Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б08-мм

Обфускация самомодифицирующимся кодом

Чернобровкина Юлия Владиславовна

Отчёт по учебной практике
в форме «Производственное задание»

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

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

Санкт-Петербург
2025

Оглавление

Введение **3**

1. Постановка задачи **5**

2. Обзор **6**

2.1. Обзор технологии самомодификации кода 6

2.2. Обзор существующих решений 8

2.3. Обзор используемых технологий 11

3. Реализация **14**

3.1. Алгоритм шифрования 14

3.2. Изменения в кодогенерации LLVM 14

3.3. Создание функции-шифратора 15

3.4. Добавление обрамляющих блоков 16

3.5. Шифрование тела функции 19

4. Ограничения **20**

4.1. Ограничение на уровне релокаций 20

4.2. Ограничения на структуру функций 20

5. Эксперимент **22**

5.1. Тестовый стенд 22

5.2. Оценка качества обфускации 23

5.3. Выводы 29

Заключение **30**

Список литературы **31**

Введение

В современном мире безопасность является критически важным качеством для ПО. Одной из ключевых проблем в этой области является защита ПО от анализа и взлома, что особенно актуально для проприетарных приложений или приложений, работающих с персональными данными. Для достижения высокого уровня защиты ПО были разработаны разнообразные методы и специализированные инструменты.

В частности, обеспечить защиту кода от взлома помогает обфускация — преобразование кода, сохраняющее его функциональность, но значительно затрудняющее понимание и анализ. Обфускация в свою очередь разбивается на три вида: обфускация исходного кода, промежуточного представления или двоичных кодов. Для каждого вида обфускации разработаны собственные специфичные для него методы и приемы [3].

Компания ООО «Софтком» разрабатывает обфускатор, основанный на LLVM — фреймворке для разработки компиляторов, анализаторов и оптимизаторов. Обфускатор уже реализует различные преобразования на уровне промежуточного представления LLVM IR [13], [11], [14], следующим решением стала реализация трансформаций на уровне двоичных кодов. В частности обфускатор нуждается в преобразованиях самомодифицирующимся кодом.

Самомодифицирующаяся программа — это программа, которая меняет часть своего кода непосредственно во время исполнения. Описанный прием может применяться в различных случаях, например для оптимизации чувствительных к скорости исполнения участков кода. Отдельного внимания заслуживает применение самомодифицирующегося кода при разработке вредоносного ПО, а именно полиморфных вирусов [12]. Для данной работы существенны обфусцирующие свойства самомодифицирующегося кода, так как этот прием зарекомендовал себя в качестве эффективного средства противостояния статическому анализу — одного из средств реверс-инжиниринга [7].

Таким образом, в рамках данной работы предлагается исследовать

возможность реализации метода обфускации на основе самомодифицирующегося кода непосредственно на этапе компиляции с использованием инфраструктуры LLVM. Предполагается разработка прототипа метода и изучение ограничений подхода. Тема данной работы предложена компанией «Софтком».

1. Постановка задачи

Целью работы является исследование возможности реализации метода самомодификации кода на этапе компиляции в инфраструктуре LLVM для обфускатора компании «Софтком».

Для её выполнения были поставлены следующие **задачи**:

1. провести обзор существующих решений и используемых технологий;
2. реализовать прототип метода самомодификации кода на основе инфраструктуры LLVM;
3. выявить ключевые ограничения, возникающие при реализации самомодификации на этапе компиляции;
4. провести экспериментальное исследование полученного прототипа.

2. Обзор

2.1. Обзор технологии самомодификации кода

Тема самомодификации кода сохраняет свою актуальность на протяжении многих лет. Наряду с ранними работами исследователей защищённости ПО, таких как [1], [9], [8], [4], появляются современные исследования, расширяющие границы использования этого метода.

Со временем взгляд на применение самомодификации в коде изменился. Если авторы ранних работ (например, [4]) выделяли самомодификацию в отдельную категорию защитных методов наравне с обфускацией, шифрованием и фрагментацией кода, то современные подходы интегрируют её в другие техники защиты. Так, в работах [6] и [10] самомодификация рассматривается как элемент обфускации.

Кроме того, самомодифицирующийся код нашёл применение вне сферы защиты ПО. Например, исследование [2] описывает его использование в сфере искусственного интеллекта, а в работе [5] анализируется роль самомодификации в создании вредоносных программ.

В данной работе самомодификация рассматривается как метод обфускации, поэтому дальнейший обзор сосредоточен на анализе её свойств в контексте противодействия реверс-инжинирингу.

На уровне инструкций самомодифицирующийся код можно описать так: инструкция p заменяет другую инструкцию q на инструкцию r непосредственно во время исполнения программы, пример на рисунке 1.

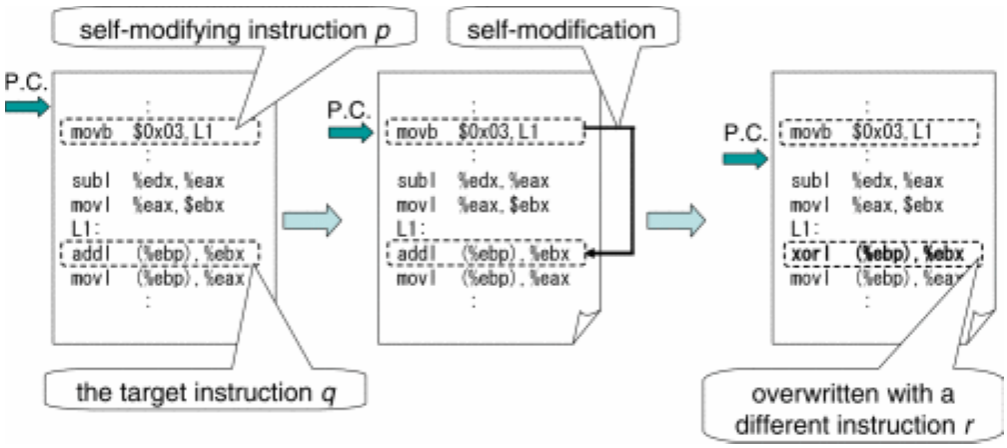

Рис. 1: Иллюстрация самомодификации кода на уровне инструкций. Изображение взято из источника https://ieeexplore.ieee.org/document/1245338(дата обращения: 24 марта 2025 г.).

Таким образом, имея на входе оригинальную программу, обфусцирующий механизм преобразует ее в новую, состоящую из зашифрованных инструкций (∘ на рисунке 2) и инструкций шифрования/дешифрования (□ и △ на рисунке 2).

