УДК 681.3.06:620.9.002.56

С.Ю. Петрик, В. Н. Ярмолик

ИССЛЕДОВАНИЕ ЛЕКСИЧЕСКОГО МЕТОДА ЗАПУТЫВАНИЯ ИСХОДНЫХ ТЕКСТОВ ПРОГРАММ С ЦЕЛЬЮ ИХ ЗАЩИТЫ*

Проводится анализ исходных кодов программ для определения целесообразности запутывания его методом запутывающего кодирования (obfuscation) – заменой имен идентификаторов – как одним из наиболее распространенных способов противодействия технологиям обратной инженерии (reverse engineering). Для анализа выбран перспективный язык программирования C#. Исследуются практические возможности запутывания кода программ в зависимости от различных факторов, в том числе от его длины. Изучается содержание в коде программ идентификаторов различной длины. Кроме того, рассматривается автоматический запутывающий кодер и его функциональные возможности.

Введение

На сегодняшний день широкое распространение получила технология обратной инженерии (reverse engineering) [1], задача которой – восстановление структурной схемы и алгоритма работы по исходным текстам [2]. Существуют четыре основных метода защиты программы от приемов обратной инженерии:

1. Выполнение программы на стороне сервера (Server-Side Execution) – пользователь подключается к серверу разработчика и запускает программу удаленно. Злоумышленник, пытающийся получить информацию об алгоритмах работы программы, не имеет прямого доступа к приложению и, следовательно, не может использовать методы обратной инженерии. Очевиден недостаток такого метода защиты интеллектуальной собственности – низкая производительность программы, запущенной удаленно, и большая стоимость реализации этого метода.

2. Передача пользователю внутреннего (собственного) кода ЭВМ (Native Code) – переданная программа подходит для компьютера конкретного пользователя. Она защищается и другими средствами, чаще всего используется цифровая подпись, позволяющая обеспечить целостность и аутентичность приложения. Декомпиляция такого кода возможна, но понимание его при этом будет затруднено.

3. Шифрование (Encryption) – программа распространяется в зашифрованном виде и расшифровывается на стадии выполнения. На сегодняшний день разработаны средства, позволяющие обойти такую защиту [2].

4. Запутывающее кодирование (Obfuscation – дословно озадачивание, сбивание с толка) – перед окончательной сборкой проекта производитель пропускает его код через автоматический кодировщик. Кодирование с целью запутывания – преобразование программы в функционально идентичную оригинальной, но менее подверженную декомпиляции и приемам обратного проектирования. Этот метод не способен полностью защитить программу от вскрытия, его задача заключается в другом – сделать это вскрытие технически весьма сложным и неоправданно дорогим для злоумышленников. Преимущество запутывающего кодирования заключается в его дешевизне относительно других методов защиты [3].

Существует достаточно много способов запутывания исходного кода программы. Один из наиболее распространенных – изменение имен идентификаторов. Для них разработчики чаще всего используют имена, достаточно много говорящие о назначении называемого объекта. Многие разработчики обратили свое внимание на так называемую «венгерскую нотацию» [4]. Эти и другие правила именования призваны ускорить процесс понимания программ, что, однако, облегчит понимание кода и злоумышленникам. Для ликвидации данного недостатка используются запутывающие кодеры, переименовывающие идентификаторы. Существуют раз-

* Работа выполнена при поддержке гранта BMBF BLR 02/006 «DSP-based control systems for safety-related applications».

личные правила переименования: в длинные последовательности символов с малым хэмминговым расстоянием; в короткие имена типа «знак подчеркивания, число»; в имена, которые носят смысловую нагрузку, однако не ту, которая была до кодирования. Кроме чистых методов применяются и их комбинации [2].

Цель настоящей работы заключается в оценке целесообразности и возможности использования данного метода для защиты интеллектуальной собственности на программный продукт. Количественно это можно оценить по доле кода программы, которая поддается замене, а значит и запутыванию.

1. Проблема безопасности платформы .NET

Одной из последних и наиболее интересных разработок в области программного обеспечения представляется платформа .NET (dot NET). Об этом свидетельствует ряд факторов [5, 6]:

– широкие возможности по взаимодействию с уже существующим кодом;

– абсолютное межъязыковое взаимодействие (вплоть до межъязыкового наследования, обработки исключений и отладки);

– общая среда выполнения для приложений .NET вне зависимости от использованного языка;

– библиотека базовых классов – целостная модель для всех языков программирования .NET;

– упрощение развертывания приложений (фактически решена проблема «ад DLL» – разные версии одного и того же модуля DLL могут сосуществовать на одном компьютере).

