Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Южно-Уральский государственный университет»
(национальный исследовательский университет)
Факультет математики, механики и компьютерных наук
Кафедра дифференциальных и стохастических уравнений

РАБОТА ПРОВЕРЕНА
Рецензент, кандидат технических наук,
доцент
_____ / Б.М. Кувшинов /
«___» _____2016 г.

ДОПУСТИТЬ К ЗАЩИТЕ
Зав. кафедрой дифференциальных и
стохастических уравнений, ЮУрГУ,
доктор физ-мат. наук, доцент
_____ / С.А. Загребина /
«___» _____2016 г.

РАЗРАБОТКА ПРИЛОЖЕНИЯ ДЛЯ ВНЕДРЕНИЯ И ВЕРИФИКАЦИИ ВОДЯНЫХ ЗНАКОВ В ИСХОДНЫЕ КОДЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
ЮУрГУ – 09.04.04.2016.129-246.ВКР

Руководитель, канд. физ.-мат. наук,
доцент
_____ / С.М. Елсаков /
«___» _____2016 г.

Автор, студент группы ММиКН-293
_____ / Д.С. Комяков /
«25» мая 2016 г.

Нормоконтролер, канд. физ.-мат. наук,
доцент
_____ / М.А. Сагадеева /
«2» _____2016 г.

Челябинск 2016

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Южно-Уральский государственный университет»
(национальный исследовательский университет)
Факультет математики, механики и компьютерных наук
Кафедра уравнений математической физики

З А Д А Н И Е

студенту группы ММиКН-293
Комякову Дмитрию Сергеевичу
на выпускную квалификационную работу
по направлению 09.04.04 – ПРОГРАММНАЯ ИНЖЕНЕРИЯ

1. **Тема диссертации:** «Разработка приложения для внедрение и верификации водяных знаков в исходные коды программного обеспечения».
   (Утверждена приказом по университету от «15» апреля 2016г. № 661)
2. **Перечень подлежащих исследованию вопросов**
   2.1. Литературный обзор, история вопроса и результаты предыдущих исследований
   2.2. Постановка задачи
   2.3. Методы исследования и ожидаемые результаты
3. **Календарный план подготовки выпускной квалификационной работы**

Наименование этапов дипломной работы	Срок выполнения этапов работы	Отметка о выполнении
1. Обзор существующих методов и решений	01.02.16 – 14.02.16	
2. Выбор и разработка методов и алгоритмов решения	15.02.16 – 28.02.15	
3. Реализация и тестирование программного обеспечения	29.02.16 – 15.04.16	
4. Подготовка текста выпускной квалификационной работы	15.04.16 – 15.05.16	
5. Проверка и рецензирование работы руководителем, исправление замечаний	15.05.16 – 25.05.16	
6. Подготовка доклада и текста выступления	26.05.16 – 10.06.16	
7. Внешнее рецензирование	20.05.16 – 05.06.16	
8. Защита дипломной работы	10.06.16 – 20.06.16	

4. **Дата выдачи задания** «01» февраля 2016 г.

Руководитель работы
кандидат ф.- м. наук, доцент _____ (подпись) ***Елсаков С.М.***

Задание принял к исполнению _____ (подпись) ***Комяков Д.С.***

АННОТАЦИЯ

Комяков Д.С. Разработка приложения для внедрение и верификации водяных знаков в исходные коды программного обеспечения. – Челябинск: ЮУрГУ, 2016. – 36 с. Илл. 12. Табл. 1. Библ. – 21 наимен. Прилож. – 1.

Разработано приложение для внедрения и верификации водяных знаков в исходные коды программного обеспечения. Рассмотрены методы внедрения водяных знаков и существующие решения. На их основе составлены требования к разрабатываемой системе.

Разработаны алгоритмы внедрения и верификации водяных знаков в исходный код программного обеспечения и на их основе реализовано приложение. Подано заявление на регистрацию приложения и опубликована научная статья. Также возможна последующая модификация разработанного программного обеспечения.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Нелицензионное использование программного обеспечения, также известное как пиратство, является копированием лицензионного приложения с последующим его незаконным распространением, вне зависимости от того является ли целью кражи извлечения прибыли или нет. Законные способы защиты ПО, такие как закон об авторском праве, патенты и лицензионные соглашения часто являются недостаточным аргументом для предотвращения кражи ПО, в особенности в развивающихся странах, где стоимость лицензионного ПО относительно высока [8; 1].

По оценкам IDC, в 2014 г. российские компании потратили на устранение последствий технических рисков от использования пиратского ПО – обнаружение проблем и восстановление данных – около $5 млрд [3].

Поэтому вопрос защиты авторских прав и лицензионного использования ПО остается весьма актуальным для Российской Федерации. Один из способов защиты ПО от нелицензионного использования – введение водяных знаков в исходный код программного обеспечения.

Введение водяных знаков в исходный код программного обеспечения представляет из себя внедрение уникальных идентификаторов в различные компоненты ПО в целях сопротивления краже ПО. Введение водяных знаков не предотвращает кражу, но вместо этого предоставляет средства для идентификации правообладателя ПО и /или происхождения украденного ПО [17]. Скрытый водяной знак может быть затем извлечен с помощью некого распознавателя чтобы затем доказать права на обладание украденным ПО. Также возможно разместить уникальный продажный идентификатор в каждой копии ПО, который поможет разработчику определить индивидуальные случаи пиратства. Необходимо чтобы был водяной знак скрыт, что воспрепятствует его обнаружению и удалению. В большинстве случаев также необходимо чтобы водяной знак был устойчив к семантическим преобразованиям (таким как оптимизация и обфускация, т.е. запутывание исходного кода приложения).

Данная проблема особенно актуальна для программ, написанных на языках программирования, использующих какую-либо промежуточную среду исполнения, таких как Java и C#.

Программы, написанные на таких языках, поддаются дизассемблированию вплоть до именования методов и переменных, поэтому для их защиты необходимо принимать дополнительные меры.

В данной работе будет реализована система внедрения водяных знаков в исходный код программного обеспечения для программ, написанных на языке C#, целью которой

является уменьшение времени разработки и внедрения водяных знаков в исходный код программного обеспечения.

1 МЕТОДЫ ВНЕДРЕНИЯ ВОДЯНЫХ ЗНАКОВ В ИСХОДНЫЙ КОД ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1 Методы внедрения водяных знаков

Водяные знаки широко используются в индустрии развлечений для идентификации файлов мультимедиа таких как аудио, фото и видео-файлы, эта концепция была расширена под нужды индустрии ПО. Водяные знаки не преследуют цель увеличить трудоемкость кражи, как например обфускация, а предоставляют возможность доказать сам факт нелицензионного использования.

Введение водяных знаков в исходный код программного обеспечения включает в себя несколько подзадач. Водяные знаки должны удовлетворять следующим требованиям:

1) Размер исходного кода ПО не должен значительно увеличиваться после внедрения водяных знаков.

2) Производительность ПО не должна значительно падать после внедрения водяных знаков.

3) Водяные знаки должны быть устойчивы к семантическим преобразованиям.

4) Водяные знаки должны быть достаточно хорошо скрыты, чтобы избежать их удаления.

5) Водяные знаки должны быть легко извлекаемы для их владельца.

Наиболее сложной задачей является скрытие водяных знаков от злоумышленников одновременно с предоставлением правообладателю возможности извлечь водяной знак. Если водяной знак слишком легко извлечь, злоумышленнику не составит труда просто удалить его, в то время как необходимо сохранить возможность извлечения водяного знака правообладателем. Некоторые утилиты для введения водяных знаков (как например, Sandmark [5]) используют метки в местах хранения водяных знаков, что облегчает их обнаружение.

Водяные знаки должны быть устойчивы к сохраняющим семантику преобразованиям. Результатом сохраняющих семантику преобразований являются программы, отличающиеся синтаксисом, но сохраняющим свое поведение. Злоумышленник может попытаться, используя такие трансформации, предоставить эквивалентную семантически программу с удаленными водяными знаками. В идеале, нужно чтобы водяные знаки можно было распознать, располагая только частью программы, а также чтобы водяные знаки были устойчивы к декомпиляции и ребилдингу.

Исходный код водяного знака должен быть локально неотличимым от остальной части программы, чтобы быть скрытым от злоумышленников [20]. К примеру, представим водяной знак, который представляет собой фиктивный метод с сотней переменных – такие методы будут выделяться при простом анализе кода, например, с помощью вычисления метрик кода [12]. Программно сгенерировать код, который будет тяжело отличим от написанного человеком, крайне сложно, но статистический анализ оригинальной программы может помочь в генерации подходящих водяных знаков [13].

Водяные знаки должны быть эффективными по нескольким критериям: затраты на вложение, затраты на выполнение и временные затраты на распознавание.

Затраты на вложение могут быть разделены на две области: время разработки и "стоимость" самого их вложения.

Первые означают время, которое разработчик тратит на внедрение водяного знака, тогда как вторые означают время работы утилиты внедряющей водяной знак. Затраты на вложение не самая значимая проблема.

Время разработки важно в применении к водяным знакам, так как разработчик не должен тратить большой промежуток времени на создание самого водяного знака. Сложность водяного знака пропорциональна его устойчивости, это так, чем больше разработчик тратит на разработку водяного знака – тем труднее для злоумышленника ее обойти. Например, разработчик может потратить дни, вводя «тонкое» семантическое свойство, которое делает ПО уникальным и очень трудно отыскивается.

Оптимальным вариантом является полуавтоматический водяной знак, который включает подготовку программы разработчиком перед тем как внедряющая утилита размещает водяной знак. Подготовка может содержать расстановку меток в местах размещения кода водяного знака или создание фиктивных методов которые должен использовать водяной знак. Они описывают алгоритм, который требует создание фиктивного метода в программе для хранения водяного знака[14].

Затраты на запуск зависят от эффекта который оказали трансформации на размер и время выполнения. К примеру, было вычислено, что размер программы с водяными знаками, введенными по алгоритму Дэвидсона-Мирвольда, увеличивался на 24%, а производительность уменьшилась на 24%[9, 11].

Фиктивные методы, которые не выполняются, будут оказывать минимальный эффект на затраты на запуск в то время как динамические водяные знаки могут иметь высокие затраты на запуск, т.к. водяной знак формируется в ходе выполнения программы. Качество водяных знаков или «степень в которой водяной знак портит оригинальное содер-

жимое», также должно быть принято во внимание, например встраивание водяного знака может привести к непредвиденным ошибкам [18].

Время распознавания водяного знака в идеале должно быть довольно коротким, но в некоторых случаях имеет смысл искусственно его увеличить для предотвращения атак типа oracle [18]. Такие атаки основаны на повторяющемся исполнении распознавания, таким образом, быстрое время распознавания водяного знака помогает злоумышленнику.

Водяные знаки ПО могут быть разделены на две широкие категории: статические и динамические [18]. Первая включает водяные знаки в данных и/или коде программы, тогда как вторая включает водяные знаки в структуре данных, построенных в ходе выполнения программы.

Также по функциональному назначению водяные знаки можно разделить на 4 типа: водяные знаки, идентифицирующие автора ПО; водяные знаки, уникально определяющие каждый экземпляр ПО; валидационные водяные знаки, подтверждающие подлинность и неизменность оригинального ПО; водяные знаки, противодействующие попытке нелицензионной активации ПО [7].

В данной работе будут рассмотрены статические водяные, по своей функциональности относящиеся к первым двум группам из представленных выше.

Среди атак, цель которых нейтрализовать водяные знаки в исходном коде приложения, можно выделить 3 категории: аддитивные атаки, субстрактивные атаки и искажающие атаки.

Аддитивные атаки обеспечивают внедрение нового водяного знака в исходный код приложения, который перекрывает собой оригинальные водяные знаки. Этот метод обычно срабатывает, если внедряемый водяной знак обладает одним типом с существующим водяным знаком, в противном случае его работа не гарантируется [16].

Субстрактивные атаки стараются удалить часть кода, в которой расположен водяной знак, сохраняя при этом работоспособность программы. Это может быть достигнуто за счет статистического анализа или фрагментирования исходного кода программы.

