Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
(ФГБОУ ВО «АмГУ»)

Институт компьютерных и инженерных наук
Кафедра информационных и управляющих систем Направление подготовки 09.04.04 – Программная инженерия
Направленность (профиль) образовательной программы Управление разработкой программного обеспечения

ДОПУСТИТЬ К ЗАЩИТЕ
Зав. кафедрой
_____А.В. Бушманов
«___» _____2025 г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему: Разработка приложения для защиты программного кода

Исполнитель
студент группы 3105-ом1 _____ (подпись, дата) В.А. Степаненко

Руководитель
доцент, канд. техн. наук _____ (подпись, дата) Л.В. Никифорова

Руководитель научного содержания программы магистратуры
профессор, доктор техн. наук _____ (подпись, дата) И.Е. Ерёмин

Нормоконтроль
инженер кафедры _____ (подпись, дата) В.Н. Адаменко

Рецензент
доцент, канд. техн. наук _____ (подпись, дата) Т.В. Труфанова

Благовещенск, 2025

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
(ФГБОУ ВО «АмГУ»)

Институт компьютерных и инженерных наук
Кафедра информационных и управляющих систем

УТВЕРЖДАЮ
Зав. кафедрой
_____А.В. Бушманов
«___» _____2025 г.

З А Д А Н И Е

К магистерской диссертации студента группы 3105-ом1

Степаненко Владислава Андреевича

1. Тема магистерской диссертациии:

Разработка приложения для защиты программного кода

(Утверждено приказом от 06.03.2025 № 609-уч)

2. Срок сдачи студентом законченной работы: 10.06.2025

3. Исходные данные к магистерской диссертации: обратная разработка, методы обфускации, интернет ресурсы, учебная литература

4. Содержание магистерской диссертации (перечень подлежащих разработке вопросов): Обзор методов обфускации, проектирование алгоритма решения задачи, разработка приложения

5. Перечень материалов приложения (наличие чертежей, таблиц, графиков, схем, программных продуктов, иллюстративного материала и т.п.): нет

6. Рецензент магистерской диссертации: Труфанова Т.В., доцент, канд. техн. наук

7. Дата выдачи задания: 03.02.2025

Руководитель выпускной квалификационной работы:

Л.В. Никифорова, доцент, канд. техн. наук

(фамилия, имя, отчество, должность, уч.степень, уч.звание)

Задание принял к исполнению

РЕФЕРАТ

Магистерская диссертация содержит 75 страниц, 22 рисунка, 50 источников, 7 таблиц

ОБФУСКАЦИЯ, ЗАЩИТА ПО, ОБРАТНАЯ РАЗРАБОТКА, IL-КОД, .NET

Целью данной работы является разработка программного инструмента для защиты кода .NET-приложений путем обфускации без изменения его функциональности и производительности.

Разрабатываемое приложение обеспечивает защиту промежуточного IL-кода от реверс-инжиниринга, затрудняя анализ и декомпиляцию с помощью стандартных инструментов. В системе реализованы модули лексической обфускации, обфускации хранения и потока управления, а также предоставлен графический интерфейс, упрощающий взаимодействие с кодом для разработчиков.

Разработка программного средства включала следующие этапы:

- анализ существующих методов защиты IL-кода;
- формулирование требований к архитектуре и функциональности;
- проектирование расширяемой модульной архитектуры;
- реализация основных методов обфускации;
- создание графического интерфейса пользователя;
- тестирование и оценка эффективности приложения.

Предложенное решение соответствует современным требованиям к защите программных продуктов на платформе .NET и может применяться как в учебных, так и в прикладных целях для повышения устойчивости приложений к анализу и модификации.

СОДЕРЖАНИЕ

Введение 6
1 Проблема защиты программного кода .NET от реверс инжиниринга 8
1.1 Реверс инжиниринг и декомпиляция кода 8
1.1.1 Понятие обратной разработки 8
1.1.2 Проблемы защиты программного кода 9
1.1.3 Уязвимость языков .NET к обратной разработке 11
1.2 Обфускация как метод защиты 13
1.2.1 Понятие обфускации 13
1.2.2 Лексическая обфускация 14
1.2.3 Обфускация хранения 15
1.2.4 Обфускация потока управления 17
1.3 Существующие инструменты обфускации 20
1.4 Предлагаемое решение 22
2 Проектирование приложения для защиты IL-кода 26
2.1 Требования к ПО 26
2.1.1 Функциональные требования 26
2.1.2 Нефункциональные требования 28
2.2 Выбор средств разработки 29
2.2.1 Выбор модели жизненного цикла разработки 29
2.2.2 Выбор инструментов разработки 31
2.3 Синтез эффективных алгоритмов обфускации IL-кода 36
2.3.1 Алгоритм лексической обфускации 37
2.3.2 Алгоритм обфускации хранения 37
2.3.3 Алгоритм обфускации потока управления 39
2.4 Архитектура Simple IL-fuscator 43
3 Реализация и тестирование приложения 46

3.1 Организация входных и выходных данных 46
3.2 Реализация лексической обфускации 47
3.3 Реализация обфускации хранения 49
3.4 Реализация обфускации потока управления 52
3.5 Реализация пользовательского интерфейса 55
3.6 Тестирование приложения 60
Заключение 66
Библиографические ссылки 68
Библиографический список 70

ВВЕДЕНИЕ

Современные программные продукты, особенно ориентированные на платформу .NET, широко используются как в корпоративной, так и в потребительской среде. Одним из преимуществ данной платформы является использование промежуточного языка (IL – Intermediate Language), который позволяет реализовать кроссплатформенные решения и упростить переносимость кода. Однако эта же особенность становится серьёзным недостатком с точки зрения информационной безопасности, поскольку промежуточный код легко поддаётся декомпиляции. Это делает .NET-приложения уязвимыми для реверс-инжиниринга, кражи интеллектуальной собственности, внедрения вредоносного кода и иных форм компрометации.

На сегодняшний день существует ряд коммерческих и открытых решений для обфускации .NET-кода, однако многие из них имеют закрытую архитектуру, ограниченный функционал или высокую стоимость. Кроме того, большинство таких инструментов ориентированы на статическую обфускацию и не предусматривают гибкого расширения, что ограничивает возможности адаптации под конкретные требования безопасной разработки.

Актуальность исследования обусловлена необходимостью создания доступного и расширяемого инструмента, позволяющего защищать .NET-приложения на уровне промежуточного кода. В условиях активного использования платформы .NET в государственных, корпоративных и научных проектах защита программных решений становится приоритетной задачей. Разработка инструмента, обеспечивающего обфускацию IL-кода без ущерба для производительности и корректности выполнения, представляет собой важный шаг в сторону повышения безопасности программного обеспечения.

Практическая значимость работы заключается в создании прикладного инструмента, который может быть использован специалистами в области безопасной разработки, тестирования, аудита безопасности, а также студентами и преподавателями в учебных целях. Открытая архитектура приложения, ориентированная на

модульность, позволяет адаптировать его к различным задачам и требованиям в сфере защиты кода.

Предметом исследования в данной работе является промежуточный код .NET-приложений (IL-код) как объект защиты от анализа и декомпиляции, а также методы его трансформации с целью затруднения реверс-инжиниринга.

Целью настоящей магистерской диссертации является разработка программного инструмента для защиты кода .NET-приложений путем обфускации без изменения его функциональности и эффективности.

Для достижения поставленной цели необходимо решить следующие задачи:

– проанализировать существующие методы и подходы к защите IL-кода;

– сформулировать требования к программному обеспечению, обеспечивающему защиту il-кода, включая функциональные и нефункциональные характеристики;

– разработать архитектуру приложения с возможностью расширения модулей обфускации;

– реализовать модули: лексической обфускации, обфускации хранения, обфускации потока управления;

– разработать графический интерфейс пользователя для обеспечения удобной работы с IL-кодом;

– провести тестирование реализованного программного средства;

– оценить его эффективность.

Данная работа направлена на решение прикладной задачи, связанной с обеспечением защиты кода приложений, разрабатываемых под платформу .NET, с использованием современных подходов к обфускации. Результаты исследования могут применяться как в образовательной среде, так и в профессиональной сфере разработки безопасного программного обеспечения.

1 ПРОБЛЕМА ЗАЩИТЫ ПРОГРАММНОГО КОДА .NET ОТ РЕВЕРС ИНЖИНИРИНГА

1.1 Реверс инжиниринг и декомпиляция кода

Процессы обратного проектирования и декомпиляции программного кода являются двусторонними технологиями, применяемыми как в конструктивных целях для анализа и оптимизации ПО, так и в деструктивных намерениях. В условиях современной цифровизации общества данные техники приобретают всё большую распространённость и совершенствуются, порождая значительные риски для защиты информации, авторских прав на программные решения и надёжности функционирования IT-систем. Данный раздел посвящён изучению фундаментальных принципов декомпиляции, анализу её потенциала и связанных с ней угроз, а также демонстрации практических случаев, подтверждающих важность рассматриваемой проблематики.

1.1.1 Понятие обратной разработки

Обратное проектирование представляет собой методологию изучения программных решений, направленную на выявление их архитектуры, функциональности и механизмов работы при отсутствии доступа к первоначальному коду. Данный подход включает множество направлений: исследование исполняемых файлов, информационных структур, коммуникационных протоколов, системной архитектуры, анализ сетевого взаимодействия, а также мониторинг поведения приложений во время исполнения. В экосистеме .NET обратное проектирование преимущественно сосредоточено на изучении промежуточного IL-представления, анализе метаинформации модулей, применении средств отладки и трассировки.

Декомпиляция является специализированным направлением обратного проектирования, в рамках которого исполняемый или промежуточный код (например, байт-код) трансформируется в высокоуровневое представление исходного текста. Данная технология обеспечивает возможность восстановления

программной логики даже в случае недоступности оригинальных исходников. Декомпиляция приобретает особую значимость при работе с языками программирования, транслируемыми в промежуточные представления, включая Java (байт-код) и C# (IL-код платформы .NET). В отличие от дизассемблирования, которое конвертирует машинные инструкции в ассемблерный код, декомпиляция нацелена на восстановление структуры программы, максимально приближенной к первоначальной, например, идентификаторы, процедуры и управляющие блоки.

Главная проблема процесса декомпиляции состоит в утрате данных на стадии компиляции – исчезают наименования идентификаторов, аннотации разработчика, происходит оптимизация кодовой структуры. Тем не менее, современные аналитические платформы (Ghidra, IDA Pro, Jadx, ILSpy) применяют алгоритмы эвристического анализа и исследование потоков данных для максимального восстановления утраченной информации.

1.1.2 Проблемы защиты программного кода

Технологии обратного проектирования и декомпиляции создают существенные риски для информационной безопасности и охраны авторских прав на программные разработки. Первостепенной угрозой выступает компрометация коммерческих секретов. Организации, создающие инновационные алгоритмы, к примеру, для машинного обучения или криптографической защиты, сталкиваются с опасностью извлечения их собственных технологий посредством инструментов анализа кода и последующего использования конкурирующими структурами. Подобные инциденты влекут за собой значительный экономический ущерб.

Дополнительной угрозой служит выявление и использование программных уязвимостей. Изучение исполняемых модулей дает возможность злоумышленникам обнаруживать недочеты в коде и создавать эксплойты, угрожающие как отдельным приложениям, так и комплексным IT-системам.

Важно отметить направление исследования вредоносного программного обеспечения. Специалисты по информационной безопасности применяют

декомпиляцию для понимания принципов функционирования malware и создания защитных решений, однако идентичные техники используются киберпреступниками для изучения встроенных защитных механизмов легального ПО с целью их последующего преодоления.

Практические случаи демонстрируют реальность данных угроз. Показательным примером служит инцидент с мобильным банковским приложением одного из европейских банков в 2023 году, когда исследователи безопасности с помощью инструмента ILSpy обнаружили в Xamarin-приложении жестко закодированные криптографические ключи и уязвимости в логике аутентификации. Аналогично, в рамках соревнований по информационной безопасности, таких как DEF CON CTF, участники регулярно используют Ghidra и Radare2 для анализа защищенных бинарных файлов и восстановления алгоритмов шифрования. Известен случай 2022 года, когда с помощью IDA Pro была взломана система защиты популярного коммерческого архиватора, что привело к массовому распространению пиратских версий.

Указанные риски усугубляются структурными особенностями защиты программного кода. Открытость промежуточного IL-представления в .NET обеспечивает доступность для анализа структуры классов, методов и атрибутов. Стандартные модули обычно лишены встроенных средств противодействия анализу, а восстановленный и модифицированный код без труда подвергается повторной компиляции. Традиционные подходы, включающие лицензионные проверки или контроль целостности сборок, без комплексной защиты демонстрируют недостаточную эффективность.

Данная ситуация обуславливает потребность в применении комплексных защитных решений, включая обфускацию кода, криптографическую защиту модулей, противодействие отладке и внедрение anti-debugging механизмов. Безусловно, ни одно из перечисленных решений не гарантирует абсолютной защиты, однако их комплексное применение существенно повышает сложность процесса обратного проектирования и минимизирует вероятность успешной компрометации.

1.1.3 Уязвимость языков .NET к обратной разработке

Языки высокого уровня, такие как C#, VB.NET и F#, использующие платформу .NET, подвергаются трехступенчатому процессу формирования исполняемого кода, что представляет определенные уязвимости к реверс-инжинирингу. На первом этапе исходный код программы преобразуется в промежуточный язык (IL-код) [1]. Затем выполняется его оптимизация, в ходе которой устраняются операции, не влияющие на конечный результат работы программы, такие как неиспользуемые переменные и их модификации. На завершающем этапе оптимизированный IL-код преобразуется в финальный исполняемый код. Схема компиляции языка программирования C# до машинного кода представлена на рисунке 1.

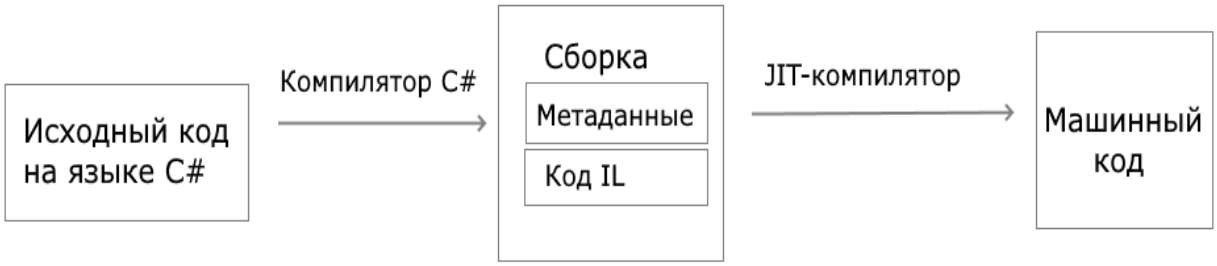

Рисунок 1 – Схема компиляции C#

Такая многоступенчатая компиляция значительно снижает эффективность обфускации исходного кода. Алгоритмы оптимизации компилятора устраняют многие «запутывающие» изменения, внесенные в исходный код, нивелируя усилия по усложнению анализа кода. Таким образом, традиционные методы обфускации на уровне исходного кода оказываются неэффективными для языков платформы .NET.

Для повышения уровня защиты программного обеспечения, разрабатываемого на языках .NET, целесообразно применять обфускацию не на уровне исходного кода, а на уровне IL-кода и финального исполняемого кода. Это позволяет сохранить запутывающие изменения и значительно усложнить задачу реверс-

инжиниринга для потенциальных злоумышленников, обеспечивая более высокий уровень безопасности и защиты интеллектуальной собственности.

Кроме того, в IL-коде сохраняются имена методов, классов и переменных, а также структура классов, сигнатуры методов и логика управления потоком выполнения. Это значительно облегчает декомпиляцию и позволяет получить близкий к исходному код на высокоуровневом языке.

IL-код сохраняет множество деталей исходного кода, что упрощает анализ и декомпиляцию. Рассмотрим пример IL-кода на рисунке 2:

```
// =============== CLASS MEMBERS DECLARATION ===================

.class public auto ansi beforefieldinit Program
       extends [System.Runtime]System.Object
{
  .method public hidebysig static void  Main() cil managed
  {
    .entrypoint
    // Code size       44 (0x2c)
    .maxstack  2
    .locals init (int32 V_0,
             int32 V_1,
             int32 V_2)
    IL_0000:  nop
    IL_0001:  ldc.i4.5
    IL_0002:  stloc.0
    IL_0003:  ldc.i4.s   10
    IL_0005:  stloc.1
    IL_0006:  ldloc.0
    IL_0007:  ldloc.1
    IL_0008:  call       int32 Program::Add(int32,
                                            int32)
    IL_000d:  stloc.2
    IL_000e:  ldstr      "Sum: "
    IL_0013:  ldloca.s   V_2
    IL_0015:  call       instance string [System.Runtime]System.Int32::ToString()
    IL_001a:  call       string [System.Runtime]System.String::Concat(string,
                                                                      string)
    IL_001f:  call       void [System.Console]System.Console::WriteLine(string)
    IL_0024:  nop
    IL_0025:  call       string [System.Console]System.Console::ReadLine()
    IL_002a:  pop
    IL_002b:  ret
  } // end of method Program::Main
```

Рисунок 2 – Пример IL-кода

В этом примере метод Main демонстрирует последовательность операций, выполняемых для вычисления суммы двух чисел и вывода результата на консоль. IL-код отражает:

– инициализацию локальных переменных (V_0, V_1, V_2);

– вызов пользовательского метода Program::Add для сложения чисел;

– форматирование результата в строку и его вывод с помощью метода System.Console::WriteLine.

Пример иллюстрирует, насколько легко IL-код можно интерпретировать и декомпилировать. Его основные уязвимости:

– все методы, переменные и вызовы отражены в явной форме;

– даже без исходного кода можно восстановить логику программы;

– il-код можно исследовать и модифицировать с использованием инструментов, таких как ilspy, dotpeek или dnspy.

Чтобы повысить защищенность промежуточного кода, разработчики могут применить методы, которые включают:

- изменение имён переменных и методов на нечитаемые строки;
- запутывание потока управления, делая логику менее очевидной;
- шифрование строковых литералов.

