

Санкт-Петербургский Государственный Политехнический Университет  
Институт информационных технологий и управления  
Кафедра измерительных информационных технологий

Проект допущен к защите

Зав. кафедрой

Г.Ф. Малыгина

« 6 » июня 2014 г.

**ДИПЛОМНЫЙ ПРОЕКТ**

**Тема: Внедрение элементов защиты в .NET приложения  
без необходимости модификации исходного кода**

Направление: 090900 — Информационная безопасность

Специальность: 090104 — Комплексная защита объектов информатизации

Выполнил:

студент гр. 53505/2



С.А. Гладышев

Руководитель:

канд. техн. наук, доцент




В.Ю. Сальников

Консультанты:

по экономической части

канд. экон. наук, доцент



И.Б. Корокин

по вопросам охраны труда

канд. физ.-мат. наук, доцент



Г.В. Струйков

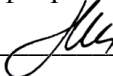
Санкт-Петербург  
2014

**ФГБОУ ВПО «САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ  
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»  
Кафедра «Измерительных информационных технологий»**

УТВЕРЖДАЮ

« 6 » июня 2014 г.

Зав кафедрой Г.Ф. Малыхина



**ЗАДАНИЕ  
на дипломное проектирование**

студенту Гладышеву Сергею Александровичу

**1. Тема проекта (работы)**

Внедрение элементов защиты в .NET приложения без необходимости модификации исходного кода

**2. Срок сдачи студентом законченного проекта (работы)**

10 июня 2014

**3. Исходные данные к проекту (работе)**

Исходными данными к проекту являются существующие технологии защиты приложений и представленный на рынке решения для защиты.

4. Содержание расчетно-пояснительной записки (перечень подлежащих разработке вопросов)

В расчетно-пояснительной записке представлены обзор аналогичных решений по защите приложений, рассмотрены их преимущества и недостатки, определено направление разработки. Описаны использованные инструментны и технологии, приведен алгоритм разработки и результаты тестирования созданного решения

5. Перечень графического материала (с точным указанием обязательных чертежей)

- Схема классов решения
- Алгоритмы функций по защите данных

6. Консультанты по проекту (с указанием относящихся к ним разделов проекта (работы))

Консультант по экономической части:

канд. экон. наук, доцент И.Б. Корокин.

Консультант по вопросам охраны труда:

канд. физ.-мат. наук, доцент Г.В. Струйков.

7. Дата выдачи задания

25 февраля 2014

Руководитель



Задание принял к исполнению

25.02.2014

(дата)



(подпись студента)

## **РЕФЕРАТ**

С. 81. Рис. 11. Табл. 7. Черт. 3

### **АСПЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ, ИНЪЕКЦИЯ КОДА, РЕФЛЕКСИЯ, ОТРАЖЕНИЕ, ЗАЩИТА ПАМЯТИ, МОДИФИКАЦИЯ СБОРКИ**

Разработано программное обеспечение для внедрения элементов защиты в .NET приложения. Используемые алгоритмы позволяют динамически модифицировать память приложения и тем самым, усложнять доступ и модификацию скрытой информации. Созданное решение является законченным продуктом и может использоваться для защиты в комплексе с остальными методами и технологиями.

Проведенное тестирование эффективности используемых методов защиты показало на практике работоспособность разработанного программного обеспечения.

В работе приведено экономическое обоснование проекта и освещены вопросы охраны труда, техники безопасности и проектирования производственной среды.

## **SUMMARY**

P. 81. Fig. 11. Tables 7. Plots 3

### **ASPECT-ORIENTED PROGRAMMING, CODE INJECTION, REFLECTION, MEMORY PROTECTION, ASSEMBLY MODIFICATION**

The software for the injection of security features to the .NET application was developed. The used algorithms permit to dynamically modify the application memory and thereby complicate the access and modification of the hidden information. Created solution is complete and can be used for protection in combination with other methods and technologies.

Conducted testing the effectiveness of used methods has shown protection efficiency of the developed software in practice.

In this project economic feasibility of the project and issues of health, safety and design of the working environment was presented.

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	9
ТЕХНИЧЕСКОЕ ЗАДАНИЕ.....	12
1. ОБЗОР ТЕХНОЛОГИЙ ЗАЩИТЫ.....	15
1.1. Стеганография и водяные знаки.....	15
1.2. Обфускация.....	17
1.3. Упаковка.....	19
1.4. Системы лицензирования.....	20
1.5. Защита от модификаций.....	20
1.6. Соккрытие данных .....	21
1.7. Комплексный подход.....	22
1.8. Обзор имеющихся решений.....	22
2. ТЕОРЕТИЧЕСКАЯ РАЗРАБОТКА.....	25
2.1. Аспектно-ориентированное программирование.....	25
2.2. CLR и .NET Framework.....	29
2.3. Отражения.....	32
2.4. Атрибуты.....	33
2.5. Метапрограммирование и Mono.Cecil .....	34
2.6. Сборка проектов при помощи MSBuild.....	36
2.7. Реверс-инжиниринг и проверка эффективности защиты .....	38
3. РЕАЛИЗАЦИЯ.....	41
3.1. Библиотека SecureField.....	42
3.2. MonoInjections и модификация сборок .....	46
3.3. Практическое использование.....	49
4. ТЕСТИРОВАНИЕ .....	51

4.1.	Объект испытаний.....	51
4.2.	Цель испытаний.....	51
4.3.	Требования к программе и программной документации.....	51
4.4.	Средства и порядок испытаний .....	51
4.5.	Методы испытаний .....	52
5.	ОРГАНИЗАЦИЯ РАБОЧЕГО МЕСТА .....	58
5.1.	Требования к помещениям для работы с ПЭВМ .....	58
5.2.	Требования к микроклимату, содержанию энтропинов и вредных химических веществ в воздухе .....	59
5.3.	Требования к уровням шума и вибрации .....	59
5.4.	Требования к освещению .....	60
5.5.	Требования к уровню электромагнитных полей .....	61
5.6.	Требования к визуальным параметрам ВДТ .....	62
6.	ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ .....	63
6.1.	Определение трудоемкости выполнения разработки.....	63
6.2.	Расчет затрат на разработку .....	64
	ЗАКЛЮЧЕНИЕ .....	68
	СПИСОК ЛИТЕРАТУРЫ.....	70
	Приложения .....	72

## ПЕРЕЧЕНЬ ТЕРМИНОВ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

**CLR** (Common Language Runtime) – общезыковая исполняющая среда. Исполняющая среда, интерпретирующая код на языке IL в байт-код, в который компилируются программы, написанные, в частности, на .NET-совместимых языках программирования.

**DLL** (Dynamic Link Library) – динамически подключаемая библиотека. Динамическая библиотека, позволяющая многократное использование различными программными приложениями.

**Easter Egg** («пасхально яйцо») – разновидность секрета, оставляемого в программном обеспечении создателями, особенность которого состоит в том, что его содержание, как правило, не вписывается в общую концепцию.

**IL** (CIL, MSIL, Common Language Runtime) – промежуточный язык, разработанный фирмой Microsoft для платформы .NET Framework.

**ILASM/ILDASM** (IL assembler/IL disassembler) – служебные программы, ассемблер и дизассемблер, поставляемые совместно с VisualStudio.

**JIT-компиляция** (Just-in-time) – динамическая компиляция, технология увеличения производительности программных систем, использующих байт-код, путём компиляции его в машинный код непосредственно во время работы программы.

**MSBuild** – платформа сборки проекта, разработанная Microsoft и позволяющая автоматизировать сборку и назначить дополнительные задачи.

**WinApi-приложение** (Windows application programming interfaces) – приложения, использующие набор базовых функций интерфейсов программирования приложений операционных систем семейств Microsoft Windows.

**ПО** – программное обеспечение.

**Сборка** (англ. assembly) – двоичный файл, содержащий управляемый код (метаданные и инструкции языка IL), получаемый на выходе компилятора.



## ВВЕДЕНИЕ

В наши дни информационные технологии являются основной движущей силой прогресса и захватывают все большие области современной жизни. Информационные технологии – это совокупность программно-технических средств вычислительной техники и способов их применения для обработки информации в различных областях. В связи с таким широким их распространением, серьезно встает вопрос защиты обрабатываемых и хранимых данных. Ведь если раньше, вы были уверены в том, что никто не получит доступ к вашим документам, лежащим в сейфе, без ключа или пароля. Теперь же большая часть информации хранится в электронном виде и необходимы уже другие способы защиты.

Этот принцип действует и для разрабатываемых программ. Если раньше, программы были по большей части низкоуровневыми и хранились в виде машинных кодов конкретной архитектуры, то с течением времени появляется все больше различных высокоуровневых инструментов для облегчения разработки, развиваются и усложняются сами языки программирования.

Чтобы снять с разработчика обязанности прямой работы с памятью, контроля типов, а также сделать программу более гибкой для различных платформ и языков, приложения выполняются специальной исполняющей средой (виртуальной машиной). Примеры таких виртуальных машин: JVM (виртуальная машина Java), CLR (общезыковая исполняющая среда), Dalvik Virtual Machine (часть мобильной платформы Android). Эти средства значительно упрощают разработку, но они разработаны так, что требуют трансляции приложения в промежуточный байт-код (промежуточный язык, язык высокоуровневого ассемблера), который затем в момент выполнения компилируется в машинный код JIT-компиляторами (компиляция «на лету»). Вследствие чего, приложения хранятся на диске в виде инструкций промежуточного языка, который легко декомпилируется обратно в исходные

коды с помощью различных инструментов, таких как .NET Reflector, FernFlower (декомпилятор Java) и другие.

Соответственно существуют простые способы обхода лицензирования, получения исходных кодов и модификации под свои нужды незащищенных приложений написанных на языках .Net (и других подобных языков программирования, таких как Java). При этом для них не существует абсолютной защиты, как и не существует такой защиты для низкоуровневых приложений (например Win-32, написанных на C++), отсюда исходит основной принцип защиты программных продуктов:

*Защита приложения лишь увеличивает время его взлома и в идеале, защита должна окупить себя, т.е. время, затраченное на защиту приложения должно быть сопоставимо со временем, затраченным на взлом этой защиты.*

Существуют несколько основных технологий или методик защиты .Net приложений. Их можно разделить на две части: те, которые работают с кодом программы и те, которые работают с памятью.

Для сокрытия кода используются различные обфускаторы и упаковщики. Обфускаторы используют для защиты запутывание кода (англ. control flow), переименование методов и классов, шифрование ресурсов, добавление невалидных конструкций и др. Упаковщики модифицируют сборку, и на выходе получается WinApi-приложение, содержащие в себе код сборки и модуль распаковки. Упаковщики и обфускаторы могут применяться одновременно.

В дополнение к защите кода может применяться защита данных, хранящихся в памяти во время выполнения программы. Здесь возможно множества решений, которые зависят от конкретного приложения и его архитектуры.

Так же существует еще один метод защиты – модификация кода во время выполнения приложения. С помощью перехвата вызовов функций (англ. hooking) можно изменять поведение программы, что дает возможность более сложной защиты приложений.

Данная работа направлена на рассмотрение существующих решений, выявления преимуществ и недостатков и разработку нового решения, основанного одновременно на нескольких методах и технологиях, как защиты, так и упрощения ее внедрения в готовый проект.

## ТЕХНИЧЕСКОЕ ЗАДАНИЕ

### 1. Введение

Данный программный продукт предназначен для защиты данных, используемых в приложениях, написанных с использованием .NET Framework. Программа модифицирует целевое приложение и тем самым защищает данные, находящиеся в памяти во время его работы.

### 2. Основания для разработки

Разработка решения ведется на основании обзора и анализа существующих решений в сфере защиты программного обеспечения в рамках текущего дипломного проекта.

### 3. Назначение разработки

Программа представляет собой отдельное приложение, позволяющее модифицировать .NET сборки путем внедрения кода, обеспечивающего определенную защиту. Внесенные изменения должны предоставлять эффективную защиту данных приложения, хранящихся в памяти и по возможности препятствовать их модификации.

### 4. Требования к программному продукту

Требования к функциональным характеристикам.

Программа должна уметь работать с запускаемыми exe файлами и библиотеками функций DLL. Цель модификации – внедрение конструкций, отвечающих за выполнение логики по защите данных.

Логика защиты данных описана в отдельной DLL библиотеке, которая должна быть подключена к модифицируемому приложению и поставляться вместе с ним конечному пользователю.

Входными данными для приложения является валидная .NET сборка, транслированная в промежуточный язык IL. На выходе необходимо получить приложение с тем же функционалом и не нарушенной логикой работы.

Должна быть предусмотрена возможность модификации приложения в ходе сборки проекта с использованием платформы MSBuild.

#### Требования к надежности

При некорректном завершении работы приложения по каким-либо причинам, состояние модифицируемой сборки должно быть восстановлено до исходного.

#### Требования к составу и параметрам технических средств

Минимальные технические характеристики:

- Windows XP SP1 и выше;
- Microsoft Visual Studio 2005 и выше;
- .NET Framework 2.0 и выше.

#### Требования к информационной и программной совместимости

Модифицируемые сборки должны быть написаны на платформе .NET Framework версии 2.0 и выше на следующих языках программирования: C#, F#, Visual Basic.

#### Специальные требования

Специальных требований к характеристикам программы не предъявляется.

### **5. Технико-экономические показатели**

Разрабатываемый продукт является бесплатным для распространения.

Время, затраченное на разработку, составит 600 человеко-часов. На составление технической документации необходимо выделить 120 часов. Так же 30 часов необходимы для подготовки к началу разработки.

### **6. Стадии и этапы разработки**

#### Эскизный проект – 20 часов

Разрабатывается структура программы, алгоритм модификации сборок и алгоритм защиты данных. Определяется система взаимодействия с модифицируемыми приложениями.

### Тестовый проект – 65 часов

Создание рабочего решения, позволяющего защищать приложения с помощью простого алгоритма. Отсутствие проверок и восстановления после ошибок.

### Технический проект – 40 часов

Написание эффективного алгоритма по защите данных, добавление обработки ошибок и восстановления после некорректного завершения работы.

### Рабочий проект – 35 часов

Реализация автоматизации процесса, интеграция с процессом сборки модифицируемых решений. Подготовка контрольно-отладочного примера.

### Тестирование – 60 часов

Тестирование работоспособности решения. Попытка противодействовать используемой защите.

## **7. Порядок контроля и приемки**

Контроль и приемка разработки осуществляются на основе проведенных испытаний контрольно-отладочных примеров. При этом проверяется выполнение всех функций программы.

## 1. ОБЗОР ТЕХНОЛОГИЙ ЗАЩИТЫ

В данной главе будут рассмотрены основные технологии защиты программного обеспечения, исполняемого в среде .Net и приведены популярные существующие решения для этих областей. По завершению этой главы будет проведен анализ преимуществ и недостатков рассмотренных средств защиты.

### 1.1. Стеганография и водяные знаки

Стеганография – это метод защиты программного обеспечения (ПО) путем встраивания секретной информации в приложение (код или данные). Это делается для идентификации владельца ПО и позволяет избежать или детектировать создание нелегальных копий приложений. Подобное внедрение водяных знаков (англ. watermarks) позволяет владельцам лицензионного ПО определить подлинность копии путем извлечения водяных знаков[1].

По методу встраивания водяные знаки можно разделить на две категории: статические (англ. static) и динамические (англ. dynamic). К статическим относятся: модификация данных (строк) и кода. Для детектирования таких водяных знаков не обязательно запускать приложение, но они очень легко снимаются. К динамическим относятся: «пасхальные яйца» (англ. dynamic Easter Egg watermarks) – изменения в незаметном или редко используемом месте программы, изменения алгоритма работы программы (англ. dynamic execution trace watermarks) или изменение динамических данных, доступных извне (англ. dynamic data structure watermarks), например, глобальных переменных или областей памяти[1].

Чтобы понять эффективность тех или иных мер защиты, необходимо понимать потенциальные направления атак. Рассмотрим возможные виды атак на водяные знаки[2]:

- Удаление водяных знаков (англ. subtractive attack). Заключается в нахождение измененного места и вырезания защиты.
- Искажение водяных знаков (англ. distortive attack). Цель такой атаки – исказить значение так, чтобы владелец не смог идентифицировать свое приложение.
- Добавление водяных знаков (англ. additive attack). Добавление новых водяных знаков затрудняет доказательство владельцем подлинности продукта.