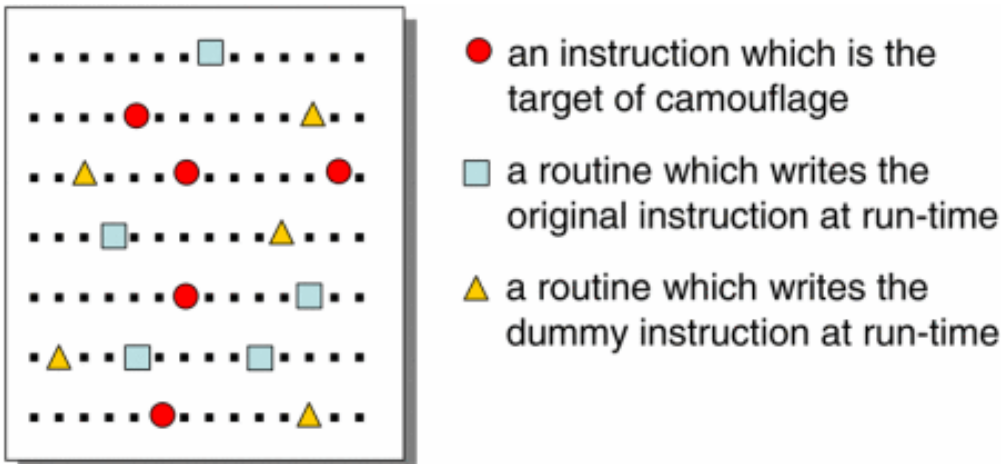

Рис. 2: Структура самомодифицирующегося кода. Изображение взято из источника https://ieeexplore.ieee.org/document/1245338(дата обращения: 24 марта 2025 г.).

Следует отметить, что наличие однообразия и паттернов в шифрующих инструкциях напрямую влияет на возможность выявлять шифрующие модули в коде, упрощая реверс-инжиниринг. Поэтому имеет смысл использовать различные схемы шифрования и стремиться к рандомизации выбора инструкций.

В большинстве упомянутых работ утверждается, что самомодификация способна значительно усложнить анализ кода, тем самым увеличивая затраты на реверс-инжиниринг и снижая его целесообразность. Несмотря на описанные преимущества, нельзя не упомянуть ограничения на использование самомодификации:

1. **Ограниченная поддержка платформ**. Самомодификация кода предполагает, что одна и та же область памяти исполняема и записываема одновременно, что противоречит $W \hat{} X$ принципу (Write Xor Execute). Если платформа поддерживает указанный принцип и не допускает одновременно исполняемых и записываемых секций, то использование самомодификации оказывается невозможным.

2. **Ложные срабатывания антивирусов**. Программа, содержащая самомодифицирующийся код, может вызывать срабатывания антивирусных программ, так как самомодификация также используется во вредоносном ПО.

Описанные выше проблемы подробно рассмотрены в работе [9].

2.2. Обзор существующих решений

2.2.1. Требования компании

Приведем основные требования компании, предъявляемые к реализации метода обфускации самомодифицирующимся кодом:

1. Разрабатываемый обфускатор основан на фреймворке LLVM, поэтому требуемый метод обфускации должен быть реализован через интерфейс LLVM и интегрирован в существующую кодовую базу.

2. Обфускация должна производиться на уровне двоичных кодов для x86-64 архитектуры.

Принимая во внимание приведенные требования, перейдем к обзору существующих решений.

2.2.2. Tigress

Tigress — это некоммерческий обфускатор, разработанный в обучающих и исследовательских целях. Обфускация осуществляется на уровне исходного кода на языке C. Проект поддерживает широкий набор обфусцирующих методов, подробное описание которых можно найти на официальном сайте проекта [1]. Среди них самомодификация кода реализована методами `Jit` и `JitDynamic`. Рассмотрим подробнее каждый метод:

1. Jit (Just-In-Time Compilation) — данный метод преобразует части целевой функцию в инструкции в промежуточном представлении, которые при вызове функции компилируются в исполняемый код. Таким образом, исполняемый код для части функции генерируется в момент непосредственного исполнения, что значительно затрудняет статический анализ кода. Для улучшения обфусцирующих свойств метод рандомизирует инструкции промежуточного представления при каждом запуске Tigress.

2. JitDynamic — схожий с Jit метод, выполняющий шифрование базовых блоков функции с использованием инструкций промежуточного представления. До исполнения блок находится в закодированном состоянии, непосредственно перед исполнением блок декодируется, исполняется и вновь кодируется. Для использования метода доступны разные схемы шифрования: шифрование xor, xtea и другие.

Tigress является проектом с закрытым исходным кодом, а целевым для него является язык программирования C, поэтому использование Tigress для решения задач компании невозможно.

2.2.3. Themida

Themida — это коммерческий инструмент защиты ПО, предоставляющий пользователям обширный набор средств противостояния реверс-

[1] Официальный сайт обфускатора Tigress, https://tigress.wtf/(дата обращения: 24 марта 2025 г.)

инжинирингу [2]. Среди них можно выделить средства анализа целостности кода, защита от дизассемблирования и декомпиляции, антиотладчик и антидампер.

Themida также использует самомодификацию кода, а именно динамическое шифрование кода во время исполнения. Принцип этого метода схож с описанным JitDynamic методом обфускатора Tigress.

Исходный код продукта является закрытым, а описание поддерживаемых методов недостаточно подробно для какого-либо анализа и использования.

2.2.4. Obsidium

Obsidium — коммерческий продукт для защиты 32 и 64-битных Windows приложений [3]. Инструмент обеспечивает защиту от реверс-инжиниринга, модификации и незаконного распространения кода. Obsidium предоставляет широкий диапазон методов защиты: шифрование и обфускация данных и кода, виртуализация кода, система лицензирования и прочее.

Инструмент так же, как и рассмотренные ранее решения, поддерживает метод самомодификации через шифрование и дешифрование кода в момент исполнения [4].

Исходный код закрыт, а сфера применения ограничена Windows системами.

2.2.5. Выводы из обзора существующих решений

Все рассмотренные решения реализуют защиту методом самомодификации кода, причем схема метода одна: предварительно зашифрованный по некоторому алгоритму код перед самим исполнением дешифруется, после чего исполняется и шифруется вновь. При этом в качестве шифрующего алгоритма возможно применение разных вариантов.

[2]Официальный сайт Themida, https://www.oreans.com/Themida.php (дата обращения: 24 марта 2025 г.)

[3]Официальный сайт Obsidium, https://www.obsidium.de/home (дата обращения: 24 марта 2025 г.)

[4]Поддерживаемые Obsidium методы защиты ПО, https://www.obsidium.de/product/sps/features (дата обращения: 24 марта 2025 г.)

Однако ни одно из решений не может в полной мере удовлетворить требованиям компании, что делает невозможным их интегрирование в проект обфускатора. Таким образом, предлагается реализовать самомодификацию кода на основе динамического шифрования, как это сделано в рассмотренных решениях, ведя разработку в инфраструктуре LLVM с учетом требований заказчика.

2.3. Обзор используемых технологий

Поскольку разработка обфускатора компании ведется в инфраструктуре LLVM, целесообразно описать устройство LLVM и предоставляемый для разработчиков интерфейс.