Искажающие атаки производят трансформации исходного кода, сохраняющие саму семантику программы, такие как обфускация и оптимизация. Эти трансформации, как правило, нацелены на водяные знаки, завязанные на синтаксисе исходного кода. Например: переименование переменных, циклические трансформации, встраивание функций и т.д..

Как статические, так и динамические водяные знаки могут быть уязвимы к трансформирующим атакам. Майлзом была проведена оценка динамической и статической версий алгоритма Эрбойта с помощью внедрения водяных знаков в коллекцию тестовых

файлов и их последующей обфускации [16]. В ходе оценки было выяснено, что динамическая версия лишь незначительно превосходит статическую, и обе версии алгоритма могли быть нейтрализованы искажающими атаками[2].

По этой причине произведена оценка только статических методов внедрения водяных знаков в исходный код приложений.

Обзор методов производился на примере программ на языке Java, ввиду широкой их распространенности и возможности дизассемблирования байт-кода Java.

Для обзора были выбраны следующие методы:

1) Добавление выражения – добавление фиктивного выражение, содержащего водяной знак для файла класса.

2) Добавление инициализации – добавление фиктивных локальных переменных в какой-либо метод файла-класса.

3) Добавление метода и полей – водяной знак разделяется на две части: первая часть хранится в имени фиктивного поля, а вторая в имени фиктивного метода. Фиктивный метод обращается к фиктивному полю, в то время как случайным образом выбранные методы обращаются к фиктивному методу чтобы сделать его похожим на часть программы.

4) Добавление оператора switch – добавление водяного знака в значения меток case оператора switch, который добавляется в начало случайным образом выбранного метода.

5) Алгоритм Дэвидсона-Мирвольда – водяной знак вводится за счет изменения порядка вызова базовых блоков в подходящем для этого методе[9].

6) Графовый водяной знак [20] – водяной знак вводится в граф потока управления оригинальной программы [20].

7) Алгоритм Маундэна – водяной знак вводится с помощью замены кодов операций в фиктивном методе, сгенерированном утилитой Sandmark [14].

8) Алгоритм Ку-Потконьяка – водяной знак добавляется в выделенные локальные переменные с помощью добавления ограничений в граф несовместимости [15; 19].

9) Регистрация типов – водяной знак вводятся добавлением локальных переменных некоторых стандартных типов библиотек Java.

10) Статический алгоритм Эрбойта – вставляет метод с помощью кодирования водяного знака в труднопонимаемое выражение и затем вставляет это выражение в выбранную ветку [4; 16].

11) Алгоритм Штерна – водяной знак внедряется в виде объекта, который создается с использованием частотного вектора представления кода [6].

12) Строковая константа – простой водяной знак, хранящийся в строковой константе в каком-либо классе.

В ходе исследования, проведенного Хэмилтоном в ходе которого jar-файлы с встроенными в них водяными знаками подвергались различным искажающим трансформациям [10]. Результаты представлены в таблице 1.1.

Таблица 1.1 – Число распознанных водяных знаков.

Атака	Добавление выражения	Добавление инициализации	Добавление метода и полей	Добавление оператора switch	Алгоритм Дэвидсона-Мирвольда	Графовый водяной знак	Алгоритм Маундэна	Алгоритм Ку-Потконьяка	Регистрация типов	Статический алгоритм Эрбойта	Алгоритм Штерна	Строковая константа
1	2	3	4	5	6	7	8	9	10	11	12	13
Исходный набор	60	56	35	59	15	47	58	0	51	19	46	60
Array Folder	60	56	35	59	15	47	56	0	51	19	43	60
Array Splitter	60	56	35	59	12	47	58	0	51	19	46	60
Block Marker	60	56	35	59	14	47	56	0	51	19	45	60
Constant Pool Reorderer	57	54	33	55	13	45	55	0	49	18	44	57
Dynamic Inliner	60	56	30	59	15	45	58	0	49	12	44	60
FalseRefactor	60	56	35	59	15	47	58	0	51	19	45	60
Integer Array Splitter	60	56	35	59	15	47	58	0	51	19	46	60
Interleave Methods	57	55	7	59	12	29	28	0	50	3	39	60
Overload Names	60	56	6	59	15	47	58	0	9	19	45	60
ParamAlias	60	56	34	59	12	47	58	0	51	19	46	60
Rename Registers	0	56	35	59	15	47	58	0	0	19	45	60
Split Classes	28	56	29	59	7	46	58	0	15	19	45	60
String Encoder	60	56	35	59	8	47	58	0	51	19	46	60
Class Splitter	60	44	23	58	15	47	58	0	32	19	45	60
Field Assignment	60	56	32	59	15	47	58	0	51	19	45	60
Method Merger	60	56	35	59	15	47	57	0	51	1	45	60
Objectify	60	56	35	59	13	47	58	0	6	19	45	60
Publicize Fields	60	56	35	59	15	47	58	0	51	19	45	60

1	2	3	4	5	6	7	8	9	10	11	12	13
Simple Opaque Predicates	60	55	35	59	15	1	7	0	51	0	1	60
Static Method Bodies	24	55	35	59	15	47	58	0	5	19	45	60
Bludgeon Signatures	60	56	35	59	11	47	58	0	51	19	45	60
Boolean Splitter	60	55	35	59	12	47	48	0	51	19	46	60
Branch Inverter	60	56	35	59	13	47	55	0	51	19	45	60
Duplicate Registers	60	5	35	59	8	47	56	0	51	19	45	60
Insert Opaque Predicates	60	56	35	59	7	33	31	0	51	3	10	60
Irreducibility	59	56	35	59	15	0	44	0	51	19	26	60
Merge Local Integers	56	0	35	59	11	47	25	0	51	19	16	60
Opaque Branch Insertion	60	0	35	59	6	1	32	0	51	0	42	60
Promote Primitive Registers	0	0	35	59	4	0	44	0	51	0	0	60
Random Dead Code	47	56	35	59	14	47	56	0	51	19	22	60
Reorder Instructions	60	10	35	59	13	47	58	0	51	19	46	60
Reorder Parameters	60	51	35	59	13	47	58	0	51	19	45	60
Transparent Branch Insertion	59	48	35	59	9	2	58	0	51	10	45	60
Variable Reassigner	1	56	35	59	2	47	58	0	1	19	45	60
Inliner	10	56	27	59	13	45	56	0	8	2	42	60
Proguard Optimize	2	1	35	1	8	0	7	0	1	2	2	60

Было выявлено следующее:

1) Водяные знаки не всегда могли быть внедрены в исходный код. Худшим в этом плане оказался алгоритм Ку-Потконьяка. Это можно объяснить тем, что файлы тестовой выборки были слишком маленькими (порядка 30КБ). Водяные знаки были введены только в 80% файлов.

2) До применения трансформирующих атак только 87,6% водяных знаков были успешно распознаны.

3) Наиболее эффективно искажающим атакам противостоял водяной знак в виде строковой константы, но в ходе анализа кода его можно просто удалить.

4) При применении комплексов трансформационных атак ни один водяной знак за исключением строковой константы не показал свою эффективность.

1.2 Коды Рида – Соломона

Из обзора методов внедрения статических водяных знаков можно заметить, что при применении комплекса трансформационных атак лучше всего себя показывают простые водяные знаки, представляющие из себя строковые константы, но знаки этого типа можно легко вычленить в ходе простого статистического анализа.

Чтобы это предотвратить, можно закодировать водяной знак таким образом, чтобы можно было восстановить его, не обладая при этом им целиком, а также разделить его на несколько частей и внедрить его таким образом, что при изменении или удалении одной из частей исходный код программного обеспечения терял свою работоспособность.

Коды Рида — Соломона — недвоичные циклические коды, позволяющие исправлять ошибки в блоках данных. Элементами кодового вектора являются не биты, а группы битов (блоки). Очень распространены коды Рида — Соломона, работающие с байтами (октетами).

Код Рида — Соломона является одним из наиболее мощных кодов, исправляющих многократные пакеты ошибок. Применяется в каналах, где пакеты ошибок могут образовываться столь часто, что их уже нельзя исправлять с помощью кодов, исправляющих одиночные ошибки.

Коды Рида-Соломона (РС) определены над полем $GF(p^m)$. В дальнейшем рассматриваются коды над конечными полями характеристики $p=2$, как наиболее распространенные. Поэтому символы кодовых слов являются элементами поля $GF(2^m)$. Коды РС, таким образом, это не двоичные, а 2^m- ичные коды БЧХ длины $N=2^m-1$. Они являются циклическими кодами с порождающим многочленом

$$g(x)=(x-\alpha^{\nu})(x-\alpha^{\nu+1})..(x-\alpha^{\nu+\delta-2}). \qquad (2.1)$$

где α – примитивный элемент поля $GF(2^m)$; ν – целое; δ – конструктивное расстояние кода.

Выбор длины кода $N=2^m-1$ гарантирует, что многочлен (2.1) является делителем x^N-1, и следовательно, всегда будет выполнено требование к порождающему многочлену циклического кода. Действительно, все ненулевые элементы поля $\beta=\alpha^i$, $i=0,1,2,\ldots,N-1$ являются корнями многочленах x^N-1, в чем легко убедиться подстановкой $x=\beta$: $\beta^N-1=(\alpha^N)^i-1=1-1=0$, т.к. $\alpha^N=1$. Поэтому многочлен x^N-1 пред-

ставим в виде произведения линейных множителей $(x-\alpha^i)$ и делится на любой многочлен (2.1).

Слово кода РС имеет вид: $A_1A_2...A_KB_1B_2...B_{N-K}$, где A_i и B_j – соответственно информационные и избыточные символы, которые могут быть представлены m-разрядными двоичными векторами. Число информационных символов $K=(N-cm.g(x))$, где $cm.g(x)$ – степень порождающего полинома $g(x)$. Коды РС являются систематическими кодами с максимальным кодовым расстоянием $D_0=N-K+1$, равным конструктивному δ.

Обозначение символов слов, длины кода, размерности, кодового расстояния прописными буквами A_i, B_j, N, K и D_0 введено для того, чтобы отличить их от аналогичных параметров двоичных кодов.

Коды РС являются линейными кодами, поэтому, кроме задания с помощью порождающего многочлена (2.1), они могут быть заданы также проверочным многочленом $h(x)=(x^N-1)/g(x)$, порождающей G и проверочной H матрицами.

Коды РС существуют для всех $K=1,2,...,N$. При $K=1$ код содержит нулевое слово и 2^m-1 ненулевых слов, полученных умножением одного базисного вектора порождающей матрицы на все 2^m-1 ненулевых элементов конечного поля. Если в (2.1) положить $v=1$, то для $K=1$ базисный вектор равен единичному. Тогда все ненулевые кодовые слова составлены из повторяющихся N раз ненулевых элементов поля. Очевидно, кодовое расстояние равно N. Это пример кода с повторением над произвольным конечным полем.

При $K=N$ код РС становится безызбыточным, его словами являются все $(2^m)^N$ последовательностей, символами в которых служат элементы поля $GF(2^m)$. Минимальное расстояние этого кода равно 1.

При $K=N-1$ в кодовом слове имеется один проверочный символ, который равен взвешенной сумме информационных символов. Весами служат элементы поля $GF(2^m)$. Так как последовательности из K информационных символов имеют минимальное расстояние, равное 1, то добавление проверочного символа, выбранного специальным образом, может увеличить кодовое расстояние до 2. Напомним, что для двоичных кодов один проверочный символ может быть связан с информационными единственным способом: он

равен сумме всех информационных, что при $K = N - 1$ приводит к кодам с простой проверкой на четность. Таким образом, переход к недвоичным кодам (расширению двоичного поля, над которым задается код) предоставляет большие возможности для конструирования новых кодов.

Отметим и другие особенности кодов РС по сравнению с двоичными кодами. Так, расстояние между векторами над полем $GF(2^m)$ по-прежнему определяется как расстояние Хэмминга, т. е. равно числу пар несовпадающих компонентов, хотя символы кодовых слов как m-разрядные векторы могут отличаться друг от друга в $1,2,3,\ldots,m$ разрядах. Такое определение расстояния не учитывает, насколько сильно различаются символы слов, но зато позволяет упростить анализ кодов.