Любая из указанных атак считается эффективной, если не нарушена работоспособность приложения и в случае добавления или искажения, механизм создания новых водяных знаков должен использовать алгоритм владельца.

Исходя из особенностей атак на водяные знаки мы можем определить стратегию защиты ПО с использованием данного метода:

- Водяные знаки должны быть скрыты от обнаружения;
- Водяные знаки должны обладать устойчивостью к модификации приложения (обфускации, деобфускации);
- Чем больше используется различных способов внедрения водяных знаков и их количество, тем эффективнее защита;
- Внедрение защиты не должно сказываться на производительности приложения;
- Водяные знаки следует генерировать с использованием закрытого алгоритма или ключей (асимметричное шифрование), что позволит избавиться от проблемы добавления новых водяных знаков злоумышленником.

Как пример эффективного решения можно привести следующий алгоритм: в программе создается несколько статических массивов и, чтобы они не были «мертвыми», нам необходимо вовлечь их в какие-либо вычисления. Генерировать значения водяных знаков будем с помощью асимметричного алгоритма шифрования (например, AES) и будем использовать хеширование



для точной идентификации водяных знаков в памяти. Секретный ключ для генерации хранится у разработчика. Проверять приложения на подлинность будем путем анализа памяти и нахождения значений наших водяных знаков.

Подобные методы защиты могут быть реализованы как в отдельных продуктах, так и в комплекте с обфускатором.

## 1.2. Обфускация

*Обфусцированной* (англ. obfuscated, запутанной) называется программа, которая на всех допустимых для исходной программы входных данных выдаёт тот же самый результат, что и оригинальная программа, но более трудна для анализа, понимания и модификации[3].

Соответственно, *обфускация* – это процесс применения к исходной программе запутывающих преобразований, затрудняющих дальнейший ее анализ. Обфускация может осуществляться на разных уровнях, соответствующих типам информации, на которые она нацелена, так, основываясь на таксономии запутывающих преобразований [4], можно выделить следующие уровни обфускации:

### 1. Запутывание форматирования исходного текста (англ. layout obfuscation).

Сюда относятся изменение имен методов и классов, объединение классов и пространств имен, удаление форматирования, комментариев, изменение метаданных. Данные преобразования могут применяться как к исходному тексту программы на языке высокого уровня, так и к скомпилированной сборке. Особо не влияют на сложность анализа.

### 2. Запутывание данных (англ. data obfuscation).

Изменения могут затрагивать размещение данных в памяти (имеется ввиду перенос переменных в глобальную область, объединение нескольких переменных в одну, изменение порядка в массиве), кодирование, агрегацию и перемешивания строк и ресурсов.

### 3. Запутывание потока управления (англ. control flow obfuscation).

В отличие от запутывания данных, данный метод способен достаточно сильно увеличить сложность программы. Можно выделить следующие способы преобразований[5]:

- Преобразования агрегации. Вставка и переплетение функций, раскрутка циклов, клонирование кода;
- Преобразования вычислений. Вставка избыточного кода, и усложнение вычислительных преобразований;
- Использование непрозрачных предикатов. Непрозрачный предикат – предикат, значение которого известно в момент запутывания, но трудноустанавливаемо после [3], это такие выражение, которые не зависимо от значений аргументов, всегда истины, либо ложны.
- Табличная интерпретация. Добавление таблиц переходов (например, больших CASE структур), сильно увеличивающих ветвление кода. Добавление интерпретатора (виртуальной машины) – свойственно упаковщикам.

#### **4. Превентивные трансформации (англ. preventive transformation).**

Такие трансформации ориентированы на отладчики, декомпиляторы, деобфускаторы. Могут быть как врожденными (затрудняющие распутывание существующими методами) и целевыми (использующие известные проблемы конкретных решений).

На рынке существует множество различных обфускаторов, отличающихся своим набором возможностей для обфускации, рассмотренных выше, возможностями противодействия деобфускации, цена и зависимостью от версий сборок и платформы. В общем, обфускаторы можно разделить на две группы по используемым средствам воздействия на обфусцируемую сборку[6]:

- Stand-alone обфускаторы, использующие свои интерфейсы для доступа к метаданным и генерации обфусцированной сборки;
- Зависимые от сервисов .Net (ILASM/ILDASM). Функционал таких обфускаторов заведомо ограничен тем набором инструментов, которые предоставляют сервисы.

### 1.3. Упаковка

*Упаковка* – способ модификации сборки, при котором на выходе получается WinApi-приложение, содержащее в себе упакованный (не обфусцированный) IL код и распаковочный модуль.

Такой способ вносит некоторые ограничения:

- Сборка остается рабочей только под операционной системой Windows;
- Из-за этого теряется преимущество JIT-компиляции среды CLR, дающее гибкость для x64 и x86 платформ (генерация различных инструкций, в зависимости от платформы);
- Упаковка немодифицированных сборок не дает серьезной защиты и легко обходится;
- Упаковщики могут перехватывать вызовы виртуальной машиной функций библиотеки `mscorlib.dll`, что предоставляет дополнительную защиту от его снятия. Но на работу таких программ может влиять антивирус и они сильно зависимы от версии .Net Framework.

Упаковщики помимо своей основной функции обычно содержат множество средств дополнительной защиты, основными из которых являются:

- Отслеживание отладки. Позволяет запретить или детектировать выполнение приложения под отладчиком. В продвинутом варианте, оболочка (упаковщик) приложения может передать сигнал на нижний уровень, где, например, может произойти затирание отлаживаемого кода в памяти.
- Обнаружение виртуальной машины. Запрещает выполнение приложения под виртуальной машиной
- Защита памяти и проверка целостности данных. Отслеживание и выявление обращений к памяти и ее модификации.
- Защита импорта и создание DLL оберток. Позволяет либо скрыть информацию об используемых библиотеках, либо исключить их хранение на диске путем включения DLL в состав исполняемого файла.

#### 1.4. Системы лицензирования

Данный метод защиты используется для защиты лицензионного ПО от копирования. Основная суть заключается в выдаче вместе с продуктом уникального ключа, который однозначно идентифицирует данную копию продукта. Отслеживание подлинности продуктов может проводиться с использованием сервера лицензирования и активации, это значительно повышает устойчивость к взлому. Сервер активации предполагает, что при установке приложения, оно должно быть активировано с помощью ключа активации, использование которого строго отслеживается, таким образом, избегается возможность повторной установки.

Часто такие системы входят в состав продуктов, называемых протекторами. *Протекторы* (англ. native processor wrapper) – приложения, совмещающие в себе упаковщик и обфускатор (обычно довольно слабый). Протекторы являются комплексным решением.

При генерации упакованной сборки, протектор внедряет в нее блок активации. Многие протекторы, имеющие систему лицензирования предлагают различные дополнительные сервисы[7]:

- Сервер активации;
- Генератор ключей. Возможность генерировать ключи на сервере в автоматическом режиме;
- Деактивация ключей. Позволяет пользователю перенести программу на другой компьютер и активировать ее заново;
- Блокировка ключей. В приложение встраивается база скомпрометированных ключей активации;
- Привязка к аппаратному обеспечению.

#### 1.5. Защита от модификаций

Суть данного метода заключается в обнаружении нарушения целостности оригинального ПО. Может применяться совместно с обфускацией приложения и защищать его от попыток деобфусцирования.

Самый простой способ защиты от модификаций (англ. tamper-resistant)— это подсчет хеш-суммы всего приложения или его модулей (в случае использования упаковщика – хеш-суммы упакованного кода). Полученное значение сравнивается с оригинальным значением, рассчитанным для данной сборки. Минус статического расчета хеш-суммы в возможности модификации приложения уже после его запуска.

Так же существуют методы динамической самопроверки (англ. integrity verification). В отличие от проверки статического образа приложения (файл на диске), программа сравнивает значения в памяти. Могут сравниваться значения переменных в памяти на выходе из функций, хеш-сумма последних выполненных инструкций или значений в стеке [8].

Более продвинутое метод заключается в шифровании исходного кода в памяти. Секции кода, незадействованные в текущих вычислениях могут быть зашифрованы в любое время и расшифрованы при необходимости их выполнения. Таким образом, исключается модификация кода без знания алгоритма шифрования. Данный способ можно усложнить еще больше, транслировав весь код в код виртуальной машины[9].

## **1.6. Соккрытие данных**

Рассмотренные ранее методы защиты помогают защитить код от кражи и модификации, позволяют отслеживать подлинность используемого ПО или противодействовать отладке, но они не помогают защитить данные приложения. Запущенное приложение в какой-то момент хранит все свои данные в оперативной памяти, откуда злоумышленник может их получить. От этого может спасти модификация данных, перед записыванием их в память. Возможных реализаций этого способа защиты существует множество, и они будут рассмотрены позже.

В ходе анализа не было найдено продуктов, предоставляющих подобную защиту, но она, несомненно, используется в готовых приложениях.

### 1.7. Комплексный подход

Как уже было сказано ранее, нельзя полностью защитить приложение от взлома, можно лишь увеличить необходимые на это затраты. Соккрытие данных не придаст приложению дополнительной защиты без использования других методов, т.к. такую защиту легко снять, то же самое относится к системе лицензирования, защите от модификаций, да и любого рода проверкам.

Для эффективной защиты необходимо применять комплексный подход, который и реализован в готовых предлагаемых решениях, которые будут рассмотрены далее.

### 1.8. Обзор имеющихся решений

В основном на рынке инструменты для защиты приложений представлены в виде двух групп: обфускаторы и протекторы.

На основании обзоров и анализов обфускаторов[6][10] был выбран Spices.Net Obfuscator как один из лучших в данной группе, благодаря следующим возможностям:

- Неограниченная по функционалу версия для некоммерческих проектов;
- Интеграция в MSBuild и VisualStudio;
- Возможность защиты сборки от фальсификации (tamper-resistance);
- Противодействие декомпиляции;
- Запутывание данных (с противодействием подмене благодаря технологии tamper-resistance) и потока управления (технология CodeAnonymizer, запутывающая IL код без возможности восстановления);
- Оперативная поддержка на русском языке.

Из множества протекторов, работающих с .Net сборками можно выделить следующие[7]: Obsidium, Enigma Protector, Oreans Themida, .NET Reactor. Все они имеют примерно одинаковую цену (\$150-200) и за исключением .NET Reactor'а схожие возможности. .NET Reactor больше ориентирован на .Net

приложения и включает в себя довольно мощный обфускатор, что достаточно важно для .Net приложений с открытым исходным кодом. В нем так же предусмотрена достаточно гибкая система лицензирования.

В качестве протектора для сравнения был выбран Enigma Protector, благодаря более богатому набору функционала для защиты.

Лучшие отобранные решения будут сравниваться по описанным методам защиты приложений (табл. 1). Если данный метод реализован, напротив него ставится «+», если нет – «-», если информация не предоставлена – «?».

Таблица 1.1

Сравнение различных решений по степени эффективности защиты

Возможности	Spices.Net Obfuscator	.NET Reactor	Enigma Protector	Мое решение
Обфускация кода				
Защита кода	+	+	+	+/-
Защита от дизасемблирования	+	+	-	-
Запутывание потока управления	+	+	-	-
Упаковка	-	+	+	-
Система лицензирования	-	+	+	-
Tamper-resistant	+	-	+	-
Watermarks	+	-	+	-
Соккрытие данных	-	-	-	+
Поддержка различных версий и языков .Net	+	+	?	+
Цена	\$400	\$180	\$150	\$0
	<a href="http://www.9rays.net/TourStep.aspx?TourStepID=17">http://www.9rays.net/TourStep.aspx?TourStepID=17</a>	<a href="http://www.eziriz.com/dotnet_reactor.htm">http://www.eziriz.com/dotnet_reactor.htm</a>	<a href="http://enigma-protector.com/ru/about.html">http://enigma-protector.com/ru/about.html</a>	

**Выводы:** исходя из проведенного обзора, можно сделать некоторые выводы и рекомендации. Enigma Protector предоставляет множество различных методов защиты, но можно предположить, что из-за достаточно простой

обфускации, эту защиту достаточно легко снять. Spices.Net Obfuscator представляет собой очень мощный инструмент для обфускации приложений и если не нужна система лицензирования, это, пожалуй, лучший вариант. .NET Reactor – что-то среднее между двумя описанными решениями.

Но ни одно из этих решений не предоставляет инструмента для защиты данных, что так же является довольно важным элементом защиты. Этот метод защиты и будет представлен в моем решении, которое может быть использовано как часть комплексной подхода к защите приложения.



## 2. ТЕОРЕТИЧЕСКАЯ РАЗРАБОТКА

В ходе разработки данного решения были использованы различные инструменты и технологии, представленные на рисунке 2.1:

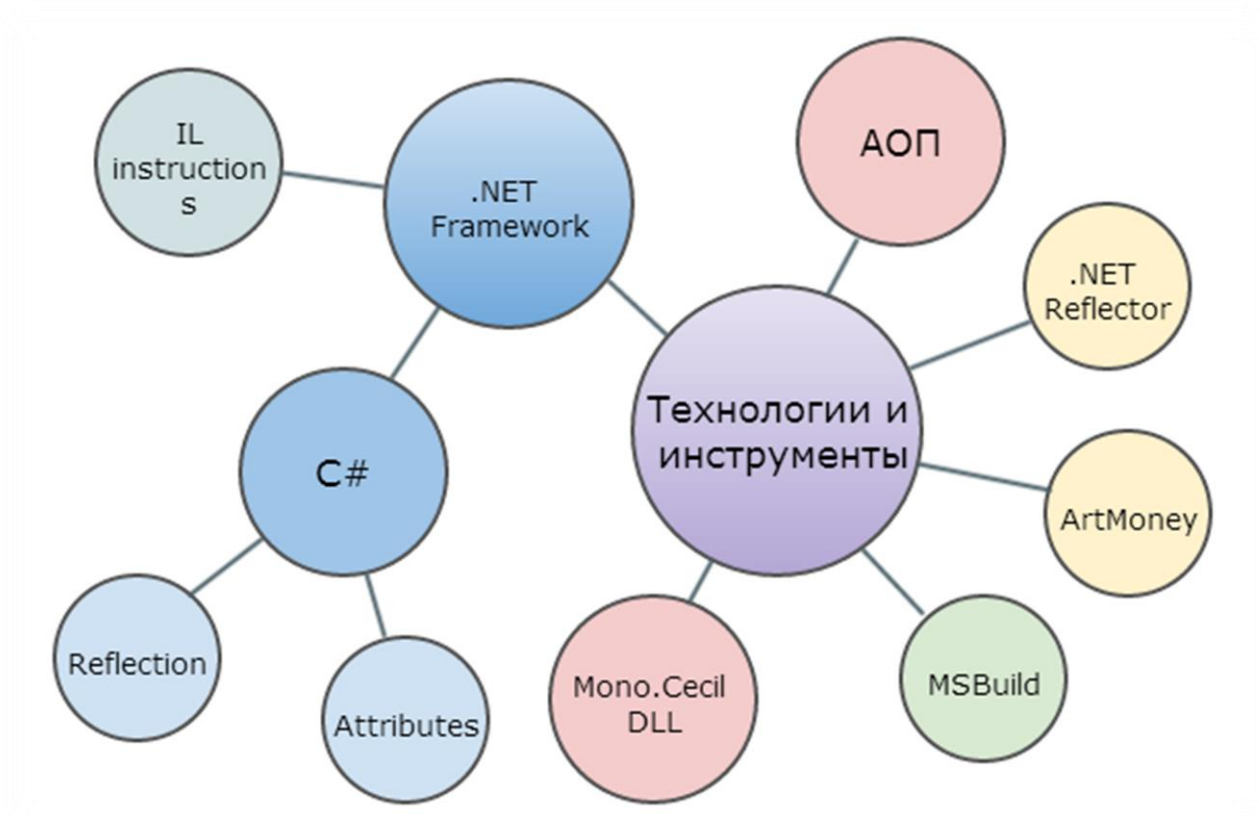


Рис. 2.1. Используемые инструменты и технологии

В данной главе рассмотрены основные моменты, примененных на практике технологий.

### 2.1. Аспектно-ориентированное программирование

*Парадигма программирования* — это совокупность идей и понятий, определяющих стиль написания компьютерных программ. Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером. Существуют несколько основных парадигм (моделей, концепций) программирования, вот основные из них:

- Императивное программирование:
  - Процедурное программирование;

- Структурное программирование;
- Объектно-ориентированное программирование (ООП);
- Аспектно-ориентированное программирование.
- Декларативное программирование:
  - Функциональное программирование;
  - Логическое программирование;
- Метaprogramмирование.