Эти методы повышают сложность анализа IL-кода и защищают интеллектуальную собственность. Такой подход особенно эффективен для платформы .NET, где обфускация на уровне исходного кода не приносит существенных результатов из-за особенностей компиляции [2].

1.2 Обфускация как метод защиты

1.2.1 Понятие обфускации

Обфускация программного кода представляет собой процесс изменения структуры и содержания исходного кода с целью усложнения его понимания, анализа, а также предотвращения несанкционированного использования и изменения. Основная цель обфускации заключается в том, чтобы сделать код менее читаемым для человека, при этом сохраняя его работоспособность и производительность [3].

К основным видам обфускации относятся следующие:

– лексическая обфускация, благодаря которой происходит изменение внешнего вида программного кода для усложнения его восприятия и анализа;

– обфускация хранения, которая изменяет типы и виды хранения, их замена на пользовательские типы (поток данных в программе становится неочевидным);

– обфускация потока управления, что изменяет и запутывает логические структуры, добавляет запутывающие блоки, которые не влияют на результат выполнения программы.

В ходе написания магистерской диссертации рассмотрены все вышеперечисленные методы обфускации.

1.2.2 Лексическая обфускация

Лексическая обфускация – это один из ключевых методов защиты исходного кода, направленный на усложнение его чтения и анализа без изменения функциональности программы. Этот метод включает в себя изменение внешнего вида кода, удаление комментариев и изменение форматирования текста, таких как удаление пробельных символов, табуляций и символов перехода на новую строку.

Основным приемом лексической обфускации является изменение имен идентификаторов. Переменные, функции, классы и другие элементы программы получают бессмысленные, случайные имена вместо осмысленных, описательных названий. Например, переменная price может быть переименована в «G9$2flvvg2ln» Это значительно усложняет понимание кода, так как теряется связь между именем и его назначением. На рисунке 3 изображен пример проведенной лексической обфускации.

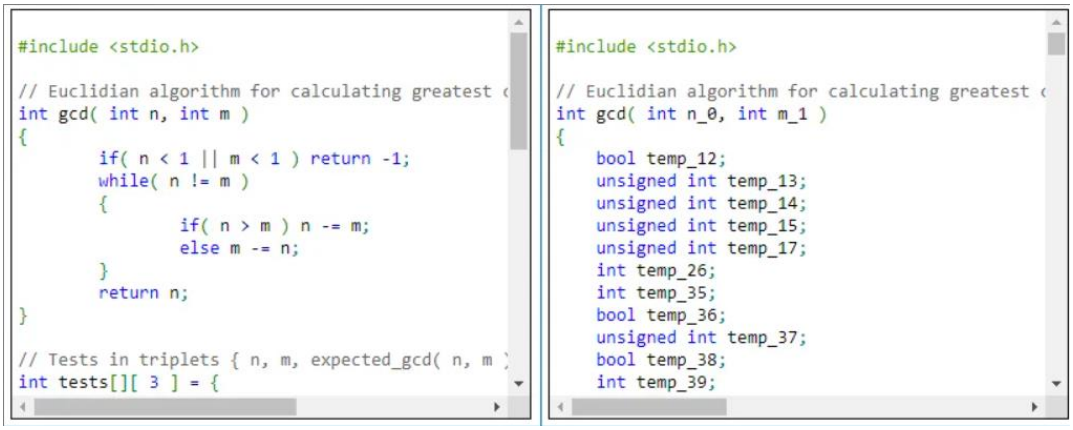

Рисунок 3 – Пример лексической обфускации

Хотя функциональность программы остается неизменной, её понимание становится значительно сложнее.

Дополнительной техникой лексического запутывания выступает внедрение избыточного программного кода. Данные конструкции не несут функциональной нагрузки и не оказывают воздействия на конечный результат выполнения приложения, однако расширяют объем кодовой базы и создают дополнительные пути исполнения. В качестве примера можно привести условные конструкции или итеративные блоки, которые никогда не достигаются в процессе выполнения, но визуально усложняют структуру программы и повышают трудозатраты на ее исследование.

В современных платформах обфускации применяются автоматизированные алгоритмы лексического преобразования, которые существенно снижают временные затраты разработчиков на реализацию защитных механизмов. Подобные решения осуществляют анализ первоначального кода, применяют к нему предварительно настроенные техники запутывания и формируют защищенную версию с минимальным человеческим вмешательством.

Лексическое запутывание представляет собой ключевой компонент в многоуровневой системе защиты программных продуктов. При интеграции с альтернативными защитными подходами, включая криптографическое шифрование и механизмы контроля доступа, данная технология обеспечивает надежный уровень безопасности и создает значительные препятствия для злоумышленников при попытках обратного проектирования и декомпиляции программного обеспечения.

1.2.3 Обфускация хранения

Обфускация хранения является одной из сложных, но эффективных техник защиты программного кода, направленной на усложнение анализа и модификации данных, которые используются внутри программы. В отличие от лексической обфускации, которая фокусируется на изменении видимой части кода (имена переменных, методов, классов), обфускация хранения направлена на

преобразование способов хранения и использования данных. Это делает статический анализ программы и её данных значительно сложнее для реверс-инженеров.

Основные принципы обфускации хранения:

– изменение структуры хранения данных. Прямое хранение данных заменяется на процедурное, например, строковые значения могут быть закодированы и восстановлены только в процессе выполнения программы;

– использование нестандартных типов данных. Вместо привычных и ожидаемых типов данных (например, int, string), используются собственные, кастомизированные типы или структуры, что усложняет анализ кода и его структуры;

– динамическое получение данных. Хранение критически важных данных, таких как ключи шифрования или строки, может быть замаскировано через вызовы функций, которые возвращают эти данные только при соблюдении определённых условий. Это уменьшает вероятность того, что реверс-инженер сможет легко вычленить важную информацию из кода;

– разделение данных на части. Важные данные могут быть разбиты на несколько частей и восстановлены только в момент их использования, что делает задачу анализа и изменения кода значительно сложнее.

Программа для лексической обфускации IL-кода, о которой шла речь ранее, должна быть дополнена механизмами обфускации хранения, что значительно повысит уровень защиты кода от реверс-инжиниринга.

Алгоритм обфускации хранения с генерацией идентификаторов позволяет значительно усложнить анализ и обратное проектирование кода без значительных затрат вычислительных ресурсов. Использование пользовательских правил для генерации новых имен переменных и функций эффективно маскирует логику программы, сохраняя её работоспособность.

Общий процесс обфускации хранения показан в виде UML-диаграммы последовательности на рисунке ниже:

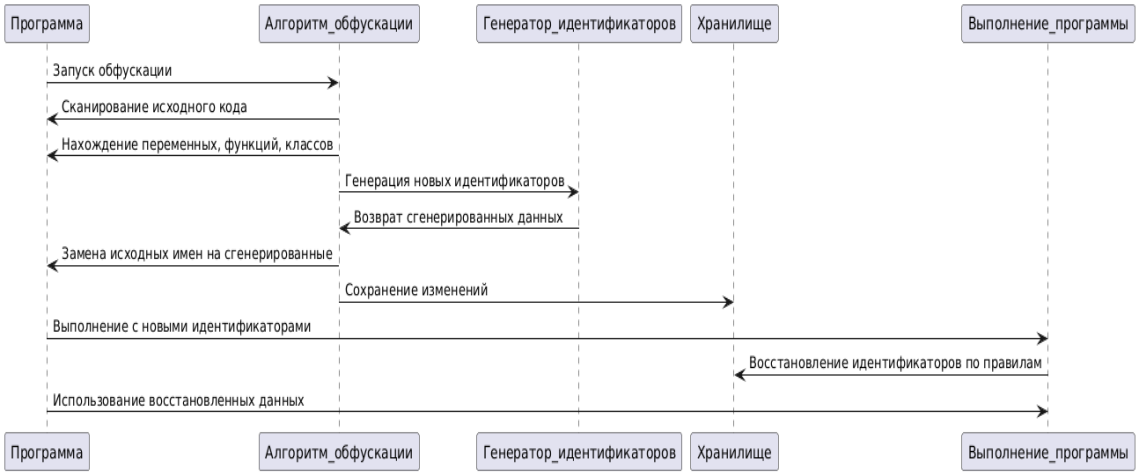

Рисунок 4 – Диаграмма последовательности обфускации хранения

Объяснение диаграммы:

– программа передает управление алгоритму обфускации для анализа и модификации кода;

– алгоритм обфускации находит элементы кода, которые подлежат изменению (переменные, функции, классы);

– генератор создает новые уникальные идентификаторы по заранее заданным правилам, которые заменят старые имена;

– старые идентификаторы заменяются на сгенерированные имена, что усложняет понимание кода;

– модифицированные элементы программы сохраняются, и они будут использоваться при выполнении;

– в процессе выполнения программы возможны вызовы процедур восстановления идентификаторов для корректной работы.

Такой подход снижает риск несанкционированного использования или модификации кода.

1.2.4 Обфускация потока управления

Обфускация потока управления (Control Flow Obfuscation) является одной из ключевых техник защиты программного кода от анализа, реверс-инжиниринга

и атак. Данная методика направлена на усложнение структуры исполнения программы таким образом, чтобы затруднить понимание её логики, но при этом сохранить корректность выполнения. В отличие от традиционной обфускации, которая работает на уровне отдельных инструкций или данных, обфускация потока управления модифицирует саму последовательность исполнения команд, создавая запутанную и нелогичную структуру кода.

Существует несколько популярных методов обфускации потока управления, каждый из которых имеет свои особенности, достоинства и недостатки. Основные техники включают в себя:

Вставка ложных путей исполнения:

Этот метод основан на добавлении в код псевдологических ветвлений и блоков, которые никогда не выполняются при нормальной работе программы, но усложняют анализ её логики. Такие ложные пути могут быть замаскированы под реальные условия, увеличивая сложность статического анализа.

Достоинства:

– затрудняет анализ графа потока управления (Control Flow Graph, CFG);

– осложняет автоматическую деобфускацию и анализ с помощью инструментов реверс-инжиниринга.

Недостатки:

– может увеличивать размер бинарного кода;

– при некачественной реализации может замедлять выполнение программы.

Использование непрямых переходов:

Данный метод включает замену прямых переходов (jump, call) на непрямые (indirect jumps), управляемые переменными или сложными вычислениями. Это делает невозможным простое статическое определение порядка выполнения команд.

Достоинства:

– осложняет построение графа потока управления;

– усложняет анализ с помощью дизассемблеров.

Недостатки:

– может ухудшить предсказуемость выполнения кода процессором, снижая производительность;

– некоторые отладчики способны отслеживать изменения в адресах переходов.

Использование таблиц переходов:

При данной технике управление передаётся через таблицы переходов, хранящие адреса функций или кодовых блоков. Вместо явных условных конструкций (if-else, switch-case) программа определяет, какой блок выполнить следующим, обращаясь к таблице с заранее подготовленными адресами.

Достоинства:

- усложняет анализ за счёт размытости связей между операторами;
- позволяет избежать предсказуемых ветвлений, что делает анализ сложнее.

Недостатки:

- требует дополнительной памяти для хранения таблиц;
- может быть частично устранена динамическим анализом.

Вставка ненужных вычислений и состояний:

Данный метод заключается в добавлении в код бесполезных математических или логических операций, а также фиктивных состояний программы, которые усложняют анализ логики выполнения. Это может включать избыточные переменные, фиктивные циклы или вызовы функций, не влияющие на результат.

Достоинства:

- усложняет анализ кода человеком;
- может быть эффективен против автоматизированных систем реверс-инжиниринга.

Недостатки:

- может увеличивать потребление ресурсов (CPU, память);

– динамический анализ выявляет ненужные вычисления и удаляет их.

Искусственная рекурсия и самовызовы:

Использование рекурсивных вызовов вместо простых итеративных конструкций может значительно запутать поток управления, особенно если рекурсия скрыта через сложные вызовы функций.

Достоинства:

– усложняет анализ за счёт множества рекурсивных вызовов;

– может привести к запутанному стеку вызовов.

Недостатки:

– может существенно замедлить работу программы;

– современные анализаторы могут идентифицировать и упрощать такие конструкции.

1.3 Существующие инструменты обфускации

Актуальные решения в области запутывания кода предлагают различные стратегии защиты программного обеспечения, включая трансформацию символьных идентификаторов, криптографическую защиту строковых литералов, модификацию IL-структур и интеграцию механизмов противодействия отладке. В настоящем разделе детально анализируются наиболее востребованные и результативные инструменты обфускации: Dotfuscator, ConfuserEx, ILProtector и DeepSea Obfuscator.

Dotfuscator от PreEmptive представляет собой передовой инструмент шифрования C# и обфускации кода, предназначенный для обеспечения безопасности приложений и защиты пользовательских данных. Данная платформа является одним из ведущих коммерческих решений для защиты кода в экосистеме .NET. Продукт распространяется в двух редакциях: Dotfuscator Community, входящий в состав всех версий Visual Studio с базовыми функциями защиты, и Dotfuscator Professional, доступный для оценки или приобретения с корпоративными возможностями защиты.

Основные технические характеристики платформы:

– трансформация символьных обозначений: классы, процедуры и переменные получают неинформативные наименования, что делает первоначальный код непригодным для анализа;

– исключение неактивного кода: устранение избыточных фрагментов для оптимизации размера сборки;

– криптографическая защита строковых ресурсов: все текстовые данные маскируются посредством шифрования и дешифруются исключительно в процессе исполнения;

– внедрение псевдо-путей исполнения: добавление кодовых блоков, не влияющих на функциональность, для повышения сложности анализа;

– механизмы защиты runtime-среды: детектирование попыток отладки или исследования с возможностью аварийного завершения работы программы.

ConfuserEx представляет собой open-source обфускатор для C#, который использует собственную версию dnlib для манипуляций со сборками. Этот бесплатный инструмент с открытым исходным кодом предназначен для защиты .NET-приложений и предоставляет функциональность, сопоставимую с коммерческими аналогами.

Ключевые технические возможности:

– замена наименований переменных и методов на произвольные символьные последовательности;

– шифрование текстовых констант с дешифровкой во время выполнения программы;

– интеграция алгоритмов обнаружения отладочных инструментов с автоматическим прерыванием выполнения при их детектировании;

– обфускация логики управления: реструктуризация алгоритмов работы программы для затруднения анализа;

– слияние методов: объединение множественных процедур в единую с изменением логики, что значительно усложняет код.

ILProtector специализируется на защите IL-кода. В отличие от обфускаторов, работающих исключительно с исходным кодом, ILProtector концентрируется на защите байт-кода.

Основные технические решения:

- криптографическая защита IL-кода – шифрование с дешифровкой исключительно в процессе исполнения, что предотвращает анализ через декомпиляторы типа ILSpy или dotPeek;
- противодействие отладчикам и прочим инструментам обратного проектирования;
- минимальное воздействие на производительность благодаря оптимизированным криптографическим алгоритмам;
- универсальная совместимость, поддержка всех .NET-сборок независимо от версии платформы.

DeepSea Obfuscator делает обфускацию .NET-сборок интуитивной и интегрированной частью процесса разработки продукта. Эта платформа предназначена для комплексной защиты .NET-приложений, включая обфускацию IL-кода, и предоставляет гибкие настройки для проектов любого масштаба.

Ключевые функциональные возможности:

- запутывание архитектуры IL-инструкций для усложнения анализа;
- создание ложных путей исполнения;
- криптографическая защита метаданных;
- интеграция в процессы сборки – автоматизация обфускации при создании исполняемых модулей.

Указанные инструменты активно применяются в проектах, требующих не только защиты от декомпиляции, но и обеспечения максимального усложнения анализа внутренней архитектуры программного обеспечения.

1.4 Предлагаемое решение

Обфускация IL-кода занимает центральное место в системе защиты .NET-приложений от обратного проектирования, однако сталкивается с множественными ограничениями. Указанные проблемы затрагивают как результативность

защитных механизмов, так и быстродействие программных решений, а также усложняют процессы их внедрения и сопровождения. Проанализируем основные аспекты данной проблематики.

Несмотря на то, что обфускация повышает сложность анализа программного кода, её стойкость к современным методам обратного проектирования остается недостаточной:

– статический анализ кода: множество техник, включая трансформацию наименований переменных или внедрение псевдо-путей исполнения, могут быть частично преодолены с использованием аналитических платформ типа ILSpy или Ghidra, которые восстанавливают архитектуру кода даже после применения обфускации;

– динамический анализ: запутанный код подвержен трассировке исполнения, что обеспечивает выявление фактических путей выполнения с игнорированием ложных ветвлений;

– инструменты деобфускации: распространенные решения (например, de4dot) автоматически нейтрализуют базовые типы обфускации, такие как переименование идентификаторов, понижая уровень защищенности;

– ограниченная многоуровневость защиты: к примеру, лексическое запутывание (устранение комментариев, замена наименований) не воздействует на программную логику, что создает уязвимость для восстановления алгоритмов.

Однако процессы обфускации часто влекут за собой замедление работы приложений:

– интеграция избыточных вычислений или фиктивных итераций повышает нагрузку на вычислительные ресурсы;

– применение таблиц переходов или косвенных вызовов нарушает предсказуемость исполнения, что ухудшает процессорную оптимизацию кода;

– накладные расходы на криптографические операции: методы, объединяющие обфускацию с шифрованием, требуют дополнительных ресурсов для расшифровки кода в процессе исполнения. Согласно исследованию Иванникова В.,

некоторые алгоритмы обфускации (в частности, сглаживание потока управления в O-LLVM) способны замедлить выполнение кода на 15–30% [4].

Сложность внедрения и сопровождения имеет следующие факторы:

– IL-код .NET подвергается оптимизации компилятором, который может устранить часть обфусцированных конструкций (например, неиспользуемые переменные), нивелируя усилия по защите;

– обфускаторы, функционирующие на уровне IL-кода (например, на базе LLVM), не всегда учитывают специфику целевой архитектуры, что может снижать их эффективность;

– трудности отладки – запутанный код сложно анализировать даже опытным разработчикам, что увеличивает временные затраты на выявление и устранение ошибок.