Коэффициентами порождающего $g(x)$ и проверочного $h(x)$ полиномов служат элементы поля $GF(2^m)$, а не 0 и 1, как у соответствующих многочленов двоичных кодов. Получение кодовых слов с помощью порождающей матрицы означает не простое суммирование строк матрицы, а, как уже указывалось, суммирование базисных векторов, умноженных на различные элементы поля $GF(2^m)$.

Проверочная матрица кода РС есть проверочная матрица кода БЧХ длины N, являющаяся частью матрицы ДПФ последовательностей с компонентами из поля $GF(2^m)$.

Особенности декодирования кодов РС по сравнению с двоичными кодами БЧХ связаны с тем, что вектор ошибки имеет компоненты ε_i из поля $GF(2^m)$. Поэтому при декодировании кодов РС недостаточно указать номера искаженных символов, надо еще определить, насколько искажены символы, т.е. найти значения ε_i вектора ошибки. Последнее требование усложняет процедуру декодирования кодов РС по сравнению с декодированием двоичных кодов БЧХ.

Пример 5.14. Пусть имеется код РС, исправляющий одиночные, двойные и тройные ошибки. Корнями порождающего многочлена являются элементы $\alpha, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6$, где α – примитивный элемент поля $GF(2^4)$, образованного с помощью неприводимого полинома $p(x) = x^4 + x + 1$. Спектральные коэффициенты ДПФ кодовых слов в точках α^i, $i = 1,2,\ldots,6$, равны 0. Параметры кода: $N = 15$, $K = 9$, $\delta = D_0 = N - K + 1 = 7$, $q_{ис} = 3$.

Так как синдром не зависит от передаваемого слова S, то для рассмотрения процедуры декодирования достаточно задать вектор ошибки E. Допустим, что многочлен,

описывающий вектор ошибки, $E(x) = \alpha^7 x^3 + \alpha^{11} x^{10}$, т.е. произошла двукратная ошибка, исказившая в слове $S = (s_0 s_1 s_2 \ldots s_{14})$ символы s_3 и s_{10}, причем принятые символы $y_3 = s_3 + \varepsilon_3 = s_3 + \alpha^7$, $y_{10} = s_{10} + \varepsilon_{10} = s_{10} + \alpha^{11}$ остальные – $y_i = s_i$.

Декодер по реализации Y должен найти оценку вектора ошибки $\hat{E}$, а затем оценку переданного слова $\hat{S} = Y + \hat{E}$. Если оценка $\hat{E} = E$, то $\hat{S} = S + E + \hat{E} = S$, и ошибка исправляется. Если $\hat{E} \neq E$, ошибка не исправляется.

Процедура нахождения $\hat{E}$ зависит от описания вектора ошибки E. Для указания номеров искаженных символов воспользуемся многочленом локаторов ошибок $\sigma(z)$. Для двоичных кодов многочлен $\sigma(z)$ полностью определяет вектор ошибки E. В случае кодов над полем $GF(2^m)$, каковыми являются коды РС, необходимо еще указать значения ошибок ε_i, что делается с помощью так называемого многочлена значений ошибок $\Omega(z)$. Для нахождения ε_i требуется знание обоих многочленов $\sigma(z)$ и $\Omega(z)$, так как

$$\varepsilon_i = \Omega(\alpha^{-i}) / \sigma'(\alpha^{-i}). \tag{2.2}$$

Где i – номер искаженного символа; $\sigma'(\alpha^{-i})$ – формальная производная многочлена $\sigma(z)$ в точке α^i, которая для кодов, заданных над полями характеристики 2, содержит только четные степени. Коэффициенты при нечетных степенях производной являются четными числами и обращаются в ноль.

Итак, произвольный вектор ошибки над полем $GF(2^m)$ может быть задан с помощью двух многочленов $\sigma(z)$ и $\Omega(z)$. Отметим, что эти многочлены определены с точностью до постоянного множителя β – элемента поля $GF(2^m)$. Многочлен $\beta\sigma(z)$ имеет множество корней, совпадающее с множеством корней многочлена $\sigma(z)$, и следовательно, оба определяют одни и те же номера искаженных символов.

Главная задача декодера состоит в оценке многочленов $\sigma(z)$ и $\Omega(z)$. Если эти оценки правильно описывают реальную ошибку, т.е. множества корней о $\hat{\sigma}(z)$ и $\sigma(z)$ совпадают, то такая ошибка исправляется. В противном случае исправления не произойдет.

Многочлены локаторов $\sigma(z)$, значений ошибок $\Omega(z)$ и синдрома $C(z)$ связаны ключевым уравнением :

$$\sigma(z)C(z) \equiv \Omega(z) \bmod \left(z^{2q_{\text{ис}}}\right). \tag{2.3}$$

Трудность решения этого уравнения состоит том, что в соответствии с принципом максимального правдоподобия должны быть найдены многочлены $\hat{\sigma}(z)$ и $\hat{\Omega}(z)$ минимальных степеней, причем степень $\hat{\sigma}(z)$ должна быть больше степени $\hat{\Omega}(z)$, но менее степени $q_{\text{ис}}$.

Разработаны различные методы решения ключевого уравнения, некоторые из которых уже упоминались в третьем разделе.

Заданная в примере конфигурация двукратной ошибки в соответствии с (5.29) описывается многочленом локаторов, имеющим корни, обратные элементам α^3 и α^{10}:

$$\sigma(z) = \beta(1-\alpha^3 z)(1-\alpha^{10} z) = \beta\left(1+(\alpha^3+\alpha^{10})z+\alpha^3\alpha^{10}z^2\right) = \beta\left(1+\alpha^{12}z+\alpha^{13}z^2\right).$$

Напомним, что вычисления производятся в поле $GF(2^4)$. Сложение удобно выполнять, используя полиномиальное или векторное представление элементов поля, а умножение – степенное представление. Поэтому $\alpha^3+\alpha^{10} = 1000+0111 = 1111 = \alpha^{12}$.

Этапы декодирования данного кода при сделанных предположениях о характере искажении описываются следующими операциями.

Этап 1. По принятой реализации символов Y вычисляются 6 компонентов синдрома $C_1, C_2, \ldots, C_6$, являющихся коэффициентами ДПФ последовательности Y. Для заданного искажения $E(x) = \alpha^7 x^3 + \alpha^{11} x^{10}$ компоненты синдрома определяются по формуле

$$C_j = Y(\alpha^j) = E(\alpha^j) = \alpha^7\alpha^{3j} + \alpha^{11}\alpha^{10j}, \quad j = 1,2,\ldots,6.$$

$$C_j = E(\alpha) = \alpha^7\alpha^3 + \alpha^{11}\alpha^{10} = \alpha^{10} + \alpha^{21} = \alpha^{10} + \alpha^6 = 0111+1100 = 1011 = \alpha^7.$$

Результатом выполнения первого этапа является многочлен синдрома

$$C(z) = \alpha^{14}z^5 + \alpha^{14}z^4 + \alpha^{12}z^3 + \alpha^6 z^2 + \alpha^{12}z + \alpha^7.$$

Этап 2. Декодер решает ключевое уравнение (2.3) при $q_{\text{ис}} = 3$, оценивает многочлены $\sigma(z)$ и $\Omega(z)$ с помощью алгоритма Евклида нахождения НОД. Не вдаваясь в детали этого алгоритма, с которыми можно познакомиться в [26], приведем результат решения (2.3):

$$\hat{\sigma}(z) = \alpha^9 z^2 + \alpha^8 z + \alpha^{11}, \quad \hat{\Omega}(z) = \alpha^2 z + \alpha^3.$$

Анализ показывает, что $\sigma(z) = \alpha^{-4}\left(\alpha^{13}z^2 + \alpha^{12}z + 1\right) = \alpha^{-4}\sigma(z)$, т.е. в данном случае многочлен локаторов определен правильно.

Этап 3. Исправление ошибок производится при сложении символов принятой комбинации Y с компонентами ε_i вектора ошибки на позициях, номера которых обратны корням многочлена $\sigma(z)$. Корни $\sigma(z)$ определяются путем непосредственной подстановки в многочлен всех ненулевых элементов поля $\beta = \alpha^{-i}$, $i = 0,1,2,\ldots,2^m - 1$. Если $\sigma\left(\alpha^{-i}\right) = 0$, то предполагается, что искажен символ ε_i.

Проделав такую подстановку, найдем корни многочлена $\sigma(z)$: $z = \alpha^{-3}$ и $z = \alpha^{-10}$. Действительно, для элемента α^{-3}:

$$\sigma\left(\alpha^{-3}\right) = \alpha^9\alpha^{-6} + \alpha^8\alpha^{-3} + \alpha^{11} = 1000 + +0110 + 1110 = 0000.$$

Для оценки вектора ошибки $\hat{E}(x) = \varepsilon_3 x^3 + \varepsilon_{10}x^{10}$, т.е. для определения ε_3 и ε_{10}, найдем производную многочлена локаторов $\sigma'(z) = 2\alpha^9 z + \alpha^8 = \alpha^8$ и с помощью формулы (2.2) Получим:

$$\varepsilon_3 = \Omega\left(\alpha^{-3}\right)\Big/\alpha^8 = \left(\alpha^2\alpha^{-3} + \alpha^3\right)\Big/\alpha^8 = \alpha^{-9} + \alpha^{-5} = \alpha^6 + \alpha^{10} = \alpha^7,$$

$$\varepsilon_{10} = \Omega\left(\alpha^{-10}\right)\Big/\alpha^8 = \left(\alpha^2\alpha^{-10} + \alpha^3\right)\Big/\alpha^8 = \alpha^{11}.$$

Оценка многочлена ошибки $l/m + 1$ совпадает с $E(x)$, по предположению исказившим кодовое слово. Поэтому $\hat{S} = S$, и двукратная ошибка исправлена.

В заключение рассмотрим декодирование данного кода, если кратность ошибки больше трех, например, $q_{ис} = 4$. Такая ошибка описывается многочленом локаторов 4-й степени. Декодер, конечно, «не знает» о кратности произошедших искажений и выполняет алгоритм декодирования, предназначенный для исправления ошибок кратности не более 3.

На первом этапе определяется синдром, содержащий по-прежнему 6 компонентов. Синдром ненулевой, так как кратность ошибки меньше кодового расстояния $D_0 = 7$, и ошибка кратности 4 обнаруживается. В результате решения ключевого уравнения (если это окажется возможным) будет найден многочлен локаторов, степень которого не превышает 3. Ясно, что множества корней многочленов разных степеней не совпадают (мно-

гочлен локаторов не имеет кратных корней). Поэтому корни $\sigma(z)$, найденные на третьем этапе декодирования, будут неправильно указывать номера искаженных символов. Ошибка кратности 4 обнаруживается, но не исправляется.

Коды РС имеют важное теоретическое и практическое значение, так как при заданных N и K имеют максимальное кодовое расстояние, используются для обнаружения и исправления пакетов ошибок и построения высокоэффективных каскадных кодов.

Исходя из проведенного анализа методов внедрения водяных знаков для реализации системы были выбраны метод Эрбойта с генерацией фиктивного метода и внедрения строковой константы с применением кодов Рида-Соломона.

Метод Эрбойта относительно неплохо показал себя при применении различных трансформационных атак, тогда как строковые константы оказались устойчивы даже к комплексам трансформационных атак. Для того чтобы затруднить вычленение строковых констант на этапе статистического анализа из исходного кода программного обеспечения, было принято решение закодировать данные с применением кодов Рида-Соломона и разделить их на несколько частей.

1.3 Существующие системы внедрения водяных знаков в исходный код программного обеспечения, написанного на языке C#

Далее будет приведен краткий обзор некоторых уже существующих решений для защиты программного обеспечения, написанного на языке C#, и их анализ.

1.3.1 Codeveil

Codeveil является прекрасным протектором .NET сборок и dll-библиотек. Основа защиты построена на крипто-движке с мощной антиотладкой и антитрэйсингом.