*Аспектно-ориентированное программирование* (АОП) представляет собой одну из концепций программирования, которая является дальнейшим развитием процедурного и объектно-ориентированного программирования (ООП) [11]. Современные программы постоянно усложняются и параллельно с их усложнением, разрабатываются способы снижения сложности программного кода. Так в свое время появилось ООП и возможность выделять код с близким функционалом в отдельные классы. Но некоторую функциональность с помощью предложенных методов невозможно выделить в отдельные сущности. Это сквозная функциональность, это функциональность, необходимая в нескольких модулях системы или приложения. Соответственно ее реализация становится разбросанной по различным модулям, и сопровождение такого кода становится затруднительным, а количество потенциальных ошибок повышается. АОП предполагает наличие языковых средств, позволяющих выделять сквозную функциональность в отдельные модули. Примерами такой функциональности могут служить:

- логирование,
- обработка транзакций,
- обработка ошибок,
- авторизация и проверка прав,
- кэширование,
- элементы контрактного программирования.

С точки зрения АОП в процессе разработки достаточно сложной системы программист решает две ортогональные задачи[11]:

- Разработка компонентов, то есть выявление классов и объектов, составляющих словарь предметной области;
- Разработка сервисов, поддерживающих взаимодействие компонентов, то есть построение структур, обеспечивающих взаимодействие объектов, при котором выполняются требования задачи.

У каждой парадигмы программирования есть свои особенности, но главным отличием является понятие основной единицы программы, например, для императивного – это инструкции, для функционального – функции, для ООП – объект, для логического – факт. В АОП такой единицей является *аспект*. Аспект (aspect) – это модуль или класс, в который вынесена основная сквозная функциональность. Аспектный модуль – это результат выявления и декомпозиции явлений, понятий и событий, применимых к группе компонентов. Но прежде чем рассматривать подробнее сущность аспектно-ориентированного подхода, необходимо дать определения и другим важнейшим средствам АОП:

- точка соединения (join point) – точка в выполняемой программе (вызов метода, создание объекта, обращение к переменной), где следует применить сквозную функциональность (совет);
- совет (advice) – дополнительная логика, код, который должен быть вызван из точки соединения. Совет может быть выполнен до, после или вместо точки соединения;
- срез (pointcut) – набор точек соединения. Срез определяет, подходит ли данная точка соединения к заданному совету;
- внедрение (introduction) – изменение структуры класса и/или изменение иерархии наследования для добавления функциональности аспекта в инородный код;
- цель (target) – объект, к которому будут применяться советы;

- переплетение, вшивание (weaving) – связывание объектов с соответствующими аспектами (возможно на этапе компиляции, загрузки или выполнения программы).

Таким образом, в аспектном модуле описываются срезы и советы, связанные с конкретными срезами.

Существуют несколько реализаций АОП, самыми известными из которых можно назвать AspectJ (Java) и PostSharp (C#). В них представлен набор конкретных срезов, т.е. мест, куда привязывается совет, например: вызов методов определенного класса, возвращающих значение определенного типа или определенное значение, принимающих определенные аргументы; все манипуляции с конкретным объектом; вызов статических методов определенного класса; выполнение точки соединения, цель которой помечена аннотацией или атрибутом.

Все эти принципы теряют свое основное преимущество без автоматической компоновки аспектных модулей с модулями программы (вшивания). Реализация автоматической компоновки во многом определяет возможности той или иной аспектно-ориентированной платформы. В настоящее время обсуждаются два подхода к интеграции аспектов:

- Статическая интеграция на этапе компиляции;
- Динамическая интеграция на этапе выполнения программы.

Основными преимуществами использования АОП являются:

- Инкапсуляция функциональности, которая не может быть представлена в отдельном модуле или функции, тем самым улучшается декомпозиция и упрощение структуры системы;
- Упрощение сопровождения программной системы и внесения в нее изменений;
- Простая возможность повторного использования кода.

Но так же аспектно-ориентированный подход пока что обладает рядом недостатков. Концепции АОП реализованы только на уровне языковых расширений, такая реализация представляется неполной и неэффективной.

Различные реализации отличаются по своим возможностям и не стандартизованы.

В рамках моего проекта методология АОП была применена для внедрения функционала модификации данных (совет) для защищенного хранения в памяти в местах применения атрибутов (точка соединения) к свойствам защищаемого класса. Внедрение происходит после компиляции целевой программы путем модификации скомпилированной сборки с использованием сторонней библиотеки Mono.Cecil путем применения рефлексии (отражений) .NET Framework.

## 2.2. CLR и .NET Framework

2.2.1. **.NET Framework** — программная платформа (программный комплекс) разработки и исполнения прикладных программ, выпущенная компанией Microsoft. Термин “платформа” может применяться в двух разных смыслах. С одной стороны, это “концепция” (идеи, спецификации и т. д.), с другой — набор вполне конкретных объектов (файлов, документации и пр.). Эта двойственность в полной мере относится к .NET Framework.

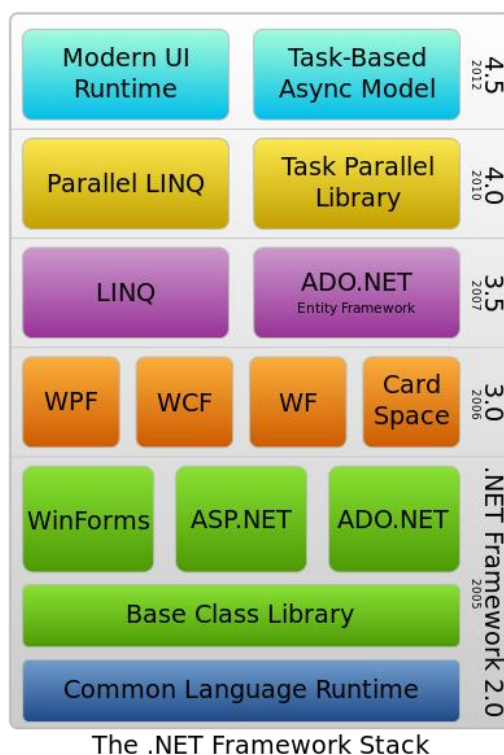


Рис. 2.2. Стек технологий .NET Framework

.NET Framework состоит из двух главных компонентов: библиотеки базовых классов FCL (Framework Class Library) и общезыковой среды разработки CLR (Common Language Runtime). Более полно архитектура .NET Framework представлена на Рис. 2.2.

2.2.2. Благодаря **CLR**, .NET Framework поддерживает несколько различных языков разработки, которые в общем случае не влияют на конечный результат выполнения программы. Программа, написанная на любом поддерживаемом языке программирования, сначала переводится компилятором в единый для .NET промежуточный байт-код Common Intermediate Language (IL), образуя сборку (англ. assembly), затем код выполняется виртуальной машиной CLR (Рис. 2.3). Фактически во время выполнения программы в среде CLR неизвестно, на каком языке программирования разработчик написал исходный код.

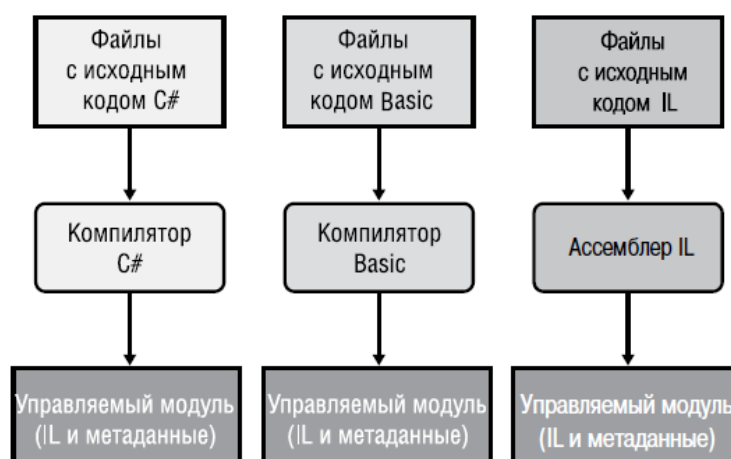


Рис. 2.3. Компиляция исходного кода в управляемые модули

CLR сборка состоит из стандартного заголовка PE-файла Windows, заголовка CLR, метаданных и собственное IL кода. Для выполнения любого управляемого модуля на машине конечного пользователя должна быть установлена среда CLR (в составе .NET Framework).

2.2.3. Запуск .NET приложения. При запуске исполняемого файла Windows анализирует PE-заголовок EXE-файла и загружает в адресное пространство процесса соответствующую версию библиотеки MSCorEE.dll (в зависимости от разрядности процесса). В библиотеке MSCorEE.dll определен

метод, который инициализирует CLR. Основным потоком инициализирует CLR, загружает сборку и переводит управление на метод Main. На этом запуск приложения заканчивается и начинается его исполнение.

Для выполнения какого-либо метода его IL-код должен быть преобразован в машинные команды. Этим занимается JIT-компилятор (Just-In-Time) среды CLR. Перед исполнением метода среда CLR находит все типы данных, на которые ссылается код данного метода (п. 2.3) и для всех методов найденных типов заменяет записи, содержащие адрес реализации метода на вызов функции JIT-компилятора. Когда впервые вызывается какой-то метод, управление передается на внутреннюю функцию JIT-компилятора, она ищет в метаданных сборки IL-код этого метода, компилирует его в машинные команды, сохраняет их в динамически выделенном блоке памяти, затем заменяет в метаданных адрес вызываемого метода адресом блока памяти, содержащего готовые машинные команды и передает ему управление [12].

2.2.4. **FCL** – набор сборок в формате DLL, содержащих несколько тысяч определений типов, каждый из которых предоставляет некоторую функциональность. Ниже перечислены некоторые разновидности приложений, которые могут создаваться разработчиками при помощи этих сборок [12]:

- Веб-службы (технологии ASP.NET и WCF (Windows Communication Foundation));
- Приложения Web Forms/приложения MVC на базе HTML;
- Приложения Windows с расширенным графическим интерфейсом (технологии WPF (Windows Presentation Foundation) и Windows Forms);
- Консольные приложения Windows;
- Службы (services)Windows, управляемые через Windows SCM (Service Control Manage);
- Библиотеки компонентов (DLL).

## 2.3. Отражения

*Отражение* или *рефлексия* (англ. reflection) - это процесс, во время которого программа может отслеживать и модифицировать собственную структуру и поведение во время выполнения. В .NET Framework это становится возможным благодаря пространству имен System.Reflection. В этом пространстве имен имеется несколько типов, позволяющих писать код разбора таблиц метаданных. Точнее, типы из этого пространства имен предоставляют объектную модель для работы с метаданными сборки или модуля.

Метаданные – это блок двоичных данных, состоящий из набора таблиц. Метаданные хранятся в исполняемом файле непосредственно перед кодом и после PE32 и CLR заголовков. Существуют три категории таблиц: определений, ссылок и манифестов.

Основные таблицы определений включают в себя таблицы для: модуля (содержит одну запись, идентифицирующую текущий модуль), типов, методов, полей, свойств, параметров, событий (содержат по одной записи для каждой соответствующей сущности) и т.д. Для каждой сущности, определяемой в компилируемом исходном тексте, компилятор генерирует строку в одной из таблиц.

В ходе компиляции исходного текста компилятор также обнаруживает сущности, на которые имеются ссылки в исходном тексте текущего модуля. Все сведения о найденных сущностях регистрируются в нескольких таблицах ссылок, составляющих метаданные. В них же содержатся записи о других сборках, на которые ссылается модуль [12].

Типы пространства имен System.Reflection дают возможность запрашивать поля, методы, свойства и события типа путем разбора соответствующих таблиц метаданных. Можно узнать, какими атрибутами помечена та или иная сущность метаданных. Есть даже классы, позволяющие определить указанные сборки и методы и возвращающие в методе байтовый IL-поток.



В моем решении отражения используются в совете (внедряемом функционале). Приложение использующее Mono.Cecil модифицирует код определенных свойств (property) целевой сборки, а точнее методов доступа к этим свойствам – чтения (get) и записи (set). Внутри этих методов с помощью отражения получается текущий метод, затем он передается в класс, отвечающий за защиту данных, в нем из объекта типа `MethodBase` описывающего методы получается свойство, из метода которого был произведен вызов.

## 2.4. Атрибуты

Атрибуты в CLR – это декларативные аннотации, наделяющие код дополнительными возможностями. Они выступают в роли определенных меток или ярлыков, привязанных к определенной сущности, такой как метод, класс, поле, свойство и т.д. Функционал атрибутов в среде .NET довольно обширный и его описание не является целью данной работы.

В CLR так же есть мощный механизм настраиваемых или пользовательских атрибутов (custom attributes). Настраиваемые атрибуты служат лишь средством передачи дополнительной информации, которая хранится в метаданных модуля. Само по себе определение атрибутов бесполезно и никак не влияет на работу программы [12].

В соответствии с методикой АОП атрибут можно назвать срезом, к которому может быть применен определенный совет. Применение атрибута в определенном месте кода программы – точка соединения.

В моей библиотеке для защиты данных определен класс `SecureFieldAttribute`, унаследованный от `System.Attribute`. Область применения этого атрибута распространяется только на свойства классов, это достигается путем применения к классу атрибута, другого атрибута `AttributeUsage` с параметром `AttributeTargets.Property`.

## 2.5. Метапрограммирование и Mono.Cecil

В пункте 2.3 было описано пространство имен `System.Reflection`, которое содержит другое пространство имен `System.Reflection.Emit`, позволяющее создавать метаданные и инструкции языка CIL, а так же формировать на диске PE-файл. Это же умеет и библиотека `Mono.Cecil`, в этом пункте будут описаны основные отличия двух подходов и приведено обоснование моего выбора в пользу `Mono.Cecil` для модификации целевой сборки.

Метапрограммирование – вид программирования, связанный с созданием самомодифицирующихся программ, которые меняют свой код во время выполнения (процесс динамической генерации кода).

`Mono.Cecil` — это библиотека, позволяющая работать со сборкой как с массивом байтов. Она предоставляет широкий спектр классов, как для создания своих собственных сборок, так и для модификации уже существующих.

Кардинальное отличие этих двух подходов обусловлено различными изначальными целями. `System.Reflection` изначально разрабатывалось для использования паттернов внедрения зависимостей (англ. *dependency injection*, DI) и динамического связывания (англ. *late binding*), т.е. в общем случае для выявления типов объектов в ходе исполнения приложения. `Mono.Cecil` сразу разрабатывалась для анализа и модификации CIL кода сборок.

`System.Reflection` не способна оперировать кодом как сырыми данными. При загрузке сборки она собирается в объектную модель, где объектами являются такие сущности как типы, поля, методы и т.д. Это накладывает определенные ограничения на нее, как технологию модификации исходного кода:

- Во время загрузки сборки может быть выброшено исключение `CodeAccessSecurityException` (CAS), т.к. данные которые обрабатываются, все еще являются кодом. CAS – это механизм защиты, используемый в .NET Framework, позволяющий ограничивать доступ коду к ресурсам компьютера;
- Низкая производительность;

- Большое потребление памяти (данные преобразуются в объектную модель). Так же однажды загруженная сборка, больше не может быть выгружена до завершения работы приложения;
- Нельзя загрузить две разные версии одной сборки, т. о. нельзя анализировать стандартные сборки CLR версии отличной от используемой в приложении;
- Есть информация о нестабильной работе и исключениях при работе со сборками не AnyCPU (64х/32х битные версии), содержащих определения статических конструкторов и смешанных сборок, содержащих модули, написанные на C++/CLI.

Так же у System.Reflection есть некоторые ограничения по умолчанию, не заложенные в функционал:

- Нет возможности анализировать IL код напрямую;
- При попытке получения информации об элементах, на которые ссылается код сборки, определенных в не подгруженной сборке выбрасывается исключение UnresolvedException. Это происходит из-за отсутствия различия между ссылкой и определением (типы TypeRef и TypeDef в Mono.Cecil соответственно);
- Нет доступа к метаданным TypeSpec, отвечающих за определение типов обобщенных (англ. generic) методов.

Все эти ограничения не говорят о том, что System.Reflection – это плохая технология, у нее есть свои преимущества и предпосылки для развития динамического программирования. Но когда речь заходит об анализе IL кода, лучшим средством для этого будет библиотека Mono.Cecil [13].