LLVM — это компиляторная инфраструктура, позволяющая разрабатывать компиляторы, оптимизаторы, анализаторы и другие инструменты. LLVM предоставляет набор библиотек и хорошо документированное API на языке C++, с использованием которого разработчики могут создавать анализаторы, оптимизаторы, а также необходимый фронтенд или бэкенд — компиляторы из высокоуровневого языка в промежуточное представление LLVM IR и из LLVM IR в двоичный код соответственно.

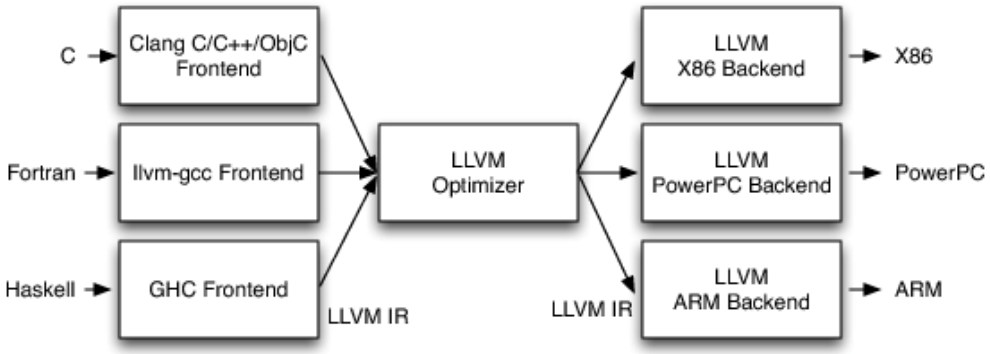

Рис. 3: Структура компиляторов в LLVM. Изображение взято из источника https://www.google.com/url?sa=i&url=https%3A%2F%2Faosabook.org%2Fen%2Fv1%2Fllvm.html&psig=AOvVaw0HHcNQkVY_ra1GPmFwC87A&ust=1742917232148000&source=images&cd=vfe&opi=89978449&ved=0CBcQjhxqFwoTCODm6OqGo4wDFQAAAAAdAAAAABAR(дата обращения: 24 марта 2025 г.).

Через соответствующие классы API определяются абстракции для таких объектов, как модуль программы, функция, базовый блок и инструкция.

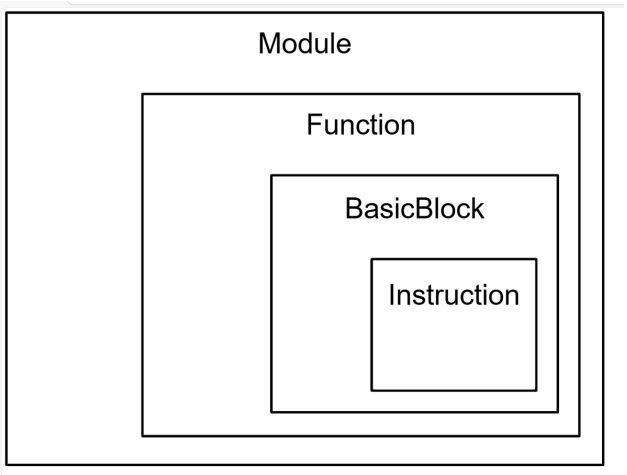

Рис. 4: Классы LLVM API. Изображение взято из источника https://sungsoo.github.io/2016/04/24/introduction-to-llvm.html(дата обращения: 24 марта 2025 г.).

В контексте данной работы наиболее интересно устройство API для создания бэкенда, его и предлагается рассмотреть подробнее.

Для реализации бэкенда существуют классы целевой архитектуры `Target`, генератора ассемблера `ASMPrinter`, секций кода `Section` и многое другое. Все вместе они представляют абстракцию для процесса генерации двоичного кода. С использованием API для реализации нового бэкенда достаточно в исходном коде LLVM определить свойства целевой архитектуры (регистры, секции, инструкции процессора) и переопределить методы генерации кода [5]. По данной схеме реализованы бэкенды для архитектур x86-64, MIPS, ARM, RISCV и других платформ.

Такой подход позволяет разработчикам значительно экономить время и ресурсы, используя готовые шаблоны и кодовую базу. Дополнительным преимуществом становится возможность использовать существующие фронтенды и оптимизаторы в связке с созданным бэкендом и получать сразу несколько готовых компиляторов высокоуровневых языков с настраевыми компонентами.

[5]Туториал по созданию нового бэкенда с помощью LLVM, https://llvm.org/docs/WritingAnLLVMBackend.html(дата обращения: 24 марта 2025 г.)

Обфускатор компании «Софтком» определяет отдельную целевую архитектуру для x86-64, в рамках которой генерируется обфусцированный код. Именно в рамках этого таргета предполагается реализовывать прототип метода самомодификации кода с использованием описанного API.

3. Реализация

Для низкоуровневой обфускации, не вписывающейся в рамки плагинов LLVM, создан таргет X86_64_OBF, который является полной копией стандартного X86 таргета. Таким образом для генерации обфусцированного кода достаточно в опциях компилятора указать целевую архитектуру как X86_64_OBF.

Разработка метода самомодификации кода велась на языке C++ в директории указанного таргета, при этом код LLVM вне директории не менялся, чтобы исключить влияние на генерацию кода без обфускации.

3.1. Алгоритм шифрования

В качестве алгоритма выбрано прибавление фиксированной константы к каждому байту тела функции. Таким образом в памяти функция содержится в зашифрованном виде, перед исполнением дешифруется путем вычитания из каждого байта той же константы и после исполнения вновь шифруется.

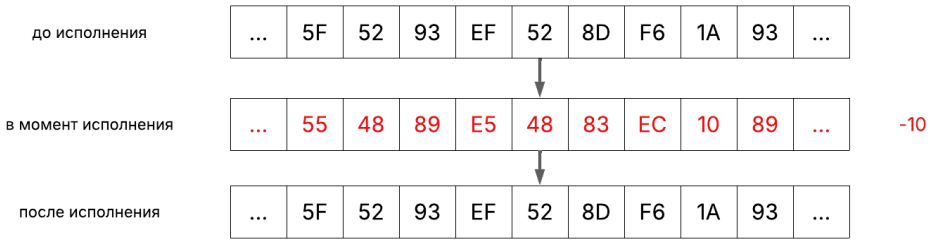

Рис. 5: Байты кода обфусцированной функции до исполнения, в момент и после исполнения.

Параметры метода (название и константа) пользователь указывает в комментариях к функции в исходном коде, которые обрабатываются механизмом разметки кода и сохраняются в метаданных функции.

3.2. Изменения в кодогенерации LLVM

Для реализации метода самомодификации понадобилось внести изменения в разные стадии кодогенерации:

1. **уровень LLVM IR**: с помощью технологии проходов (LLVM Passes) в модуль добавляется функция шифратор;

2. **уровень `MachineInstr` (ассемблерный уровень)**: тело шифруемой функции обрамляется входным и выходным блоками — они отвечают за шифрование/дешифрование тела функции в рантайме. На этом же уровне происходит извлечение метаданных функции;

3. **уровень `MCInst` (машинные инструкции)**: к формируемым байтам кода прибавляется шифрующая константа.