CodeVeil добавляет машинный код для сборки для расшифровки сборку перед среда .NET необходим доступ к данным. Декодер включает в себя анти-отладки, антиотслеживание, отслеживание модификации, а также различные другие передовые технологии для предотвращения несанкционированного доступа и изменения сборок. Также завуалированные сборки могут быть использованы, как и любые другие сборки.

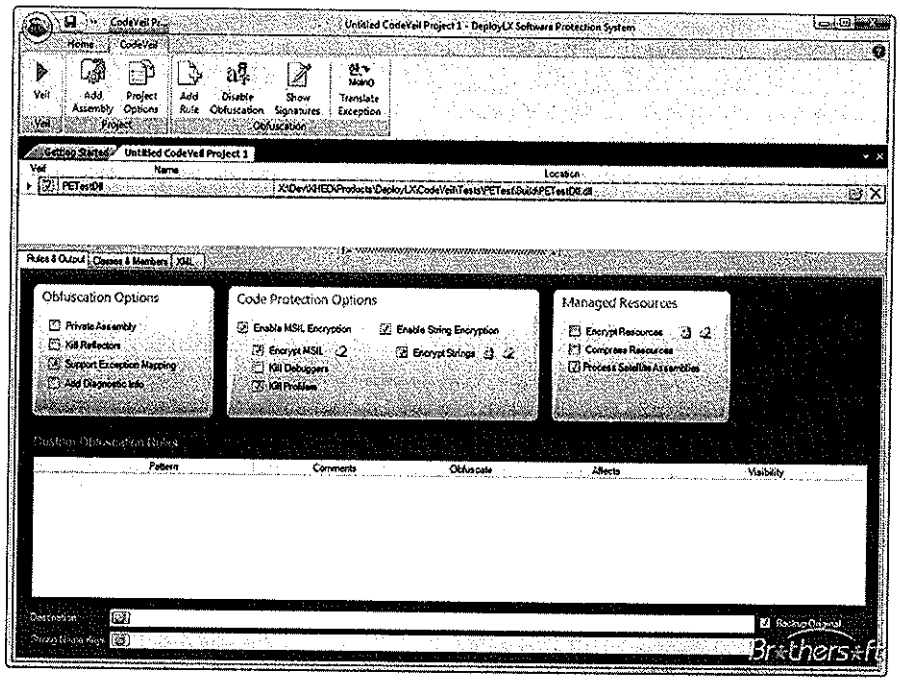

Рис. 1.1 Codeveil

1.3.2 Dotfuscator

Приложение Dotfuscator, разработанное компанией PreEmptive Solutions, является одним из лучших средств .NET в области запутывания кода, сжатия и создания водяных знаков, что способствует защите программ от декомпиляции, одновременно уменьшая их размеры и делая более эффективными. Кроме того, программа Dotfuscator and Analytics обеспечивает дополнительные предварительно реализованные функциональные возможности, обеспечивающие отслеживание использования и окончания срока действия, а также определение незаконного изменения в приложениях .NET.

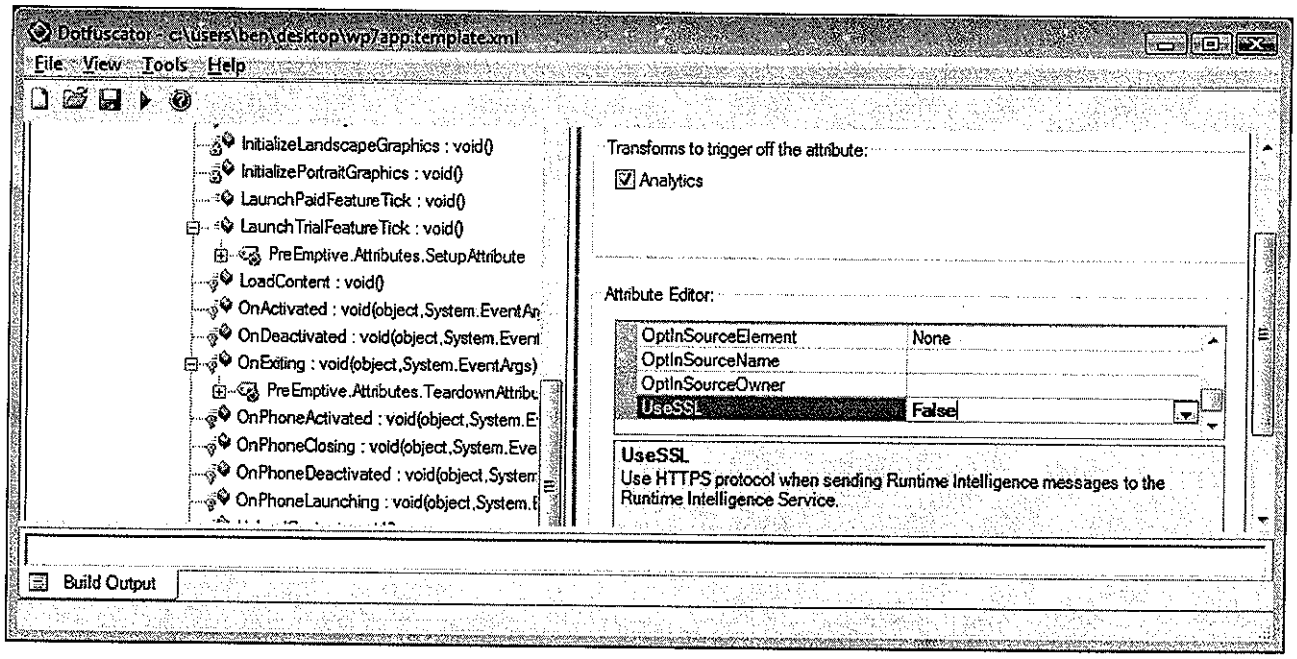

Рис. 1.2 Dotfuscator

1.3.3 .NET Reactor

.NET Reactor – продвинутый инструмент, предназначенный для организации защиты программного обеспечения, созданного по .NET-технологиям.

Благодаря .NET Reactor разработчики могут обеспечить первоклассную защиту своих программ, используя простые и эффективные инструменты. Утилита, не затрагивая оригинальные .NET-файлы, создает барьер, который крайне трудно преодолеть взломщикам.

В программе есть мастер, благодаря которому можно за несколько кликов организовать защиту как новых, так и уже готовых программ. .NET Reactor обладает интерфейсом, понятным многим пользователям.

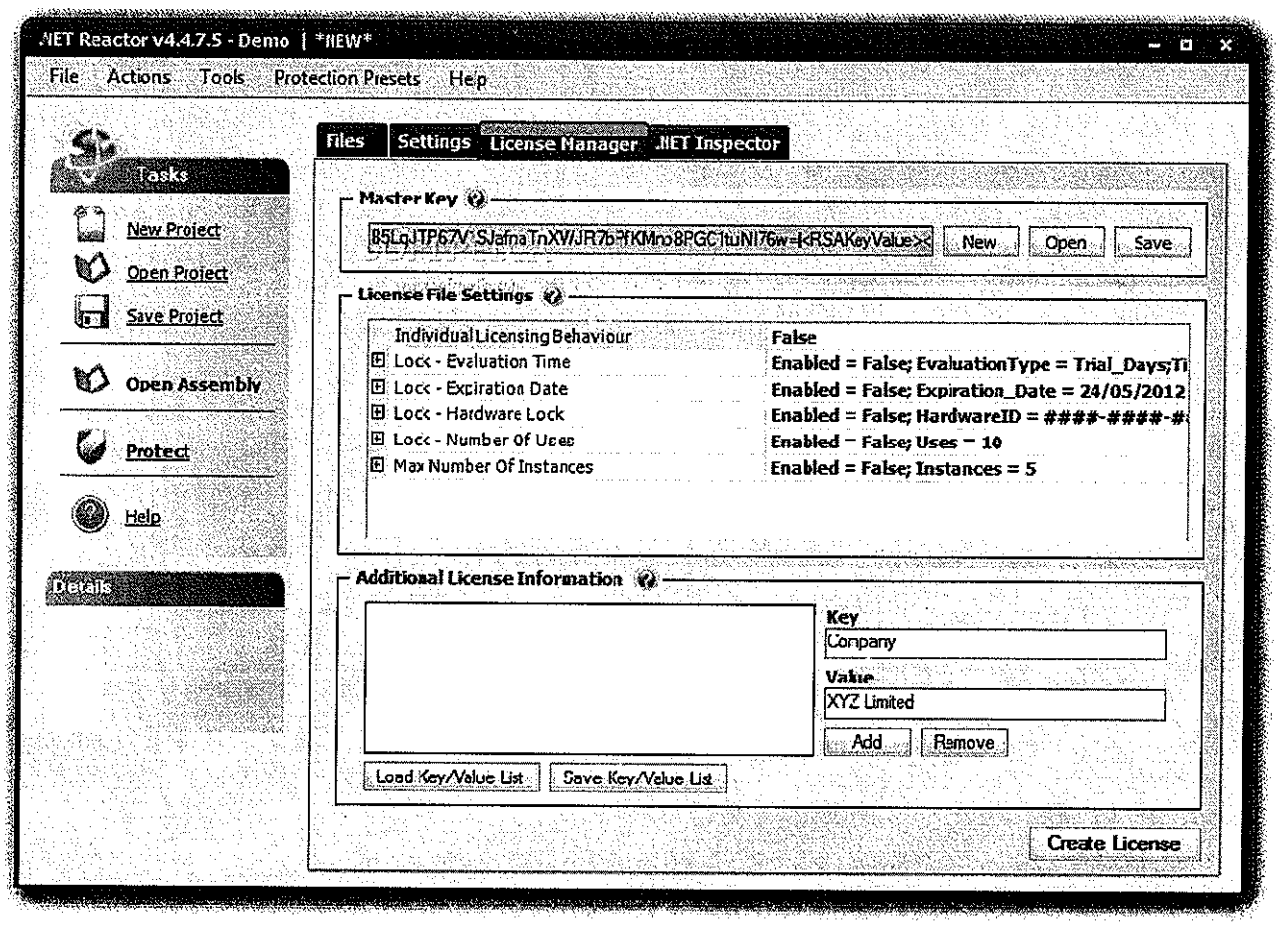

Рис. 1.3 .NET Reactor

1.3.4 Прочие коммерческие решения

Также существует ряд других коммерческих решений, таких как {SmartAssembly}, Xenocode PostBuild, Spices.NET, Obfuscator.NET, DesaWare и т.д., но многие из этих решений обладают существенными недостатками. Например, результаты работы этих решений легко поддаются взлому (в случае {SmartAssembly} и Xenocode PostBuild), не поддерживают .NET 4.0 и более поздние платформы (DesaWare), часто приводят защищаемые сбор-

ки в нерабочее состояние (Obfuscator.NET), шифруют всю сборку целиком (Spices.NET) и т.д..