Производительность Mono.Cecil в несколько раз выше, чем System.Reflection и она имеет полную совместимость с CLR сборками и позволяет напрямую оперировать IL кодом.

Mono.Cecil оперирует следующими сущностями: сборки, модули, типы, поля, свойства и методы (Рис. 2.4) [14].

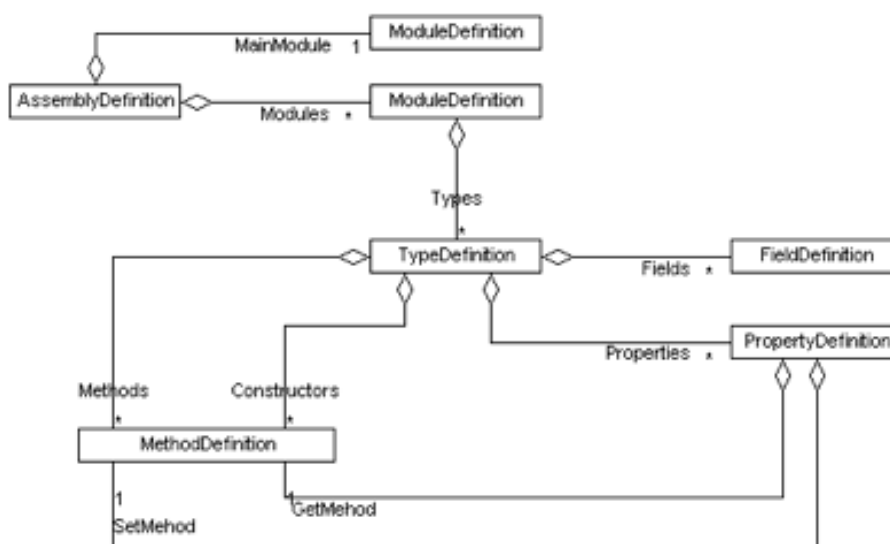


Рис. 2.4. Сущности, представленные Mono.Cecil

Объект типа `AssemblyDefinition` создается с помощью `AssemblyFactory`, которая оперирует файлом сборки. `MethodDefinition` содержит в себе объект типа `MethodBody`, через который можно получить доступ к IL инструкциям, с которыми можно работать посредством IL процессора (`ILProcessor`), предоставляющего широкий функционал, который будет рассмотрен в следующей главе.

Таким образом, использования данной библиотеки в моем проекте является обоснованным, но от `System.Reflection` я не отказался, и этот подход использован во внедряемом коде для динамического определения типа свойств, для которых вызываются методы.

## 2.6. Сборка проектов при помощи MSBuild

Для небольших проектов обычно достаточно компилятора командной строки и пакетного файла, содержащего его вызов. Для больших проектов такой метод становится не удобным, так как появляется необходимость отслеживать зависимости и версии файлов. Для решения этой проблемы была создана программа `Make`. Она обладала довольно простым функционалом, на основе информации из `make`-файла, содержащего описание зависимостей, принималось решение о необходимости построения тех или иных файлов, и

затем вызывалась внешняя команда оболочки ОС, что не позволяло собирать кроссплатформенные приложения.

В дальнейшем такой принцип развился в продукты подобные MSBuild. MSBuild представляет собой платформу сборки проекта, поставляемую вместе с .NET Framework начиная с версии 2.0. Теперь для выполнения различных действий используются внутренние команды, проецируемые на специально создаваемые классы. В качестве основы формата файла проекта MSBuild использует XML. Корневой тег Project содержит следующие элементы:

- PropertyGroup – группы свойств;
- ItemGroup – группы элементов;
- Import – элемент, импортирующий другой файл проекта.

Свойства определяют статические значения, используемые для конфигурирования проекта. Элементы используются для определения файлов и папок входящих в проект. С помощью элементов и свойств определяются данные проекта [15].

Такие действия как сборка (build), отчистка (clean), пересборка (rebuild) проекта и другие являются целями (targets). Для каждой цели определяется набор задач (task), которые необходимо выполнить для ее достижения (Рис. 2.5).

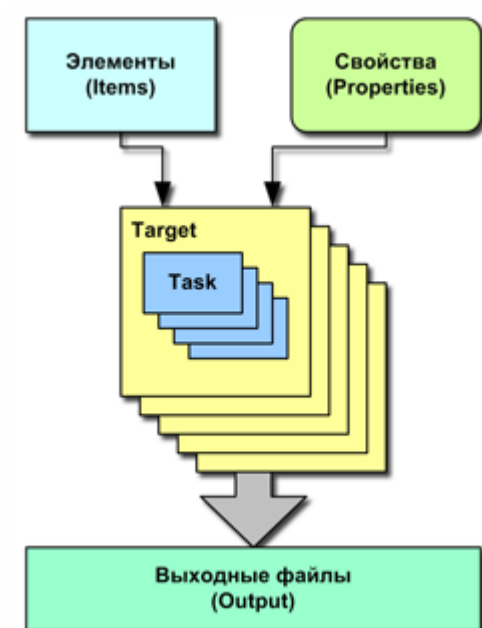


Рис. 2.5. Схема обработки файла проекта.

Для каждой цели можно указать условия ее выполнения и список зависимостей, состоящий из других целей, которые нужно вызвать перед ней. На этом основаны средства расширения MSBuild. API MSBuild позволяет создавать свои задачи, для этого необходимо создать класс, наследуемый от `Microsoft.Build.Utilities.Task` и скомпилировать его в отдельную dll. Чтобы затем вызвать такое задание, необходимо в XML файле описать его с помощью тега `UsingTask`, указав его имя и путь к сборке.

В моем проекте существует необходимость использовать средства дополнительные MSBuild для упрощения постройки проекта. Это связано с необходимостью запустить исполняемый файл, встраивающий защиту, после компиляции проекта.

## **2.7. Реверс-инжиниринг и проверка эффективности защиты**

Реверс-инжиниринг (обратная разработка) – процесс исследования некоторой программы или устройства с целью понимания принципа его работы. Обычно осуществляется с целью модификации, копирования, написания генератора ключей, получения закрытых сведений об алгоритмах и данных. В рамках текущего проекта реверс-инжиниринг использовался на стадии разработки для анализа промежуточного кода (IL кода), получаемого после компиляции. На основе полученной информации и была написана программа для внедрения защиты в сборки, путем модификации ее исходного кода.

Для получения доступа к исходному коду сборки использовалась утилита .NET Reflector (Рис. 2.6). Она позволяет декомпилировать сборку на различные языки .NET, такие как C#, Visual Basic, IL и некоторые другие.

Так же эта утилита имеет анализатор зависимостей, который позволяет увидеть, где используется текущий метод и от каких библиотек он зависит.

Для проверки эффективности защиты использовалась программа ArtMoney (Рис. 2.7), позволяющая производить динамический анализ памяти и искать вхождения различных значений. Цель моего проекта – достигнуть результата, при котором в памяти всегда будет храниться модифицированное

значение переменной, и такими программами как ArtMoney нельзя будет найти переменную, значение которой (не оригинальное) пользователь знает, для дальнейшего изменения ее значения для своих нужд.

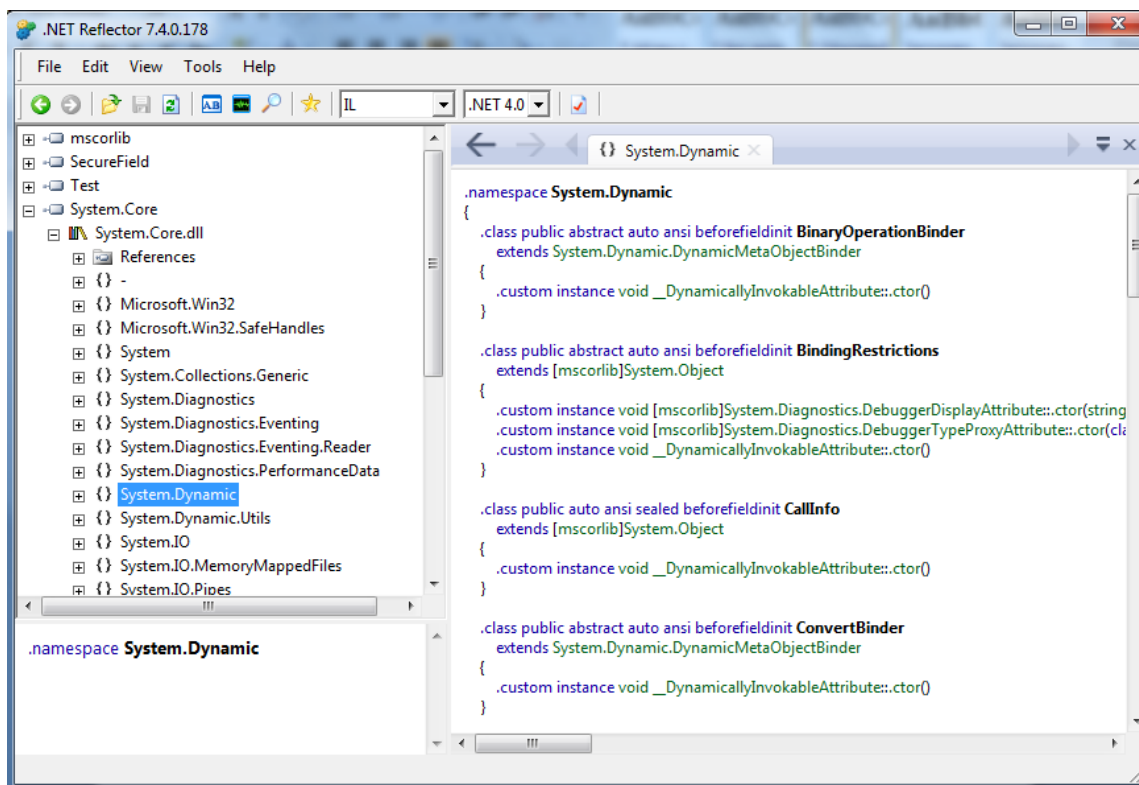


Рис. 2.6. Графический интерфейс .NET Reflector

Важной возможностью данной программы является способность отсеивать результаты, т.е. при изменении значения переменной, искать только в тех областях памяти, которые содержали предыдущие значения.

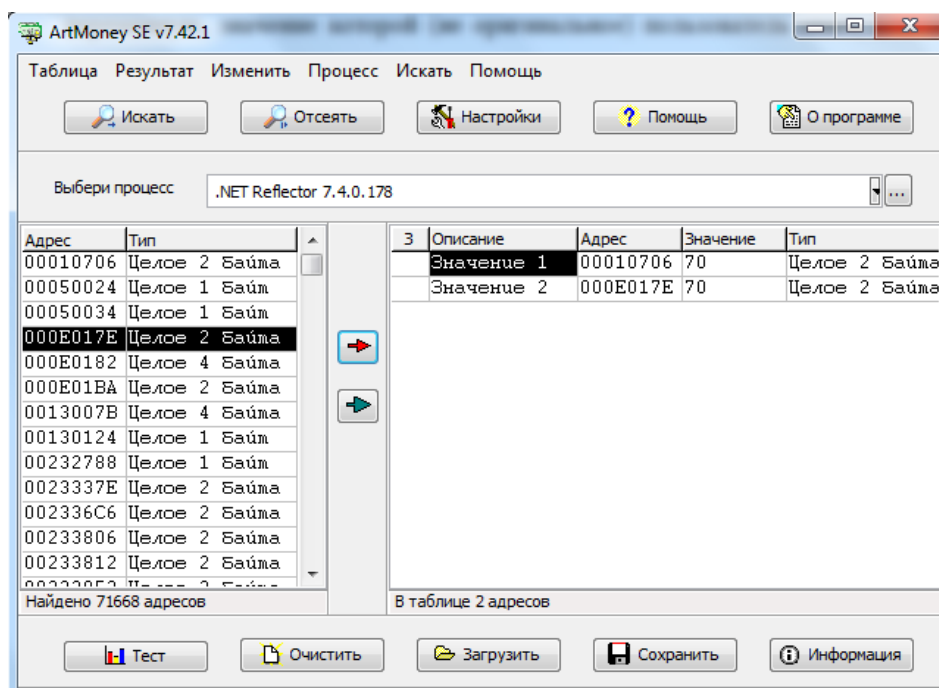


Рис. 2.7. Графический интерфейс ArtMoney

**Выводы:** в данной главе были описаны основные технологии и инструменты, используемые в ходе работы над проектом. Подробное описание их реализации и применения будут описаны в следующей главе.



### 3. РЕАЛИЗАЦИЯ

В данной главе будет последовательно описан процесс разработки проекта и на практике рассмотрено применение технологий и инструментов, о которых говорилось в предыдущей главе.

Разрабатываемый проект состоит из двух частей, это библиотека SecureField, предоставляющая методы защиты и приложение MonoInjections, которое осуществляет внедрение этой защиты в целевую сборку. Для использования данного решения, необходимо подключить к своему проекту библиотеку SecureField и обозначить необходимые для защиты свойства атрибутом SecureFieldAttribute, затем после компиляции проекта, модифицировать его приложением MonoInjections.

На Рис. 3.1 представлена схема классов проекта:

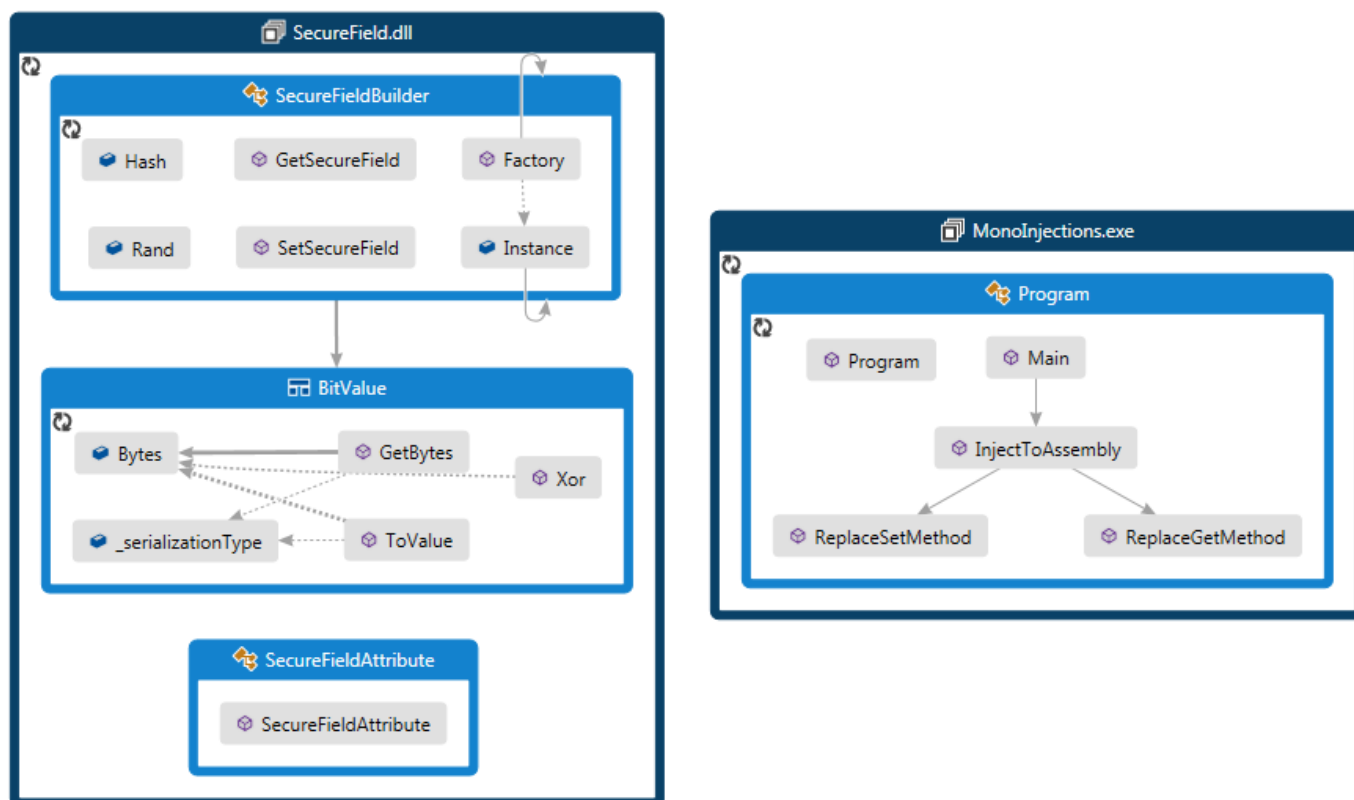


Рис. 3.1. Схема классов проекта

### 3.1. Библиотека SecureField

#### 3.1.1. Класс SecureFieldBuilder и методы защиты.

Этот класс содержит в себе методы, производящие модификацию значений и список всех измененных значений в единственном экземпляре. Для этого необходимо воспользоваться шаблоном проектирования одиночка (singleton). Применение этого шаблон гарантирует, что в однопоточном приложении экземпляр данного класса будет единственным, обладающий глобальной точкой доступа.