Ограничения методов обфускации:

– опытные аналитики могут идентифицировать недостижимые ветви посредством динамического анализа;

– таблицы переходов – структура может быть восстановлена через анализ шаблонов адресации;

– искусственная рекурсия может быть упрощена современными декомпиляторами, которые автоматически преобразуют её в итеративные конструкции.

Также имеет место и воздействие на совместимость и целостность, что описывается следующим:

– отдельные методы обфускации противоречат компиляторным оптимизациям, приводя к непредсказуемым сбоям;

– модификации в IL-коде могут нарушить функционирование Just-In-Time компилятора .NET, особенно при использовании виртуализации или динамической генерации кода;

– контрольные суммы и антиотладочные методы, интегрируемые в обфусцированный код, часто легко обходятся через патчинг.

Все перечисленные проблемы указывают на необходимость разработки инновационных подходов.

В рамках предлагаемого решения планируется спроектировать и реализовать приложение, учитывающее ограничения современных обфускаторов и направленное на повышение устойчивости к статическому и динамическому анализу. Основное внимание будет уделено усложнению графа потока управления, а также адаптации методов защиты под специфику IL-кода .NET. Такой подход позволит повысить сложность работы существующих деобфускаторов, минимизировать потери производительности и снизить риски, связанные с JIT-компиляцией, обеспечивая при этом совместимость с экосистемой .NET и улучшая общую стабильность и поддерживаемость защищённых приложений.

2 ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ ДЛЯ ЗАЩИТЫ IL-КОДА

2.1 Требования к ПО

Для разработки приложения для защиты IL-кода необходимо определить требования, которые обеспечат его корректную работу и эффективность обфускации. Основной задачей программного обеспечения является защита кода от анализа и модификации, что достигается путём преобразования промежуточного языка (IL) в сложночитаемую и трудновосстанавливаемую форму.

Важной частью требований является сохранение работоспособности защищённого кода. Все трансформации должны учитывать особенности платформы .NET и обеспечивать корректное выполнение программы после обфускации [5]. Это предполагает минимизацию возможных ошибок при изменении IL-кода и внедрение проверок для контроля целостности преобразований.

В разделе рассмотрены функциональные и нефункциональные требования, определяющие основные характеристики разрабатываемого ПО.

2.1.1 Функциональные требования

Функциональные требования к ПО сформированы и показаны ниже:

загрузка и обработка исходного IL-кода – приложение должно позволять пользователю загружать файлы с исходным IL-кодом (.il файлы) для дальнейшей обработки. Необходимо поддерживать чтение и анализ IL-кода для построения внутреннего представления программы;

реализация лексической обфускации – приложение должно автоматически убирать комментарии, пробельные символы, и символы форматирования в исходном коде для усложнения его анализа. Важно предусмотреть возможность исключений для отдельных частей кода (не изменять важные секции, которые могут быть полезны для отладки);

обфускация хранения – приложение должно предоставлять механизм для преобразования данных, хранимых в программе, таким образом, чтобы статические данные были заменены на динамически вычисляемые выражения (например, строки на функции, возвращающие значения). Реализованный алгоритм

должен быть управляемым, чтобы пользователь мог выбирать объекты для обфускации и способ преобразования данных;

обфускация потока управления - приложение должно предоставлять механизм для преобразования структуры потока управления программы с целью усложнения его анализа и обратной инженерии. Данный процесс включает изменение последовательности выполнения команд, внедрение ложных путей, использование несводимых участков в графе потока управления и запутывание условий переходов;

предварительный просмотр обфусцированного кода – пользователь должен иметь возможность просматривать результат обфускации до окончательного сохранения измененного IL-кода. Интерфейс должен показывать исходный и обфусцированный код в режиме сравнения для наглядности изменений;

приложение должно поддерживать разбор файлов для получения внутреннего представления IL-кода, необходимого для анализа и обфускации. После внесения изменений необходимо выполнять обратную сборку, создавая новый .il файл или компилируя его обратно в исполняемый формат (PE-файл);

генерация и сохранение обфусцированного IL-кода – приложение должно уметь сохранять обфусцированный IL-код в новый файл с возможностью сохранения исходного формата (.il) или экспорта в другие форматы, если это необходимо. Сохранение должно быть осуществлено с минимальными потерями производительности программы [6];

поддержка пользовательских настроек – пользователь должен иметь возможность задавать параметры обфускации (например, уровень сложности, исключения, правила генерации имен). Настройки должны сохраняться и быть доступны для последующих сеансов работы с приложением;

журналирование процесса обфускации – приложение должно вести лог процесса обфускации, чтобы пользователь мог отслеживать выполненные операции и их влияние на исходный код. Логи должны содержать информацию о том, какие объекты были изменены, каким образом и в какой момент времени.

Показана на рисунке 5 диаграмма прецедентов, отражающая функциональные требования приложения.

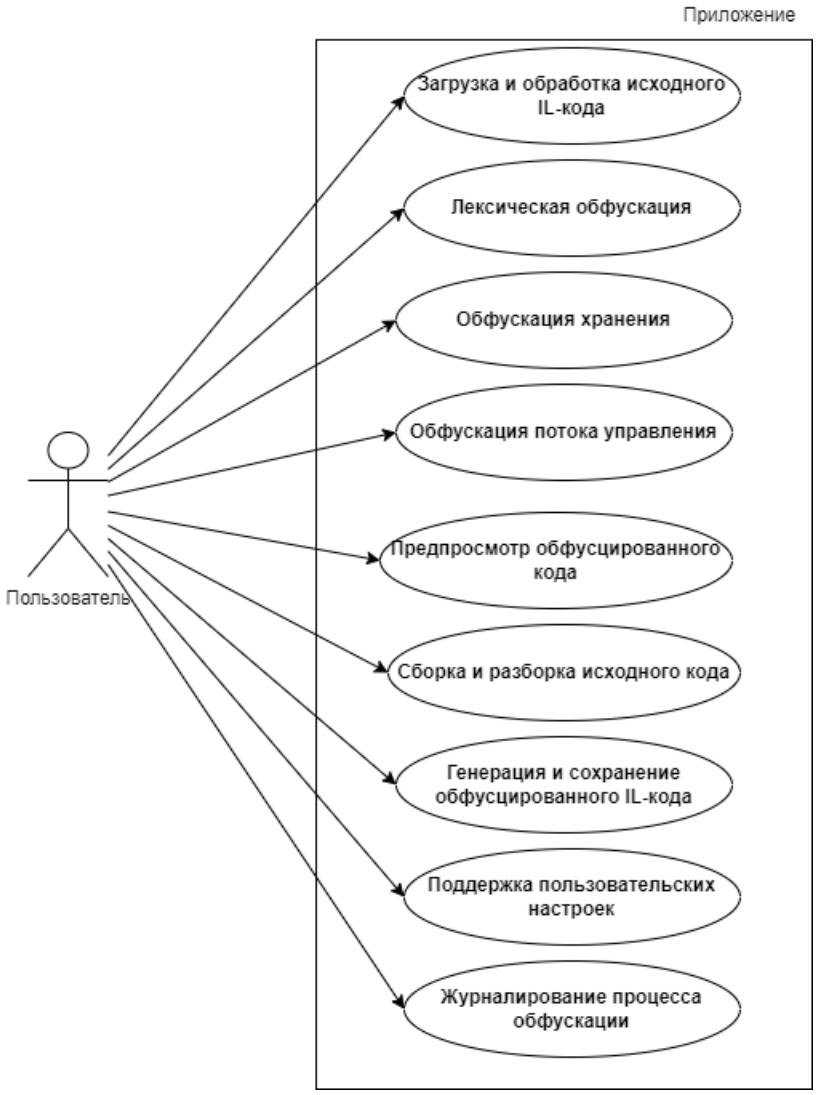

Рисунок 5 – Диаграмма прецедентов для приложения

Диаграмма отражает функциональные требования для приложения.

2.1.2 Нефункциональные требования

Программное обеспечение должно функционировать на стандартных настольных ПК и удовлетворять следующим требованиям:

Минимальные системные характеристики:

– процессор: 2-ядерный с частотой 2 ггц и выше;

– оперативная память: 4 гб;

– свободное место на диске: 1 гб;

– операционная система: Windows 10 и выше.

Программные зависимости:

– установленная версия .NET 8.0 или выше;

– nuget-пакеты ILASM и ILDASM для работы с IL-кодом;

– Python версии 3.10 и выше.

Требования к обработке данных – поддержка IL-кода, соответствующего стандартам .NET Framework и .NET Core.

Требования к совместимости:

– совместимость с инструментами анализа IL-кода, такими как ILSpy и dnSpy;

– использование библиотеки re для работы с регулярными выражениями;

– библиотека pyqt5 для работы с графическим интерфейсом.

2.2 Выбор средств разработки

2.2.1 Выбор модели жизненного цикла разработки

Для разработки ПО рассмотрим методологии SCRUM и RAD.

Scrum – это популярная разновидность методологий Agile, основанная на итеративном подходе к разработке программного обеспечения. Основной акцент делается на гибкости и адаптивности к изменениям, что важно в исследовательских проектах [7].

Основные принципы Scrum:

- разработка делится на короткие циклы, называемые спринтами (обычно 2–4 недели);
- каждую итерацию команда выпускает работающий прототип или инкремент продукта;
- роли включают владельца продукта, scrum-мастера и команду разработки;
- важными событиями являются ежедневные встречи (daily scrum), планирование спринта и итоговая демонстрация (sprint review);

– приоритеты работы задаются через список задач (product backlog), который может меняться в зависимости от обратной связи.

Ниже в виде таблицы показаны достоинства и недостатки данной технологии:

Таблица 1 – Достоинства и недостатки Scrum

Достоинства Scrum	Недостатки Scrum
– возможность изменять приоритеты задач в процессе выполнения; – регулярные встречи позволяют команде оставаться в курсе прогресса; – частая проверка позволяет своевременно выявлять ошибки и отклонения от цели.	– команда должна быть достаточно зрелой, чтобы эффективно управлять процессами; – в случае недостаточного планирования спринты могут затягиваться.

RAD – это методология, ориентированная на быструю разработку прототипов и сокращение времени на создание программного обеспечения. В ней основной упор делается на скорость и итеративность процесса.

Основные принципы RAD:

– быстрая разработка через создание прототипов и частую обратную связь с заказчиком;

– инкрементное улучшение продукта на основе прототипов;

– преимущество отдается выполнению небольших частей проекта, которые могут быть протестированы и усовершенствованы.

Ниже в виде таблицы показаны достоинства и недостатки RAD:

Таблица 2 – Достоинства и недостатки Scrum

Достоинства RAD	Недостатки RAD
– скорость разработки: rad позволяет быстро разрабатывать и внедрять новые функции;	– требует вовлеченности заказчика: заказчик должен активно участвовать в процессе разработки;

Продолжение таблицы 2

– гибкость: возможность мгновенной обратной связи и внесения изменений; – снижение затрат на исправление ошибок: раннее тестирование прототипов снижает стоимость их устранения.	– риск низкой масштабируемости: быстро созданные прототипы могут оказаться сложными для масштабирования и поддержки в долгосрочной перспективе.

Scrum и RAD представляют собой два гибких подхода, которые могут эффективно применяться для разработки инструмента обфускации IL-кода. Scrum позволит гибко управлять проектом, особенно если предполагаются частые изменения требований и необходимость экспериментов. RAD, в свою очередь, обеспечит быстрое прототипирование и тестирование функциональности, что полезно на этапе ранней разработки инструмента. Выбор этих методологий позволит поддерживать высокую динамику разработки и адаптироваться к возможным изменениям в ходе выполнения магистерской диссертации.

2.2.2 Выбор инструментов разработки

При создании программного комплекса для защиты промежуточного кода .NET-приложений посредством обфускации необходимо тщательно проанализировать доступные технологии и инструменты разработки. Выбор технологического стека должен учитывать специфику задач обработки IL-кода, требования к производительности системы, а также обеспечивать удобство сопровождения и расширения функциональности. Данный раздел содержит детальное обоснование выбранных решений.

В качестве базового языка программирования был принят Python, что обусловлено рядом существенных преимуществ для решения поставленных задач.

Процесс написания кода на Python значительно ускоряется, что критически важно при разработке сложных алгоритмов обфускации. Минималистичная структура языка способствует быстрому прототипированию и итеративной разработке Python располагает обширной экосистемой инструментов для анализа,

парсинга и генерации программного кода, что существенно упрощает реализацию механизмов трансформации IL-инструкций.

Через специализированные библиотеки (pythonnet, clr-модули) обеспечивается прямое взаимодействие с компонентами .NET Framework, что является принципиальным требованием для корректной обработки промежуточного кода.

Поддержка различных операционных систем позволяет создать универсальное решение, не привязанное к конкретной платформе.

Следует отметить, что интерпретируемая природа Python может негативно сказываться на производительности при обработке больших объемов IL-кода по сравнению с компилируемыми языками. Для критичных по времени выполнения операций предусмотрена возможность вынесения вычислительно-интенсивных участков в отдельные модули или применение JIT-компиляции.

Для разработки пользовательского интерфейса была выбрана библиотека PyQt – Python-обертка над мощным фреймворком.

PyQt предоставляет доступ к полному спектру возможностей Qt-фреймворка, включая продвинутую работу с оконными системами, обработку пользовательских событий и гибкую настройку визуального оформления.

Qt из себя представляет кроссплатформенную среду разработки приложений для создания графических пользовательских интерфейсов, а также кроссплатформенных приложений, которые работают на различных программных и аппаратных платформах, таких как Linux, Windows, macOS, что гарантирует стабильную работу приложения в различных операционных средах. PyQt поддерживает как процедурную, так и объектно-ориентированную парадигмы программирования, что позволяет выбирать оптимальный подход для конкретных компонентов интерфейса.

Для создания графического интерфейса применяется Qt Designer — специализированная среда визуальной разработки GUI-приложений на базе Qt-технологий [8].

Система drag-and-drop позволяет интуитивно размещать элементы управления без необходимости ручного кодирования разметки, что значительно ускоряет процесс создания пользовательского интерфейса.

Результатом работы в Qt Designer являются XML-файлы с расширением .ui, которые могут быть легко интегрированы в Python-код с помощью утилиты pyuic или загружены динамически.

Поддержка различных типов макетов (горизонтальных, вертикальных, сеточных, формовых) обеспечивает адаптивность интерфейса при изменении размеров окон приложения.

Встроенный редактор свойств позволяет детально настраивать внешний вид, поведение и доступность элементов интерфейса без программирования.

Внешний вид программы Qt Designer показан на рисунке ниже:

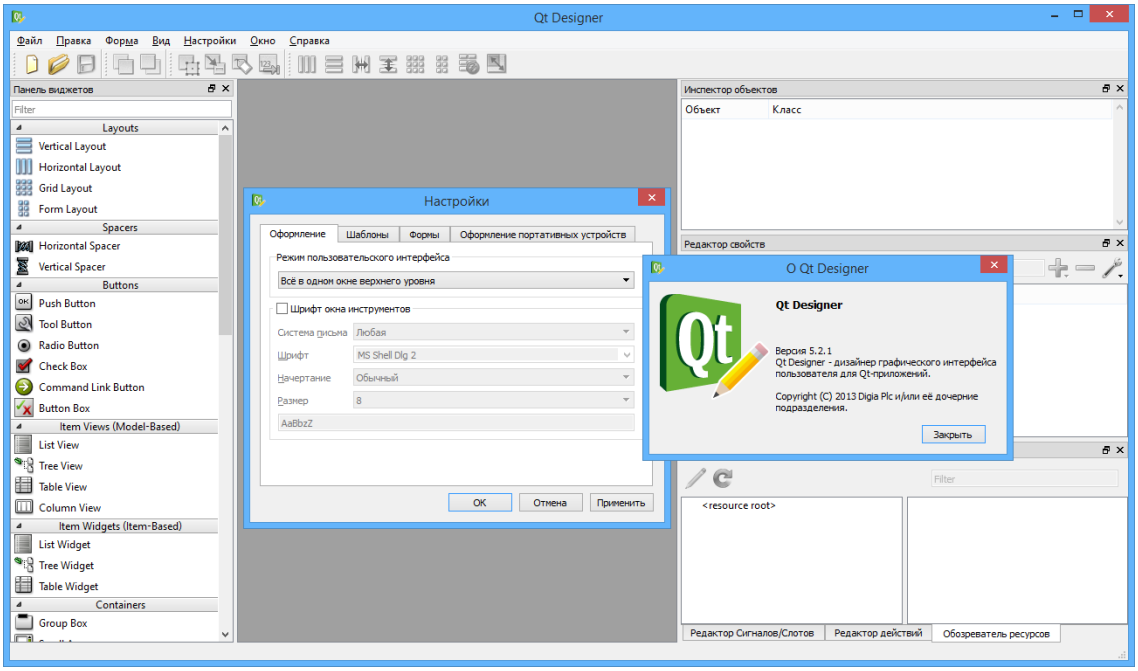

Рисунок 6 – Qt designer

Механизм сигналов и слотов Qt Designer поддерживает визуальную настройку связей между пользовательскими действиями и обработчиками событий. Поддержка контейнеров, групп элементов, вкладок и таблиц упрощает создание сложных интерфейсов.

По сравнению со стандартным Tkinter, связка PyQt + Qt Designer обеспечивает более широкие функциональные возможности, профессиональный внешний вид и удобство сопровождения сложных интерфейсов.

В качестве основной среды разработки используется PyCharm по следующим причинам:

– поддержка автодополнения, рефакторинга и встроенный анализатор кода;

– удобная система контроля версий;

– встроенные инструменты позволяют анализировать производительность и искать ошибки в коде;

– упрощает управление зависимостями проекта в виртуальном окружении.