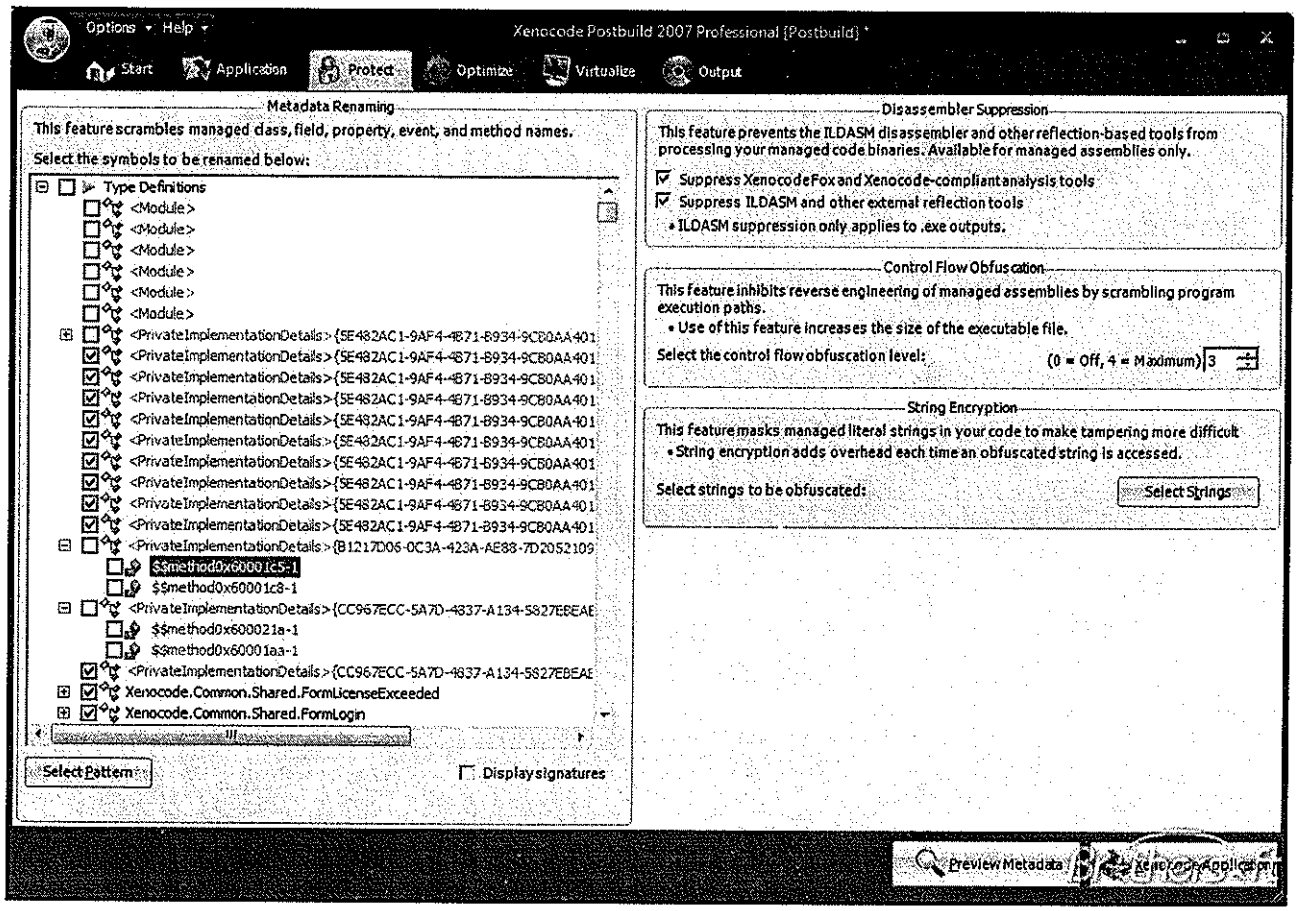

Рис. 1.4 Xenocode Postbuild

1.3.5 Open-source решения

Существует ряд open-source решений для защиты программного обеспечения на языке C#, таких как Assemblur, Eazfuscator.NET, NetOrbiter, Obfuscar, SharpObfuscator, Phoenix Protector и т.д..

Подробно рассматривать их не имеет особого смысла, т.к. они чаще всего выполняют простое переименование переменных, не поддерживают актуальных версий платформы .NET либо их поддержка прекращена.

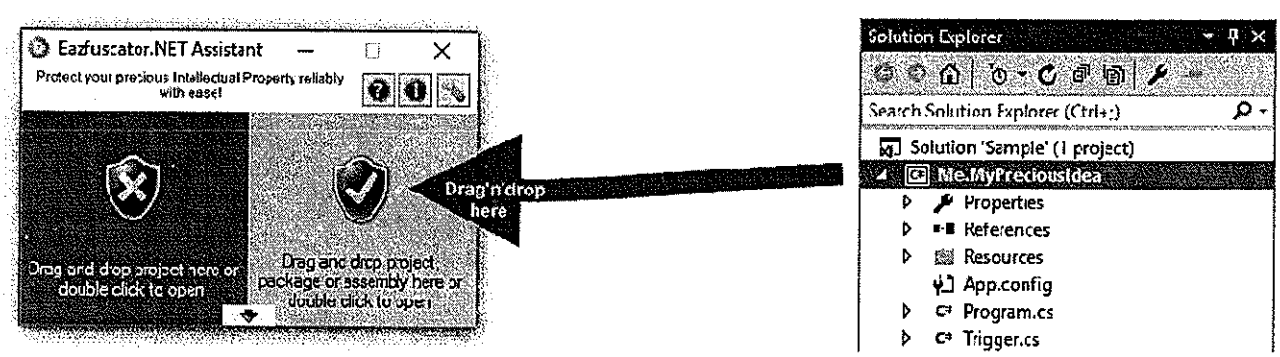

Рис. 1.4 Eazfuscator.NET

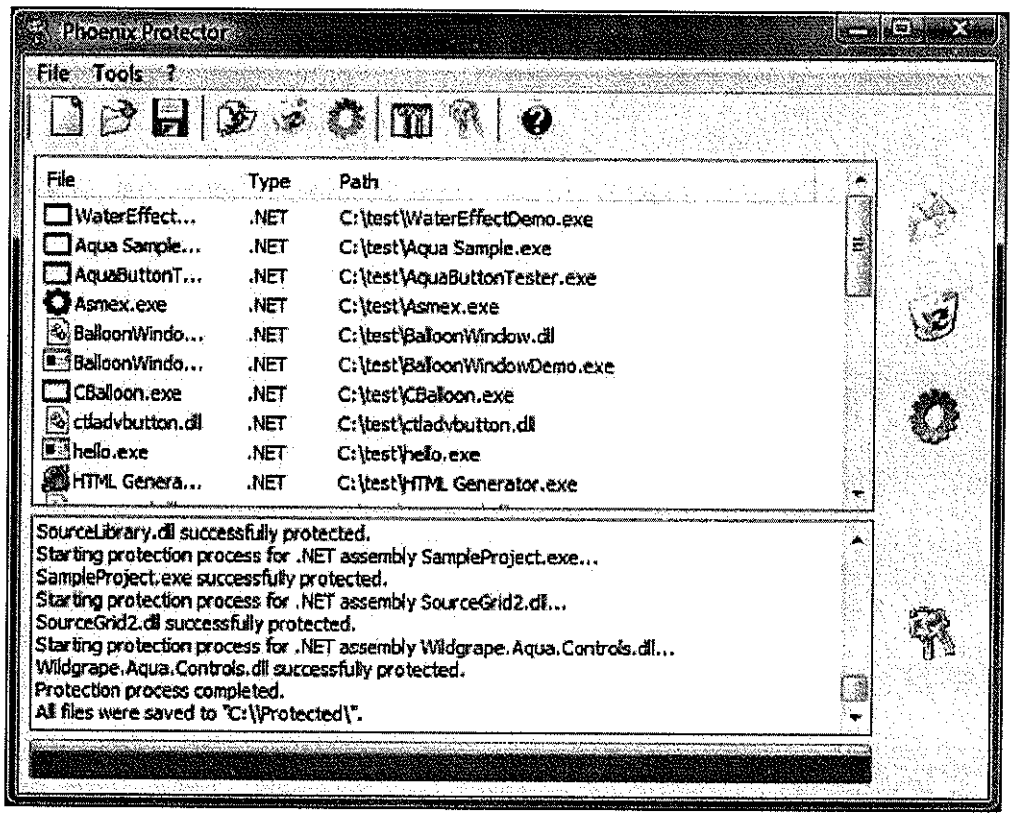

Рис. 1.5 Phoenix Protector

1.3.6 Результаты анализа существующих решений

Исходя из анализа существующих систем, были выработаны 2 основных требования: разрабатываемая система должна поддерживать работу с актуальными версиями платформы .NET и не должна приводить сборки в неработоспособное состояние.

1.3.7 Постановка задачи

Целью данной работы является разработка системы внедрения водяных знаков в исходный код программного обеспечения. Система должна удовлетворять следующим требованиям:

- должна работать в ограниченном объеме оперативной памяти компьютера;
- должна корректно работать с кодом, используемым с разными версиями платформы .NET;
- не должна вносить функциональных изменений в код исходного программного обеспечения;
- должна быть устойчива к трансформационным атакам на исходный код программного обеспечения;
- должна обладать удобным графическим интерфейсом.

Для решения данной задачи необходимо выполнить следующие шаги:

1) разработать алгоритм внедрения водяных знаков в исходный код программного обеспечения;

2) разработать алгоритм извлечения водяных знаков из исходного кода программного обеспечения;

3) разработать пользовательский интерфейс;

4) разработать программу, реализующую полученные алгоритмы;

5) отладить и протестировать программу;

6) оценить работу алгоритма.

2 ПРОГРАММА ДЛЯ ВНЕДРЕНИЯ И ВЕРИФИКАЦИИ ВОДЯНЫХ ЗНАКОВ В ИСХОДНЫЙ КОД ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ, НАПИСАННОГО НА ЯЗЫКЕ C#

Для решения поставленной задачи были разработаны алгоритмы, необходимые для внедрения и извлечения водяных знаков в исходном коде программного обеспечения.

2.1. Выбор компонентов для разработки

Для реализации разработанной системы были выбраны следующие компоненты:

- Среда разработки Microsoft Visual studio 2012;
- Платформа .NET 4.5;
- NRefactory;
- ZXing.NET.

NRefactory – свободно распространяемая библиотека, устанавливаемая как пакет NuGet и используемая для анализа и модификации кода, написанного на языке программирования C#.

NRefactory включает в себя парсер языка программирования C#, абстрактное дерево синтаксиса с поддержкой сравнения с образцом, семантический анализатор языка программирования C# (на данный момент стабильная версия библиотеки поддерживает версию C# 5.0, поддержка версии C# 6.0 находится на стадии бета-тестирования), исполнитель кода на языке C#, автоформатирование исходного кода на языке C#, а также большое число методов рефакторинга исходного кода ПО.

ZXing.NET - свободно распространяемая библиотека, устанавливаемая как пакет NuGet и используемая для преобразования какой-либо информации в различные типы QR-кодов, как частный случай кодов Рида-Соломона. В последствии из полученных изображений можно получить текстовые данные, используя кодировку Base64 для последующего внедрения в исходный код программного обеспечения.

2.2 Интерфейс программного обеспечения

На главной форме отражаются самые необходимые элементы управления, используемые для выполнения алгоритмов внедрения и извлечения водяных знаков.

В верхней части главного окна разместим элементы управления, необходимые для внедрения водяного знака:

- выбор папки, в которой расположен исходный код приложения;
- форма для ввода текста, который будет использован для формирования водяного знака;

– выбор папки, в которой будет размещен файл, содержащий метаданные водяного знака;
– кнопка запуска алгоритма внедрения водяного знака.

В нижней части главного окна разместим элементы управления, необходимые для извлечения водяного знака:

– выбор файла, содержащего в себе метаданные водяного знака;
– выбор папки, в которой расположен исходный код приложения;
– кнопка запуска алгоритма извлечения водяного знака и показа отчета о работе алгоритма.

Главная форма изображена на рисунке 2.1.

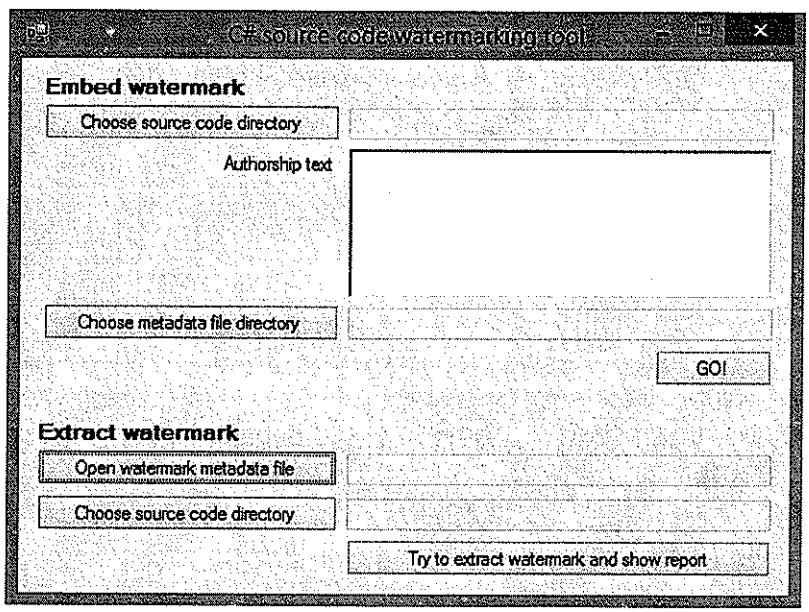

Рис. 2.1 Главная форма приложения

Для удобства показа и сохранения отчета необходимо предоставить отдельную форму, показанную на рисунке 2.2.