В проекте используется самая простая реализация данного паттерна. В классе создается статическое поле Instance, в которое при первом обращении к классу сохраняется экземпляр этого класса (здесь и далее некоторый код опущен, полный исходный код можно найти в приложении 1):

```
private static readonly SecureFieldBuilder Instance = new
    SecureFieldBuilder();
```

Для его получения используется открытый метод Factory, который возвращает значение данного поля:

```
static public SecureFieldBuilder Factory()
{
    return Instance;
}
```

Конструктор класса является закрытым, поэтому нет другой возможности создать экземпляр.

В классе объявлено закрытое поле Hash типа Hashtable, который предоставляет коллекцию пар "ключ-значение", упорядоченных по хэш-коду ключа. В качестве ключа используется полное имя свойства вида [Class].[Property], где Class – имя класса, в котором оно объявлено, Property – имя свойства. В качестве значения, хранится маска, которая используется для шифрования каждого отдельно свойства. Для полного исключения конфликтов имен, можно добавить к ключу пространство имен, т.к. в разных пространствах имен могут быть объявлены классы с одинаковыми именами и свойствами, что приведет к ошибке.

Основной функционал данного класса реализован в двух методах: `GetSecureField` и `SetSecureField`, которые вызываются из методов доступа свойств. В качестве аргументов в них передается объект `method` типа `MethodBase`, описывающий метод из которого вызвана функция и само свойство `obj`. Для метода `Set` так же передается значение, которое нужно установить, а метод `Get` возвращает значение, которое получается после дешифрования:

```
public object GetSecureField(MethodBase method, object obj)
public void SetSecureField(MethodBase method, object obj, object val)
```

Функционал этих методов местами схож, один из них производит запись, другой считывание (блок-схемы алгоритмов приведены в приложении). Изначально в обоих методах на основании объекта `method` и рефлексии получаем имя свойства `fullPropName` и объект `numField` типа `FieldInfo`, описывающий приватное поле, которое добавляется динамически на этапе модификации сборки, его имя получается добавлением в начало символа подчеркивания и приведение первого символа в нижний регистр. Объект `numField` дает нам доступ к значению поля, а строка `fullPropName` является ключом для коллекции масок, как было сказано выше.

Для метода `Set` значение, которое нужно установить приводится к массиву байт с использованием класса `BitValue`, о котором будет сказано позже, стоит отметить, что он может работать одинаково с любыми значащими типами. В качестве маски используется массив случайных байтов соответствующей длины, получаемых с использованием стандартного для .NET ГСЧ (генератора случайных чисел), ее значение добавляется в коллекцию `Hash` (или обновляется, если оно там уже присутствует). Далее над значением и маской производится операция логического сложения (исключающего или, XOR), полученный массив байт преобразуется обратно в значение соответствующего типа и устанавливается как значение текущего поля.

Для метода `Get` значение поля получается с использованием рефлексии с помощью `obj` и `numField`, а значение маски берется из коллекции, если его там нет (не было записи или инициализации), возвращается значение данного типа

поля по умолчанию. Затем производится операция логического сложения над маской и байтовым представлением значения поля, и возвращается преобразованное уже преобразованное значение необходимого типа.

Вызов данных методов производится из методов доступа свойств и выглядит следующим образом:

```
public <type> <propertyName> { get; set; }
{
    get
    {
        var currentMethod = MethodBase.GetCurrentMethod();
        var builder = SecureFieldBuilder.Factory();
        return (<type>)builder.GetSecureField(currentMethod, this);
    }
    set
    {
        var currentMethod = MethodBase.GetCurrentMethod();
        var builder = SecureFieldBuilder.Factory();
        builder.SetSecureField(currentMethod, this, (object)value);
    }
}
```

Где propertyName – имя свойства, а type – его тип. С помощью рефлексии получается информация по текущему методу и объект класса SecureFieldBuilder, затем вызывается соответствующий метод.

### 3.1.2. Класс BitValue и динамическое приведение типов.

Изначально в проекте в качестве маски использовалось случайное целое число, и оно складывалось со значением, но такой подход возможен лишь для одного типа данных и такое решение не имеет смысла, т.к. оно должно быть комплексным и по возможности предоставлять полный функционал для защиты данных. Поэтому оно было расширено до поддержки всех значимых типов (кроме булевого). Для этого нужно было решить две проблемы: возможность привести все типы к одному определенному, для проведения одинаковых операций по шифрованию, и динамическое определение типа свойств на этапе модификации сборки.

В качестве общего типа был выбран массив байт, так как с ним очень легко проводить операцию логического сложения, которая была выбрана для защиты. Для этого FCL предоставляет класс System.BitConverter со статическим

перегруженным методом `GetBytes`, который принимает на вход значение определенного типа и возвращает массив байт, проблема в том, что нет перегруженного метода для объектов типа `object`, поэтому нужно было реализовать динамическое приведение типов. Для этого был реализован класс, содержащий поле `Bytes` типа `byte[]` и приватное поле `_serializationType`, хранящее значение типа из перечисления. Тип объекта определяется с помощью метода `GetType()` и на основании имени типа, устанавливается значение из перечисления. Это значение в дальнейшем используется для вызова методов `GetBytes` и `To[type]` класса `BitConverter`.

Эту проблему можно было бы решить проще, используя ключевое слово `dynamic`, которое позволяет разрешать операции над объектами данного типа во время выполнения. Компилятор пакует сведения об операции, затем эти сведения используются для оценки операции во время выполнения. Как часть процесса, переменные типа `dynamic` компилируются в переменные типа `object`. Поэтому тип `dynamic` существует только во время компиляции, но не во время выполнения. Это бы решило первую проблему, но усложнило бы вторую, связанную с определением типа на этапе внедрения защиты, о ней будет сказано в следующих пунктах.

### 3.1.3. Атрибут `SecureFieldAttribute`

Как было сказано в предыдущей главе (п.2.4) большинство атрибутов ничего сами не делают, а лишь используются в роли метки. Так и в этом случае, описание атрибута выглядит следующим образом:

```
[AttributeUsage(AttributeTargets.Property)]
public class SecureFieldAttribute : Attribute
{
    public SecureFieldAttribute()
    {
    }
}
```

Стоит обратить внимание на атрибут `AttributeUsage`, применяемый к нашему классу. Он ограничивает область применения атрибута путем указания в качестве аргумента элементов перечисления флагов `AttributeTargets`. В нашем

случае, атрибут может применяться только к свойствам, поэтому для него установлено ограничение `AttributeTargets.Property`.

## 3.2. MonoInjections и модификация сборок

`MonoInjections` является консольным приложением, на вход которого подается путь к целевой сборке, на выходе получается модифицированная сборка с внедренными элементами защиты и так же неизменная резервная копия.

### 3.2.1. Анализ целевой сборки

За модификацию целевой сборки отвечает метод `InjectToAssembly`. В `Mono.Cecil` сборка представляется типом `AssemblyDefinition`, одним из элементов которого является коллекция всех типов (классов) сборки. Нам необходимо получить все свойства, помеченные атрибутом `SecureFieldAttribute`.

Для каждого свойства генерируется имя соответствующего приватного поля, происходит его объявление в определениях `Mono.Cecil` и созданный объект добавляется в коллекцию полей данного класса. Затем поочередно вызываются методы для изменения методов доступа текущего свойства.

Но для начала нам нужно получить все типы, которые будут использованы во внедряемом коде, в `Mono.Cecil` это делается путем импорта типов в сборку:

```
assembly.MainModule.Import(typeof(<type>));
assembly.MainModule.Import(typeof(<type>).GetMethod("<methodName>"));
```

Где `type` – имя класса, а `methodName` – имя метода. В первом случае будет получен объект типа `TypeReference`, во втором – `MethodReference`.

### 3.2.2. Модификация исходного кода

Модификацией методов доступа свойств занимаются два метода `ReplaceSetMethod` и `ReplaceGetMethod`. Реализация этих методов довольно однотипна и отличается только добавляемыми инструкциями.

Для начала на основе объекта свойства получаем необходимый метод (`get` или `set`) и получаем его IL процессор:

```
var ilProc = method.Body.GetILProcessor();
```

IL процессор позволяет оперировать IL командами метода. Сначала нужно отчистить набор инструкцию, который может быть определен пользователем, так как на данный момент поддержка дополнительного функционала в методах доступа к защищаемым свойствам не поддерживается, и установить флаг наличия локальных переменных, иначе JIT компилятор не сможет получить к ним доступ:

```
ilProc.Body.Instructions.Clear();
ilProc.Body.InitLocals = true;
```

После этого создаем необходимые локальные переменные путем добавления их в соответствующую коллекцию:

```
var variableDef = new VariableDefinition(typeRef);
ilProc.Body.Variables.Add(variableDef);
```

Для добавления новых инструкций будет использован метод, добавляющий новую инструкцию непосредственно перед указанной:

```
ilProc.InsertBefore(target, Instruction.Create(...));
```

Первой добавленной инструкцией будет инструкция `ret`, возвращающая управление и помещающая возвращаемое значение из стека вызванной функции (при его наличии) в стек вызываемой:

```
ilProc.Append(Instruction.Create(OpCodes.Ret));
```

В дальнейшем все инструкции будут добавляться поочередно непосредственно перед ней.

### 3.2.3. IL код измененных методов

Суть модификации сборки состоит в замене тела методов доступа свойств. Чтобы понять, на что их заменять, необходимо провести анализ генерируемого компилятором IL кода, что и было сделано. Было создано тестовое приложение, в котором был объявлен класс с одним свойством, код `set` и `get` методов приведен в конце п.3.1.3.

После компиляции приложения и просмотра сборки в .NET Reflector'e получен следующий результат:

```
.method public int32 get_Number()
{
    .locals init (
        [0] class [mscorlib]System.Reflection.MethodBase currentMethod,
        [1] class [SecureField]SecureField.SecureFieldBuilder builder)
    L_0000: nop
    L_0001: call class
        [mscorlib]System.Reflection.MethodBase::GetCurrentMethod()
    L_0006: stloc.0
    L_0007: call class [SecureField]SecureField.SecureFieldBuilder::Factory()
    L_000c: stloc.1
    L_000d: ldloc.1
    L_000e: ldloc.0
    L_000f: ldarg.0
    L_0010: callvirt instance object [SecureField]
        SecureField.SecureFieldBuilder::GetSecureField(class
        [mscorlib]System.Reflection.MethodBase, object)
    L_0015: unbox.any int32
    L_001e: ret
}

.method public void set_Number(int32 'value')
{
    .locals init (
        [0] class [mscorlib]System.Reflection.MethodBase currentMethod,
        [1] class [SecureField]SecureField.SecureFieldBuilder builder)
    L_0000: nop
    L_0001: call class
        [mscorlib]System.Reflection.MethodBase::GetCurrentMethod()
    L_0006: stloc.0
    L_0007: call class [SecureField]SecureField.SecureFieldBuilder::Factory()
    L_000c: stloc.1
    L_000d: ldloc.1
    L_000e: ldloc.0
    L_000f: ldarg.0
    L_0010: ldarg.1
    L_0011: box int32
    L_0016: callvirt instance void [SecureField]
        SecureField.SecureFieldBuilder::SetSecureField(class
        [mscorlib]System.Reflection.MethodBase, object, object)
    L_001c: ret
}
```

Стоит упомянуть, что в IL перед вызовом метода в стек кладется объект, для которого вызывается метод и после него все аргументы метода, это делается с помощью команд ldloc и ldarg. Для записи значения со стека в локальную переменную используется команда stloc. Нулевой аргумент в функции – это объект this (ldarg.0 помещает в стек this). Подробное описание используемых IL команд можно найти в приложении 2.



Так как возвращаемое значение `GetSecureField` и последний аргумент `SetSecureField` типа `object`, нам необходимо привести значение значимого типа к ссылочному. Эти операции в .NET называются упаковка и распаковка (`boxing`, `unboxing`). Без явного приведения типов можно было бы обойтись, если использовать `dynamic` тип. Тут мы и подходим к этой описанной выше проблеме. Как уже было сказано, после компиляции переменные типа `dynamic` принимают тип `object`, а компилятор сохраняет информацию об операции, генерируя множество кода. И этот код пришлось бы повторить в `MonoInjections`, поочередно добавляя каждую операцию в коллекцию инструкций тела метода. С упаковкой, мы устраняем эту проблему, а получить тип текущего свойства не сохраняет труда, т.к. `Mono.Cecil` предоставляет нам такую возможность.

Теперь для функционирования нашего приложения нам необходимо повторить все эти команды и обозначить целевое свойство необходимым атрибутом. После понимания необходимости упаковывать значимые типы, повторить IL команды с использованием IL процессора `Mono.Cecil` не составляет труда.

### 3.3. Практическое использование

Данное решение поставляется в комплекте из двух файлов:

- `SecureField.dll`
- `MonoInjections.exe`

Первый из них является подключаемой библиотекой, которая должна быть добавлена в зависимости (англ. `references`) проекта, второй является исполняемым файлом, отвечающим за внедрение защиты в приложение.

Предлагаемая защита применяется к свойствам без дополнительного функционала методов доступа:

```
public int Property { get; set; }
```

Для того чтобы применить защиту к нужным свойствам, необходимо обозначить их атрибутом:

```
[SecureField]
public int Property { get; set; }
```

Затем программа MonoInjections.exe копируется в папку с исполняемым файлом проекта и запускается с именем файла в качестве параметра (случай без использования MSBuild), исполняемый файл проекта перезаписывается и приведенный выше код заменяется на следующий:

```
private int _property;

[SecureField]
public int Property
{
    get
    {
        MethodBase currentMethod = MethodBase.GetCurrentMethod();
        return (int) SecureFieldBuilder.Factory()
            .GetSecureField(currentMethod, this);
    }
    set
    {
        MethodBase currentMethod = MethodBase.GetCurrentMethod();
        SecureFieldBuilder.Factory()
            .SetSecureField(currentMethod, this, value);
    }
}
```

При использовании MSBuild должен быть модифицирован файл проекта путем добавления запуска данной программы после компиляции проекта.

**Выводы:** в данной главе был пошагово описан процесс разработки и описаны некоторые тонкости и пути решения возникших в ходе разработки проблем. Так же предложены несколько возможностей модификации данного решения, которые будут еще раз обозначены в заключении.

## **4. ТЕСТИРОВАНИЕ**

### **4.1. Объект испытаний**

Объектом испытаний является приложение, написанное на языке C#, использующее защиту предоставленную библиотекой SecurityField и внедренное с использованием программы MonoInjections.

### **4.2. Цель испытаний**

Целью испытаний является проверка работоспособности программы MonoInjections и модифицированного целевого приложения на наличие ошибок и необработанных исключений. Так же проверяется эффективность внедренной защиты путем попытки получить зашифрованное значение из оперативной памяти без дезасемблирования целевого приложения с целью его модификации.

### **4.3. Требования к программе и программной документации**

В качестве программной документации предъявляемой на испытания служит текущая дипломная работа и в частности техническое задание на разработку данного решения.

Основные требования к программе описаны в техническом задании к этому проекту, в дополнение к ним должны быть реализовано резервное копирование и сохранность данных, предоставлена возможность одинаково качественно защищать данные любых значимых типов (целые числа, числа с плавающей запятой, символьные переменные).

### **4.4. Средства и порядок испытаний**

В качестве средств испытаний используется программа ArtMoney для поиска значений в оперативной памяти и тестовое приложение с логированием реальных значений переменных и значений хранимых в памяти.