Интерфейс среды разработки PyCharm представлен на рисунке 7:

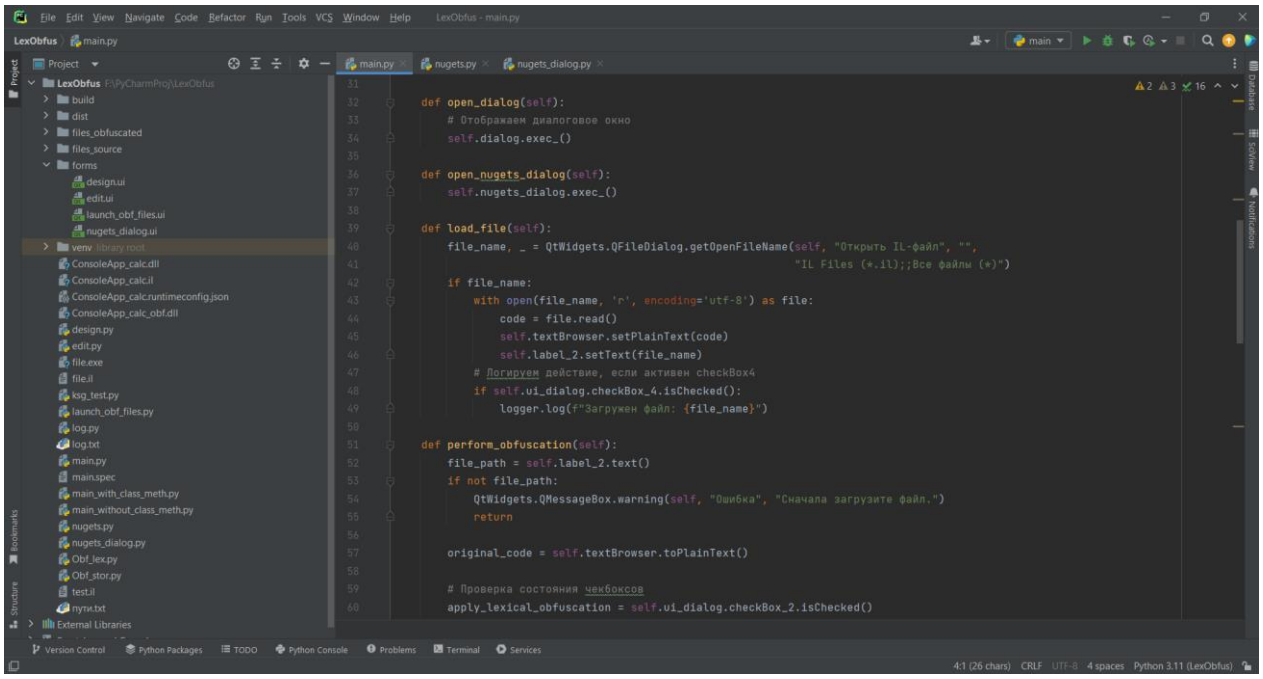

Рисунок 7 – Среда разработки Pycharm

Хотя существуют альтернативы (VS Code, Sublime Text, PyScripter), PyCharm обеспечивает наиболее мощный инструментарий для работы с Python в контексте средних и крупных проектов.

Успешная реализация системы обфускации IL-кода требует наличия специализированных средств для манипуляции с промежуточным представлением .NET-приложений. Центральную роль в этом процессе играют два взаимодополняющих инструмента из экосистемы .NET SDK – ассемблер промежуточного языка (ILASM) и дизассемблер (ILDASM).

Данные утилиты обеспечивают полный цикл трансформации программного кода: от извлечения IL-представления из скомпилированных сборок до повторной генерации исполняемых файлов после внесения модификаций. Такой подход позволяет выполнять глубокую трансформацию кода на уровне промежуточного языка без потери функциональности исходного приложения.

ILASM представляет собой компилятор, выполняющий трансляцию текстового IL-представления в исполняемые сборки формата PE (Portable Executable). Основное назначение инструмента заключается в создании готовых к выполнению .exe и .dll файлов из модифицированного IL-кода.

Компилятор обеспечивает следующие ключевые операции: генерация исполняемых модулей, версионная совместимость, метаданные сборки.

Синтаксис командной строки для компиляции исполняемого файла:

ilasm source_code.il /output=application.exe /exe

Создание библиотеки динамической компоновки:

ilasm library_code.il /dll

ILASM выполняет финальный этап защиты кода, пересобирая модифицированные IL-файлы после применения техник обфускации.

ILDASM является инструментом обратной инженерии, преобразующим бинарные .NET-сборки в текстовое IL-представление. Утилита обеспечивает полный доступ к внутренней структуре скомпилированных модулей, включая код, метаданные и манифест сборки.

К его функционалу относится конвертация исполняемых файлов в читаемое текстовое представление промежуточного языка. Также, детальное исследование таблиц методов, описаний классов, полей и связанных метаданных.

При запуске в интерактивном режиме предоставляет визуальное представление архитектуры сборки.

Дизассемблирование сборки в IL-файл для использования в командной строке:

ildasm target_assembly.exe /output=disassembled_code.il

ILDASM инициирует процесс защиты кода, извлекая IL-представление для последующего анализа и модификации.

Следует учитывать, что рассматриваемые утилиты изначально разрабатывались для Windows-окружения и могут иметь ограниченную функциональность в других операционных системах. В Linux и macOS доступность ILASM/ILDASM зависит от конкретной реализации .NET и может потребовать использования платформы Mono.

Для расширения возможностей анализа IL-кода могут применяться специализированные инструменты:

dnSpy как комплексная среда декомпиляции с поддержкой интерактивного редактирования и отладки .NET-сборок;

ILSpy представляет собой открытый декомпилятор с возможностью анализа структуры сборок и экспорта в различные форматы;

профессиональный инструмент dotPeek от JetBrains для декомпиляции и исследования .NET-кода.

В Unix-подобных системах функциональность ILASM может быть заменена Mono IL assembler, хотя его возможности могут отличаться от оригинальной Microsoft-реализации.

Комбинирование ILASM и ILDASM обеспечивает полный контроль над процессом трансформации IL-кода.

2.3 Синтез эффективных алгоритмов обфускации

В данном разделе представлены разработанные алгоритмы обфускации, созданные на основе известных методов, но адаптированные для повышения эффективности и усложнения анализа IL-кода. В процессе разработки были учтены

особенности IL-кода и выбраны подходы, обеспечивающие баланс между степенью защиты и производительностью программы.

2.3.1 Алгоритм лексической обфускации

Лексическая обфускация представляет собой процесс преобразования видимой части кода с целью затруднить его чтение и анализ. На первом этапе загружается исходный файл программы, после чего выполняется анализ структуры кода для определения всех идентификаторов, таких как переменные, методы и классы. Затем каждый идентификатор подвергается замене на уникальное, нечитаемое имя, сохраняя при этом функциональность. После замены выполняется валидация синтаксиса, чтобы убедиться, что изменения не привели к ошибкам. Если обнаруживаются ошибки, процесс откатывается до начального состояния.

Процесс лексической обфускации можно изобразить в виде диаграммы состояний на рисунке 8.

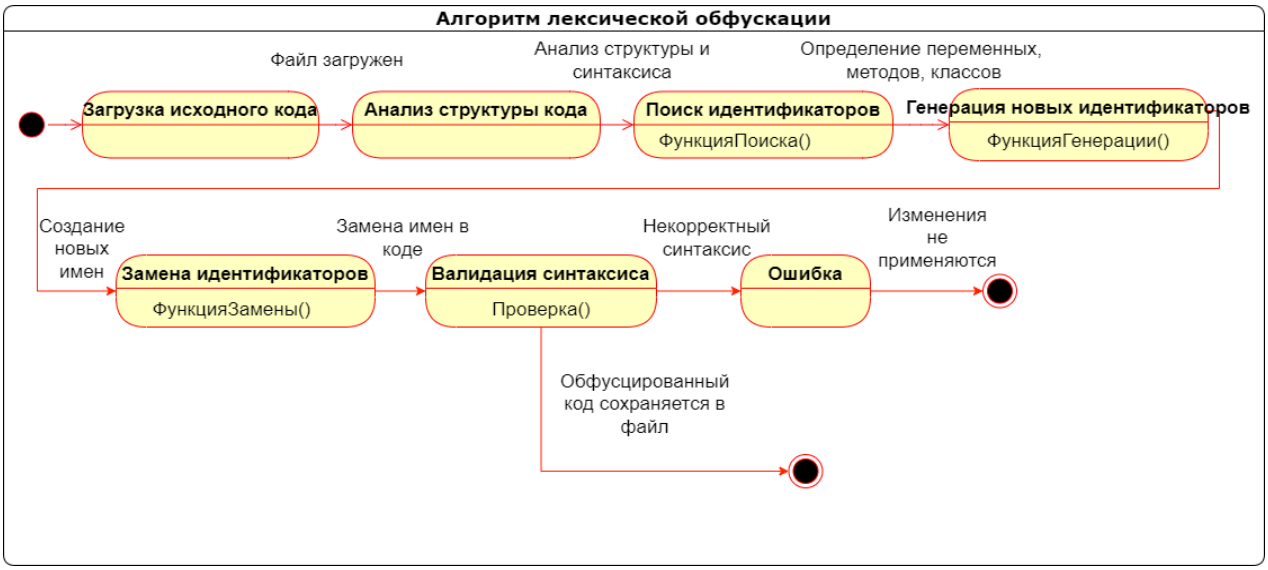

Рисунок 8 – Диаграмма для алгоритма лексической обфускации

После успешной проверки завершается обфускация, и пользователь получает изменённый код, который остаётся функциональным, но становится сложным для чтения и понимания.

2.3.2 Алгоритм обфускации хранения

Обфускация хранения данных направлена на изменение способов хранения и использования данных, что делает их труднодоступными для анализа

статическими и динамическими инструментами. В данном разделе рассматриваются наиболее эффективные методы обфускации хранения, их особенности, а также обоснование выбора метода, который реализован в рамках данной работы.

Для реализации обфускации хранения необходимо выбрать метод, который обеспечит баланс между уровнем защиты, сложностью реализации и производительностью. Из рассмотренных выше методов наиболее перспективными являются разбиение данных, запутывание доступа к данным и использование кастомизированных структур, так как они позволяют скрыть структуру хранения без значительных затрат ресурсов.

В данной работе предлагается метод динамического представления данных, основанный на создании пользовательских структур, маскирующих реальные значения. Суть метода заключается в том, что данные не хранятся в стандартных переменных, а формируются во время выполнения программы с помощью вычислений и логических преобразований. Например, числовые значения могут быть представлены в виде комбинации нескольких элементов структуры, а строки – в виде набора отдельных символов с динамическим восстановлением.

Разработанный метод для программной реализации обфускации хранения показан на рисунке 9.

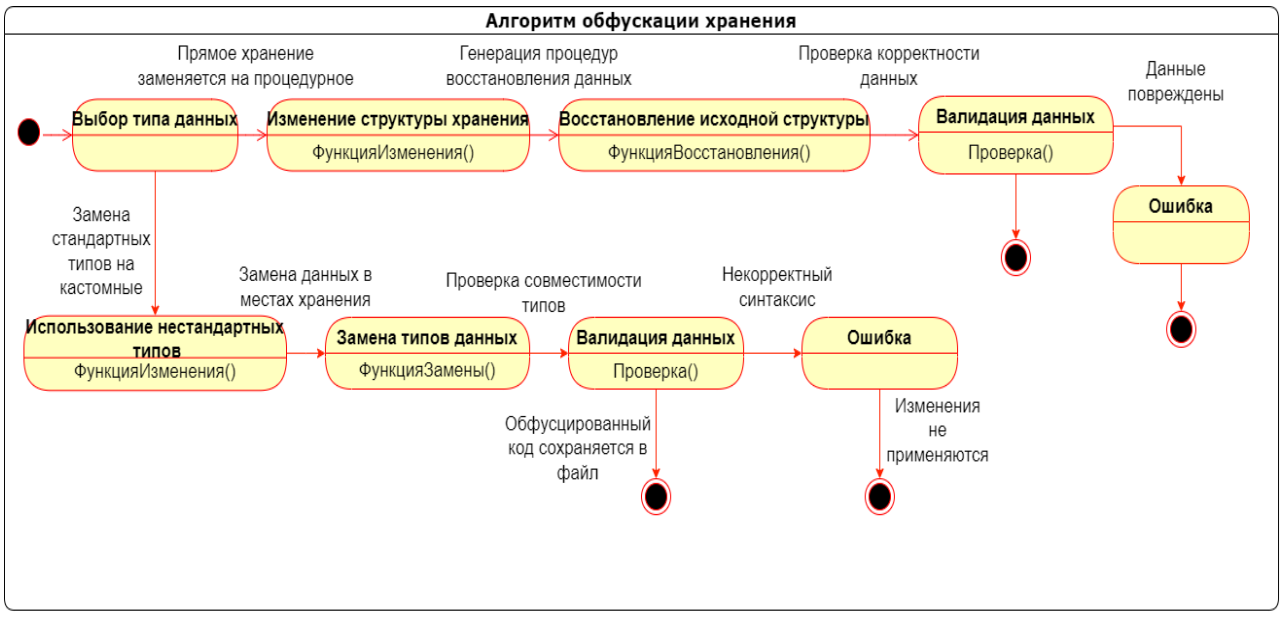

Рисунок 9 – Диаграмма для алгоритма обфускации хранения

Процесс начинается с выбора данных, подлежащих обфускации, после чего их структура хранения преобразуется: данные кодируются или хранятся в ином формате. Вместо стандартных типов данных применяются кастомизированные структуры, что дополнительно затрудняет разбор кода. Для корректной работы программы создаются процедуры, которые восстанавливают данные во время выполнения. После замены типов данных проводится проверка их корректности и совместимости с системой. В случае обнаружения проблем изменения отменяются. Успешная обфускация завершает процесс, оставляя данные защищёнными за счёт использования нестандартных методов хранения и сложных структур.

2.3.3 Алгоритм обфускации потока управления

Обфускация потока управления представляет собой метод защиты программного кода, направленный на усложнение анализа и обратной разработки путем изменения структуры графа потока управления (Control Flow Graph, CFG). Ниже на рисунке 10 представлен пример граф потока управления, используемый компилятором Rust для генерации кода.

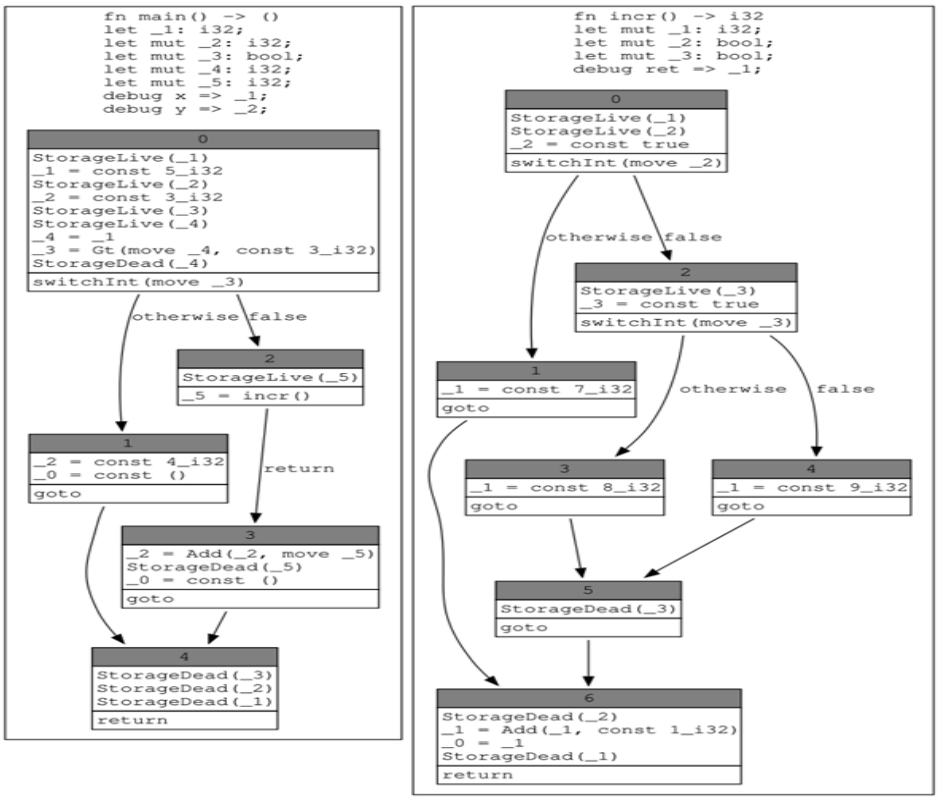

Рисунок 10 – Пример графа потока управления

Граф потока управления представляет собой структуру, где вершины обозначают базовые блоки кода (участки без разрывов), а рёбра – возможные переходы между ними [9].

Традиционно программный код характеризуется CFG с линейной или древовидной архитектурой, что обеспечивает предсказуемость выполнения программы. Модификация структуры графа потока управления посредством обфускации создает препятствия для понимания программной логики и существенно осложняет применение техник статического анализа кода.

Для реализации обфускации потока управления в современной практике разработки защищенного программного обеспечения применяются различные методологические подходы, каждый из которых обладает специфическими характеристиками и областями применения.

Метод выбранный для интеграции в разрабатываемое приложение в связи с его способностью существенно усложнять анализ кода без создания чрезмерной нагрузки на вычислительные ресурсы, представляет собой генерацию несводимых участков в графе потока управления. Несводимый участок в CFG представляет собой фрагмент кода, который не поддаётся упрощению стандартными алгоритмами анализа. Формирование таких участков достигается введением циклических конструкций, обратных переходов и сложных условий, нарушающих линейность графа и затрудняющих его интерпретацию.

Достижение несводимости обеспечивается внедрением ложных переходов через вставку переходов в произвольные участки кода, которые не выполняются в действительности, но усложняют анализ. Параллельно применяется добавление избыточных циклов посредством создания дополнительных циклических структур, увеличивающих сложность CFG без модификации логики выполнения. Дополнительно используется разделение логики с динамическими переходами через реализацию конечного автомата, где каждое последующее действие определяется сложным выражением вместо линейного выполнения.