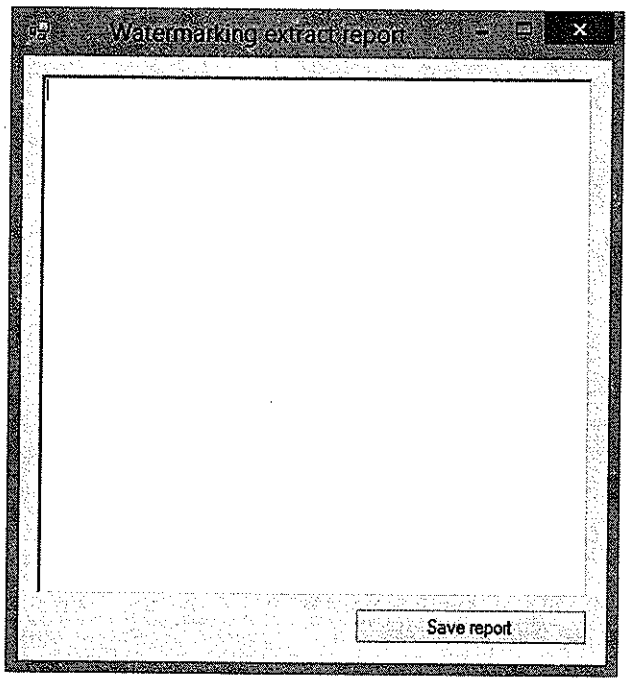

Рис. 2.2 Форма для вывода и сохранения отчета извлечения водяного знака

Для выбора папок и файлов используются стандартные компоненты Windows, показанные на рисунках 2.3 и 2.4.

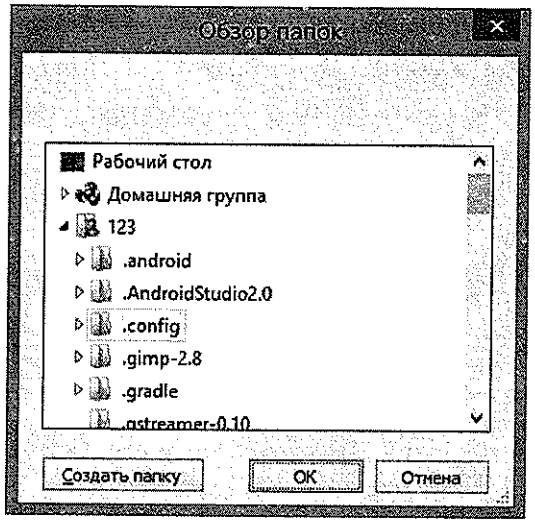

Рис. 2.3 Форма для выбора папок

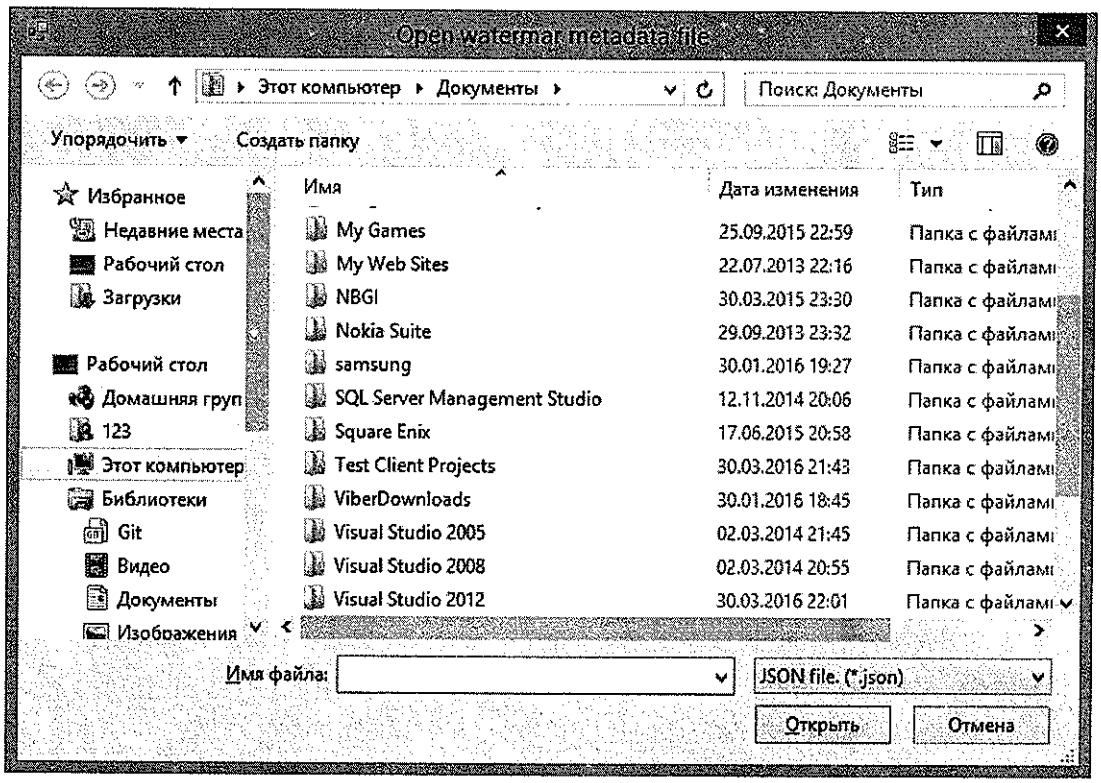

Рис. 2.4 Форма для выбора файлов

2.3 Реализация алгоритма внедрения водяных знаков

Алгоритм внедрения водяных знаков основан на алгоритме Эрбойта и внедрении строковых констант с последующей записью метаданных водяного знака в формате JSON.

Схема алгоритма внедрения водяных знаков представлена на рисунке 2.5.

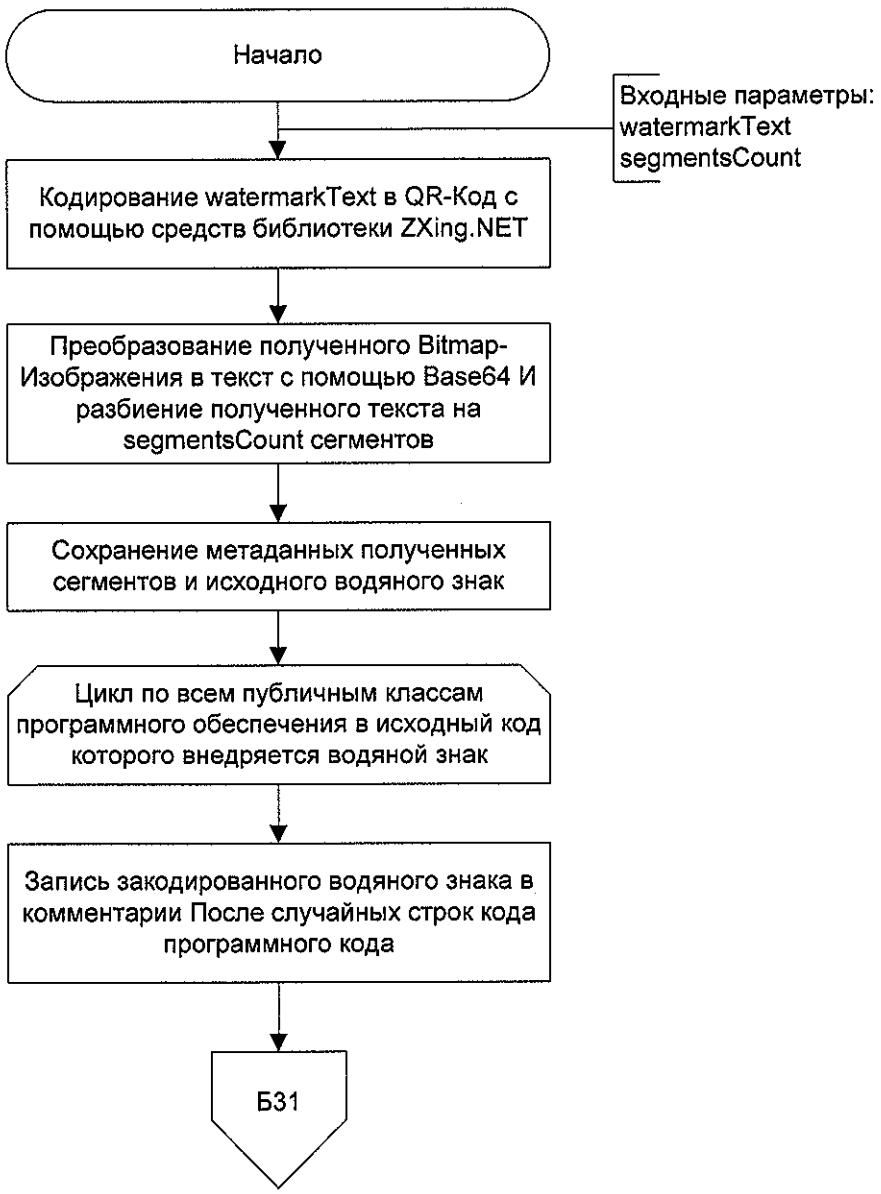

Рис. 2.5 Схема алгоритма внедрения водяного знака.

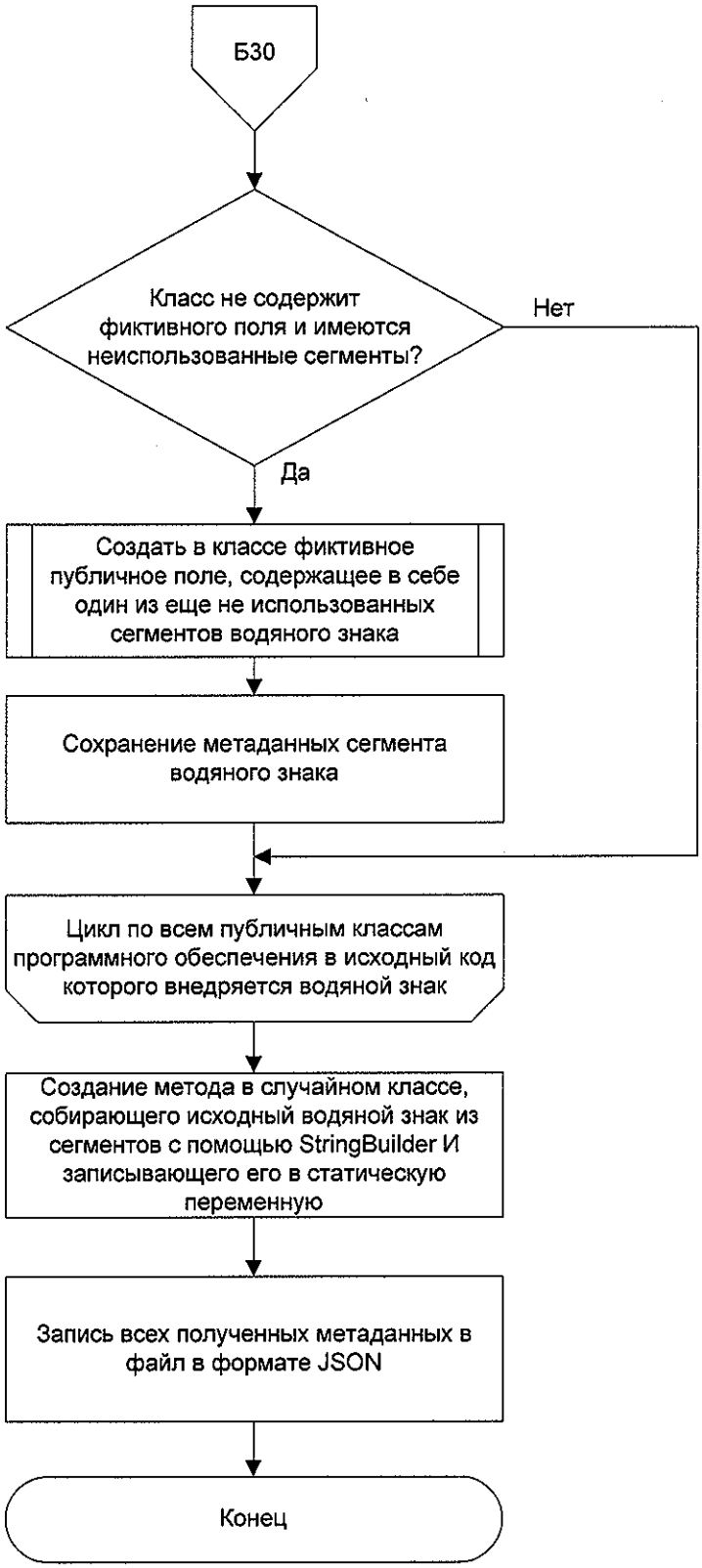

Рис. 2.5 Окончание алгоритма внедрения водяного знака.