Для реализации этих идей были использованы передовые технологии:

CLR (Common Language Runtime) – среда выполнения для платформы .NET;

CTS (Common Type System) – система типов платформы;

CLS (Common Language Specification) – набор правил, определяющих подмножество общих типов данных, что позволяет использовать код при взаимодействии с любым языком .NET.

Сейчас полноценно существует платформа .NET только для OS Windows, однако активно ведутся разработки платформы Mono, которая представляет собой ту же .NET для альтернативных систем, в первую очередь для Linux и Unix. Таким образом, программа, написанная для .NET, является независимой от операционной и аппаратной сред.

Однако за всеми этими бесспорными преимуществами остро стоит проблема защиты программных продуктов, написанных для платформы .NET [5], которые компилируются не как обычное Win32 приложение – в машинные инструкции, а в сборки, написанные на промежуточном языке MSIL (Microsoft Intermediate Language) и содержащие в себе манифест, метаданные и непосредственно инструкции на MSIL [7]. Концептуально сборка на MSIL напоминает байт-код Java и имеет те же проблемы: достаточно простая декомпиляция делает программы очень уязвимыми для злоумышленников (в комплекте поставки Visual Studio .NET входит утилита IL Dasm, представляющая собой дизассемблер языка MSIL [8, 9]). Фактически достаточно легко можно получить исходный код программы, что позволит исследовать ее алгоритмы [2]. Существует возможность изменять сборки для .NET таким образом, чтобы они не поддавались декомпиляторам (они будут восприниматься как содержащие ошибки, но при этом правильно выполняться). Этот выход временный – декомпиляторы постоянно совершенствуются и нет никаких гарантий, что не существуют декомпиляторы, которые не будут реагировать на внесенную ошибку. Разработчики Java – Sun Microsystems – для защиты байт-кода использовали приведенный выше метод шифрования (Encrypting): в среду выполнения Java встраивалась функция выполнения зашифрованного кода. Однако, как было уже отмечено, шифрование уязвимо для взломщиков. Вероятно, оно будет использовано и в платформе .NET, но полностью проблему защиты это не решит. Кроме того, вопросом шифрования ученые занимаются уже достаточно долго и плодотворно. Новым и интересным методом борьбы с несанкционированным изучением программы представляется ее запутывающее кодирование [4].

Таким образом, данная работа ориентирована на платформу .NET, что достаточно актуально. В частности, исследуется язык C#. Он был разработан специально для платформы .NET и имеет достаточно много преимуществ по сравнению даже с признанными лидерами – C++,

предыдущими версиями – их просто нет. Поэтому нет необходимости поддерживать потенциально опасные и неэффективные конструкции (что является серьезной проблемой в C++). C# вообще не поддерживает (явно) одну из самых небезопасных конструкций – указатели (что делают все остальные перечисленные языки) [5]. В этом языке полностью обеспечена поддержка всех принципов объектно-ориентированного программирования (чего нет в VB). Эти и многие другие особенности, по всей вероятности, сделают его вскоре одним из самых популярных языков для платформы .NET. Как следствие именно он будет подвергаться атакам, направленным на вскрытие кода [10].

2. Метод запутывающего кодирования заменой имен идентификаторов

Таким образом, в качестве платформы для применения запутывающего кодирования выбрана платформа .NET, а в качестве исследуемого метода – метод замены имен идентификаторов.

Примером кодирования этим методом могут служить следующие фрагменты исходного текста программы:

Листинг 1. **Исходный код до проведения запутывающего кодирования**

```
private Hashtable getFrequency(string fileName)
{
        int readByte;
        FileStream fs = new FileStream
                (fileName,FileMode.Open,FileAccess.Read,FileShare.None);
        Hashtable frequency = new Hashtable();
        while ((readByte=fs.ReadByte())!=-1)
        {
        if (frequency[readByte]!=null)
                        frequency[readByte]=(int)(frequency[readByte]) + 1;
        else
                        frequency.Add(readByte,1);
        }
        fs.Close();
      return frequency;
}
```

В листинге 1 приведен один из методов класса, реализующего функции криптографического кодирования. Этот метод анализирует файл с именем fileName и на выходе возвращает хэш-таблицу frequency, содержащую коды всех символов, встреченных в файле, и их количество.