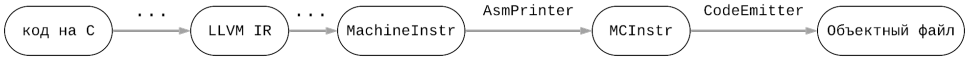

Рис. 6: Схема трансформаций инструкций при компиляции в LLVM.

Рассмотрим подробнее описанные изменения.

3.3. Создание функции-шифратора

Задача шифрующей функции следующая: в цикле пройти по всем элементам переданного массива и прибавить к каждому из них константу. Ее эквивалент на C можно записать следующим образом:

```
1  void encrypt_func(int8_t *array, size_t size, int8_t key) {
2      for (size_t i = 0; i < size; ++i)
3          array[i] += key;
4  }
```

Рис. 7: Код функции-шифратора на C.

Добавление шифрующей функции в модуль осуществляется с помощью *LLVM Passes*: в структуре LLVM регистрируется проход, проверяющий функции модуля. Если встретилась функция, подлежащая обфускации, в модуль добавляется функция-шифратор.

Поскольку проход работает на уровне LLVM IR, доступен класс `llvm::Function` и методы для работы с ним, в частности `llvm::IRBuilder`. Процедура создания функции стандартна и состоит из следующих шагов:

1. Создание объекта класса `llvm::Function` (вызывается метод `Create()` с переданными сигнатурой, типом линковки, именем и родительским модулем).

2. Формирование трех базовых блоков: входной, блок с циклом и выходной. Создать блоки и привязать их к функции можно методом `Create()`.

3. Заполнение блоков соответствующими инструкциями через объект класса `llvm::IRBuilder`.

Добавление функции шифратора на уровне LLVM IR имеет свои преимущества: представление LLVM IR платформо-независимое, поэтому возможно переиспользовать проход для обфускации под отличные от X86 архитектуры. На уровне `MachineInstr` и ниже данное преимущество отсутствует.

3.4. Добавление обрамляющих блоков

Задача входного блока — вызвать функцию-шифратор, передав адрес начала тела функции, ее размер и константу обфускации со знаком минус. Аналогичные шаги выполняет выходной блок и, к тому же, осуществляет возврат из функции.

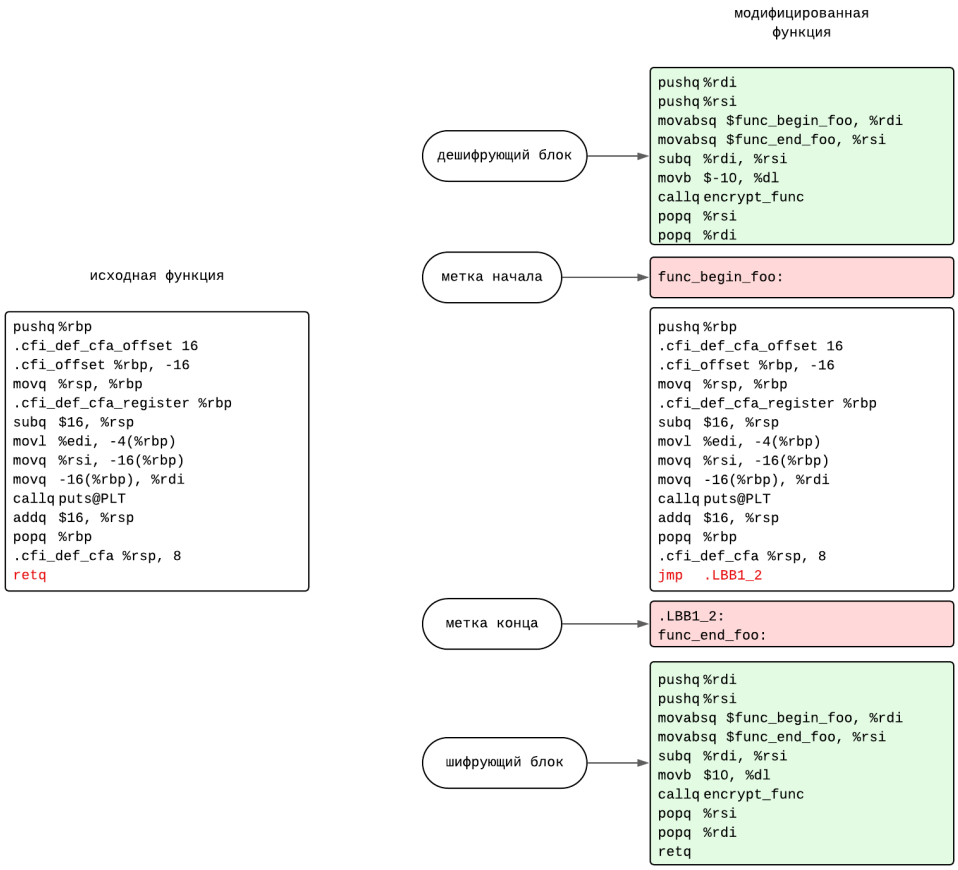

Рис. 8: Тело функции до добавления обрамляющих блоков и после. Код на асемблере x86-64.

Поскольку адрес и размер функции неизвестны на уровне LLVM IR, необходимо добавлять обрамляющие блоки на более низких уровнях, в частности на уровне `MachineInstr`. Создание блоков происходит в классе `AsmPrinter`, содержащем логику генерации ассемблерного кода.

Проблема вычисления размера функции и ее адреса решается следующим образом:

1. внутри функции генерируются метки `func_begin_name`, `func_end_name`;
2. размер вычисляется как `func_end_name - func_begin_name`;
3. адрес передается через метку начала функции `func_begin_name`.

На уровне ассемблерных кодов используются отличные от уровня LLVM IR классы и методы создания инструкций и базовых блоков. За

базовые блоки отвечает класс `llvm::MachineBasicBlock`, за генерацию инструкций — `llvm::BuildMI`. Класс `BuildMI` создает ассемблерные инструкции, поэтому фактически вызов функции шифратора пишется на уровне ассемблера x86-64 в соответствии с ABI архитектуры.

Кроме добавления обрамляющих блоков необходимо изменить поток выполнения внутри функции: заменить все инструкции возврата из функции безусловными переходами на выходной блок.