2.4. Реализация алгоритма извлечения водяных знаков

Алгоритм извлечения водяных знаков основан на сборе значений строковых констант из исходного кода программного обеспечения и попытке декодировать сообщение с помощью кодов Рида-Соломона и кодировки Base64, а также сравнением с метаданными водяного знака, сохраненными в формате JSON.

Схема алгоритма представлена на рисунке 2.6.

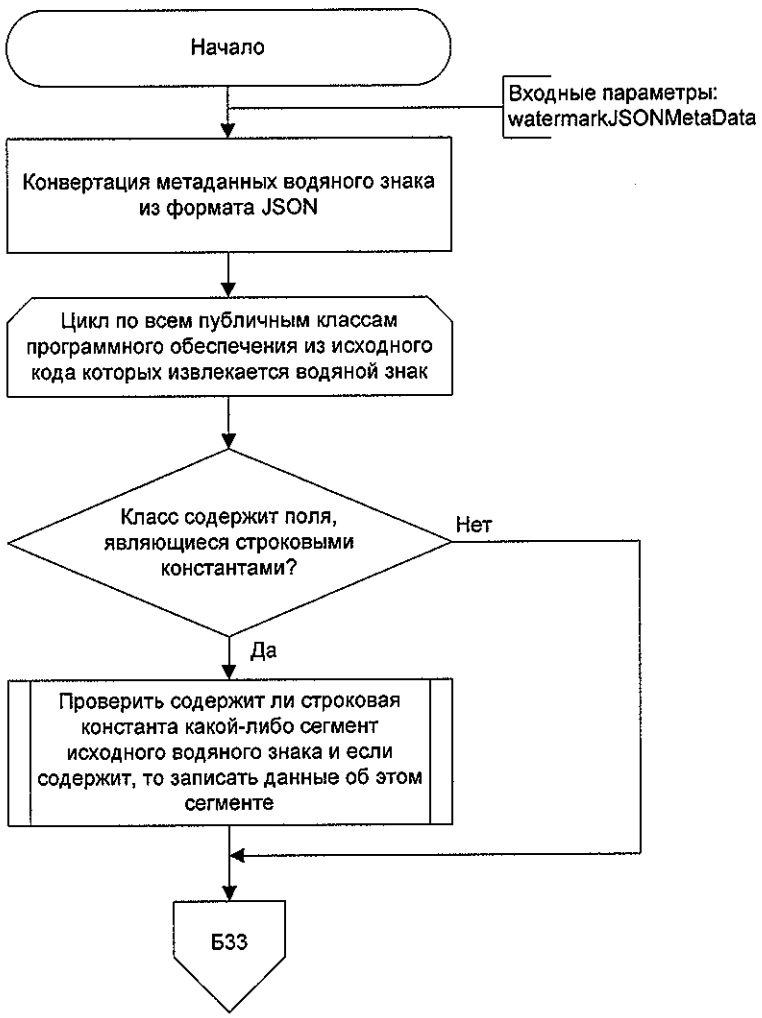

Рис. 2.6 Схема алгоритма извлечения водяного знака

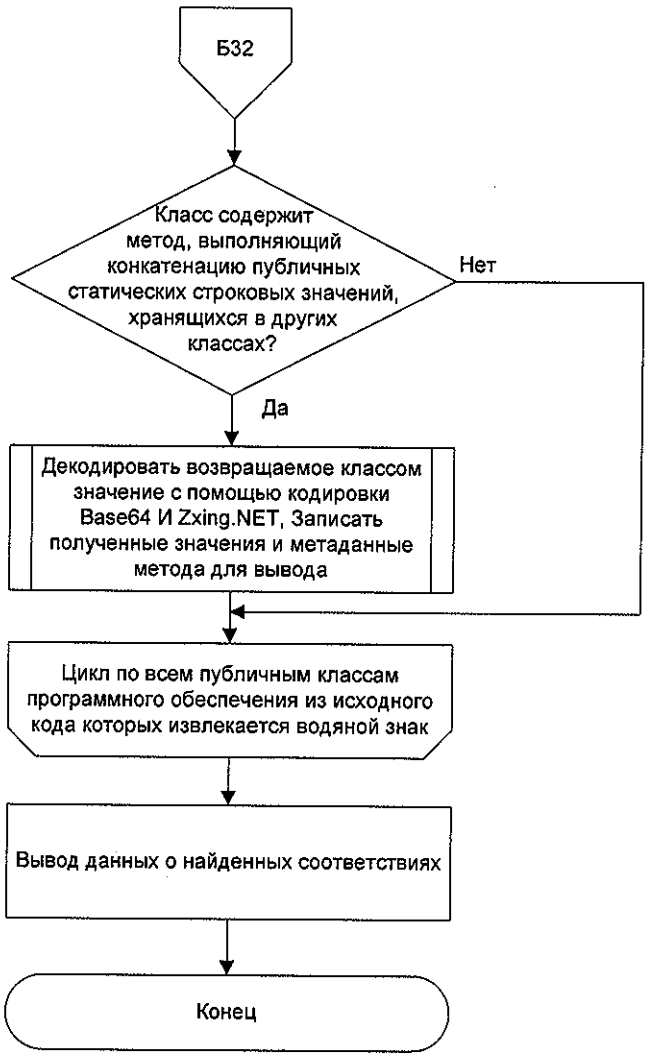

Рис. 2.6 Окончание алгоритма внедрения водяного знака.

2.5 Тестирование программного обеспечения

Для тестирования программы использовался исходный код библиотеки Dapper. Dapper – это популярная открытая библиотека, которая используется для маппинга сущностей из баз данных в объекты, являющиеся объектами различных классов. Данная библиотека была выбрана, поскольку она активно поддерживается и является хорошим примером востребованного программного обеспечения. Исходный код библиотеки полностью доступен на GitHub.

В ходе тестирования был применен алгоритм внедрения водяного знака с кодовой фразой, и затем код был обфусцирован с помощью .NET Reactor.

После декомпиляции кода с помощью JetBrains dotPeek была получена нерабочая некомпилируемая версия приложения, но в ходе работы алгоритма извлечения удалось извлечь все фрагменты водяного знака, сгенерированного с помощью ZXing. Получить и выполнить фиктивный метод, вызывающий эти поля не удалось, поскольку декомпиляция кода была усложнена .NET Reactor.

Из полученных фрагментов водяного знака удалось получить исходное кодовое выражение, чего достаточно для доказательства права обладания исходным кодом программного обеспечения.

Для доказательства права обладания достаточно проверить наличие достаточного количества сегментов для восстановления исходной строки, но и наличие какого-либо количества сегментов без возможности восстановления также может быть использовано в качестве доказательства права обладания исходным кодом программного обеспечения.

ЗАКЛЮЧЕНИЕ

Был выполнен анализ программного обеспечения в области Software Watermarking, на основании которого были сформированы требования к разрабатываемой системе внедрения и верификации водяных знаков в исходные коды программного обеспечения на языке C#. Произведен обзор методов внедрения водяных знаков в исходный код программного обеспечения и на основе наилучших из них был разработаны и реализованы алгоритмы внедрения и извлечения водяных знаков. Выполнены проектирование архитектуры системы и проектирование интерфейса.

Опубликована статья на научной конференции и подано заявление на регистрацию приложения.

В результате тестирования алгоритмов внедрения и извлечения водяных знаков была оценена эффективность работы алгоритмов на примере реального программного обеспечения.

Было разработано приложение, реализующее полученные алгоритмы внедрения и верификации. Разработанное приложение обеспечивает высокую устойчивость внедренных водяных знаков к трансформационным атакам и высокое качество их последующего распознавания, что позволяет разработчику уделять меньше времени на защиту продукта и ручное внедрение водяных знаков и различных семантических признаков.

Возможна дальнейшая модификация разработанного приложения с целью увеличения устойчивости водяных знаков к субстрактивным атакам с помощью статистического анализа исходных кодов и именования переменных, исходя из принятых нотаций, возможен перенос алгоритма для защиты программного обеспечения, написанного на языке Java, а также тестирование с программным обеспечением реализованным с помощью платформ Mono и Xamarin.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Гражданский кодекс Российской Федерации. Часть четвертая : [федеральный закон : принят Гос. Думой 21 окт. 1994 г.: по состоянию на 20 янв. 2012 г.]. – Новосибирск : Норматика, 2012. – 480 с.

2. Комяков, Д.С. Обзор методов внедрения статических водяных знаков в исходный код программного обеспечения [Текст] * / Д.С. Комяков, С.М. Елсаков // Южно-Уральская молодежная школа по математическому моделированию: сб. тр. всероссийской науч. конф. – Челябинск, 2015. – С. 84-92.

3. Пиратское ПО в России (Нелицензионное ПО). [Online]. Режим доступа: http://www.tadviser.ru/index.php/Статья:Пиратское_ПО_в_России_(Нелицензионное_ПО) , свободный - Дата обращения 28.03.2015.

4. Arboit, G. A method for watermarking java programs via opaque predicates / G. Arboit // The Fifth International Conference on Electronic Commerce Research (ICECR-5), 2002. [Online]. Режим доступа: http://citeseer.nj.nec.com/arboit02method.html, свободный - Дата обращения 28.03.2015.

5. Collberg, C. Sandmark / C. Collberg // Department of Computer Science, Aug. 2004. [Online]. Режим доступа: http://www.cs.arizona.edu/sandmark//, свободный - Дата обращения 28.03.2015.

6. Collberg, C. Software watermarking in the frequency domain: implementation, analysis, and attacks / C. Collberg, T. R. Sahoo // J. Comput. Secur., vol. 13, no. 5, pp. 721–755, 2005.

7. Collberg, C. Software watermarking: Models and dynamic embeddings / C. Collberg, C. Thomborson // Principles of Programming Languages 1999, POPL'99, 1999.

8. Cronin, G. A taxonomy of methods for software piracy prevention / G. Cronin // Department of Computer Science, University of Auckland, New Zealand, Tech. Rep., 2002.

9. Davidson, R. Method and system for generating and auditing a signature for a computer program / R. Davidson, N. Myhrvold // Jun. 1996, microsoft Corporation, US Patent 5559884.

10. Hamilton, J. An Evaluation of Static Java Bytecode Watermarking / J. Hamilton, S. Danicic // Department of Computing Goldsmiths, University of London UK, 2009

11. Hattanda, K. The evaluation of davidsons digital signature scheme / K. Hattanda, S. Ichikava // IEICE TRANS. FUNDAMENTALS, vol. E87A, no. 1, 2004.

12. Kearney, Software complexity measurement / Kearney, Sedlmeyer, Thompson, Gray, Adler // Commun. ACM, vol. 29, no. 11, p. 10441050, 1986.

13. Mishra, A. A methodbased Whole-Program watermarking scheme for java class files / A. Mishra, R. Kumar, and P. P. Chakrabarti // 2008. [Online]. Режим доступа:

http://citeseerx.ist.psu.edu/viewdoc/summary10.1.1.116.2810, свободный - Дата обращения 15.10.2015.