Порядок проведения испытаний:

1. Модификация приложения, не содержащего помеченных атрибутом свойств и без подключенной библиотеки SecureField;

2. Модификация приложения с помеченными свойствами и его запуск. Проверка возможности откатиться к неизменной версии;
3. Считывание значений переменных по умолчанию без предварительной инициализации в модифицированном приложении;
4. Применение защиты для всех следующих типов данных: Char, Int16, UInt16, Int32, UInt32, Int64, UInt64, Double, Single. Проверка работоспособности;
5. Проверка отказоустойчивости: использование различных имен для свойств;
6. Проверка отказоустойчивости: попытка защитить свойства, имеющие приватное поле для хранения значения с соответствующим стандартом именем;
7. Проверка отказоустойчивости: попытка защитить свойства других типов;
8. Проверка производительности: анализ скорости чтения и записи данных с использованием и без использования защиты;

#### 4.5. Методы испытаний

Для проведения испытаний было написано тестовое консольное приложение содержащее класс Values, в котором объявлены поля всех необходимых типов и реализованы два метода для логирования значений в которых перебираются все свойства и поля с использованием рефлексии:

```
public string ValuesLog()
{
    var type = MethodBase.GetCurrentMethod().DeclaringType;
    var props = type.GetProperties(BindingFlags.Public |
        BindingFlags.Instance);
    var log = "Values log:\n";
    foreach (var prop in props)
    {
        log += string.Format("{0}\t: {1}\n", prop.PropertyType.Name,
            prop.GetValue(this, null));
    }
    return log;
}
```

```

public string MemoryLog()
{
    var type = MethodBase.GetCurrentMethod().DeclaringType;
    var fields = type.GetFields(BindingFlags.NonPublic |
        BindingFlags.Instance).Where(f => f.Name.StartsWith("_"));
    var log = "Memory log:\n";
    foreach (var field in fields)
    {
        log += string.Format("{0}\t: {1}\n", field.FieldType.Name,
            field.GetValue(this));
    }
    return log;
}

```

При необходимости тестовая программа будет дополняться, а изменения фиксироваться в отчете.

Так же был написан простой интерфейс для тестирования, представляющий собой циклический ввод однобуквенных команд в консоли. Команды позволяют записать значение в любое из свойств и вывести лог значений и памяти на печать.

#### 4.5.1. Модификация приложения, не использующего защиту.

Для проверки данного сценария, библиотека SecureField была удалена из зависимостей, так же были удалены все атрибуты. После модификации была получена валидная сборка. Изменения исходного кода не производились, так как в программе MonoInjections отслеживаются только свойства, помеченные соответствующим атрибутом. С зависимостями проблем не возникло.

#### 4.5.2. Модификация приложения и возврат к неизменной версии.

Модификация тестового приложения происходит без каких либо ошибок. Если вдруг возникают какие-то проблемы, или просто нужно откатить изменения, программа MonoInjections перед модификацией приложения делает его резервную копию и сохраняет его в той же директории с именем \*\_bak.exe. Достаточно заменить им запускаемый файл проекта для дальнейшей работы с ним.

#### 4.5.3. Считывание значений переменных без предварительно инициализации.

Для данного сценария запускалось модифицированное приложение, и сразу вызывалась команда печати лога, был получен следующий результат (Рис. 4.1. Значения переменных по умолчанию) (в выводе представлены не все типы данных):

```
Enter: L
Values log:
Char      :
Int32     : 0
UInt16    : 0
Double    : 0

Memory log:
Char      :
Int32     : 0
UInt16    : 0
Double    : 0
```

Рис. 4.1. Значения переменных по умолчанию

#### 4.5.4. Применение защиты для всех значимых типов данных

В качестве оценки эффективности защиты использовалась программа ArtMoney, позволяющая искать в памяти конкретные значения. В качестве примера рассмотрим целочисленные значения. Для начала будем анализировать не модифицированную программу, введем команду “i 10” тем самым записав в переменную типа Int32 значение 10. В ArtMoney произведем поиск точного целочисленного значения 10 (Рис. 4.2).

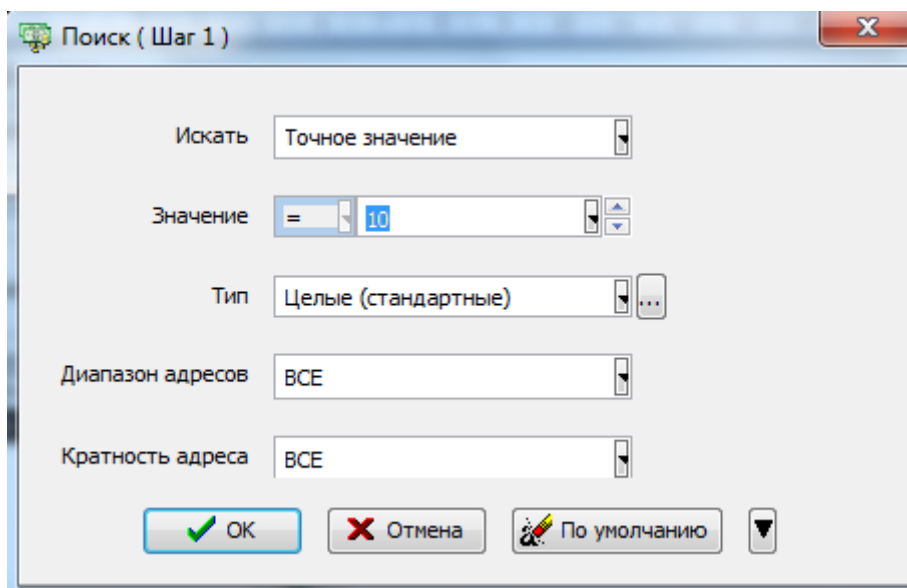


Рис. 4.2. Поиск по точному значению

В выдаче получено несколько тысяч адресов, для того чтобы найти нужны нам, изменим значение переменной используя команду “i 40” и произведем отсеивание по точному значению 40, получим следующий результат (Рис. 4.3):

Адрес	Тип		З	Описание	Адрес	Значение	Тип
01A21378	Целое 4 байта			Значение 1	01A21378	40	Целое 4 байта
04C4EE30	Целое 4 байта			Значение 2	04C4EE30	40	Целое 4 байта

Рис. 4.3. Результат отсеивания

Найдено два значения, попробуем изменить их поочередно и отслеживать изменения в программе, смотря вывод лога. Изменение первого значения дает нужный результат. Данный пример показывает, как легко изменить значения в приложении, не использующего какую-либо защиту данных.

Теперь попробуем повторить те же действия для модифицированной программы. ArtMoney находит один адрес, но изменение значения, находящегося по этому адресу, не влияет на хранимое значение в программе, что не дает никакого преимущества для пользователя.

ArtMoney также позволяет отслеживать «неизвестные» и закодированные значения. Отсевание основано на знании того изменилась переменная или нет. Было проведено порядка 20 итераций для каждого способа, не один из них не позволил найти адрес, по которому хранится модифицированное значение, либо используемая маска.

Для повышения надежности противодействия данным методам поиска можно ввести случайную составляющую: если пользователь вводит то же самое число, с определенной долей вероятности сохранять прежнее значение маски.

#### 4.5.5. Использование различных имен свойств.

Были проверены следующие возможные имена: `prop`, `PROP`, `_prop`, `_1prop`, `__prop`. Программа `MonoInjections` генерирует приватное поле, добавляя подчеркивание к названию свойства и переводя первый символ в нижний регистр. Так же в ней стоит проверка и игнорируются все свойства, начинающиеся с символа «`_`», но даже если убрать это условие, поля называются корректно.

#### 4.5.6. Попытка перезаписи поля с таким же именем.

В тестовую программу были добавлены следующие определения:

```
private double _value;

[SecureField]
public double Value { get; set; }
```

При попытке чтения значения свойства было получено исключение `AmbiguousMatchException`: "Обнаружено неоднозначное соответствие". Для исключения этой ошибки в программу `MonoInjections` была добавлена проверка на существование поля с таким же именем, при добавлении.

#### 4.5.7. Попытка защитить свойства других типов.

В тестовую программу было добавлено свойство типа `string`, при попытке записи в него выбрасывается исключение, сообщающее о том, что ожидается значение значимого типа. Согласно документации такая возможность не поддерживается, и такое поведение можно считать нормальным.

В дальнейшем возможно добавить поддержку таких типов как `string` и `decimal`. Для поддержки защиты объектов собственных классов теоретически можно сериализовывать объекты, но нет уверенности, что на практике это будет работать.

#### 4.5.8. Проверка производительности

Для проверки производительности был написан небольшой тест, в котором производилась запись и считывания для защищенного и незащищенного свойства. Для записи на каждой итерации использовалось новое случайное число, для считывания одно и то же число считывалось во временную переменную. Для каждой операции проводился миллион итераций.



Средние значения, полученные на основе 15 тестов (таблица 4.1):

Таблица 4.1

Анализ производительност. Время прохождения тестов в мс.

	Незащищенное свойство	Защищенное свойство
Чтение	25,8	6318,4
Запись	32,3	6886,5

Таким образом, обращение к защищенным полям производится примерно в 200 раз дольше, но 7мс на 1000 обращений не очень критичный результат. Тест проводился с дополнительно добавленными в коллекцию 100 свойствами.

**Выводы:** на основе проведенного тестирования, можно сказать, что разработанное решение работает довольно стабильно и показывает неплохие результаты производительности под нагрузкой. Так же в ходе тестирования были выявлены возможные пути улучшения решения и добавления новых возможностей.

## 5. ОРГАНИЗАЦИЯ РАБОЧЕГО МЕСТА

Для работы с моим программным модулем необходима персональная электронная вычислительная машина (ПЭВМ). Существуют требования и нормы к работе за ПЭВМ [16]. Для охраны здоровья и труда необходимо соблюдать все правила и требования, описанные в данной главе.

### 5.1. Требования к помещениям для работы с ПЭВМ

5.1.1. Помещения для эксплуатации ПЭВМ должны иметь естественное и искусственное освещение. Эксплуатация ПЭВМ в помещениях без естественного освещения допускается только при соответствующем обосновании и наличии положительного санитарно-эпидемиологического заключения, выданного в установленном порядке.

Естественное и искусственное освещение должно соответствовать требованиям СанПин 2.2.2/2.4.1340-03 [16]. Окна в помещениях, где эксплуатируется вычислительная техника, преимущественно должны быть ориентированы на север и северо-восток.

Оконные проемы должны быть оборудованы регулируемыми устройствами типа: жалюзи, занавесей, внешних козырьков и др.

5.1.2. Площадь на одно рабочее место пользователей ПЭВМ с ВДТ (видеодисплейные терминалы) на базе электроннолучевой трубки (ЭЛТ) должна составлять не менее 6 м<sup>2</sup>. Для ВДТ на базе плоских дискретных экранов (жидкокристаллические, плазменные) – 4,5 м<sup>2</sup>.

При использовании ПЭВМ с ВДТ на базе ЭЛТ (без вспомогательных устройств – принтер, сканер и др.), отвечающих требованиям международных стандартов безопасности компьютеров, с продолжительностью работы менее 4-х часов в день допускается минимальная площадь 4,5 м<sup>2</sup> на одно рабочее место пользователя.

5.1.3. Помещения, где размещаются рабочие места с ПЭВМ, должны быть оборудованы защитным заземлением (занулением) в соответствии с техническими требованиями по эксплуатации.

5.1.4. Не следует размещать рабочие места с ПЭВМ вблизи силовых кабелей вводов, высоковольтных трансформаторов, технологического оборудования, создающего помехи в работе ПЭВМ.

## **5.2. Требования к микроклимату, содержанию энтропинов и вредных химических веществ в воздухе**

5.2.1. В производственных помещениях, в которых работа с использованием ПЭВМ является вспомогательной, температура, относительная влажность и скорость движения воздуха на рабочих местах должны соответствовать действующим санитарным нормам микроклимата производственных помещений.

5.2.2. В помещениях, оборудованных ПЭВМ, проводится ежедневная влажная уборка и систематическое проветривание после каждого часа работы на ПЭВМ.

5.2.3. Содержание вредных химических веществ в производственных помещениях, в которых работа с использованием ПЭВМ является основной (диспетчерские, операторские, расчетные, кабины и посты управления, залы вычислительной техники и др.), не должно превышать предельно допустимых концентраций загрязняющих веществ в атмосферном воздухе населенных мест в соответствии с действующими гигиеническими нормативами.

## **5.3. Требования к уровням шума и вибрации**

5.3.1. В производственных помещениях при выполнении основных или вспомогательных работ с использованием ПЭВМ уровни шума на рабочих местах не должны превышать предельно допустимых значений, установленных для данных видов работ в соответствии с действующими санитарно-эпидемиологическими нормативами.

5.3.2. При выполнении работ с использованием ПЭВМ в производственных помещениях уровень вибрации не должен превышать допустимых значений вибрации для рабочих мест в соответствии с действующими санитарно-эпидемиологическими нормативами

В помещениях всех типов образовательных и культурно-развлекательных учреждений, в которых эксплуатируются ПЭВМ, уровень вибрации не должен превышать допустимых значений для жилых и общественных зданий в соответствии с действующими санитарно-эпидемиологическими нормативами.

5.3.3. Шумящее оборудование (печатающие устройства, серверы и т.п.), уровни шума которого превышают нормативные, должно размещаться вне помещений с ПЭВМ.

#### **5.4. Требования к освещению**

5.4.1. Рабочие столы следует размещать таким образом, чтобы ВДТ были ориентированы боковой стороной к световым проемам, чтобы естественный свет падал преимущественно слева.

5.4.2. Искусственное освещение в помещениях для эксплуатации ПЭВМ должно осуществляться системой общего равномерного освещения. В производственных и административно-общественных помещениях, в случаях преимущественной работы с документами, следует применять системы комбинированного освещения (к общему освещению дополнительно устанавливаются светильники местного освещения, предназначенные для освещения зоны расположения документов).

5.4.3. Освещенность на поверхности стола в зоне размещения рабочего документа должна быть 300 – 500 лк. Освещение не должно создавать бликов на поверхности экрана. Освещенность поверхности экрана не должна быть более 300 лк.

5.4.4. Следует ограничивать прямую блесткость от источников освещения, при этом яркость светящихся поверхностей (окна, светильники и др.), находящихся в поле зрения, должна быть не более 200 кд/м<sup>2</sup>.

5.4.5. Для освещения помещений с ПЭВМ следует применять светильники с зеркальными параболическими решетками, укомплектованными электронными пуско-регулирующими аппаратами (ЭПРА). Допускается использование многоламповых светильников с ЭПРА, состоящими из равного числа опережающих и отстающих ветвей.

Применение светильников без рассеивателей и экранирующих решеток не допускается. При отсутствии светильников с ЭПРА лампы многоламповых светильников или рядом расположенные светильники общего освещения следует включать на разные фазы трехфазной сети.

5.4.6. Для обеспечения нормируемых значений освещенности в помещениях для использования ПЭВМ следует проводить чистку стекол оконных рам и светильников не реже двух раз в год и проводить своевременную замену перегоревших ламп.

## 5.5. Требования к уровню электромагнитных полей

5.5.1. Временные допустимые уровни ЭМП, создаваемых ПЭВМ на рабочих местах пользователей представлены в таблице 5.1.

Таблица 5.1

Временные допустимые уровни ЭМП, создаваемые ПЭВМ

Наименование параметров		ВДУ ЭМП
Напряженность электрического поля	В диапазоне частот 5Гц - 2КГц	25В/м
	В диапазоне частот 2КГц - 400КГц	2,5В/м
Плотность магнитного потока	В диапазоне частот 5Гц - 2КГц	250нТл
	В диапазоне частот 2КГц - 400КГц	25нТл
Электростатический потенциал экрана видеомонитора		500В

5.5.2. Для дисплеев на ЭЛТ частота обновления изображения должна быть не менее 75 Гц при всех режимах разрешения экрана, гарантируемых нормативной документацией на конкретный тип дисплея и не менее 60 Гц для дисплеев на плоских дискретных экранах (жидкокристаллических, плазменных и т.п.).

## 5.6. Требования к визуальным параметрам ВДТ

5.6.1. Предельно допустимые значения визуальных параметров ВДТ, контролируемые на рабочих местах, представлены в таблице 5.2.

Таблица 5.2

Визуальные параметры ВДТ, контролируемые на рабочих местах

Параметры	Допустимые значения
Яркость белого поля	Не менее 35кд/кв.м.
Неравномерность яркости рабочего поля	Не более $\pm 20$
Контрастность (для монохромного режима)	Не менее 3:1
Временная нестабильность изображения (мелькания)	Не должна фиксироваться
Пространственная нестабильность изображения (дрожание)	Не более $2 \times 10^{-4}L$ , где L - проектное расстояние наблюдения, мм.