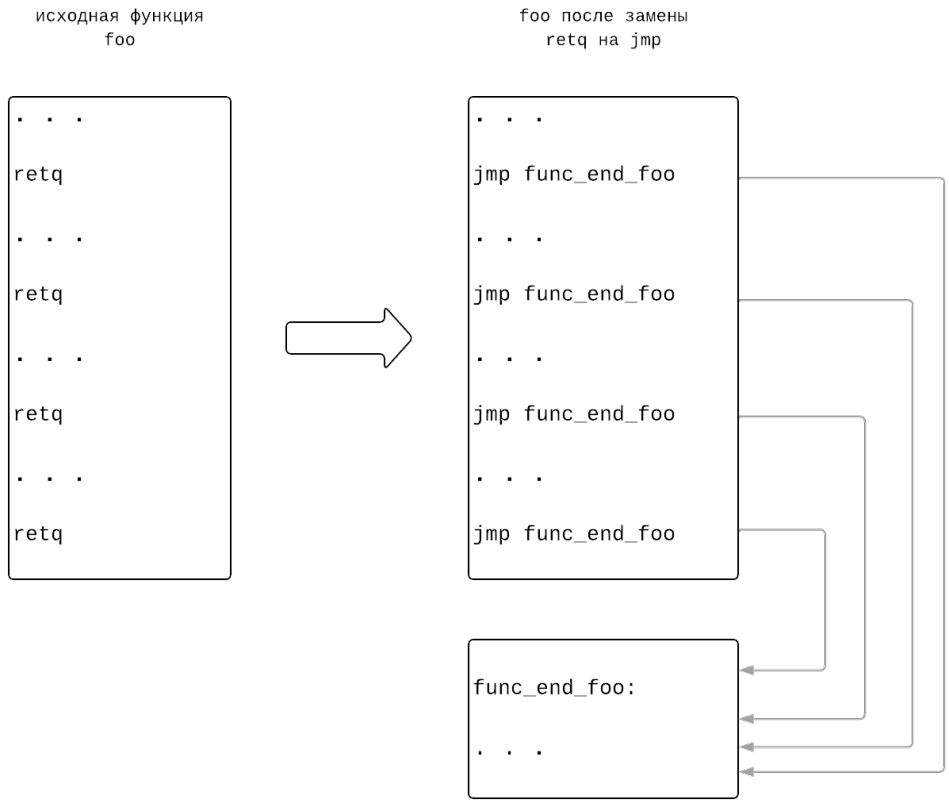

Рис. 9: Схема замены инструкций возврата на безусловный переход на шифрующий выходной блок.

Без этого шага функция после исполнения останется в незашифрованном виде и последующий ее вызов приведет к ошибке.

Также внутри класса `AsmPrinter` через метод `llvm::Function::setSection()` секция обфусцируемой функции устанавливается в `.data`.

3.5. Шифрование тела функции

За генерацию байтов исполняемого кода отвечает класс `MCCodeEmitter`. Он оперирует объектами `MCInst`, которые хранят опкод и операнды инструкции, но уже не имеют информации о родительской функции и ее метаданных.

Соответственно необходимо решить проблему определения инструкций, подлежащих обфускации, и констант. Для этого заводится соответствующая структура, цель которой — сопоставить объекту `MCInst` пару из родительского объекта `MachineInstruction` и константы. Далее достаточно поправить основной метод кодогенерации `encodeInstruction()` и вспомогательный `emitByte()`, чтобы при формировании байта кода учитывалось смещение на константу обфускации.

4. Ограничения

Разработанный прототип метода самомодифицирующегося кода имеет фундаментальное ограничение, препятствующее корректной работе метода для произвольных функций.

4.1. Ограничение на уровне релокаций

Основное ограничение связано с обработкой релокаций в скомпилированном коде. Более конкретно:

- окончательные адреса функций и глобальных переменных определяются на этапе компоновки;
- шифрование применяется на этапе компиляции до обработки релокаций;
- модификация зашифрованных адресов линкером приводит к некорректной работе кода.

В результате метод не может корректно обрабатывать:

- вызовы функций через таблицы виртуальных методов (vtable);
- косвенные вызовы функций;
- обращения к глобальным переменным;
- межмодульные вызовы функций.

4.2. Ограничения на структуру функций

Корректная работа метода возможна для класса достаточно простых функций, удовлетворяющих как минимум следующим критериям:

- отсутствие циклов и ветвлений;
- отсутствие вызовов других функций того же модуля;

- отсутствие обращения к глобальным переменным.

Эти ограничения существенно сужают область применимости прототипа.

Решить описанную проблему возможно путем модификации компоновщика LLVM для обработки релокаций, что потребует:

- изменений в логике обработки релокаций в компоновщике;
- создания новых типов релокационных записей;
- синхронизации между компилятором и компоновщиком.

Заметим, что обработка релокаций и модификация компоновщика выходит за рамки данной работы, исследующей возможности реализации самомодификации на этапе компиляции.

5. Эксперимент

В данном разделе представлено экспериментальное исследование разработанного прототипа метода самомодификации. Задачи эксперимента:

1. Исследовать качество обфускации с использованием инструментов статического анализа.

5.1. Тестовый стенд

При проведении эксперимента было использовано следующее программное обеспечение: LLVM 19.0.0, clang 19.0.0, Ghidra 11.2.1, IDA Free 9.0.

В качестве тестовых программ для оценки качества обфускации были использованы написанные тесты на языке C.

5.1.1. Подготовка тестовых программ

Опишем процесс получения обфусцированных версий кода при проведении эксперимента.

Необходимо собрать LLVM из исходников, после чего будет доступна архитектура x86_64_obf, отвечающая за обфускацию кода. Для применения прохода, добавляющего функцию-шифратор, используется опция компилятора `-fpass-plugin` с указанием пути до прохода. Для указания целевой архитектуры используется опция `--target`.

Полная команда по сборке программы имеет следующий вид:

Листинг 1: Сборка обфусцированного кода с помощью clang

```
$ clang --target=x86_64_obf-unknown-linux-gnu
-fpass-plugin=<path-to-pass> program.c -o obf_program
```

5.2. Оценка качества обфускации

Для оценки качества обфускации, осуществляемой проходом, использовались инструменты статического и динамического анализа Ghidra и IDA. В частности, были использованы реализованные в них дизассемблеры и декомпиляторы.

Схема проведения эксперимента следующая:

1. Получить бинарный файл для обфусцированной и исходной версии программы.

2. Загрузить оба варианта кода в дизассемблер.

3. Проанализировать результат дизассемблирования.

5.2.1. Критерии оценки

Обфускацию можно считать приемлемой, если дизассемблер при статическом анализе не способен восстановить код до обфускации. При этом ожидается, что применение динамического анализа кода позволяет выявить обфусцированный код.

5.2.2. Результаты

Рассмотрим подробно процесс на одной из тестовых программ, в которой вызывается библиотечная функция `printf` для вывода строки и числа. Данная функция удовлетворяет ограничениям, накладываемым на обрабатываемые методом функции, поэтому обфусцированный код работает корректно.

Для указания обфускации конкретной функции перед ее определением помещается комментарий специального вида. При обработке кода компилятором информация из комментария сохранится в метаданных функции. Выбрана константа обфускации равная десяти.