14. Monden, A. A practical method for watermarking java programs / A. Monden, H. Iida, K. ichi Matsumoto, K. Torii, K. Inoue // in COMPSAC '00: 24th International Computer Software and Applications Conference. Washington, DC, USA: IEEE Computer Society, 2000, p. 191197.

15. Myles, G. Software watermarking through register allocation: Implementation, analysis, and attacks / G. Myles, C. Collberg // International Conference on Information Security and Cryptology, 2003.

16. Myles, G. Software watermarking via opaque predicates: Implementation, analysis, and attacks / G. Myles, C. Collberg // ICECR-7, 2004.

17. Myles, G. Using software watermarking to discourage piracy/ G. Myles // Crossroads - The ACM Student Magazine, 2004.

18. Nagra, J. A functional taxonomy for software watermarking / J. Nagra, C. Thomborson, C. Collberg // Aust. Comput. Sci. Commun., M. J. Oudshoorn, Ed. Melbourne, Australia: ACS, 2002, pp. 177–186.

19. Qu, G. Hiding signatures in graph coloring solutions / G. Qu, M. Potkonjak // Information Hiding, 1999, pp. 348–367

20. Venkatesan, R. A graph theoretic approach to software watermarking / R. Venkatesan, V. Vazirani, S. Sinha // Proceedings of the 4th International Workshop on Information Hiding, 2001. [Online]. Режим доступа: http://www.cc.gatech.edu/fac/Vijay.Vazirani/water.ps, свободный - Дата обращения 15.10.2015.

21. Zhu, W. Concepts and techniques in software watermarking and obfuscation / W. F. Zhu // PhD Thesis, The University of Auckland, 2007.

ПРИЛОЖЕНИЕ 1

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Южно-Уральский государственный университет»
(национальный исследовательский университет)
Факультет математики, механики и компьютерных наук
Кафедра дифференциальных и стохастических уравнений

РАЗРАБОТКА ПРИЛОЖЕНИЯ ДЛЯ ВНЕДРЕНИЯ И ВЕРИФИКАЦИИ ВОДЯНЫХ ЗНАКОВ В ИСХОДНЫЕ КОДЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ТЕКСТ ПРОГРАММЫ
ЮУрГУ – 09.04.04.2016.129-246.ВКР

Руководитель, канд. физ.-мат. наук,
доцент
_____ / С.М. Елсаков /
«___» _____2016 г.

Автор, студент группы ММиКН-293
_____ / Д.С. Комяков /
«25» _____2016 г.

Нормоконтролер, канд. физ.-мат. наук,
доцент
_____ / М.А. Сагадеева /
«___» _____2016 г.

Челябинск 2016

АННОТАЦИЯ

Комяков Д. С. Разработка приложения для внедрения и верификации водяных знаков в исходные коды программного обеспечения. Текст программы. – Челябинск: ЮУрГУ, 2016. – 12 с.

Данный документ содержит исходные коды разработанного программного продукта на языке C#.

					ММиКН090404.16.129.246.12		
Изм.	Лист	№ докум.	Подпись	Дата			
Разраб.		Комяков Д.С.			Разработка приложения для внедрение и верификации водяных знаков в исходные коды программного обеспечения. Текст программы.	Лит.	Лист	Листов
Провер.		Елсаков С.М.				Д	38	12
Реценз.		Кувшинов Б.М.				ЮУрГУ кафедра ДиСУ		
Н. Контр.		Сагадеева М.А.						
Утверд.		Загребина С.А.						

ОГЛАВЛЕНИЕ

П2.1. ИСХОДНЫЙ ТЕКСТ ФАЙЛА «CONST.CS»

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ThesisProject
{

    public enum WatermarkTypeEnum
    {
        FakeField = 0,
        FakeMethod = 1,
        FakeMethodWithFakeField = 2
    }

    public class WatermarkType
    {
        public WatermarkTypeEnum Type { get; set; }
        public String Description { get; set; }
    }

    public static class Consts
    {
        public static IList<WatermarkType> WatermarkTypes = new
List<WatermarkType>{
            new WatermarkType() { Type = WatermarkTypeEnum.FakeField,
Description = "Fake field" },
            new WatermarkType() { Type = WatermarkTypeEnum.FakeMethod,
Description = "Fake Method" },
            new WatermarkType() { Type =
WatermarkTypeEnum.FakeMethodWithFakeField, Description = "Fake method with
fake field" },
        };
    }
}
```

П2.2. ИСХОДНЫЙ ТЕКСТ ФАЙЛА «WATERMARKINGPARAMS.CS»

```
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace ThesisProject
{
    public class WatermarkingParams
    {
        public WatermarkTypeEnum Type { get; set; }

        public Boolean WithFakeCall { get; set; }

        public Stream SourceStream { get; set; }

        public String FilePath { get; set; }

        public String AuthorshipString { get; set; }
    }
}
```

П2.3. ИСХОДНЫЙ ТЕКСТ ФАЙЛА «AUTHORSHIPDATASERVICE.CS»

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO.Compression;
using ZXing;

namespace ThesisProject
{
    public static class AuthorshipDataService
    {
        public static void GetAuthorshipData(string authorshipString)
        {
            IBarcodeWriter writer = new BarcodeWriter { Format =
BarcodeFormat.QR_CODE };
            var result = writer.Write(authorshipString);
            var barcodeBitmap = new Bitmap(result);
            using (var stream = new FileStream("test.bmp",
FileMode.OpenOrCreate, FileAccess.ReadWrite))
            {
                ImageConverter converter = new ImageConverter();
                var barcodeAsBytes =
(byte[])converter.ConvertTo(barcodeBitmap, typeof(byte[]));
                stream.Write(barcodeAsBytes, 0, barcodeAsBytes.Length);
            }
        }
    }
}
```

П2.4. ИСХОДНЫЙ ТЕКСТ ФАЙЛА «STREAMHELPER.CS»

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace ThesisProject
{
    public static class StreamHelper
    {
        public static string StreamToUTF8String(this Stream stream)
        {
            stream.Position = 0;
            using (StreamReader reader = new StreamReader(stream,
Encoding.UTF8))
            {
                return reader.ReadToEnd();
            }
        }

        public static Stream UTF8StringToStream(this string src)
        {
            byte[] byteArray = Encoding.UTF8.GetBytes(src);
            return new MemoryStream(byteArray);
        }

        public static string StreamToUnicodeString(this Stream stream)
        {
            stream.Position = 0;
            using (StreamReader reader = new StreamReader(stream,
Encoding.Unicode))
            {
                return reader.ReadToEnd();
            }
        }

        public static Stream UnicodeStringToStream(this string src)
        {
            byte[] byteArray = Encoding.Unicode.GetBytes(src);
            return new MemoryStream(byteArray);
        }
    }
}
```

П2.5. ИСХОДНЫЙ ТЕКСТ ФАЙЛА «WATERMARKINGSERVICE.CS»

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
using System.Diagnostics.Contracts;
using ICSharpCode.NRefactory;
using ICSharpCode.NRefactory.CSharp;


namespace ThesisProject
{
    public static class WatermarkingService
    {
        public static void PutWatermarkIntoSourceFile(WatermarkingParams
watermarkingParams)
        {
            if (watermarkingParams.SourceStream == null)
                throw new NullReferenceException("U must chose source code
class file for inserting watermark");

            var sourceTextReader = new
StreamReader(watermarkingParams.SourceStream);

            var parser = new CSharpParser();
            var syntaxTree = parser.Parse(sourceTextReader);
            var nameSpaceList =
syntaxTree.Members.OfType<NamespaceDeclaration>().ToList();
            var typeList = nameSpaceList.SelectMany(x =>
x.Members).OfType<TypeDeclaration>().ToList();
            var typeDeclaration = typeList.FirstOrDefault();

            var fakeField = new FieldDeclaration();
            var fakeVariableInitializer = new
VariableInitializer("FakeVariable");
            fakeField.Variables.Add(fakeVariableInitializer);
            fakeField.Modifiers = Modifiers.Protected;
            fakeField.ReturnType = new PrimitiveType("Int64");

            var fakeMethod = new MethodDeclaration
            {
                Name = "FakeMethod",
                ReturnType = new PrimitiveType("void"),
                Modifiers = Modifiers.Public,
                Body = new BlockStatement()
            };
```

```
                switch(watermarkingParams.Type){
                    case WatermarkTypeEnum.FakeField:
                        if (typeDeclaration!= null)
                            typeDeclaration.Members.Add(fakeField);
                        break;
                    case WatermarkTypeEnum.FakeMethod:
                        if (typeDeclaration != null)
                            typeDeclaration.Members.Add(fakeMethod);
                        break;
                    case WatermarkTypeEnum.FakeMethodWithFakeField:
                        if (typeDeclaration != null)
                        {
                            typeDeclaration.Members.Add(fakeField);
                            typeDeclaration.Members.Add(fakeMethod);
                        }
                        break;
                    default:
                        break;
                }
                var watermarkedSource = syntaxTree.ToString();
                watermarkingParams.SourceStream.Close();
                File.WriteAllText(watermarkingParams.FilePath,
watermarkedSource);
            }
        }
    }
```

П2.6. ИСХОДНЫЙ ТЕКСТ ФАЙЛА «FORM1.CS»

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;

namespace ThesisProject
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private WatermarkingParams watermarkingParams = new
WatermarkingParams();
        private OpenFileDialog sourceFileDialog = new OpenFileDialog();
        private FolderBrowserDialog folderDialog = new
FolderBrowserDialog();

        private void OpenSourceFileButton_Click(object sender, EventArgs e)
        {
            sourceFileDialog.Filter = "JSON file.|*.json";
            sourceFileDialog.Title = "Open watermar metadata file";
            sourceFileDialog.RestoreDirectory = true;

            if (sourceFileDialog.ShowDialog() == DialogResult.OK &&
(watermarkingParams.SourceStream = sourceFileDialog.OpenFile()) != null)
            {
                watermarkMetaDataFilePathTextBox.Text =
sourceFileDialog.FileName;
                watermarkingParams.FilePath = sourceFileDialog.FileName;
            }
        }

        private void IsWithFakeCallCheckBox_CheckedChanged(object sender,
EventArgs e)
        {
        }
```

```
        private void WatermarkTypeComboBox_SelectedIndexChanged(object
sender, EventArgs e)
        {
            //watermarkingParams.Type =
(WatermarkTypeEnum)WatermarkTypeComboBox.SelectedValue;
        }

        private void startButton_Click(object sender, EventArgs e)
        {
            watermarkingParams.AuthorshipString =
authorshipRichTextBox.Text;

AuthorshipDataService.GetAuthorshipData(watermarkingParams.AuthorshipString
);
            try
            {

WatermarkingService.PutWatermarkIntoSourceFile(watermarkingParams);
                MessageBox.Show("Watermarked!");
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }

        private void label2_Click(object sender, EventArgs e)
        {

        }

        private void button2_Click(object sender, EventArgs e)
        {
            new ExtractReportForm().Show();
        }

        private void sourceCodeDirectoryButton_Click(object sender,
EventArgs e)
        {
            folderDialog.ShowDialog();
        }

    }
}
```

П2.7. ИСХОДНЫЙ ТЕКСТ ФАЙЛА «PROGRAM.CS»

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace ThesisProject
{
    static class Program
    {
        /// <summary>
        /// Главная точка входа для приложения.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
  }
```

Формат	Зона	Позиц.	Обозначение	Наименование	Кол-во	Примечание
			Текстовые документы			
А4			ММиКН090404.16.129.246.00ПЗ	Пояснительная записка	36	
А4			ММиКН090404.16.129.246.12	Текст программы	12	

Изм.	Лист	№ докум.	Подпись	Дата
Разраб.		Комяков Д.С.		
Провер.		Елсаков С.М.		
Рецензиз.		Кувшинов Б.М.		
Н. Контр.		Сагадеева М.А.		
Утверд.		Загребина С.А.		

ММиКН090404.16.129.246.00ВД

Разработка системы кластеризации текстовых документов. Ведомость документов

Лит.	Лист	Листов
Д	49	1

ЮУрГУ
кафедра ДиСУ