В рамках исследования был создан запутывающий кодер Obfuscation Studio, который будет описан ниже. После его использования текст приведенной выше программы принял следующий вид.

Листинг 2. **Исходный код после проведения запутывающего кодирования с использованием одного из правил именования**

```
private Hashtable ll1lll11(string ll1lll1l)
{
        int ll1llll;
        FileStream ll1ll1ll = new FileStream
                (ll1lll1l,FileMode.Open,FileAccess.Read,FileShare.None);
        Hashtable ll1ll1l1 = new Hashtable();
        while ((ll1llll=ll1ll1ll. ReadByte())!=-1)
        {
        if (ll1ll1l1[ll1llll]!=null)
                        ll1ll1l1[ll1llll]=(int)(ll1ll1l1[ll1llll]) + 1;
```

```
        else
                    ll1ll1l1.Add(ll1llllll,1);
        }
        ll1ll1ll.Close();
        return ll1ll1l1;
}
```

Видно, что код стал значительно более сложным для восприятия человеком, но не компьютером – подпрограмма так же компилируется и выполняется без видимых задержек. Для подтверждения этого приведенный участок программы в обоих представлениях был выполнен 10^7 раз, разницы во времени выполнения обнаружено не было.

У данного метода кодирования есть недостаток – можно выполнить декодирование. Для этого существуют автоматические программы-декодеры. Однако они только частично делают код понятней, так как очевидно, что первоначальные осмысленные имена вернуть уже невозможно. Было проведено декодирование предыдущего фрагмента и в результате получен следующий код.

Листинг 3. **Исходный код после проведения попытки декодирования**

```
private Hashtable _3(string __0)
{
        int __1;
        FileStream __2 = new FileStream
              (__0,FileMode.Open,FileAccess.Read,FileShare.None);
        Hashtable __3 = new Hashtable();
        while ((__1=__2.ReadByte())!=-1)
        {
        if (__3[__1]!=null)
                    __3[__1]=(int)(__3[__1]) + 1;
        else
                    __3.Add(__1,1);
        }
        __2.Close();
        return __3;
}
```

Этот код понять проще, но ненамного. Он остался запутанным. Декодирование было проведено программой Obfuscation Studio, написанной в рамках исследования. Необходимо заметить, что запутывающее кодирование методом замены имен идентификаторов необратимо воздействует на текст программ. Кроме того, код, приведенный выше в качестве декодированного, содержит идентификаторы, именованные по тем же правилам, которые использует Dotfuscator – кодер, поставляемый с пакетом разработки Visual Studio 2003 от Microsoft. Это говорит о том, что метод замены имен переменных достаточно надежный, а при воздействиях на запутанный код принципиальных шагов в сторону упрощения его понимания сделать практически невозможно.

3. Возможности запутывающего кодирования кода программ

Основное внимание в настоящей статье уделено языку C#, однако интересным представляется сравнение возможности кодирования программ на этом языке с другими. В качестве экземпляра процедурно-ориентированного языка был выбран C. Этот язык, проверенный временем, очень широко используется: на нем полностью или частично написаны самые популярные операционные системы – Windows и Unix/Linux. Был исследован также язык C++ – пример смешанного языка, в котором одинаково часто используются процедурно-ориентированные и объектно-ориентированные технологии. Был, кроме того, исследован достаточно близкий к C# полностью объектно-ориентированый язык Java. Как отмечалось ранее, он имеет те же проблемы с безопасностью, что и C#. Для сбора статистики, приведенной ниже в статье, использова-

лись исходные файлы приложений, поставляемых в качестве примеров к книгам и студиям разработки по соответствующим языкам программирования; тексты, полученные через Интернет (исходные файлы мультимедиа, клиент-серверные приложения, текстовые редакторы), кроме того, исследовались исходные файлы Linux и продуктов, поставляемых с этой операционной системой, и многое другое. Всего было проанализировано более 1 000 программ.

Код любой программы, как и любой текст, состоит из слов и знаков препинания. Слова могут быть ключевыми словами самого языка; словами, определенными в подключаемых библиотеках; числами; строковыми и буквенными константами; строками комментариев либо идентификаторами, определенными программистом непосредственно в программе. Таким образом, модель кода программы можно формализовать следующим образом:

$$P' = \{V, W, C\},$$

где P’ – множество всех лексических единиц программы;

V – множество идентификаторов, определенных программистом;

W – все остальные слова, кроме комментариев (в это множество входят ключевые слова языка, имена структур данных и методов, определенных в подключаемых библиотеках, числа, отдельно взятые строки и символы и т. п.);

C – множество комментариев.