Листинг 1: Тестовая программа, вызывающая `printf`

```
1 /*?
2 {
```

```
    "type": "method",
    "obfuscate":
    {
        "add_const": {
            "type": "int",
            "value": 10
        }
    }
}
?*/
void print_str_num(int a, const char *str){
    printf("Number: %d\nString: %s\n", a, str);
}
```

Проверим, что дизассемблер IDA Free успешно справился с восстановлением функций.

```
.text:0000000000001140 ; int __fastcall print_str_num(int, const char *)
.text:0000000000001140                 public print_str_num
.text:0000000000001140 print_str_num   proc near               ; CODE XREF: main+28↓p
.text:0000000000001140
.text:0000000000001140 var_10          = qword ptr -10h
.text:0000000000001140 var_4           = dword ptr -4
.text:0000000000001140
.text:0000000000001140 ; __unwind {
.text:0000000000001140                 push    rbp
.text:0000000000001141                 mov     rbp, rsp
.text:0000000000001144                 sub     rsp, 10h
.text:0000000000001148                 mov     [rbp+var_4], edi
.text:000000000000114B                 mov     [rbp+var_10], rsi
.text:000000000000114F                 mov     esi, [rbp+var_4]
.text:0000000000001152                 mov     rdx, [rbp+var_10]
.text:0000000000001156                 lea     rdi, format     ; "Number: %d\nString: %s\n"
.text:000000000000115D                 mov     al, 0
.text:000000000000115F                 call    _printf
.text:0000000000001164                 add     rsp, 10h
.text:0000000000001168                 pop     rbp
.text:0000000000001169                 retn
.text:0000000000001169 ; } // starts at 1140
.text:0000000000001169 print_str_num   endp
.text:0000000000001169
```

Рис. 10: Восстановленная дизассемблером IDA Free функция print_str_num.

Видно, что IDA Free правильно распознала тело функции. Более того, она справилась с декомпиляцией функции.

Теперь рассмотрим результат работы дизассемблера IDA Free на обфусцированной программе.

```
1 int __fastcall print_str_num(int a1, const char *a2)
2 {
3   return printf("Number: %d\nString: %s\n", a1, a2);
4 }
```

Рис. 11: Декомпилированная с помощью IDA Free функция print_str_num.

```
.data:0000000000004030                 public print_str_num
.data:0000000000004030 print_str_num:                          ; CODE XREF: main+28↑p
.data:0000000000004030 ; __unwind {
.data:0000000000004030                 push    rax
.data:0000000000004031                 push    rdi
.data:0000000000004032                 push    rsi
.data:0000000000004033                 mov     rdi, offset func_begin_print_str_num
.data:000000000000403D                 mov     rsi, offset func_end_print_str_num
.data:0000000000004047                 sub     rsi, rdi
.data:000000000000404A                 mov     dl, 0F6h
.data:000000000000404C                 call    decrypt_func
.data:0000000000004051                 pop     rsi
.data:0000000000004052                 pop     rdi
.data:0000000000004053                 pop     rax
.data:0000000000004054
.data:0000000000004054 func_begin_print_str_num:               ; DATA XREF: .data:0000000000004033↑o
.data:0000000000004054                                         ; func_end_print_str_num+3↓o
.data:0000000000004054                 pop     rdi
.data:0000000000004055                 push    rdx
.data:0000000000004056                 xchg    eax, ebx
.data:0000000000004057                 out     dx, eax
.data:0000000000004058                 push    rdx
.data:0000000000004058 ; ---------------------------------------------------------------------------
.data:0000000000004059                 db 8Dh, 0F6h, 1Ah, 93h, 87h, 6, 52h
.data:0000000000004060                 dq 9552067F95FA7F93h, 9E99D479752FA5Fh, 909D9C2F20ABA09h
.data:0000000000004078                 dq 0A0AF3671ACE8D52h
.data:0000000000004080 ; ---------------------------------------------------------------------------
.data:0000000000004080                 or      cl, [rdx]
.data:0000000000004082
.data:0000000000004082 ; =============== S U B R O U T I N E =======================================
.data:0000000000004082
.data:0000000000004082
.data:0000000000004082 func_end_print_str_num proc near        ; DATA XREF: .data:000000000000403D↑o
.data:0000000000004082                                         ; func_end_print_str_num+D↓o
.data:0000000000004082                 push    rax
.data:0000000000004083                 push    rdi
.data:0000000000004084                 push    rsi
.data:0000000000004085                 mov     rdi, offset func_begin_print_str_num
.data:000000000000408F                 mov     rsi, offset func_end_print_str_num
.data:0000000000004099                 sub     rsi, rdi
.data:000000000000409C                 mov     dl, 0Ah
.data:000000000000409E                 call    decrypt_func
.data:00000000000040A3                 pop     rsi
.data:00000000000040A4                 pop     rdi
.data:00000000000040A5                 pop     rax
.data:00000000000040A6                 retn
.data:00000000000040A6 ; } // starts at 4030
```

Рис. 12: Восстановленная дизассемблером IDA Free обфусцированная функция print_str_num.

Сперва заметим, что тело функции теперь находится в секции `.data`, в отличие от необфусцированного кода, находящегося в секции `.text`.

IDA верно определила входной и выходной блоки, образующиеся в результате обфускации. Можно вычленить константу обфускации, которая перемещается в регистр DL в инструкции `mov dl, 0Ah`, также видно вызов функции-шифратора decrypt_func.

Тело функции, находящееся между обрамляющими блоками, оказывается зашифрованным, и IDA не может распознать байты кода как

инструкции исходной функции. По этой причине декомпилятор IDA Free не способен восстановить код функции.

Таким образом, статический анализ дизассемблером IDA Free не смог выявить код обфусцированной функции.

Проведем те же действия для инструмента Ghidra.