**Вывод:** в данной главе были установлены правил безопасности по работе с моей библиотекой.

## 6. ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ

За экономическую часть своего проекта я предлагаю расчет затрат, которые необходимы для разработки моего решения. В перечень затрат вошли как стоимость самой работы, так и стоимость начального материала и ресурсов.

### 6.1. Определение трудоемкости выполнения разработки

Трудоёмкость характеризует затраты рабочего времени на производство определённой единицы продукции или на выполнение конкретной технологической операции.

Трудоемкость определяется методом экспертных оценок по сумме трудоемкости этапов работ, выраженных в днях. Ожидаемое время выполнения работ  $T_{ож}$  определяется по формуле:

$$T_{ож} = \frac{3T_{min} + 2T_{max}}{5},$$

где  $T_{min}$  - минимально возможное время выполнения заданной работы;

$T_{max}$  - максимально возможное время.

Основные этапы работ, которые необходимо выполнить для разработки устройства, приведены в таблице 6.1.

Таблица 6.1

Трудоемкость выполнения разработки

Наименование этапов разработки	$T_{min}$ , дни	$T_{max}$ , дни	$T_{ож}$ , дни
Составление технического задания	1	1	1
Анализ существующих решений	1	2	1.4
Установка необходимого ПО	2	3	2.4
Разработка ПО	15	20	17
Тестирование	3	5	3.8
Написание технической документации	10	15	12
<b>Итого:</b>	<b>32</b>	<b>46</b>	<b>37.6</b>

## 6.2. Расчет затрат на разработку

Сметная стоимость затрат является основным документом, на основании которого осуществляется планирование и учет затрат на разработку устройства.

В состав сметной стоимости могут входить следующие статьи затрат:

1. Материалы, покупные изделия.
2. Специальное оборудование для проведения разработки.
3. Заработная плата разработчиков.
4. Отчисления на социальные нужды.
5. Затраты на электроэнергию для технологических целей.
6. Затраты на командировки
7. Контрагентские работы.
8. Прочие затраты.
9. Накладные расходы.

Сметная стоимость определяется методом сметного калькулирования. Метод сметного калькулирования основан на прямом определении затрат по определенным статьям.

Стоимость материалов, покупных изделий ( $C_M$ ) оценивается по действующим рыночным ценам с учетом величины транспортно-заготовительных расходов по формуле:

$$C_M = \sum_{i=1}^n H_{mi} C_{mi} K_{TЗ} - \sum_{i=1}^n H_{oi} U_{oi},$$

где  $n$  – число позиций применяемых материалов;

$H_{mi}$  – норма расхода материала, кг;

$C_{mi}$  – цена материала, руб/кг;

$K_{TЗ}$  – величина транспортно-заготовительных расходов.



6.2.1. Затраты по статье «Материалы, покупные изделия» приведены в табл. 6.2.

Таблица 6.2

## Затраты по статье «Материалы, покупные изделия»

Наименование материалов, покупных изделий и п/фабрикатов	Ед. изм.	Кол-во	Цена единицы (руб.)	Сумма (руб.)	Транспортно-заготовительные расходы	Итого матер. затрат (руб.)
Бумага для принтера	Пачка	1	150	150	1,2	180
Картридж для принтера	шт.	1	1100	1100	1,05	1155
<b>Итого:</b>				1250		<b>1335</b>

Остальные использованные инструменты распространяются на основе свободной лицензии или в демо версии, которой хватает для нужд проекта.

6.2.2. В качестве специального оборудования для проведения разработки использовался компьютер, балансовая стоимость которого равна **17 000** рублей.

Амортизация оборудования рассчитывается по следующей формуле:

$$A = V * k * t,$$

где V – стоимость оборудования;

k – норма амортизации;

t – время эксплуатации, год.

Норма амортизации можно рассчитать по формуле:

$$k = 1/T * 100\%,$$

где T – срок полезного использования, год.

В нашем случае срок полезного для оборудования использования составляет 7 лет, а время эксплуатации – 4 месяца, подставим значения в формулу и получим:

$$A = 17000 * 1/7 * 4/12 = 809,52$$

Остаточная стоимость оборудования составит **16 190,48** рублей.

6.2.3. Заработная плата разработчиков ( $C_3$ ) рассчитывается по формуле.

$$C_3 = 3П * (1 + K_{\text{доп}}) * T_{\text{ож}}/22,$$

где 3П – заработная плата разработчика;

$K_{\text{доп}}$  – коэффициент дополнительной заработной платы;

$T_{\text{ож}}$  – ожидаемое время выполнения работ;

22 – количество рабочих дней в месяце.

Для текущего проекта: 3П = 25 000 руб.,  $K_{\text{доп}} = 0$ ,  $T_{\text{ож}} = 37,6$  дней.

**Итого:  $C_3 = 42\,727,27$  руб.**

6.2.4. Отчисления в социальные внебюджетные фонды определяются по формуле:

$$C_{\text{сф}} = C_3 r / 100,$$

где  $r$  – суммарная величина отчислений в социальные внебюджетные фонды (30%).

$$C_{\text{сф}} = 42\,727,27 * 0,3 = \mathbf{12\,818,18 \text{ руб.}}$$

6.2.5. Затраты на электроэнергию, потребляемую ЭВМ, рассчитываются по следующей формуле:

$$C_{\text{эн}} = WTC_k K_{wi},$$

где  $W$  – установленная мощность ЭВМ, кВт;

$T$  – время использования для проведения разработки, ч;

$C_k$  – цена одного кВт/ч электроэнергии;

$K_{wi}$  – коэффициент использования мощности.

$$C_{\text{эн}} = 0,4 * 37,6 * 8 * 2,39 * 0,9 = \mathbf{258,81 \text{ руб.}}$$

6.2.6. Расходы на служебные командировки, а так же расходы на контрагентные работы в данной разработке не предусматриваются.

6.2.8. Статья «Прочие затраты» в данной работе отсутствует.

6.2.9. Накладные расходы ( $C_n$ ) начисляются в процентах к основной заработной плате (140%). Величина накладных расходов:

$$C_n = 42\,727,27 * 1,4 = \mathbf{59\,818,18 \text{ руб.}}$$

**Вывод:** на основании полученных данных по отдельным статьям затрат, составляем смету затрат на разработку в целом по форме, приведённой в таблице 6.3:

Таблица 6.3

## Смета затрат на разработку

Статьи затрат	Условные обозначения	Затраты по статьям (руб.)
1. Материалы, покупные изделия	$C_m$	1 335.00
2. Специальное оборудование для проведения разработки	$C_{об}$	16 190.48
3. Заработная плата разработчиков	$C_z$	42 727.27
4. Отчисления на социальные нужды	$C_{сф}$	12 818.18
5. Затраты на электроэнергию для технологических целей	$C_{эн}$	258.81
6. Затраты на командировки и	$C_{ком}$	-
7. Контрагентские работы	$C_{кр}$	-
8. Прочие затраты	$C_n$	-
9. Накладные расходы	$C_n$	59 818.18
<b>Общая сметная стоимость</b>		<b>133 147.92</b>

Итоговые сметные затраты на разработку проекта составили 133 147.92 рублей.

## ЗАКЛЮЧЕНИЕ

В ходе анализа существующих средств и технологий защиты программного обеспечения, написанного с использованием технологии .NET, был выявлен основной недостаток: все существующие на рынке решения, ориентированы на защиту кода программных продуктов, и не было найдено продуктов, защищающих информацию, хранящуюся в памяти приложения, от модификации. Эта причина стала основной для формулирования задачи по разработке приложения для защиты данных в памяти приложений и работы над данным проектом. В ходе решения поставленных задач, были достигнуты следующие результаты:

1. Проведен анализ существующих решений и выявлены их преимущества и недостатки;
2. Определены возможные эффективные пути по защите памяти приложения от чтения и модификации информации;
3. Реализован механизм инъекции кода без необходимости модификации кода исходного приложения;
4. Создано рабочее приложение, позволяющее эффективно защищать память .NET программ от несанкционированного доступа к памяти.

Практическая значимость данной работы заключается в разработке нового отдельного решения для такого рода защиты. Сам подход по защите не новый, но он применяется разработчиками в рамках каждого отдельного проекта, как его часть, а не как сторонний инструмент. Данное решение является простым в использовании и в будущем может быть расширено для использования разработчиками своих алгоритмов защиты.

Разработанное решение является законченным программным продуктом, но имеет несколько путей для дальнейшей модификации и расширения функционала, кратко о которых уже было сказано в данной работе.

Основными из них являются:

- Дополнительная поддержка ссылочных типов данных, как стандартных, так и пользовательских;
- Добавление простой обфускации, встраиваемой по такому же принципу и реализованной на добавлении невалидных конструкций на языке низкого уровня;
- Реализация возможности использования своего алгоритма защиты данных, путем вынесения его реализации из библиотеки и создания специальных интерфейсов по подключению дополнительных алгоритмов.

Таким образом, используя данное решение совместно с существующими на рынке обфускаторами или протекторами, можно существенно обезопасить свой продукт. Не стоит забывать, что эффективным является комплексный подход к защите, и любая защита лишь увеличивает время взлома и не является абсолютной.

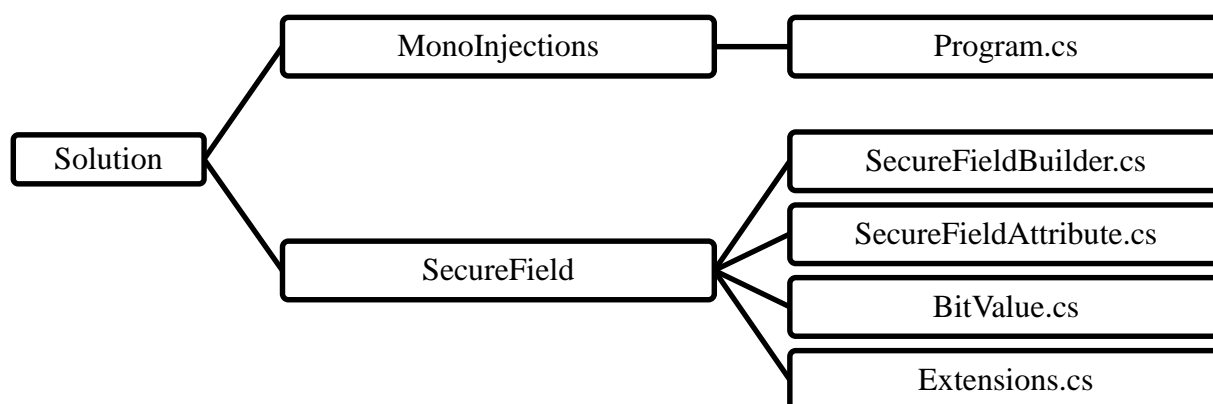
## СПИСОК ЛИТЕРАТУРЫ

1. William F.Z. Concepts and Techniques in Software Watermarking and Obfuscation // CiteSeerX. 2007 [Электронный ресурс]. – Режим доступа: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.8892&rep=rep1&type=pdf> (дата обращения: 12.04.2014).
2. Collberg C., Thomborson C. Software Watermarking: Models and Dynamic Embeddings // CiteSeerX. 1999 [Электронный ресурс]. – Режим доступа: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.33.8088&rep=rep1&type=pdf> (дата обращения: 12.04.2014).
3. Чернов А.В. Анализ запутывающих преобразований программ // ЦИТ Форум. 2003 [Электронный ресурс]. – Режим доступа: <http://citforum.ru/security/articles/analysis/> (дата обращения: 11.04.2014).
4. Collberg C., Thomborson C., Low D. A Taxonomy of Obfuscating Transformations // Laboratoire Bordelais de Recherche en Informatique. 1997 [Электронный ресурс]. – Режим доступа: <http://www.labri.fr/perso/fleury/courses/SS/download/papers/obfuscation-survey-collberg.pdf> (дата обращения: 11.04.2014).
5. Ледовских И. Метрики сложности кода // ИСП РАН. 2012 [Электронный ресурс]. – Режим доступа: [http://www.ispras.ru/ru/preprints/docs/rep\\_25\\_2013.pdf](http://www.ispras.ru/ru/preprints/docs/rep_25_2013.pdf) (дата обращения: 11.04.2014).
6. eRaider. Защита.NET приложений — всё же, во что заворачивать селёдку? // Хабрахабр. 2010 [Электронный ресурс]. – Режим доступа: <http://habrahabr.ru/post/106262/> (дата обращения: 11.04.2014).
7. Трамвон А. Обзор систем защиты ПО для Windows от нелегального использования // Хабрахабр. 2014 [Электронный ресурс]. – Режим доступа: <http://habrahabr.ru/post/215553/> (дата обращения: 11.04.2014).

8. Wang P. Tamper Resistance for Software // CiteSeerX. 2005 [Электронный ресурс]. – Режим доступа: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.8556&rep=rep1&type=pdf> (дата обращения: 12.04.2014).
9. Wurster G. A generic attack on hashing-based software tamper resistance // School of Computer Science. 2005 [Электронный ресурс]. – Режим доступа: <http://people.scs.carleton.ca/~gwurster/publications/Thesis-2005.pdf> (дата обращения: 12.04.2014).
10. Antelle. Обзор обфускаторов для.NET // Хабрахабр. 2010 [Электронный ресурс]. – Режим доступа: <http://habrahabr.ru/post/97062/> (дата обращения: 11.04.2014).
11. Павлов В. Аспектно-ориентированное программирование [ЭЛЕКТРОННЫЙ РЕСУРС]. – РЕЖИМ ДОСТУПА: <http://www.online-ane.ru/oopAOP.pdf> (дата обращения: 17.05.2014).
12. Рихтер Д. CLR via C#. 4th ed. Питер, 2013.
13. Smacchia P. Mono.Cecil vs. System.Reflection 2008 [Электронный ресурс]. – Режим доступа: <http://codebetter.com/patricksmacchia/2008/03/18/mono-cecil-vs-system-reflection/> (дата обращения: 25.05.2014).
14. Reinle F. Cecil:FAQ [Электронный ресурс]. – Режим доступа: <http://www.mono-project.com/Cecil:FAQ> (дата обращения: 25.05.2014).
15. Чистяков В., "MSBuild," *RSDN Magazine*, No. 6, 2004.
16. Министерство здравоохранения РФ. СанПиН 2.2.2/2.4.1340-03 «Гигиенические требования к ПЭВМ и организации работы» 2003 [Электронный ресурс]. – Режим доступа: [http://www.sysengineering.ru/media/511/legislation\\_02.pdf](http://www.sysengineering.ru/media/511/legislation_02.pdf) (дата обращения: 05.17.2014).

