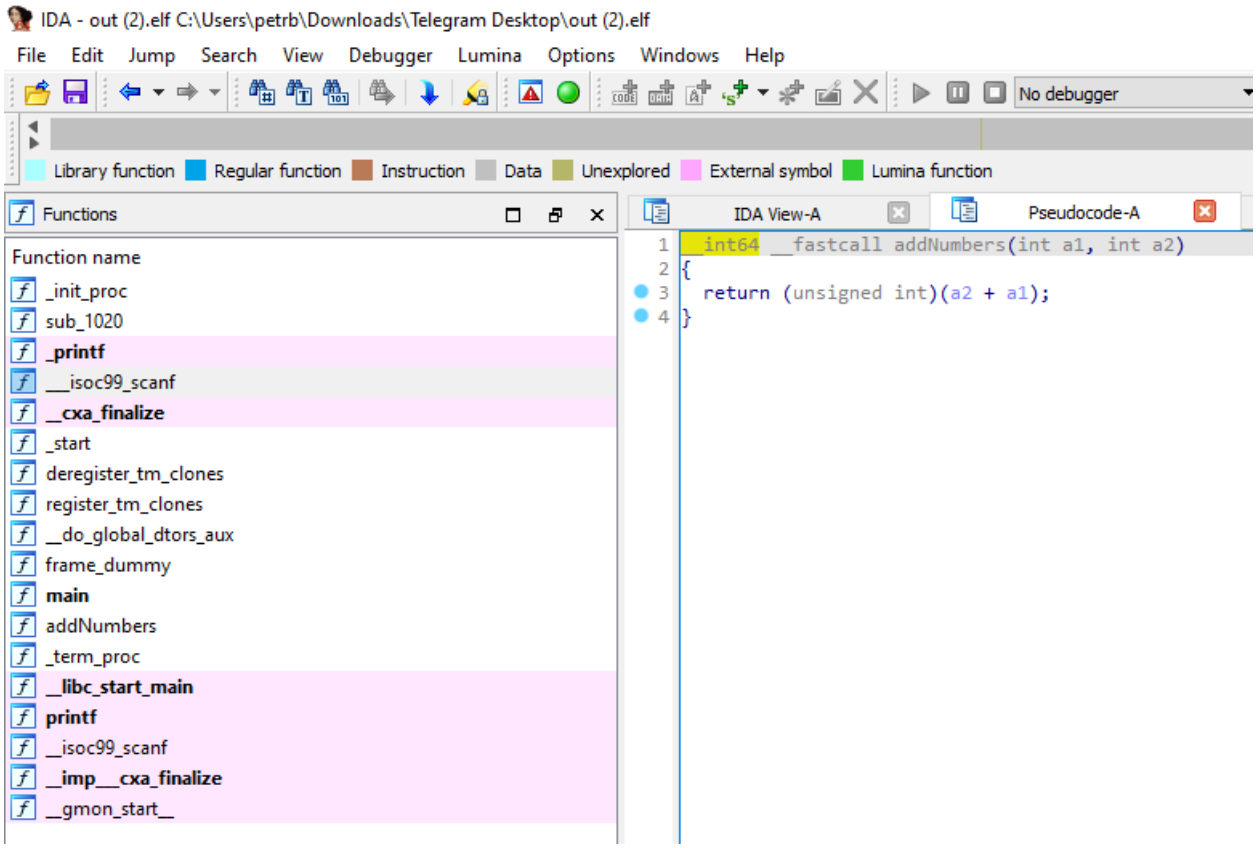

Рис. 4: Результат анализа IDA (поток управления)