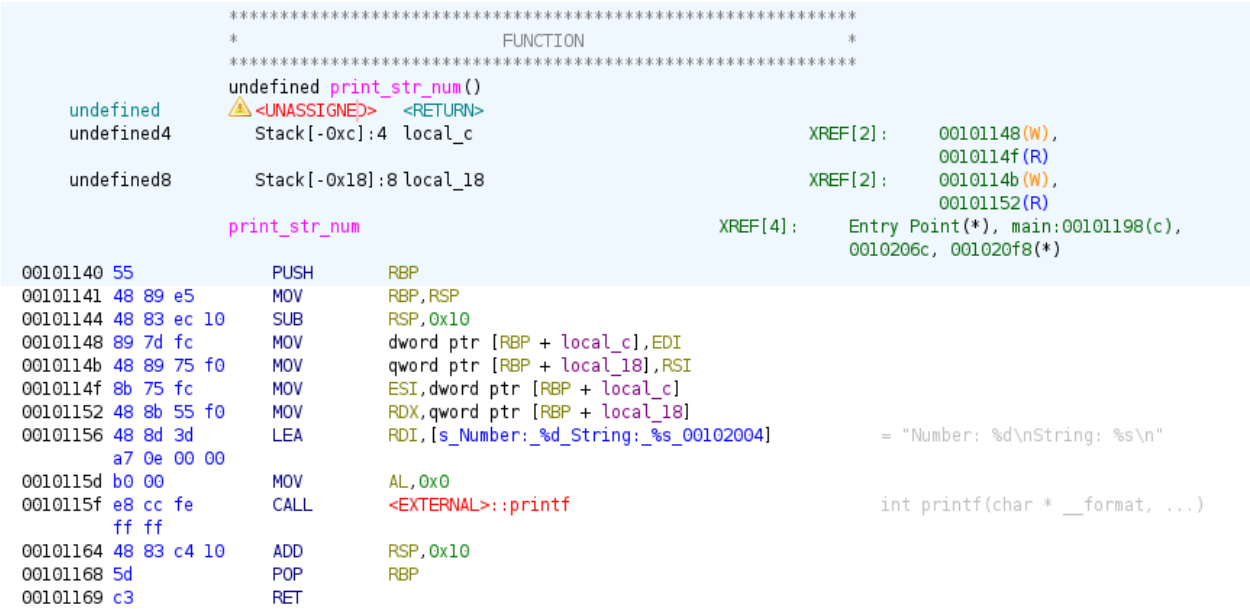

Рис. 13: Восстановленная дизассемблером Ghidra исходная функция print_str_num.

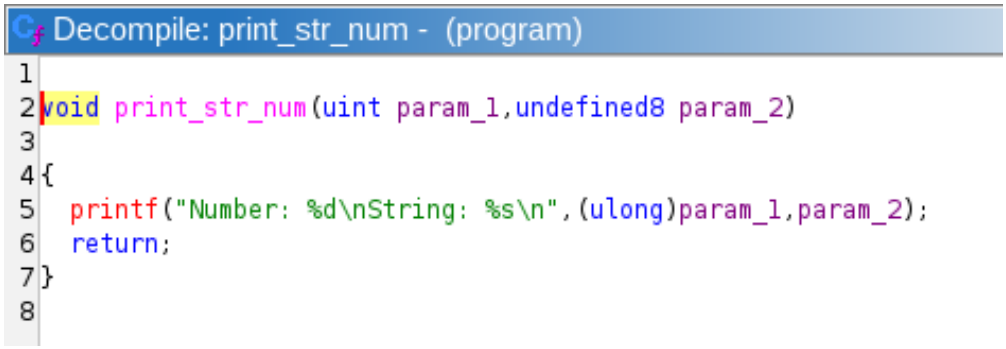

Рис. 14: Восстановленная декомпилятором Ghidra исходная функция print_str_num.

Можно заметить, что дизассемблер и декомпилятор Ghidra также успешно справляются с восстановлением необфусцированной функции.

Рассмотрим результат дизассемблирования обфусцированной функции.

```
                    undefined print_str_num()
     undefined      <UNASSIGNED>   <RETURN>
                    print_str_num                                  XREF[4]:     Entry Point(*), main:00101168(c),
                                                                                0010207c, 00102118(*)
00104030 50             PUSH        RAX
00104031 57             PUSH        RDI
00104032 56             PUSH        RSI
00104033 48 bf 54       MOV         RDI,0x104054
         40 10 00
         00 00 00 00
0010403d 48 be 82       MOV         RSI,func_end_print_str_num
         40 10 00
         00 00 00 00
00104047 48 29 fe       SUB         RSI,RDI
0010404a b2 f6          MOV         DL,0xf6
0010404c e8 2f d1       CALL        decrypt_func                       undefined decrypt_func()
         ff ff
00104051 5e             POP         RSI
00104052 5f             POP         RDI
00104053 58             POP         RAX



                    func_begin_print_str_num
00104054 5f             POP         RDI
00104055 52             PUSH        RDX
00104056 93             XCHG        EAX,EBX
00104057 ef             OUT         DX,EAX
00104058 52             PUSH        RDX
00104059 8d             ??          8Dh
0010405a f6             ??          F6h
0010405b 1a             ??          1Ah
0010405c 93             ??          93h
0010405d 87             ??          87h
0010405e 06             ??          06h
0010405f 52             ??          52h    R
00104060 93             ??          93h
00104061 7f             ??          7Fh    
00104062 fa             ??          FAh
00104063 95             ??          95h
00104064 7f             ??          7Fh    
00104065 06             ??          06h
00104066 52             ??          52h    R
00104067 95             ??          95h
00104068 5f             ??          5Fh    _
00104069 fa             ??          FAh
0010406a 52             ??          52h    R
0010406b 97             ??          97h
0010406c 47             ??          47h    G
0010406d 9d             ??          9Dh
0010406e e9             ??          E9h
0010406f 09             ??          09h
00104070 09             ??          09h
00104071 ba             ??          BAh
00104072 0a             ??          0Ah
00104073 f2             ??          F2h
```

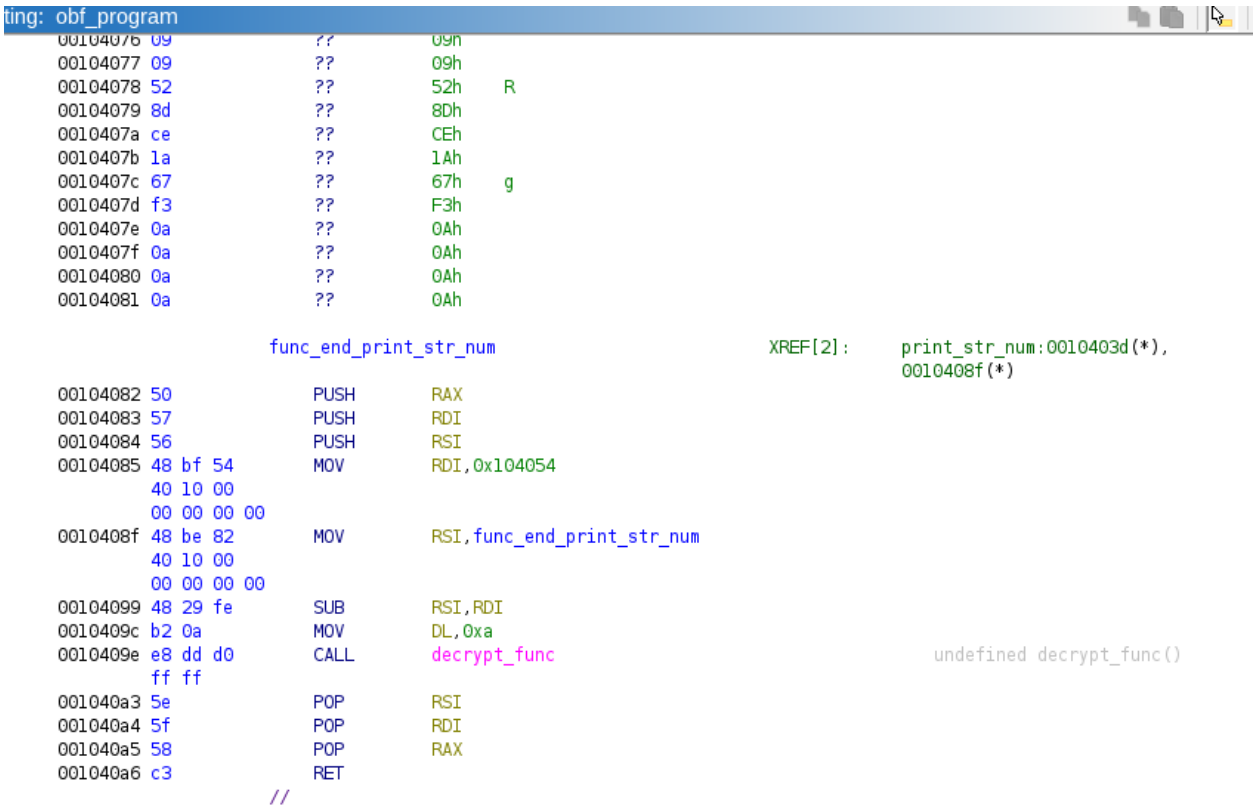

Рис. 15: Восстановленная дизассемблером Ghidra обфусцированная функция print_str_num.