В таблице 3 приведено описание методов реализации для обеспечения несводимости графа потока управления:

Таблица 3 – Методы реализации несводимости графа потока управления

Метод реализации	Техническое описание	Влияние на анализ
Внедрение ложных переходов	Вставка инструкций jmp, br в произвольные участки кода без фактического выполнения	Создание дополнительных путей в CFG, затрудняющих статический анализ
Добавление бесполезных операций	Создание циклических структур без изменения программной логики	Увеличение сложности графа и количества возможных состояний
Использование try-catch	Реализация состояний с выбором действий на основе сложных выражений	Нарушение предсказуемости потока выполнения

Для решения задачи с обфускацией потока управления методом генерации несводимых участков разработан специальный алгоритм.

Преимущества:

- значительно усложняет анализ кода, особенно для автоматизированных инструментов;
- минимальное влияние на производительность и потребление памяти;
- хорошо сочетается с другими методами обфускации.

Недостатки:

- требует тонкой настройки для балансировки сложности анализа и быстродействия;
- может усложнить отладку легитимного кода.

Алгоритм обфускации потока управления изображен на рисунке ниже:

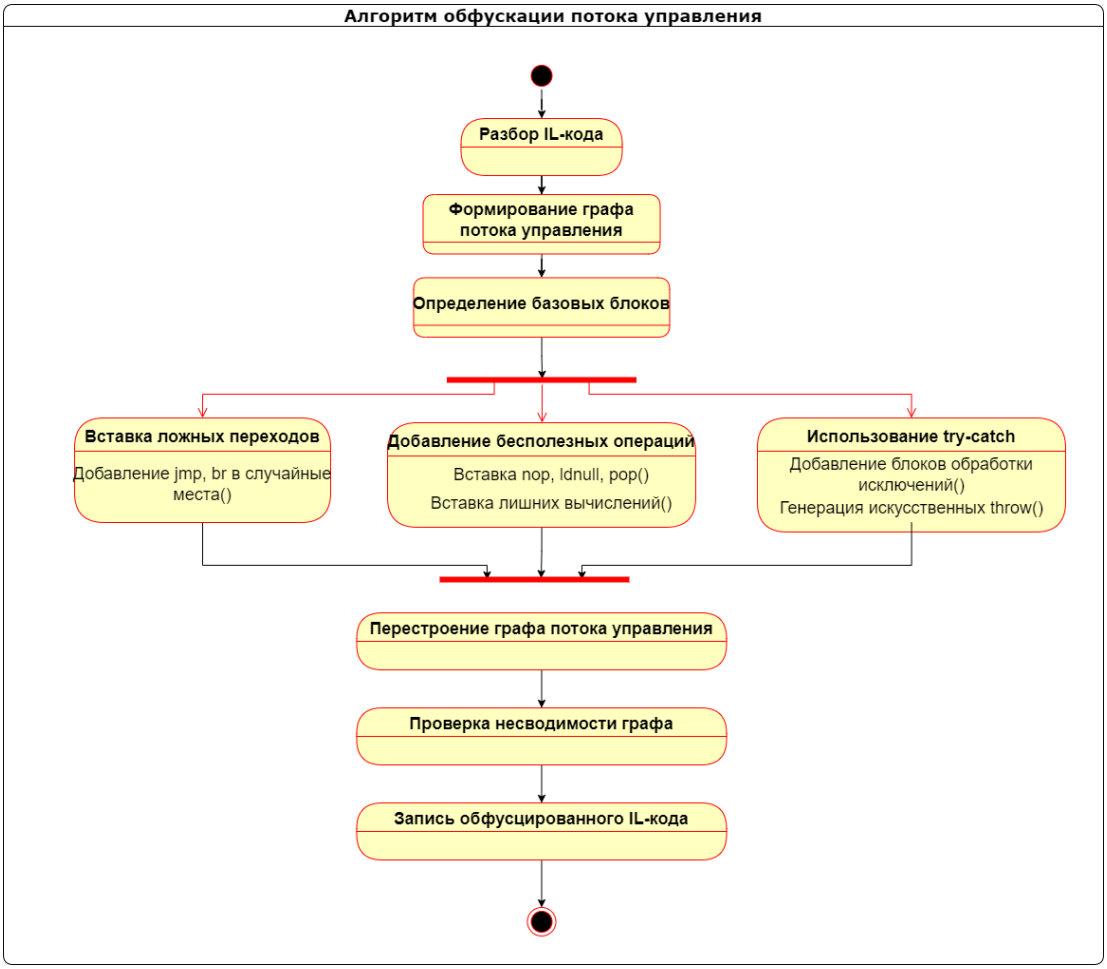

Рисунок 11 – Алгоритм обфускации потока управления

Процесс начинается с анализа исходного IL-кода и выделения ключевых участков, где могут быть внедрены изменения без нарушения логики выполнения. После этого выполняется трансформация кода по следующим функциям:

вставка ложных переходов. Для этого в IL-код добавляются инструкции br, br.s, jmp, leave, которые создают видимость альтернативных путей выполнения программы. Эти переходы указывают на неиспользуемые области кода, которые на самом деле никогда не выполняются. Это приводит к усложнению графа потока управления (CFG), затрудняя анализатору восстановление реальной последовательности команд;

добавление бесполезных операций. В код вставляются инструкции nop, ldnull, pop и другие команды, которые не влияют на выполнение программы, но увеличивают объем IL-кода;

использование try-catch блоков. Этот метод позволяет скрывать реальный поток управления за счет обработки исключений. В IL-код внедряются блоки try с искусственно создаваемыми catch, которые не несут смысловой нагрузки, но при этом усложняют анализ исполнения кода. Это особенно эффективно против инструментов статического анализа, так как они вынуждены учитывать возможность исключений, что делает поток управления менее предсказуемым.

2.4 Архитектура Simple IL-fuscator

Приложение состоит из нескольких ключевых модулей, которые обеспечивают функциональность и взаимодействие с пользователем.

На рисунке 12 представлена компонентная схема приложения, которая состоит из модулей виртуального окружения Python.

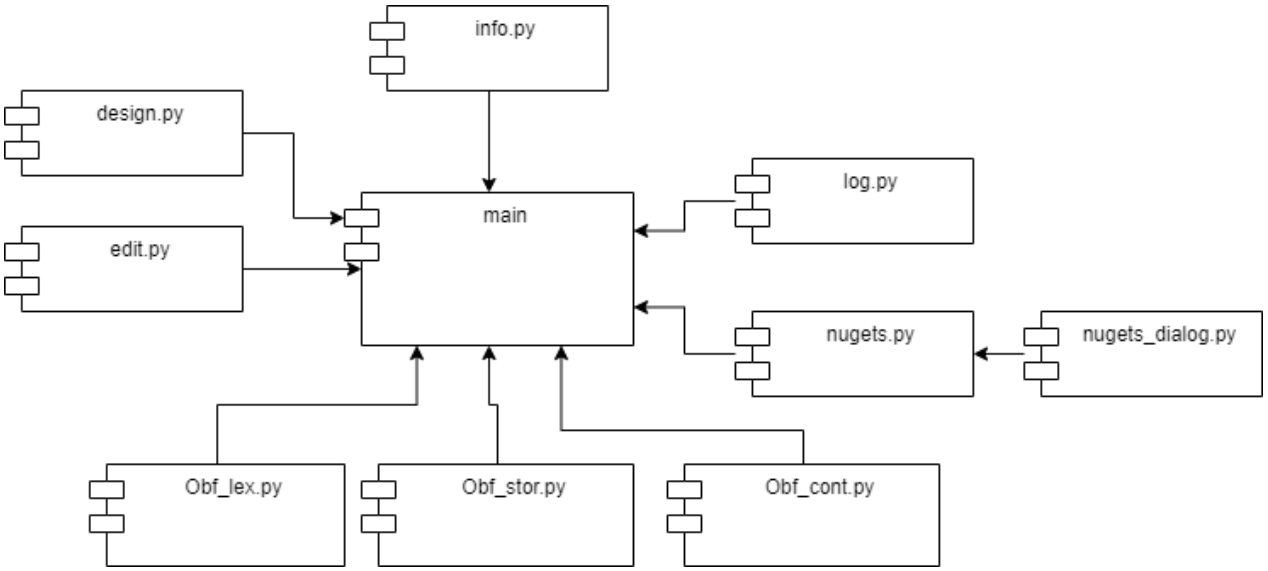

Рисунок 12 – Компонентная схема приложения

Главный модуль (main.py):

в данном модуле реализован класс MainWindow, который представляет основное окно приложения. Это центральный элемент программы, где пользователь может загружать IL-код, выбирать параметры обфускации и запускать

процесс обфускации. В нем интегрируются все остальные модули для выполнения задач.

Модуль дизайна интерфейса (design.py):

этот модуль отвечает за верстку и визуальное оформление главной формы приложения. Он содержит описание интерфейса пользователя, включая кнопки, текстовые поля, окна для отображения результатов работы программы. Все элементы интерфейса, такие как кнопки загрузки и запуска обфускации, текстовые поля для вывода ошибок и сообщений, а также панель настроек, создаются и настраиваются здесь.

Модуль для работы с диалоговыми окнами настроек (edit.py):

модуль содержит верстку диалогового окна, которое отображается для изменения настроек программы, таких как параметры лексической обфускации и обфускации хранения. Пользователь может выбрать дополнительные опции, которые влияют на работу программы и её поведение при обфускации.

Модули обфускации (Obf_lex.py, Obf_stor.py и Obf_cont.py):

Obf_lex.py – модуль лексической обфускации, который занимается изменением структуры исходного IL-кода для усложнения его анализа, замены идентификаторов и внесения других изменений на уровне лексики;

Obf_stor.py – модуль для обфускации хранения, который изменяет структуру хранения данных, чтобы предотвратить их легкий доступ и анализ.

Obf_cont.py – модуль для обфускации потока управления, который модифицирует граф потока управления программы, добавляя ложные переходы, изменяя порядок выполнения блоков и усложняя статический анализ;

Модуль логирования (log.py):

этот модуль отвечает за логирование всех действий пользователя в процессе работы с приложением. Он записывает события, такие как загрузка кода, запуск обфускации, ошибки и завершение процесса. Логирование помогает отслеживать действия программы и устранять возможные ошибки.

Модуль для работы с NuGet-пакетами (nugets):

модуль, который взаимодействует с NuGet-пакетами ilasm и ildasm, необходимыми для работы с IL-кодом. Он позволяет выполнять операции с IL-кодом, такие как компиляция и декомпиляция, используя эти пакеты. Модуль взаимодействует с интерфейсом программы и обрабатывает запросы на работу с этими инструментами.

Модуль дизайна окна для работы с NuGet-пакетами (nuget_design.py):

этот файл содержит описание модального окна, которое открывается для работы с пакетами ilasm и ildasm. Здесь пользователь может управлять установкой и настройкой пакетов NuGet, что позволяет работать с IL-кодом непосредственно в приложении;

info.py – вспомогательный модуль, содержащий документацию по использованию приложения, описание его функционала и информации о разработчике. Также включает справку по командам, настройкам и параметрам работы программы.

Основное приложение (main.py) служит центральной точкой, которая подключает все остальные модули. Через интерфейс, реализованный в design.py, пользователь взаимодействует с программой, загружает IL-код и выбирает параметры для обфускации. При запуске обфускации основные модули – Obf_lex, Obf_stor, Obf_cont выполняют свою работу, изменяя код согласно выбранным параметрам. Логирование процесса происходит в модуле log, который записывает каждый шаг работы программы.

3 РЕАЛИЗАЦИЯ ПРОГРАММНОГО ПРОДУКТА

3.1 Организация входных и выходных данных

Для корректной работы разрабатываемого программного продукта требуется чётко определить структуру входных и выходных данных. Входные данные обеспечивают приложение необходимой исходной информацией для выполнения обфускации, а выходные данные содержат результаты обработки и сведения о процессе выполнения операций.

На рисунке 13 показана организация входных и выходных данных для приложения.

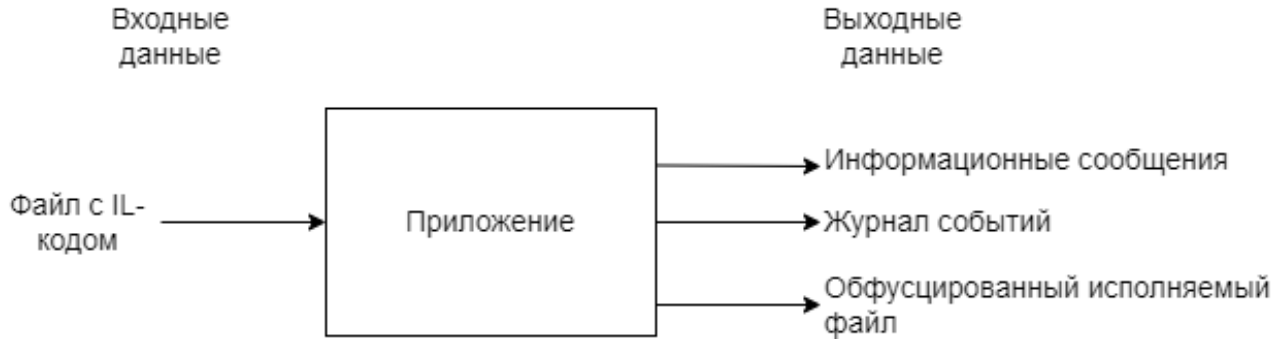

Рисунок 13 – Входные и выходные данные приложения

Входные данные:

– файл с IL-кодом – код, над которым будут совершаться манипуляции.

Выходные данные:

– информационные сообщения. Всплывающие окна или системные уведомления с результатами операций (например, «Обфускация завершена», «Ошибка чтения файла»);

– журнал событий. Содержит записи о ходе выполнения операций, включая ошибки, предупреждения и успешные действия;

– обфусцированный исполняемый файл. Файл .exe или .dll, обработанный для усложнения анализа.

Стоит отметить, что приложение использует вспомогательные файлы для правильной сборки файла обратно в исполняемый.

3.2 Реализация лексической обфускации

В рамках данной работы разработан модуль, осуществляющий лексическую обфускацию промежуточного языка IL. Ниже приводится описание этапов реализации данного модуля.

На первом этапе производится анализ исходного IL-кода для выявления имен локальных переменных, подлежащих замене. В языке IL объявления локальных переменных, как правило, располагаются в директиве .locals init, например:

.locals init ([0] int32 counter, [1] string message)

Для поиска таких строк применяется регулярное выражение, позволяющее идентифицировать объявления переменных, после чего каждая запись разбирается на отдельные элементы — тип и имя переменной. Полученные данные сохраняются в словарь, где ключом выступает имя переменной, а значением – структура, содержащая тип данных, оригинальное имя и поле для будущей замены.

Формат хранения данных переменной в словаре:

{'counter': {'type': 'int32', 'original_name': 'counter', 'obfuscated_name': ''}}

Пример обработки строки:

.locals init ([0] int32 counter, [1] string message)

Результат:

- counter – тип int32;
- message – тип string.

Таким образом, формируется полная карта локальных переменных, подлежащих обфускации.

Для корректной замены идентификаторов необходимо ограничить круг обрабатываемых типов, чтобы избежать вмешательства в системные или специфические IL-типизации. В данной реализации поддерживаются следующие стандартные типы данных:

- целочисленные типы: sbyte, byte, int16, uint16, int32, uint32, int64, uint64;
- вещественные типы: single, double, decimal;

- логический тип: boolean;
- символьные и строковые типы: char, string;
- обобщённый тип: object.

Фильтрация производится посредством сопоставления с эталонным списком типов данных. Переменные, тип которых отсутствует в данном списке, не подвергаются обфускации.

После выявления переменных осуществляется генерация новых имен. Для этого используется функция _replace, которая формирует псевдослучайную строку из букв латинского алфавита и цифр. Имя переменной начинается обязательно с буквы, что обусловлено синтаксическими требованиями IL.

Пример функции генерации:

```
def _replace(self, input_string, alphabets, min_length, max_length):
    length = random.randint(min_length, max_length)
    first_char = random.choice(self.alphabets[0])
    replaced_string = first_char + ''.join(random.choice(alphabets[i % 2]) for i in range(1, length))
    return replaced_string
```

Полученные имена записываются в ранее созданный словарь переменных в поле obfuscated_name.

На заключительном этапе осуществляется прямой обход IL-кода с целью замены всех вхождений исходных имен переменных на их обфусцированные эквиваленты. При этом используются шаблоны с регулярными выражениями, позволяющими избежать частичных совпадений, например, при наличии в коде строк, содержащих похожие подстроки [10].

Также предварительно из IL-кода удаляются все однострочные комментарии, чтобы исключить подмену имен, используемых исключительно в пояснении логики.

Пример исходного кода на IL:

```
.locals init ([0] int32 counter)
ldloc.0
```

call int32 [mscorlib]System.Console::ReadLine()

После обфускации:

.locals init ([0] int32 Zq9H2T)

ldloc.0

call int32 [mscorlib]System.Console::ReadLine()

Таким образом, переменная counter заменена на Zq9H2T, что затрудняет анализ её назначения в программе без восстановления контекста.

В таблице 4 приведён пример результата работы алгоритма лексической обфускации на примере набора переменных.

Таблица 4 – Пример результата обфускации

Тип переменной	Исходное имя	Обфусцированное имя
int32	counter	aX9b2K
string	message	qM1rTf

Данный способ замены идентификаторов является простым, но в то же время эффективным методом усложнения анализа кода.

3.3 Реализация обфускации хранения

Для реализации обфускации хранения была разработана система преобразований, адаптированная под различные типы данных: целочисленные, вещественные, строковые и логические. Основная идея заключается в представлении исходного значения в виде результата операции над двумя и более операндами. Один из операндов является случайной «маской», другой – результатом применения обратной операции.

Для целочисленных типов (int, long) применяется операция побитового исключающего ИЛИ (xor). Значение разбивается следующим образом:

- генерируется случайное число key;
- оригинальное значение value преобразуется в value ^ key;
- в коде подгружаются оба значения, а затем применяется операция xor, восстанавливающая оригинал.