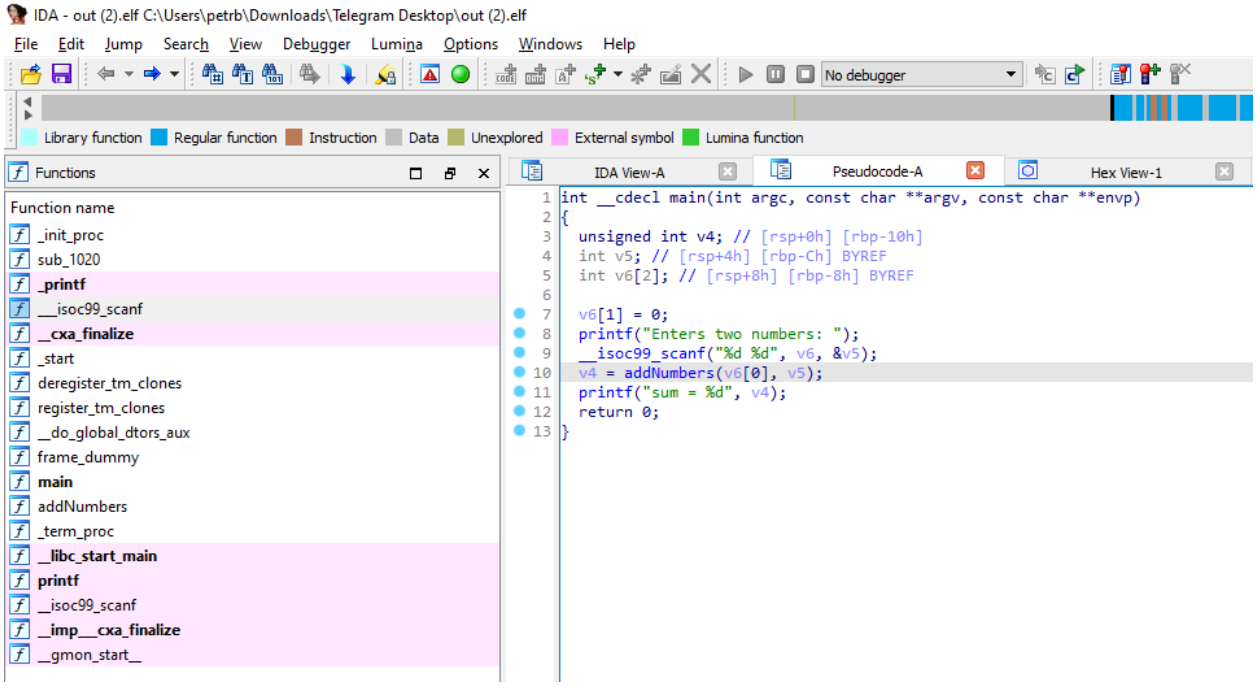

Рис. 5: Результат анализа IDA (main после просмотра потока управления)

Из Рис. 5 видно, что IDA смогла распознать исходный код программы. Но стоит отметить, что количество аргументов в main изначально не было равно двум Рис. 3, и только после перехода в поток управления, в котором оказалось два аргумента Рис. 4, функция main стала верной, что и продемонстрирвоано на Рис. 5, это служит отличным результатом для дальнейшего тестирования.

Таким образом, один из популярных дизассемблеров не смог распознать количество параметров в функции.

4.2. Второй эксперимент

Во втором эксперименте измерим время за которое выполняется исходная программа и обфусцированная.

4.2.1. Тестовый стенд

Для замеров времени работы алгоритмов использовался компьютер со следующими характеристиками:

- CPU: Intel(R) Core(TM) i5-9300H, тактовая частота: 2,40 ГГц, 4 ядра, 8 потоков;
- ОЗУ: 16,0 ГБ.

Для проведения эксперимента использовалось следующее программное обеспечение:

- Python: 3.11.6;
- Clang 14.0.0.

4.2.2. Тестовые данные

Для проведения экспериментов были взяты тесты linpack, предоставленные компанией «Софтком».

4.2.3. Замеры времени работы

Для проведения замеров времени тест запускался 10 раз в оригинальном, а также в обфусцированном формате, затем вычислили среднее время выполнения для каждого количества повторений.

Среднее время выполнения необфусцированного кода представлено в таблице 2, погрешность измерений составляет 3%.