Результаты дизассемблирования инструментом Ghidra аналогичны полученным при работе с IDA Free.

В отличие от IDA Free Ghidra смогла совершить декомпиляцию обфусцированного кода, тело функции ожидаемо некорректно.

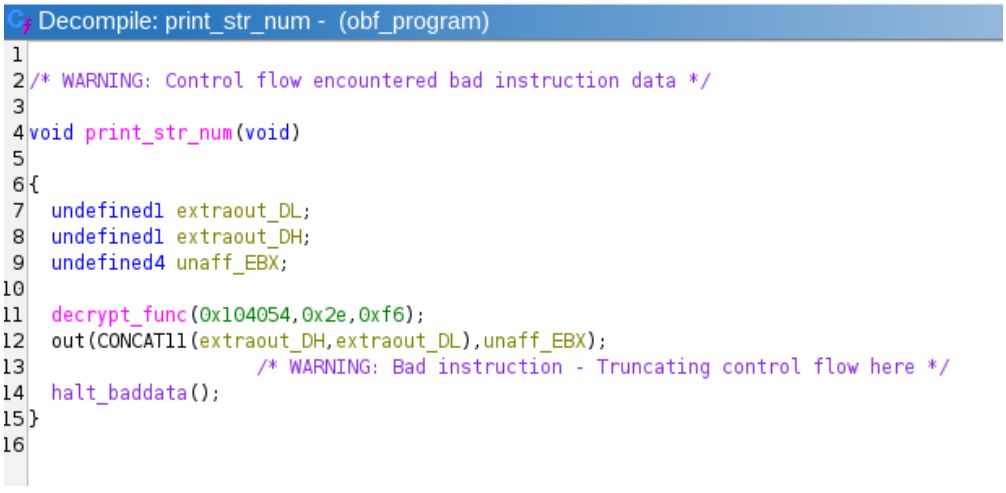

Рис. 16: Восстановленная декомпилятором Ghidra обфусцированная функция print_str_num.

5.3. Выводы

По результатам проведенного эксперимента можно сделать вывод, что реализованный прототип метода самомодификации обеспечивает приемлемый уровень обфускации, так как рассмотренные декомпиляторы и дизассемблеры оказались неспособными восстановить исходную функцию.

Заключение

В рамках данной учебной практики были выполнены следующие **задачи**:

1. проведен обзор существующих решений и используемых технологий;
2. реализован прототип метода самомодификации кода на основе инфраструктуры LLVM;
3. выявлены ключевые ограничения, возникающие при реализации самомодификации на этапе компиляции;
4. проведено экспериментальное исследование полученного прототипа с помощью инструментов статического анализа.

Исходный код проекта закрыт.

Список литературы

[1] Aucsmith David. Tamper resistant software: an implementation // Information Hiding / Ed. by Ross Anderson. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1996. — P. 317–333.

[2] Christen P. System Metamodeling of Open-Ended Evolution Implemented with Self-Modifying Code // Complex Systems. — 2024. — Vol. 32, no. 4. — P. 353–380.

[3] Collberg Christian, Thomborson Clark, Low Douglas. A Taxonomy of Obfuscating Transformations // http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial. — 1997. — 01.

[4] Exploiting Self-Modification Mechanism for Program Protection / Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, Ken-ichi Matsumoto. — 2003. — 01. — P. 170–.

[5] Henchiri Mourad M. H., Wani Sharyar. Disassembly and Detection of Self-Modifying Malicious Software // International Journal of Digital Information and Wireless Communications. — 2021. — Oct. — Vol. 11, no. 4. — P. 64+. — URL: https://link.gale.com/apps/doc/A757045570/AONE (дата обращения: 2025-05-09).

[6] Morse Gregory, Alqaradaghi Midya, Kozsik Tamás. Control flow obfuscation with irreducible loops and self-modifying code // Publicationes Mathematicae Debrecen. — 2022. — 12. — Vol. 100. — P. 617–637.

[7] Schrittwieser Sebastian, Katzenbeisser Stefan. Code Obfuscation against Static and Dynamic Reverse Engineering. — 2011. — 05. — P. 270–284.

[8] Software Protection Through Dynamic Code Mutation / Matias Madou, Bertrand Anckaert, Patrick Moseley et al. // Information Security Applications / Ed. by Joo-Seok Song, Taekyoung Kwon, Moti Yung. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2006. — P. 194–206.

[9] Towards Tamper Resistant Code Encryption: Practice and Experience / Jan Cappaert, Bart Preneel, Bertrand Anckaert et al. — 2008. — 04. — P. 86–100.

[10] Waddell Heidi. Achieving Obfuscation Through Self-Modifying Code: A Theoretical Model : Senior Honors Theses : 935 / Heidi Waddell ; Liberty University. — 2020. — URL: https://digitalcommons.liberty.edu/honors/935 (дата обращения: 2025-05-09).

[11] К.А. Дугушова. Защита программного кода путём встраивания мёртвого кода и данных. — URL: https://se.math.spbu.ru/thesis/texts/Dugushova_Ksenija_Andreevna_Autumn_practice_2nd_year_2023_text.pdf (дата обращения: 14 марта 2025 г.).

[12] Ковязин В.А., Пинигина О.И., Паюсова Т.И. Исследование возможностей самомодифицирующегося кода в области информационной безопасности. — Тюменский государственный университет, г. Тюмень. — 2021. — УДК 004.49.

[13] Т. Зекашева А. Обфускация глобальных переменных. — URL: https://se.math.spbu.ru/thesis/texts/Zekasheva_Anastasija_Temirbekovna_Spring_practice_2nd_year_2024_text.pdf (дата обращения: 14 марта 2025 г.).

[14] Ф. Немакин Н. Реализация методов обфускации строковых литералов и базовых блоков в LLVM IR. — URL: https://se.math.spbu.ru/thesis/texts/Nemakin_Nikita_Antonovich_Autumn_practice_3rd_year_2024_text.pdf (дата обращения: 14 марта 2025 г.).