## Исходный код программы

Структура файлов решения:



### ~/MonoInjections/Program.cs

```

class Program
{
    private static TypeReference _secureFieldBuilderRef;
    private static MethodReference _secureFieldFactoryRef;
    private static MethodReference _getSecureFieldRef;
    private static MethodReference _setSecureFieldRef;
    private static TypeReference _methodBaseRef;
    private static MethodReference _getCurrentMethodRef;

    static void Main(string[] args)
    {
        if (args.Length == 0)
            return;
        string assemblyPath = args[0];

        var assembly = AssemblyDefinition.ReadAssembly(assemblyPath);
        InitReferences(assembly);

        //Делаем резервную копию файла
        assembly.Write(assemblyPath + ".bak");
        assembly = InjectToAssembly(assembly);
        assembly.Write(assemblyPath);
    }
    //Инициализируем все зависимости
    private static void InitReferences(AssemblyDefinition assembly)
    {
        _methodBaseRef = assembly.MainModule.Import(typeof(MethodBase));
        _getCurrentMethodRef = assembly.MainModule
            .Import(typeof(MethodBase).GetMethod("GetCurrentMethod"));
        _secureFieldBuilderRef = assembly.MainModule.Import(typeof(SecureFieldBuilder));
        _secureFieldFactoryRef = assembly.MainModule
            .Import(typeof(SecureFieldBuilder).GetMethod("Factory"));
        _getSecureFieldRef = assembly.MainModule
            .Import(typeof(SecureFieldBuilder).GetMethod("GetSecureField"));
        _setSecureFieldRef = assembly.MainModule
            .Import(typeof(SecureFieldBuilder).GetMethod("SetSecureField"));
    }
}
  
```



```

private static AssemblyDefinition InjectToAssembly(AssemblyDefinition assembly)
{
    foreach (var typeDef in assembly.MainModule.Types)
    {
        //Проходим по всем свойствам и получаем те, к которым применен нужен атрибут
        var properties = typeDef.Properties.Where(p => p.CustomAttributes.Any(attr =>
            attr.AttributeType.Name == "SecureFieldAttribute") && !p.Name.StartsWith("_"));
        foreach (var prop in properties)
        {
            //Генерируем имя для поля
            var fieldName = String.Format("_{0}{1}", char.ToLower(prop.Name[0]),
                prop.Name.Substring(1));
            //Если поле с таким именем уже есть, не добавляем его
            if(typeDef.Fields.All(f => f.Name != fieldName))
            {
                var field = new FieldDefinition(fieldName, Mono.Cecil.FieldAttributes.Private,
                    prop.PropertyType);
                typeDef.Fields.Add(field);
            }
            ReplaceGetMethod(prop);
            ReplaceSetMethod(prop);
        }
    }
    return assembly;
}

private static void ReplaceSetMethod(PropertyDefinition prop)
{
    var method = prop.SetMethod;
    var ilProc = method.Body.GetILProcessor();
    // необходимо установить InitLocals в true, так как если он находился в false
    // (в методе изначально не было локальных переменных)
    // а теперь локальные переменные появятся - верификатор IL кода выдаст ошибку.
    ilProc.Body.Instructions.Clear();
    ilProc.Body.InitLocals = true;

    // создаем две локальных переменных для MethodBase и SecureFieldBuilder
    var currentMethodVar = new VariableDefinition(_methodBaseRef);
    var builderVar = new VariableDefinition(_secureFieldBuilderRef);
    ilProc.Body.Variables.Add(currentMethodVar);
    ilProc.Body.Variables.Add(builderVar);

    //Добавляем инструкцию Return
    ilProc.Append(Instruction.Create(OpCodes.Ret));
    var firstInstruction = ilProc.Body.Instructions[0];

    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Nop));
    // получаем текущий метод MethodBase.GetCurrentMethod();
    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Call,
        _getCurrentMethodRef));
    // помещаем результат со стека в переменную currentMethod
    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Stloc,
        currentMethodVar));
    // Вызываем SecureFieldBuilder.Factory()
    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Call,
        _secureFieldFactoryRef));
    // помещаем результат со стека в переменную builder
    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Stloc, builderVar));

    // Вызываем builder.SetSecureField(currentMethod, this, value);
    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Ldloc, builderVar));

```

```

ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Ldloc,
    currentMethodVar));
ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Ldarg_0));
ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Ldarg_1));
ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Box,
    prop.PropertyType));
ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Callvirt,
    _setSecureFieldRef));
}
private static void ReplaceGetMethod(PropertyDefinition prop)
{
    var method = prop.GetMethod;
    var ilProc = method.Body.GetILProcessor();

    ilProc.Body.Instructions.Clear();
    ilProc.Body.InitLocals = true;

    var currentMethodVar = new VariableDefinition(_methodBaseRef);
    var builderVar = new VariableDefinition(_secureFieldBuilderRef);
    ilProc.Body.Variables.Add(currentMethodVar);
    ilProc.Body.Variables.Add(builderVar);

    ilProc.Append(Instruction.Create(OpCodes.Ret));
    var firstInstruction = ilProc.Body.Instructions[0];

    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Nop));
    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Call,
        _getCurrentMethodRef));
    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Stloc,
        currentMethodVar));
    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Call,
        _secureFieldFactoryRef));
    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Stloc, builderVar));

    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Ldloc, builderVar));
    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Ldloc,
        currentMethodVar));
    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Ldarg_0));
    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Callvirt,
        _getSecureFieldRef));
    ilProc.InsertBefore(firstInstruction, Instruction.Create(OpCodes.Unbox_Any,
        prop.PropertyType));
}
}

```

### ~/SecureField/SecureFieldAttribute.cs

```

[AttributeUsage(AttributeTargets.Property)]
public class SecureFieldAttribute : Attribute
{
    public SecureFieldAttribute()
    {
    }
}

```

~/SecureField/SecureFieldBuilder.cs

```

public class SecureFieldBuilder
{
    private static readonly SecureFieldBuilder Instance = new SecureFieldBuilder();
    private static readonly Hashtable Hash = new Hashtable();
    private static readonly Random Rand = new Random();

    protected SecureFieldBuilder()
    {
    }
    public void AddHash()
    {
        var rand = Rand.Next();
        Hash.Add(rand.ToString(), rand);
    }
    static public SecureFieldBuilder Factory()
    {
        return Instance;
    }

    public object GetSecureField(MethodBase method, object obj)
    {
        var numField = method.DeclaringType.GetField(method.GetFieldName(),
            BindingFlags.NonPublic | BindingFlags.Instance);
        var fullPropName = method.GetFullPropertyName();

        var val = numField.GetValue(obj);
        var mask = (byte[])Hash[fullPropName];
        if (mask == null)
            return val;

        var byteVal = new BitValue(val);
        byteVal.Xor(mask);
        return byteVal.ToValue();
    }

    public void SetSecureField(MethodBase method, object obj, object val)
    {
        var numField = method.DeclaringType.GetField(method.GetFieldName(),
            BindingFlags.NonPublic | BindingFlags.Instance);
        var fullPropName = method.GetFullPropertyName();

        var byteVal = new BitValue(val);
        var mask = new byte[byteVal.Bytes.Length];
        Rand.NextBytes(mask);

        if (Hash.ContainsKey(fullPropName))
            Hash[fullPropName] = mask;
        else
            Hash.Add(fullPropName, mask);

        byteVal.Xor(mask);
        numField.SetValue(obj, byteVal.ToValue());
    }
}

```

~/SecureField/BitValue.cs

```

enum SerializationType { Char, Int16, UInt16, Int32, UInt32, Int64, UInt64, Double,
    Single, None };

public struct BitValue
{
    public byte[] Bytes;

    private readonly SerializationType _serializationType;

    public BitValue(object obj)
    {
        _serializationType = SerializationType.None;
        Bytes = null;

        var type = obj.GetType();
        if (!type.IsValueType)
            throw new ArgumentException("Ожидается значимый тип");

        switch (type.Name)
        {
            case "Char":           _serializationType = SerializationType.Char; break;
            case "Int16":          _serializationType = SerializationType.Int16; break;
            case "UInt16":         _serializationType = SerializationType.UInt16; break;
            case "Int32":          _serializationType = SerializationType.Int32; break;
            case "UInt32":         _serializationType = SerializationType.UInt32; break;
            case "Int64":          _serializationType = SerializationType.Int64; break;
            case "UInt64":         _serializationType = SerializationType.UInt64; break;
            case "Double":         _serializationType = SerializationType.Double; break;
            case "Single":         _serializationType = SerializationType.Single; break;
        }
        GetBytes(obj);
    }
    private void GetBytes(object obj)
    {
        switch (_serializationType)
        {
            case SerializationType.Char: Bytes = BitConverter.GetBytes((Char)obj); return;
            case SerializationType.Int16: Bytes = BitConverter.GetBytes((Int16)obj); return;
            case SerializationType.UInt16: Bytes = BitConverter.GetBytes((UInt16)obj); return;
            case SerializationType.Int32: Bytes = BitConverter.GetBytes((Int32)obj); return;
            case SerializationType.UInt32: Bytes = BitConverter.GetBytes((UInt32)obj); return;
            case SerializationType.Int64: Bytes = BitConverter.GetBytes((Int64)obj); return;
            case SerializationType.UInt64: Bytes = BitConverter.GetBytes((UInt64)obj); return;
            case SerializationType.Double: Bytes = BitConverter.GetBytes((Double)obj); return;
            case SerializationType.Single: Bytes = BitConverter.GetBytes((Single)obj); return;
        }
        Bytes = null;
    }
    public object ToValue()
    {
        switch (_serializationType)
        {
            case SerializationType.Char: return BitConverter.ToChar(Bytes, 0);
            case SerializationType.Int16: return BitConverter.ToInt16(Bytes, 0);
            case SerializationType.UInt16: return BitConverter.ToUInt16(Bytes, 0);
            case SerializationType.Int32: return BitConverter.ToInt32(Bytes, 0);
            case SerializationType.UInt32: return BitConverter.ToUInt32(Bytes, 0);
        }
    }
}

```

```

        case SerializationType.Int64: return BitConverter.ToInt64(Bytes, 0);
        case SerializationType.UInt64: return BitConverter.ToUInt64(Bytes, 0);
        case SerializationType.Double: return BitConverter.ToDouble(Bytes, 0);
        case SerializationType.Single: return BitConverter.ToSingle(Bytes, 0);
    }
    return null;
}
public void Xor(byte[] bytes)
{
    Bytes = Bytes.Xor(bytes);
}
}

```

### ~/SecureField/Extensions.cs

```

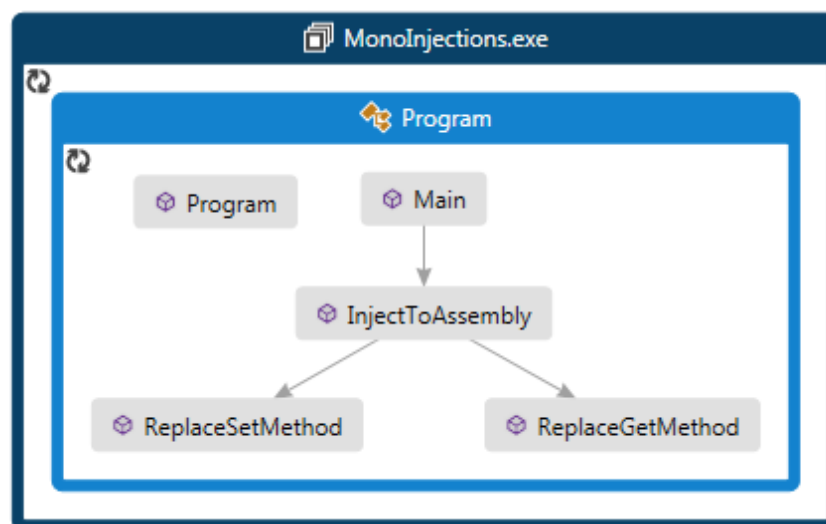
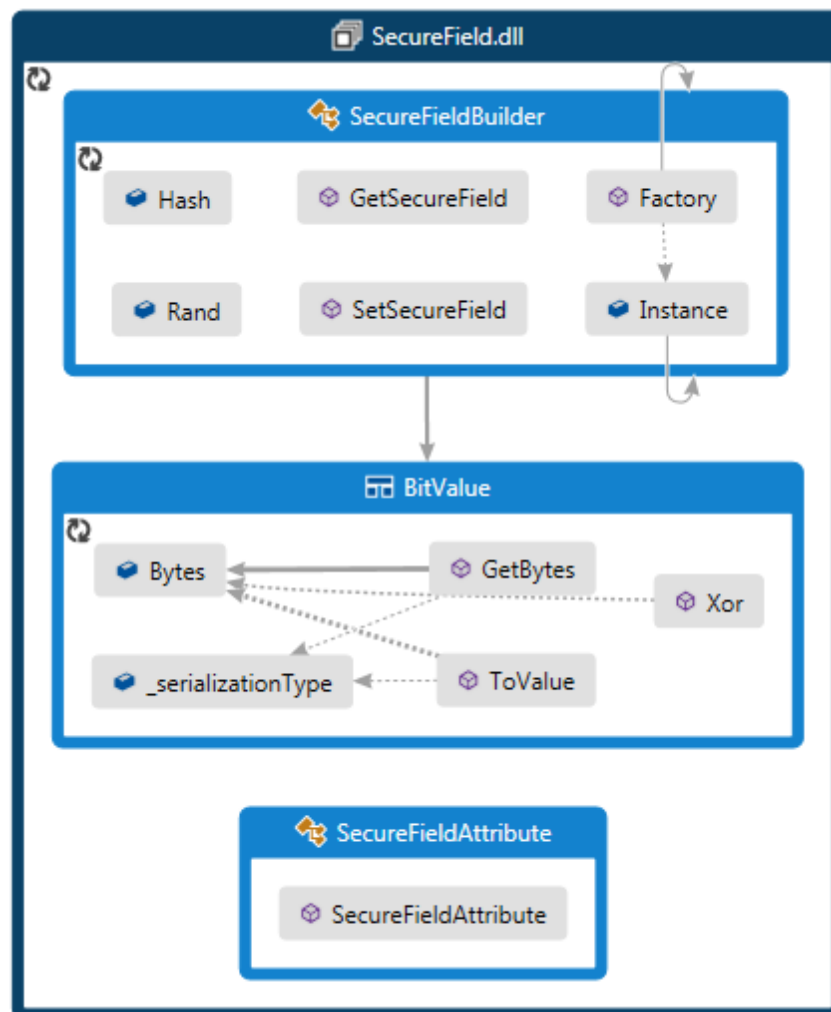
static class MethodBaseExt
{
    static public string GetFieldName(this MethodBase m)
    {
        return String.Format("_{0}{1}", char.ToLower(m.Name[4]), m.Name.Substring(5));
    }
    static public string GetFullPropertyName(this MethodBase m)
    {
        return String.Format("{0}.{1}", m.DeclaringType.Name, m.Name.Substring(4));
    }
}

public static class ByteArrayExt
{
    public static byte[] Xor(this byte[] buffer1, byte[] buffer2)
    {
        for (int i = 0; i < buffer2.Length; i++)
            buffer1[i] ^= buffer2[i];
        return buffer1;
    }
}

```

## Некоторые использованные ИЛ команды

КОП	Значение	Аргумент	Описание
<code>nop</code>	<code>0x0000</code>	-	Ничего не делает, используется для выравнивания кода.
<code>ret</code>	<code>0x002a</code>	-	Производит возврат из метода. Если используется возвращаемое значение, то значение со стека вызываемой функции помещается в стек вызывающей функции.
<code>call</code>	<code>0x0028</code>	Дескриптор метода	Вызывает метод, описанный в аргументе
<code>callvirt</code>	<code>0x006f</code>	Дескриптор метода	Вызывает метод, ассоциированный с объектом, который помещается в стек перед аргументами метода.
<code>stloc</code>	<code>0xfe0e</code>	Локальная переменная	Забирает значение со стека и помещает в указанную локальную переменную.
<code>ldloc</code>	<code>0xfe0c</code>	Локальная переменная	Помещает значение указанной локальной переменной в стек
<code>ldarg.x</code>	<code>0x000x</code>	-	Помещает указанный аргумент функции в стек.
<code>box</code>	<code>0x008c</code>	Тип	Конвертирует значение переменной в стеке в упакованный вид. В случае значимых типов, переменная просто упаковывается, в случае ссылочных, создается копия объекта. В любом случае выходное значение помещается в стек.
<code>unbox.any</code>	<code>0x00a5</code>	Тип	Для значимых типов извлекает переменную значимого типа из объекта (аналогично <code>unbox</code> ), для ссылочных типов производится приведение типов (аналогично <code>castclass</code> ).
<code>br</code>	<code>0x002b</code>	Команда	Инструкция безусловного перехода (аргумент 4 байта), короткая форма <code>br.s</code> (1 байт).
<code>ldfld</code>	<code>0x007b</code>	Тип, поле	Помещает в стек значение поля объекта, находящегося в стеке.
<code>stfld</code>	<code>0x007d</code>	Тип, поле	Перезаписывает значение поля объекта, находящегося в стеке новым значением, которое должно находиться в стеке после самого объекта.



					ИИТ.ВКРС14ГС.0001		
Изм.	Лист	№ докум.	Подпись	Дата			
Выполнил	Гладышев				Схема классов проекта		
Проверил	Сальников						
					Лит.	Лист	Листов
						1	1
					СПбГПУ зр.53505/2		



					ИИТ.ВКРС14ГС.0002		
Изм.	Лист	№ докум.	Подпись	Дата			
Выполнил	Гладышев				Блок-схема метода SetSecureField		
Проверил	Сальников						
					Лит.	Лист	Листов
						1	1
					СПбГПУ зр.53505/2		





					ИИТ.ВКРС14ГС.0003		
Изм.	Лист	№ докум.	Подпись	Дата			
Выполнил	Гладышев				Лит.	Лист	Листов
Проверил	Сальников					1	1
					СПбГПУ зр.53505/2		

Блок-схема метода  
GetSecureField