Самый простой метод запутывающего кодирования – удаление из множества P’ подмножества C, т. е. удаление из кода программы комментариев: {V, W, C} → {V, W}, P’ → P, где P = {V, W}. Удаление комментариев – первый шаг запутывающего кодирования, он тривиален, поэтому далее будем считать, что он уже проведен и будет осуществляться кодирование непосредственно кода программы.

Для исследования кода программы найдем следующие значения:

1. Количество различных слов, использованных в программе, – мощность множества P:

$$|P| \tag{1}$$

2. Количество различных идентификаторов, объявленных в программе, – мощность множества V:

$$|V| \tag{2}$$

3. Общее количество слов, составляющих программу, – сумму всех слов множества P:

$$\sum_{i=1}^{|P|} n(p_i), \tag{3}$$

где $n(p_i)$ – число использования одной и той же лексической единицы p_i в коде программы.

4. Общее количество использований идентификаторов – количество использования слов множества V:

$$\sum_{i=1}^{|V|} n(v_i). \tag{4}$$

5. Суммарную длину всех слов программы:

$$\text{length}\left(\sum_{i=1}^{|P|} n(p_i)\right). \tag{5}$$

6. Суммарную длину идентификаторов:

$$\text{length}\left(\sum_{i=1}^{|V|} n(v_i)\right). \qquad (6)$$

7. Количество всех слов в зависимости от длины (от 1 до 24 букв);
8. Количество идентификаторов в зависимости от длины (от 1 до 24 букв).

На основании приведенных абсолютных значений можно получить и характеристики, оценивающие относительные соотношения параметров программы, например (2):(1), (4):(3), (6):(5), и зависимости этих отношений от (1). В качестве базовой величины можно выбрать параметр (3), но не (5), это связано с тем, что (5) меняется в результате запутывающего кодирования, что было показано выше (кодирование влияет на параметры (5), (6) и не влияет на (1)-(4)). Верхняя и нижняя оценки представляются коэффициентами k прямых, ограничивающих область экспериментально полученных значений зависимостей (2) от (1), (4) от (3), (6) от (5) в уравнении y=a+kx.