Reps	**Time (s)**
2048	0.54
4096	1.10
8192	2.21
16384	4.39
32768	8.77
65536	17.65

Таблица 1: Среднее время выполнения оригинального кода

Среднее время выполнения обфусцированного кода представлено в таблице 2, погрешность измерений также составляет 3%.

Reps	**Time (s)**
2048	0.57
4096	1.15
8192	2.30
16384	4.58
32768	9.15
65536	18.39

Таблица 2: Среднее время выполнения обфусцированного кода

Как видно из данных выше — время выполнения обфусцированного кода увеличилось не более чем на 7%, компания ожидала, что этот показатель будет около 12%. Следовательно, результат превзошел ожидания, что свидетельствует о том, что обфускация мертвым кодом не сильно замедлила процесс, это хороший показатель.

Заключение

Подведем итоги того, что было сделано за время осенней практики:

- Реализована генерация мертвого кода и данных.
- Реализован метод вставки мертвого кода и данных в исходный код на уровне ассемблерного языка.
- Все методы и данные активны в исходном прототипе обфускатора.
- Проведено тестирование на качество обфускации, которое дало следующие результаты: IDA справилась с обфусцированным кодом, Ghidra не смогла правильно распознать количество параметров.
- Вычислено время выполнения программы в обфусцированном формате, замедление составило 7%, что является достаточно не большим показателем.

Проект обфускатора «Софтком» является проектом с закрытым кодом.

В дальнейшем планируется реализовать генерацию мертвого кода с бóльшим количеством команд (для этого потребуется глубокое погружение в работу регистров), а также добавление дополнительных математических операций, чтобы IDA и другие дизассемблеры переходя в поток управления все равно не понимали, какие команды являются лишними.

Список литературы

[1] Assessment of Source Code Obfuscation Techniques / Alessio Assessment, Leonardo Regano, Marco Torchiano et al. // 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM). — 2016. — P. 11–20.

[2] Bruce Dang Alexandre Gazet Elias Bachaalany. Practical reverse engineering. — 10475 Crosspoint Boulevard Indianapolis : John Wiley Sons, Inc., 2014. — ISBN: 1118787390.

[3] Ghidra. — URL: https://github.com/NationalSecurityAgency/ghidra.

[4] IDA. — URL: https://hex-rays.com.

[5] The LLVM Compiler Infrastructure. — URL: https://www.llvm.org/.

[6] Obfuscator-LLVM – Software Protection for the Masses / Pascal Junod, Julien Rinaldini, Johan Wehrli, Julie Michielin // Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015 / Ed. by Brecht Wyseur. — IEEE, 2015. — P. 3–9.

[7] Shiraz Matthieu Tofighi. Analysis of obfuscation transformations on binary code. — 2019. — URL: https://theses.hal.science/tel-02907117/.

[8] Suresh Anjali J., Sankaran Sriram. A Framework for Evaluation of Software Obfuscation Tools for Embedded Devices // Applications and Techniques in Information Security / Ed. by Lejla Batina, Gang Li. — Singapore : Springer Singapore, 2020. — P. 1–13.

[9] Themida. — URL: https://themida.com.

[10] Thomas Ryan Devine Maximillian Campbell Mallory Anderson Dale Dzielski. SREP+SAST: A Comparison of Tools for Reverse Engineering Machine Code to Detect Cybersecurity Vulnerabilities in Binary Executables. — 2022. — URL: https://ieeexplore.ieee.org/abstract/document/10216474/authors#authors.

[11] UnThemida: Commercial obfuscation technique analysis with a fully obfuscated program / Jae Hyuk Suk, Jae Yung Lee, Hongjoo Jin et al. // Software - Practice and Experience. — 2018. — . — Vol. 48, no. 12. — P. 2331–2349. — Funding Information: This work was supported by Institute for Information Communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2014-6-00910, Study on Security of Cryptographic Software). Publisher Copyright: © 2018 John Wiley Sons, Ltd.

[12] The tigress C obfuscator. — URL: https://tigress.wtf.