Пример для int:

ldc.i4 85 // 42 ^ 127

ldc.i4 127

xor // -> 42

Для вещественных типов (float, double) используется обратимая операция деления. Исходное значение умножается на случайный множитель mul, затем в IL-коде выполняется деление на этот множитель:

*ldc.r4 128.1 // 42.7 * 3.0*

ldc.r4 3.0

div // -> 42.7

Подобное преобразование сохраняет точность, особенно в случае double, и позволяет усложнить анализ значений.

Для строк (string) используется кодирование в формате Base64. Результатом подстановки является последовательность вызовов стандартных методов платформы .NET:

ldstr "SGVsbG8="

call class [mscorlib]System.Byte[] [mscorlib]System.Convert::FromBase64String(string)

call class [mscorlib]System.Text.Encoding [mscorlib]System.Text.Encoding::get_UTF8()

callvirt instance string [mscorlib]System.Text.Encoding::GetString(uint8[])

Таким образом, исходная строка Hello преобразуется в формат, который декодируется в процессе исполнения, без отображения оригинального текста в IL-коде.

Для логических значений (boolean) применяется преобразование аналогичное целочисленному типу: true представляется как 1, false – как 0, и далее код подменяется обфусцированной загрузкой int.

На этапе трансформации исходный IL-код проходит через регулярную замену шаблонов, соответствующих загрузке непосредственных значений. Для каждого типа реализована отдельная функция обработки. Это позволяет гибко

расширять поддержку новых форматов или адаптировать поведение под требования конкретного окружения.

Функция obfuscate_il_storage применяет регулярные выражения к коду и подменяет выражения следующего вида:

- ldc.i4 или ldc.i4.s → obfuscate_int()
- ldc.i8 → obfuscate_long()
- ldc.r4 → obfuscate_float()
- ldc.r8 → obfuscate_double()
- ldstr "..." → obfuscate_string()
- ldc.i4.0 / ldc.i4.1 → obfuscate_boolean()

Таким образом, при трансляции, следующего кода:

ldc.i4 10

ldc.r8 3.14

ldstr "admin"

ldc.i4.1

получен обфусцированный IL, изображенный на рисунке ниже:

```
ldc.i4 103
ldc.i4 97
xor               // 103 ^ 97 = 10

ldc.r8 22.596
ldc.r8 7.2
div               // 22.596 / 7.2 = 3.14

ldstr "YWRtaW4="
call class [mscorlib]System.Byte[] [mscorlib]System.Convert::FromBase64String(string)
call class [mscorlib]System.Text.Encoding [mscorlib]System.Text.Encoding::get_UTF8()
callvirt instance string [mscorlib]System.Text.Encoding::GetString(uint8[])

ldc.i4 76
ldc.i4 77
xor               // 76 ^ 77 = 1
```

Рисунок 14 – Код после обфускации хранения

Такой подход значительно затрудняет анализ и восстановление оригинальных значений при декомпиляции, что особенно критично в случаях хранения конфиденциальных данных (например, паролей, токенов, конфигурационных ключей и др.).

Реализация данной логики выполнена на языке Python, что позволяет использовать инструмент в автоматическом режиме в рамках пайплайна сборки или как отдельную утилиту для обработки промежуточного кода.

3.4 Реализация обфускации потока управления

Данный метод направлен на изменение структуры выполнения программы таким образом, чтобы она сохранила первоначальную семантику, но стала крайне сложной для статического анализа, дизассемблирования и реверс-инжиниринга.

Обфускация потока управления осуществляется путём изменения порядка исполнения инструкций, вставки ложных переходов, мёртвого кода и искусственно усложнённых логических конструкций. В данном приложении был реализован модуль, который имитирует подобную обфускацию для промежуточного кода на IL (Intermediate Language).

Ниже приведён листинг реализации класса ControlFlowObfuscator, отвечающего за запутывание управляющей логики кода:

инициализация и генерация меток:

```
import re
import random
import string
class ControlFlowObfuscator:
    def __init__(self):
        self.label_pool = set()
    def _rand_label(self, length=6):
        while True:
            label = ''.join(random.choices(string.ascii_uppercase + string.digits, k=length))
            if label not in self.label_pool:
                self.label_pool.add(label)
                return label
```

Для генерации уникальных меток переходов используется метод _rand_label(), создающий строки из случайных символов, исключающих повторы. Эти метки применяются для создания ложных точек входа и искусственных переходов, что усложняет построение корректного графа потока управления.

Ниже представлена функция вставки ложных переходов и «мусорных» инструкций:

```
def _insert_fake_jumps(self, lines: list[str]) -> list[str]:
    result = []
    for line in lines:
        result.append(line)
        if random.random() < 0.25:
            fake_label = self._rand_label()
            junk_op = random.choice(["nop", "ldc.i4.0", "ldc.i4.1", "pop"])
            result.append(f"    br.s {fake_label}")
            result.append(f"{fake_label}:")
            result.append(f"    {junk_op}")
    return result
```

Метод _insert_fake_jumps() вставляет псевдопереходы в случайных местах кода. За каждым переходом (br.s) следует фиктивная инструкция, не влияющая на логику исполнения (например, nop – операция, не выполняющая никаких действий, или pop, удаляющая верхнее значение из стека). Таким образом, программа продолжает корректно работать, но анализ переходов становится значительно сложнее.

Вставка мёртвых логических конструкций:

```
def _insert_confusing_block(self, lines: list[str]) -> list[str]:
    fake = self._rand_label()
    end = self._rand_label()
    ops = [
        "ldc.i4.0",
```

```
            "brtrue.s " + end,
            "ldc.i4.1",
            "pop",
            "nop",
            end + ":"
        ]
        insert_at = random.randint(1, len(lines) - 1)
        return lines[:insert_at] + [f"{fake}:", *["    " + op for op in ops]] + lines[in-
sert_at:]
```

В данном фрагменте добавляется условный блок, который никогда не выполняется, так как сравнение (ldc.i4.0 → brtrue.s) всегда ложно. Этот участок выглядит как ветвление, но является мёртвым кодом. Это усложняет анализ ветвлений и может ввести в заблуждение автоматические инструменты анализа.

Перестановка блоков:

```
    def _shuffle_blocks(self, lines: list[str]) -> list[str]:
        blocks = []
        current = []
        for line in lines:
            if re.match(r'^\s*\w+:$', line):
                if current:
                    blocks.append(current)
                    current = []
            current.append(line)
        if current:
            blocks.append(current)
        random.shuffle(blocks)
        return [line for block in blocks for line in block]
```

Здесь производится перестановка блоков кода, определённых по наличию меток. Порядок следования блоков меняется случайным образом, при этом

логика исполнения сохраняется за счёт явных переходов между ними. Подобная техника используется, чтобы разорвать линейность кода и затруднить чтение.

В итоге obfuscate_flow() последовательно применяет три вышеописанных техники: вставку ложных переходов, добавление запутанных блоков и перестановку сегментов кода.

На рисунке 15 приведен пример кода до обфускации потока управления и после:

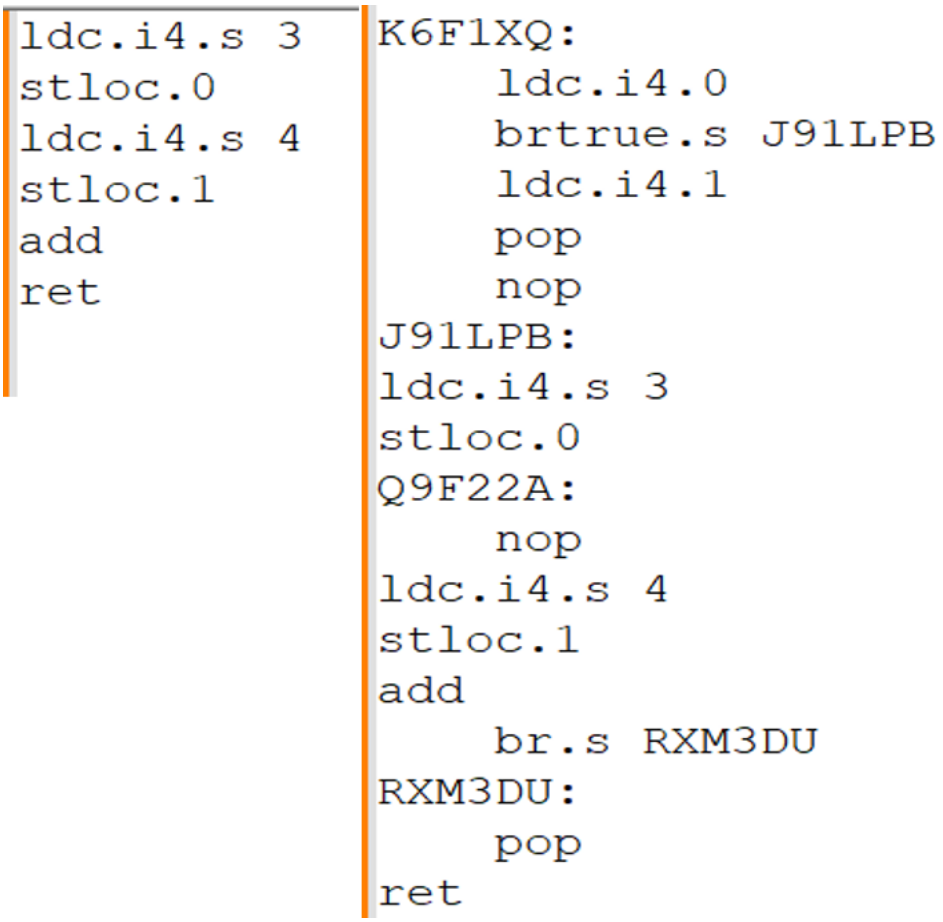

Рисунок 15 – Код до и после обфускации потока управления

Таким образом обфускация потока управления позволяет запутать для злоумышленника порядок выполнения инструкций программы.

3.5 Реализация пользовательского интерфейса

Реализованы следующие формы для удобной работы пользователя с приложением и описаны в виде таблиц ниже:

Таблица 5 – Элементы интерфейса главной формы

Идентификатор	Тип объекта	Назначение
centralwidget	QWidget	Главный контейнер для размещения всех элементов интерфейса.
textBrowser	QTextBrowser	Область для отображения содержимого исходного файла.
tableWidget	QTableWidget	Таблица для отображения информации о переменных в коде.
horizontalLayoutWidget	QWidget	Контейнер для горизонтального расположения элементов.
label	QLabel	Метка для файла.
label_2	QLabel	Метка для показа пути загруженного файла.
gridLayoutWidget	QWidget	Контейнер для размещения элементов управления.
gridLayout	QGridLayout	Сетка для размещения элементов управления, таких как кнопки.
pushButton	QPushButton	Кнопка для выполнения обфускации кода.
groupBox_2	QGroupBox	Группа для отображения обфусцированного кода.
textBrowser_2	QTextBrowser	Область для отображения обфусцированного кода.

Продолжение таблицы 5

pushButton_save_obf_file	QPushButton	Кнопка для сохранения обфусцированного кода в файл.
menubar	QMenuBar	Панель меню приложения.
menu	QMenu	Меню для работы с файлами (например, загрузка файла).
menu_2	QMenu	Меню для правки (поиск, замена и отмена).
menu_3	QMenu	Меню для настроек (настройка обфускатора).
menu_4	QMenu	Меню для помощи (документация и информация о программе).

Таблица 6 – Элементы интерфейса формы настроек

Идентификатор	Тип объекта	Назначение
gridLayoutWidget	QWidget	Контейнер для gridLayout
gridLayout	QGridLayout	Сетка для размещения элементов в таблице
checkBox_3	QCheckBox	Чекбокс «Активно» для лексической обфускации
label_2	QLabel	Метка «Лексическая обфускация»
checkBox_2	QCheckBox	Чекбокс «Активно» для обфускации хранения

Продолжение таблицы 6

label_3	QLabel	Метка «Обфускация хранения»
checkBox_4	QCheckBox	Чекбокс «Активно» для логирования
label_4	QLabel	Метка «Логирование»
formLayout	QFormLayout	Форматированный контейнер для кнопок
pushButton	QPushButton	Кнопка «Применить»
pushButton_2	QPushButton	Кнопка «Отмена»

Таблица 7 – Элементы интерфейса формы для работы с nuget пакетами

Идентификатор	Тип объекта	Назначение
buttonBox	QDialogButtonBox	Кнопки подтверждения и отмены действия в диалоговом окне.
formLayoutWidget	QWidget	Контейнер для формы ввода путей до инструментов ILDASM и ILASM.
label	QLabel	Метка для ввода пути до ILDASM.
lineEdit	QLineEdit	Поле ввода пути до ILDASM.
label_2	QLabel	Метка для ввода пути до ILASM.

Продолжение таблицы 7

lineEdit_2	QLineEdit	Поле ввода пути до ILASM.
label_3	QLabel	Метка для ввода пути к файлу для разборки (ILDASM).
lineEdit_3	QLineEdit	Поле ввода пути к файлу для разборки (ILDASM).
pushButton	QPushButton	Кнопка для выполнения разборки файла с ILDASM.
pushButton	QPushButton	Кнопка для выполнения разборки файла с использованием ILDASM.
label_4	QLabel	Метка для ввода пути к файлу для сборки (ILASM).
lineEdit_4	QLineEdit	Поле ввода пути к файлу для сборки (ILASM).
pushButton_2	QPushButton	Кнопка для выполнения сборки файла с использованием ILASM.

Экранные формы приложения обеспечивают удобство взаимодействия с инструментами и настройками. Главная форма предоставляет доступ к основным функциям обфускации, а форма настроек позволяет пользователю гибко управлять параметрами, включая активацию обфускации и логирования. Форма для

работы с ILDASM и ILASM интуитивно упрощает управление процессами разборки и сборки IL-кода, делая приложение доступным и эффективным для выполнения задач.

3.6 Тестирование приложения

Работа пользователя начинается с главной формы, которая показана ниже на рисунке.

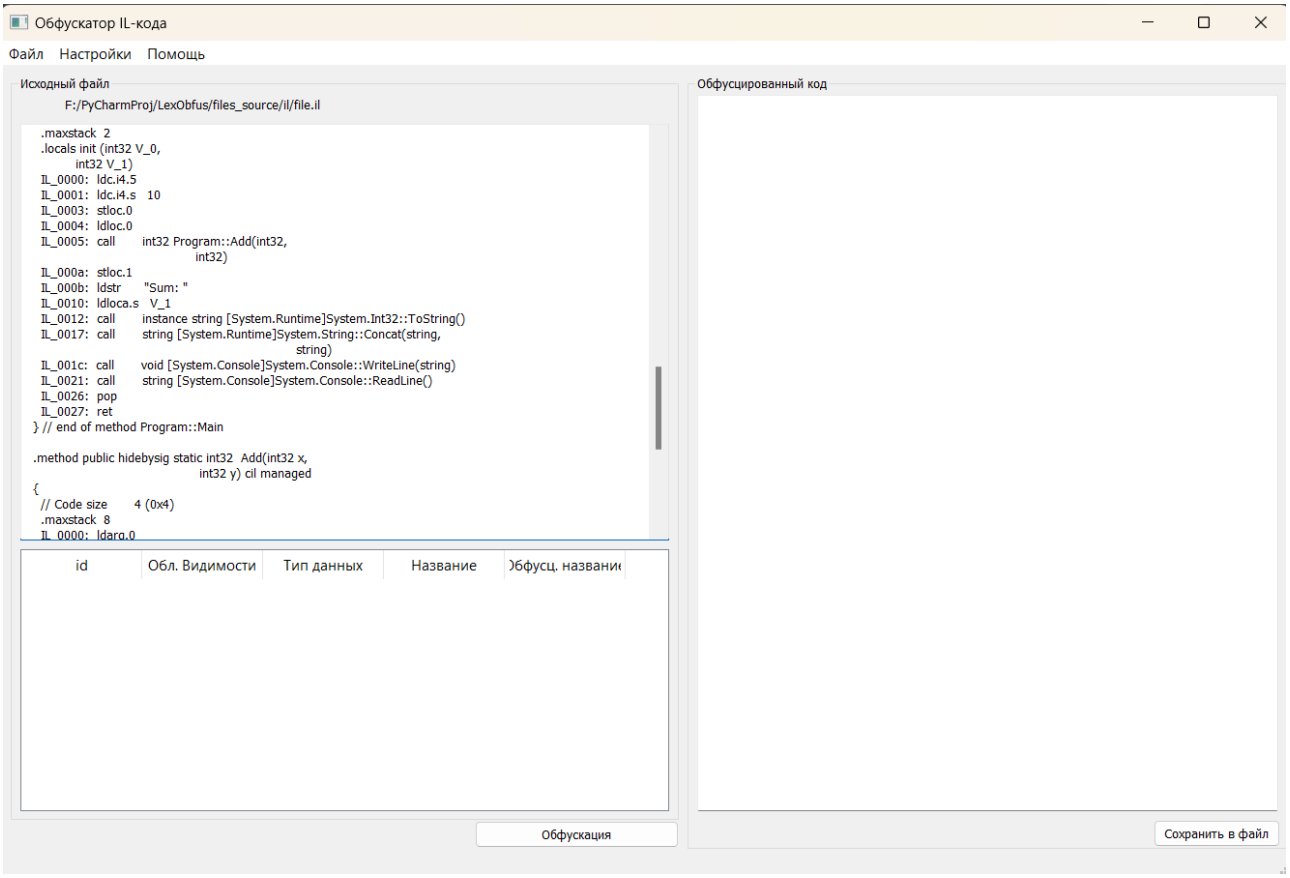

Рисунок 16 – Главная форма приложения

Пользователь выбирает действие «Загрузить файл» из пункта меню «Файл» и в диалоговом окне выбирает файл с il-кодом. По умолчанию файл с исходным il кодом находится в папке files_source/il в рабочей директории проекта.

Диалоговое окно выбора файла показано на рисунке 17.

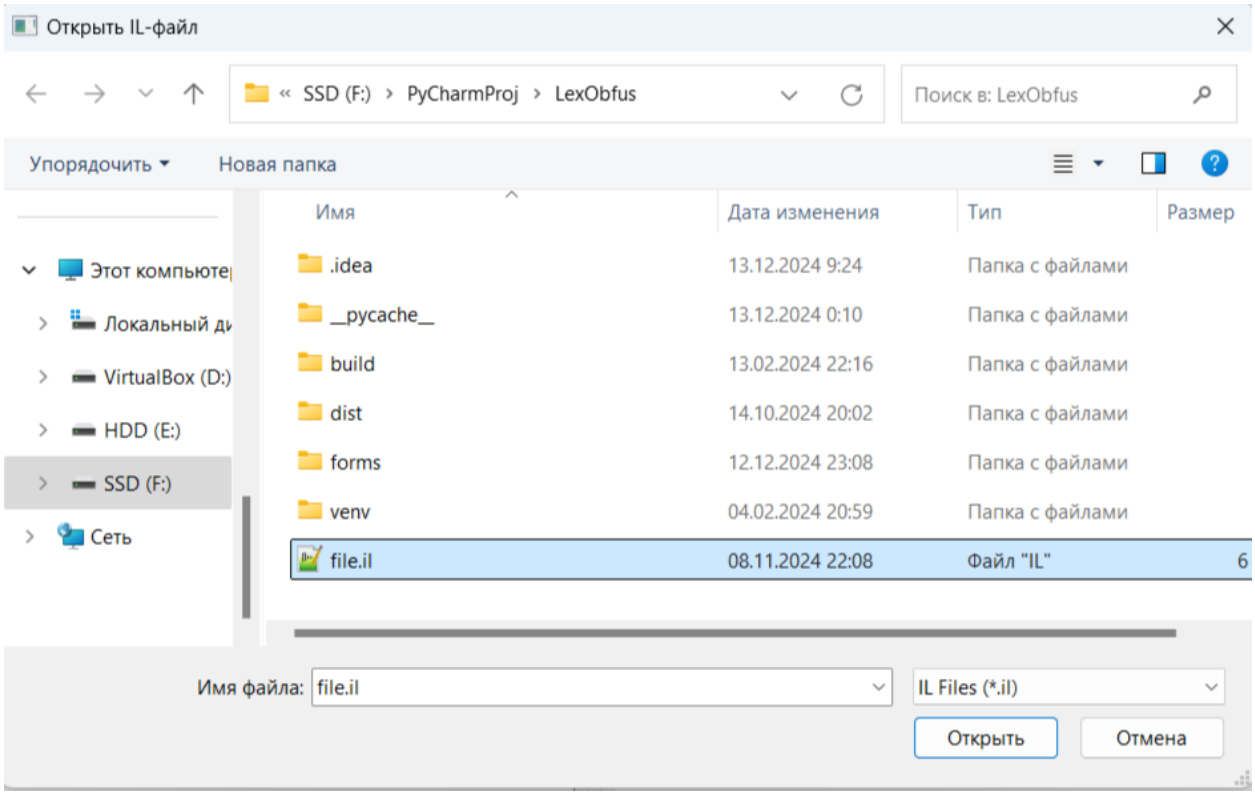

Рисунок 17 – Диалоговое окно загрузки файла

После успешной загрузки файла, его текст добавляется в левый контейнер. Текст можно пролистывать.

Затем пользователь открывает окно настроек через действие «Настройки обфускатора», где отмечает нужные ему параметры обфускации в соответствующих блоках: Лексическая обфускация, Обфускация хранения, Обфускация потока управления, Логирование.

Каждый из блоков содержит настраиваемые параметры, определяющие поведение конкретных алгоритмов обработки IL-кода.

Окно настроек показано на рисунке 18.

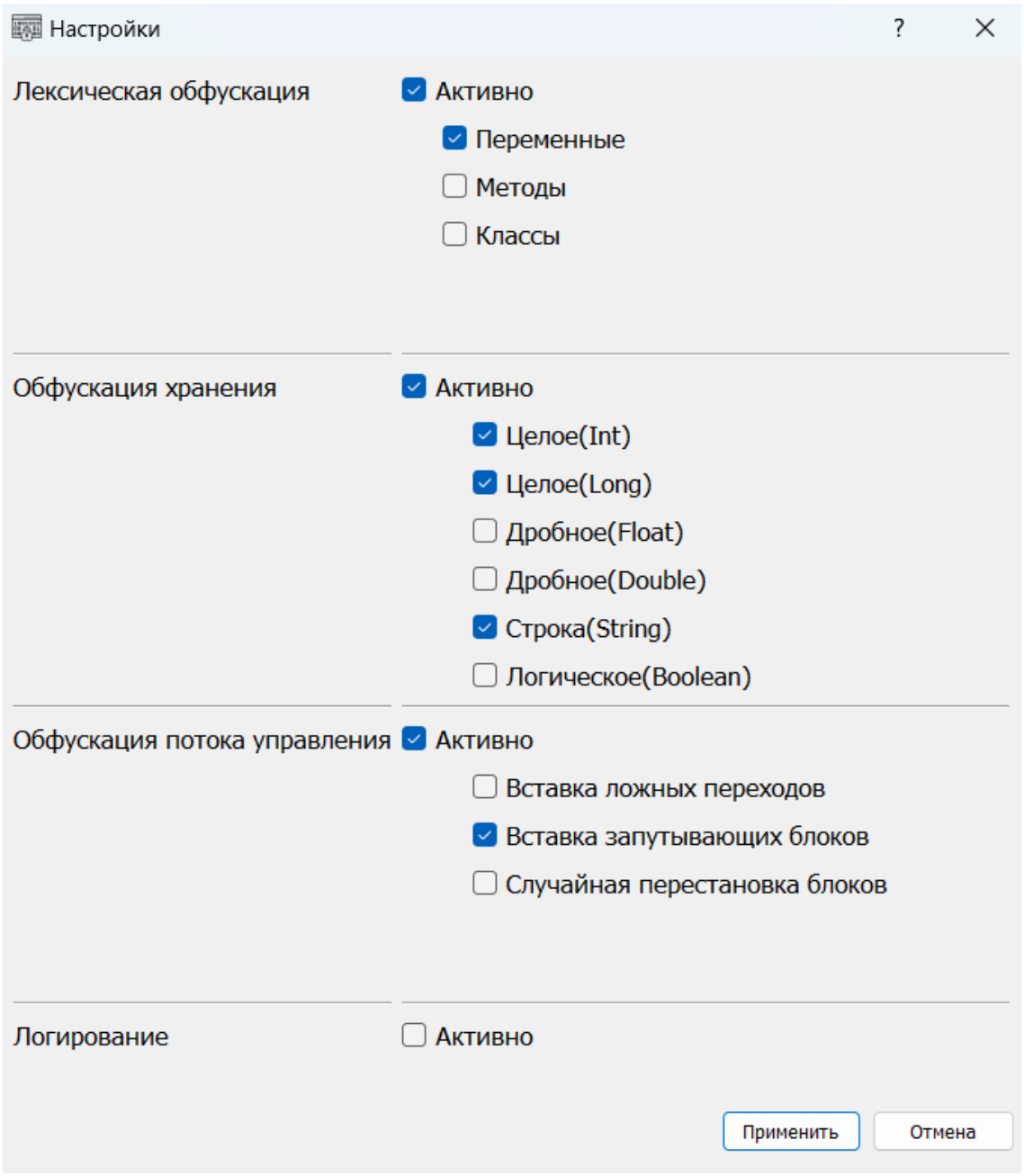

Рисунок 18 – Окно настроек

После выбора необходимых параметров из настроек обфускатора пользователь закрывает текущее диалоговое окно и нажимает кнопку «Обфускация» на главной форме и код модифицируется в соответствии с правилами обфускации.

Результат манипуляций над IL кодом показан на рисунке 19.

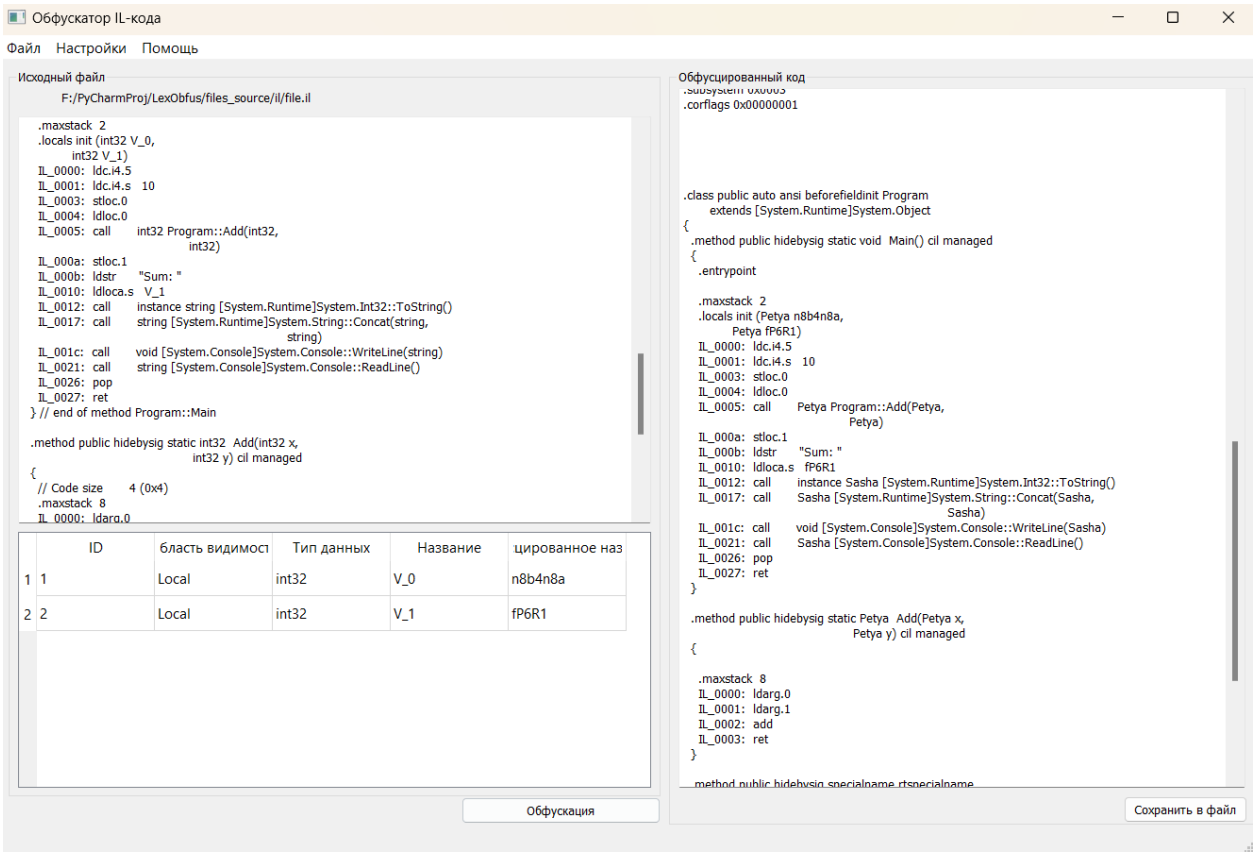

Рисунок 19 – Главная форма после применения обфускации

Далее модифицированный файл сохраняется через диалоговое окно аналогичное окну загрузки файла. При успешной операции выводится информационное сообщение, показанное на рисунке ниже.

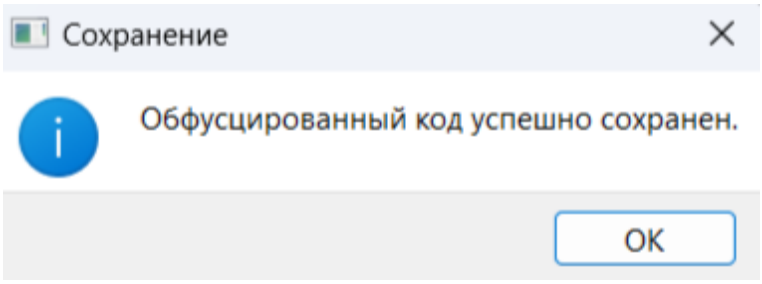

Рисунок 20 – Сообщение об успешном сохранении

Для работы с пакетами для сборки и разборки файла il нужно перейти в меню «Настройки» и выбрать «Сборка и разборка». Появится диалоговое окно, где прописывается путь до nuget пакетов – ILASM и ILDASM.

Также приписываются пути для дизассемблирования исполняемого файла (разборки) и его обратной сборки. Окно показано на рисунке ниже.

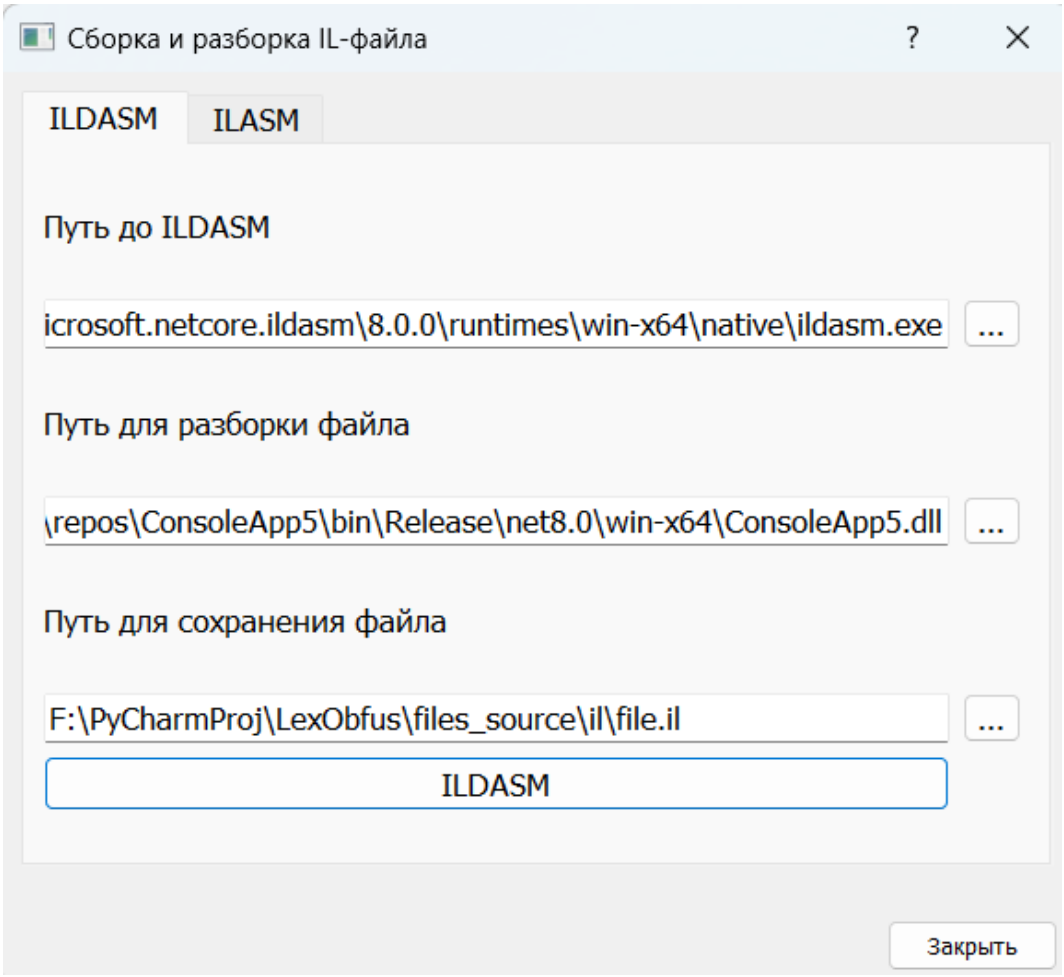

Рисунок 21 – Форма для работы с nuget пакетами

ILASM и ILDASM используют системные команды Windows для выполнения своих функций, поэтому у пользователя должна быть обязательно установлена платформа .NET, а также эти вышеупомянутые пакеты.

Модуль логирования отвечает за фиксацию всех значимых действий, выполняемых пользователем в приложении Simple IL-fuscator. Запись осуществляется в текстовый файл формата log_дд-мм-гггг.log, где в имени файла указывается текущая дата. Это позволяет удобно организовать логи по дням и быстро находить нужные записи.

Для просмотра зафиксированных событий пользователь может открыть окно журнала через главное меню программы. При этом по умолчанию

отображаются действия, совершённые в текущий день. В интерфейсе журнала доступны дополнительные возможности: просмотр записей из другого лог-файла за предыдущие дни, а также очистка текущего журнала при необходимости.

Результат записи действий пользователя показан ниже на рисунке.

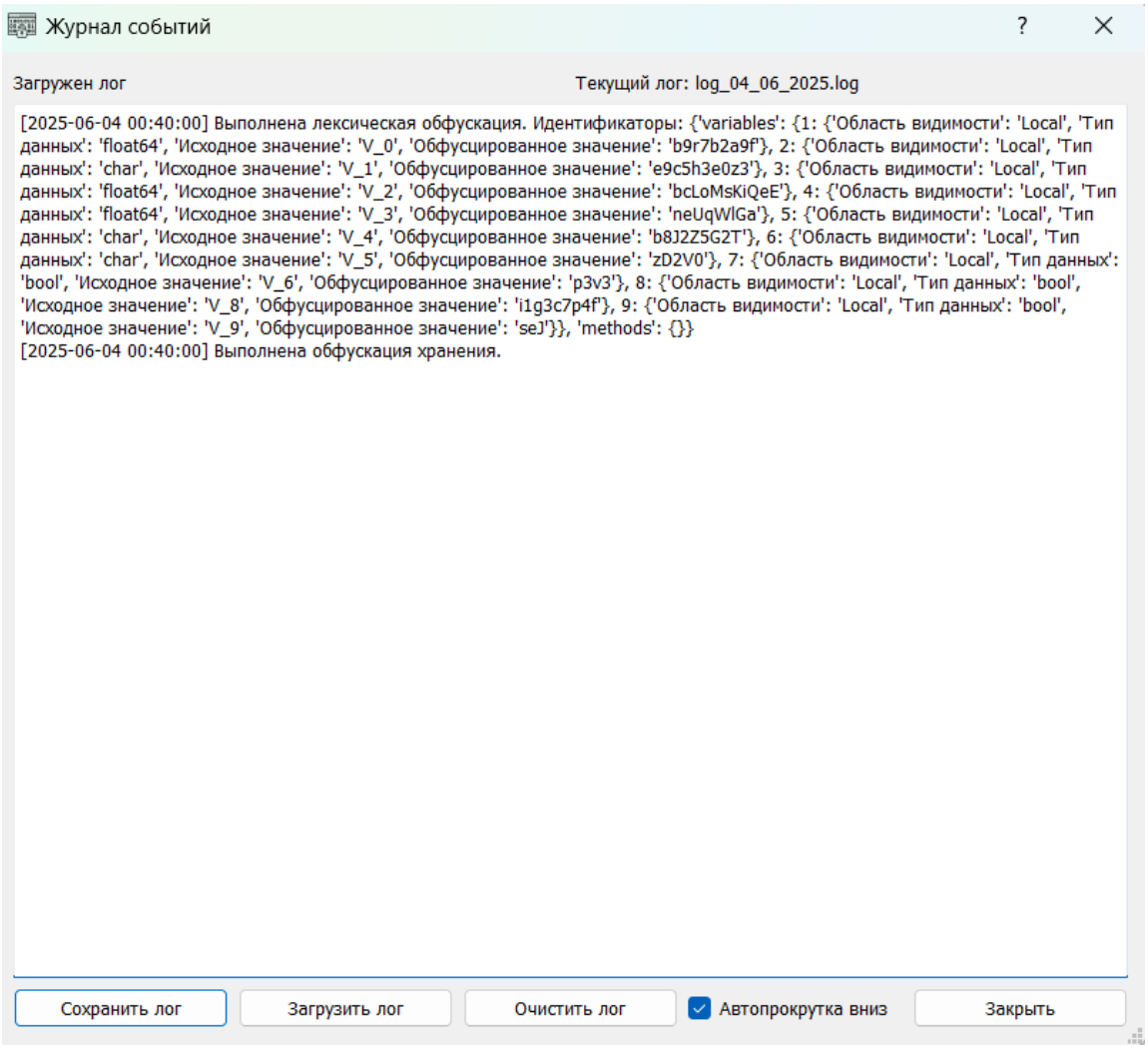

Рисунок 22 – Лог файл

Работа пользователя с приложением Simple IL-fuscator завершена.

ЗАКЛЮЧЕНИЕ

В процессе выполнения магистерской диссертации была решена задача повышения защищённости программного кода .NET-приложений от реверс-инжиниринга. В результате проведённого исследования существующих угроз, методов анализа и декомпиляции промежуточного IL-кода, а также современных подходов к обфускации, было разработано прикладное программное средство, реализующее обфускацию кода на уровне промежуточного представления.

Созданное приложение представляет собой инструмент, обеспечивающий затруднение анализа IL-кода с помощью комбинации различных техник: лексической обфускации, обфускации хранения и потока управления. При этом основным приоритетом в проектировании была сохранность исходной логики приложения и минимизация влияния на производительность. Для удобства использования специалистами в области безопасной разработки и разработчиками .NET-приложений был реализован графический интерфейс пользователя, позволяющий работать с кодом в наглядной и доступной форме.

Разработка прошла этапы проектирования архитектуры, реализации ключевых модулей, а также тестирования на предмет корректности функционирования и устойчивости к анализу. Проведённая оценка эффективности показала, что реализованные техники позволяют значительно усложнить статический и динамический анализ, при этом не нарушая логики работы приложений.

Таким образом, цель диссертации – создание программного инструмента для защиты кода .NET-приложений методом обфускации – достигнута. Решены все поставленные задачи, включая анализ существующих подходов, формализацию требований к системе, проектирование архитектуры, реализацию обфускационных модулей и тестирование готового продукта.

Разработанное приложение обладает практической ценностью и может применяться как в учебных целях, так и в процессе безопасной разработки программного обеспечения. В перспективе возможна его доработка с целью интеграции новых техник защиты и поддержки новых версий платформы .NET.

Научные публикации:

1 Степаненко В.А. Разработка приложения для защиты программного кода. Молодежь XXI века: шаг в будущее: матер. XXV регион. науч.-практ. конф. (Благовещенск, 22 мая 2024 г.) : в 2 томах. - Благовещенск : ФГБОУ ВО Амурская ГМА Минздрава России, 2024. – С. 478-479.

2 Степаненко В.А., Фомин Д.В., Никифорова Л.В. Разработка программного инструмента защиты IL-кода. Прикладная математика: современные проблемы математики, информатики и моделирования: материалы VI Всероссийской научно-практической конференции, молодых ученых, Том I, Краснодар, 15-21.04.2024 г., КубГУ. Издательство: Краснодарский ЦНТИ - филиал ФГБУ «РЭА» Минэнерго России, 2024. – С. 298-301.

БИБЛИОГРАФИЧЕСКИЕ ССЫЛКИ

1 Руководство по MSIL [Электронный ресурс]. URL: https://metanit.com/sharp/msil/ (дата обращения: 07.12.2024).

2 Документация по .NET [Электронный ресурс]. URL: https://learn.microsoft.com/dotnet (дата обращения: 07.12.2024)

3 Пимкин М. Е., Евдокимова В. В., Фомин Д. В. Лексическая обфускация языков .NET на уровне IL-кода // Вестник АмГУ, выпуск 101, 2023. Благовещенск: Амурский государственный университет, 2023. С. 158–164.

4 Иванников В. Реализация запутывающих преобразований в компиляторной инфраструктуре LLVM. [Электронный ресурс]. URL: https://www.ispras.ru/proceedings/docs/2014/26/1/isp_26_2014_1_327.pdf (дата обращения: 20.04.2025).

5 Ковров А. И., Ляпина Е. П., Савин Л. А. Инструменты реверс-инжиниринга и транскомпиляции: учебное пособие для вузов. М.: Издательство РУТ, 2024. С. 71.

6 Бортничук А. В. Обфускация кода приложений .net // Сбор. тез. докл. науч.-практ. конф. студ. Курган: Курганский государственный университет, 2021. С. 281–282.

7 Мельник Е. В., Минаев Д. П. Обзор гибкой методологии разработки программного обеспечения Scrum // Программная инженерия: современные тенденции развития и применения : сборник материалов Всероссийской конференции. Курск: Закрытое акционерное общество «Университетская книга», 2017. С. 131–137.

8 Великовский Д. А. Анализ современных средств создания графического интерфейса // Планирование, проведение и толкование итогов научных исследований : Сборник статей Международной научно-практической конференции. В 2-х частях, Воронеж, 27 мая 2024 года. Уфа: ООО «Омега сайнс», 2024. С. 65–68.

9 Control-flow graph [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Control-flow_graph (дата обращения: 15.03.2025).

10 Дрянкова Д. А. Регулярные выражения в языке программирования Python // Студенческий вестник, 2023. № 47–10(286). С. 52–56.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1 Астафурова, О. А. Практикум по программированию на языке Python / О. А. Астафурова, Т. А. Омельченко. – Волгоград : Издательско-полиграфический центр ВИУ РАНХиГС, 2022. – 56 с.

2 Бортничук, А. В. Обфускация кода приложений .net // Сбор. тез. докл. науч.-практ. конф. студ. – Курган: Курганский государственный университет, 2021. – С. 281-282.

3 Буйневич, М. В. Аналитическое моделирование работы программного кода с уязвимостями / М. В. Буйневич, К. Е. Израилов // Вопросы кибербезопасности, 2020. – № 3(37). – С. 2-12.

4 Буйневич, М. В. Основы кибербезопасности: способы анализа программ: Учебное пособие для обучающихся высших учебных заведений по укрупненной группе специальностей и направлений «Информационная безопасность» / М. В. Буйневич, К. Е. Израилов. – СПб.: Санкт-Петербургский университет Государственной противопожарной службы Министерства Российской Федерации по делам гражданской обороны, чрезвычайным ситуациям и ликвидации последствий стихийных бедствий имени Героя Российской Федерации генерала армии Е.Н. Зиничева, 2022. – 92 с.

5 Варичев, И. А. Создание графических интерфейсов в Python при помощи библиотеки PyQT / И. А. Варичев // Цифровые технологии: настоящее и будущее : Сборник статей Национальной научно-практической конференции с международным участием, Тольятти, 16 ноября 2022 года. – Тольятти: Частное образовательное учреждение высшего образования «Тольяттинская академия управления», 2022. – С. 36-40.

6 Великовский, Д. А. Анализ современных средств создания графического интерфейса / Д. А. Великовский // Планирование, проведение и толкование итогов научных исследований : Сборник статей Международной научно-

практической конференции. В 2-х частях, Воронеж, 27 мая 2024 года. – Уфа: ООО «Омега сайнс», 2024. – С. 65-68.

7 Величко, В. В. Защита .NET кода методами обфускации / В. В. Величко // Информатизация образования, 2006. – № 2(43). – С. 71-82.

8 Гагарина, Л. Г. Введение в теорию алгоритмических языков и компиляторов : Учебное пособие для студентов, аспирантов, научных сотрудников, преподавателей высших учебных заведений / Л. Г. Гагарина, Е. В. Кокорева. – 2-е издание, переработанное и дополненное. – М.: Общество с ограниченной ответственностью «Научно-издательский центр ИНФРА-М», 2025. – 207 с.

9 ГОСТ 19.701-90 (ИСО 5807-85) ЕСПД. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения: Межгосударственный стандарт: дата введения 1992-01-01 / Федеральное агентство по техническому регулированию. – Изд. официальное. – М.: Стандартинформ, 2010. – 23 с.

10 ГОСТ Р 51904-2002 Программное обеспечение встроенных систем. Общие требования к разработке и документированию.

11 ГОСТ Р 56939-2016 Защита информации. Разработка безопасного программного обеспечения. Общие требования. [Электронный ресурс] // Электронный фонд правовых и нормативно-технических документов. – Режим доступа: https://docs.cntd.ru/document/1200135525. – 12.05.2024.

12 ГОСТ Р 59793-2021. «Информационные технологии. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Стадии создания».

13 ГОСТ Р ИСО/МЭК 12207-99 Информационная технология. Процессы жизненного цикла программных средств.

14 ГОСТ Р ИСО/МЭК 12207-2010. «Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств».

15 Гупало, А. В. Анализ безопасности Python-пакетов в репозитории PyPI / А. В. Гупало // Актуальные вопросы информационной безопасности и защиты

информации : Сборник научных статей. – СПб.: Санкт-Петербургский государственный экономический университет, 2024. – С. 55-61.

16 Данилевич, В. В. Модель выполнения кода в среде CLR / В. В. Данилевич, О. В. Рычка // Программная инженерия: методы и технологии разработки информационно-вычислительных систем (ПИИВС-2022) : Сборник материалов IV Международной научно-практической конференции, Том 1. – Донецк: Донецкий национальный технический университет, 2022. – С. 39-45.

17 Документация по .NET [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/dotnet. – 07.12.2024.

18 Дрянкова, Д. А. Регулярные выражения в языке программирования Python / Д. А. Дрянкова // Студенческий вестник, 2023. – № 47-10(286). – С. 52-56.

19 Зубаль, А. М. Особенности взаимодействия Python и pyqt / А. М. Зубаль // XXVI Всероссийская студенческая научно-практическая конференция Нижневартовского государственного университета, Нижневартовск, 10–11 апреля 2024 года. – Нижневартовск: Нижневартовский государственный университет, 2024. – С. 324-329.

20 Иванников В. Реализация запутывающих преобразований в компиляторной инфраструктуре LLVM. [Электронный ресурс]. – Режим доступа: https://www.ispras.ru/proceedings/docs/2014/26/1/isp_26_2014_1_327.pdf – 20.04.2025.

21 Ивченкова, Ю. С. Виды и способы обфускации / Ю. С. Ивченкова, М. К. Савкин // Электронный журнал: наука, техника и образование. – 2016. – № 2(6). – С. 60-66.

22 Казарин, О. В. Надежность и безопасность программного обеспечения : учебное пособие для вузов / О. В. Казарин, И. Б. Шубинский. – М.: Издательство Юрайт, 2024. – 342 с.

23 Киселева, Е. А. Обзор GUI Framework на языке программирования Python / Е. А. Киселева // Постулат. – 2020. – № 1(51). – С. 135.

24 Ковров, А. И. Инструменты реверс-инжиниринга и транскомпиляции: учебное пособие для вузов. / А. И. Ковров, Е. П. Ляпина, Л. А. Савин, В. Г. Сидоренко, И. Д. Соколов. – М.: Издательство РУТ, 2024. – 71 с.

25 Козлов, С. В. Применение регулярных выражений для обработки текстовых данных / С. В. Козлов, А. В. Светлаков // International Journal of Open Information Technologies. – 2022. – Т. 10, № 9. – С. 82-89.

26 Коробейников, А. Г. Анализ методов обфускации / А. Г. Коробейников, И. М. Кутузов, П. Ю. Колесников // NB: Кибернетика и программирование, 2012. – № 1. – С. 31-37.

27 Косырев, Е. А. Проектирование пользовательских интерфейсов: современные подходы и лучшие практики / Е. А. Косырев, Д. О. Лутаев, Т. А. Коваленко // Актуальные проблемы технических и естественных наук в России и за рубежом : сборник научных статей. – М.: Издательский дом «НАУКА И СОЦИУМ», 2024. – С. 23-25.

28 Кубрин, Г. С. Выявление логических уязвимостей программного обеспечения на уровне графового представления кода. / Г. С. Кубрин, Д. П. Зегжда // Материалы докладов научно-практической конференции. – СПб.: Издательство СПбПУ, 2024. – С. 76-79.

29 Лебедева, Д. А. Сравнительный анализ программных средств обфускации // Инновации. Наука. Образование, 2021. – № 48. – С.1699-1704.

30 Малявский, Р. Д. Обфускация кода программного обеспечения / Р. Д. Малявский // Молодёжная научная школа кафедры «Защищённые системы связи», 2021. – №1(3). – С. 20-23.

31 Мельник, Е. В. Обзор гибкой методологии разработки программного обеспечения Scrum / Е. В. Мельник, Д. П. Минаев // Программная инженерия: современные тенденции развития и применения : сборник материалов Всероссийской конференции. – Курск: Закрытое акционерное общество «Университетская книга», 2017. – С. 131-137.

32 Монаппа, К. А. Анализ вредоносных программ / пер. с англ. Д. А. Беликова. – М.: ДМК Пресс, 2019. – 322 с.

33 Мукминов, Д. В. Создание GUI приложений на языке программирования Python / Д. В. Мукминов, А. Р. Шакиров // Приоритетные направления инновационной деятельности в промышленности : Сборник научных статей VI международной научной конференции – Казань: Общество с ограниченной ответственностью «КОНВЕРТ», 2021. – С. 133-135.

34 Ненашев, А. В. Обфускация данных в децентрализованных публичных системах // Математические методы в технологиях и технике. – 2024. – № 7. – С. 55-61.

35 Пимкин, М. Е. Лексическая обфускация языков .NET на уровне IL-кода / М. Е. Пимкин // День науки : Материалы XXXII научной конференции Амурского государственного университета, Благовещенск, 20 апреля 2023 года. – Благовещенск: Амурский государственный университет, 2023. – С. 66-67.

36 Пронина, С. Р. Регулярные выражения / С. Р. Пронина, И. А. Гордеева // Дни науки студентов Владимирского государственного университета имени Александра Григорьевича и Николая Григорьевича Столетовых : Сборник материалов научно-практических конференций – Владимир: Владимирский государственный университет имени Александра Григорьевича и Николая Григорьевича Столетовых, 2024. – С. 1768-1775.

37 Прохоренок, Н. А. Python 3 и PyQt : Разработка приложений / Н. А. Прохоренок. – СПб.: БХВ-Петербург, 2011. – 704 с.

38 Романов Е. Л. Основы построения трансляторов: Конспект лекций [Электронный ресурс] / Е.Л. Романов. – Новосибирск: НГТУ, 2001. – Режим доступа: https://gigabaza.ru/doc/67356pall.html. – 21.04.2025.

39 Руководство по MSIL [Электронный ресурс]. – Режим доступа: https://metanit.com/sharp/msil/. – 07.12.2024.

40 Савин, Л. А. Применение инструментов обратной разработки для выявления недекларированных возможностей программного обеспечения систем управления. / Л. А. Савин // Проблемы управления безопасностью сложных систем: материалы XXXI международной конференции. – М.: Институт проблем управления им. В. А. Трапезникова РАН, 2023. – С. 412-419.

41 Сударикov, Г. В. Основы программирования на Python / Г. В. Сударикov // Программирования в Python : учебное пособие. – Бургас : Институт гуманитарных наук, экономики и информационных наук, 2025. – С. 5-119.

42 Тахаутдинов, А. Р. Методы запутывания при защите программного кода / А. Р. Тахаутдинов, Ч. М. Лыонг Ха, А. А. Привалова // Молодежный вестник УГАТУ, 2021. – № 1(24). – С. 58-60.

43 Чернышев, С. А. Алгоритмы и структуры данных на Python / С. А. Чернышев. – М.: Общество с ограниченной ответственностью «Издательство «КноРус», 2024. – 328 с.

44 Чернышев, С. А. Принципы, паттерны и методологии разработки программного обеспечения: учебное пособие для вузов / С. А. Чернышев. – М.: Издательство Юрайт, 2021. – 176 с.

45 Control-flow graph [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/Control-flow_graph. – 15.03.2025.

46 Ildasm.exe (дизассемблер IL) [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/dotnet/framework/tools/ildasm-exe-il-disassembler. – 02.04.2025.

47 Ilasm.exe (ассемблер IL) [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/dotnet/framework/tools/ilasm-exe-il-assembler. – 02.04.2024.

48 Python 3.10 documentation [Электронный ресурс]. – Режим доступа: https://docs.python.org/3.10/. – 07.12.2024.

49 Qt Designer Manual [Электронный ресурс]. – Режим доступа: https://doc.qt.io/qt-6/qtdesigner-manual.html. – 01.10.2024.

50 Qt DOCUMENTATION [Электронный ресурс]. – Режим доступа: https://doc.qt.io/. – 04.10.2024.