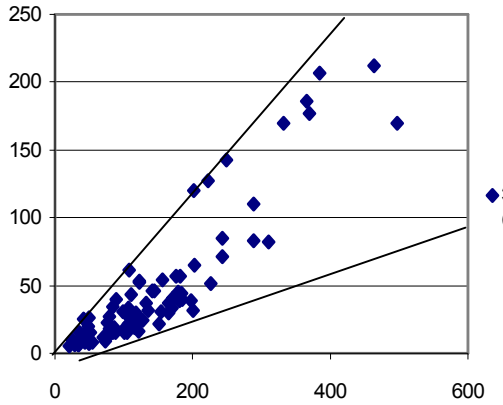

Рис. 1. Зависимость (2) от (1):
верхняя граница – k = 0,60;
нижняя – k = 0,2

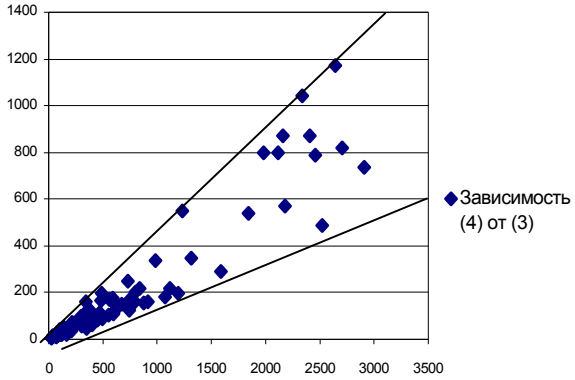

Рис. 2. Зависимость (4) от (3)
верхняя граница – k = 0,44;
нижняя – k = 0,20

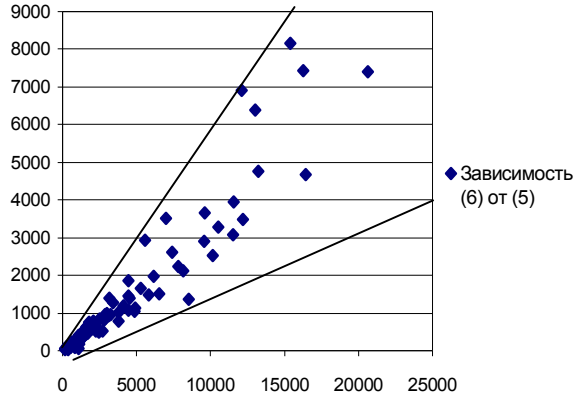

Рис. 3. Зависимость (6) от (5)
верхняя граница – k = 0,60; нижняя – k = 0,17

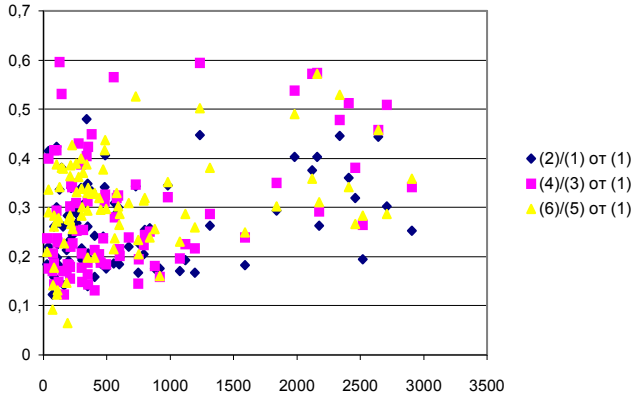

Рис. 4. Зависимость отношений
(2):(1), (4):(3), (6):(5) от (1)

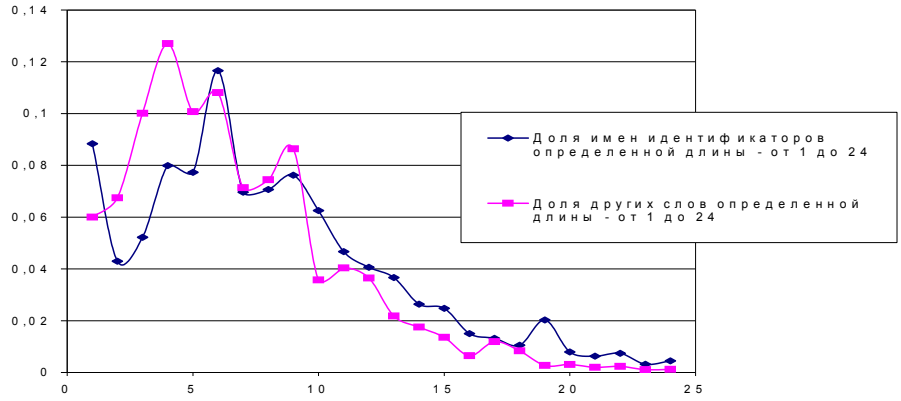

Рис. 5. Содержание в текстах программ имен идентификаторов и имен других слов длиной от 1 до 24 букв. Средняя длина идентификаторов – 8,0; средняя длина всех слов – 6,7

Для всех остальных языков диаграммы незначительно отличаются по виду [11], поэтому их характеристики приведены без графиков в табл. 1.

Таблица 1

Статистические данные для языков программирования C, C++, C#, Java

Язык программирования	Зависимость формулы (2) от (1), границы		Зависимость формулы (4) от (3), границы		Зависимость формулы (6) от (5), границы		Средняя длина	
	верхняя	нижняя	верхняя	нижняя	верхняя	нижняя	идентификаторов	всех слов
C#	0,60	0,20	0,44	0,20	0,60	0,17	8,0	6,7
C	0,63	0,26	0,42	0,30	0,47	0,15	7,8	9,3
C++	0,40	0,20	0,48	0,19	0,30	0,18	8,1	10,2
Java	0,54	0,14	0,46	0,17	0,53	0,21	6,8	6,4

4. Анализ результатов

Для всех языков наблюдается близость результатов графиков, которые для C# изображены на рис. 1 – 3, что вполне логично: чем длиннее программа, тем больше слов в ней используется. Фактически для всех языков получаются значения, лежащие в диапазоне от 15 до 55%, – это тот процент кода, который можно заменить и запутать (рис. 4). Определенный разброс для каждого из языков связан со спецификой программ и стилем написания: можно с высокой степенью уверенности утверждать, что для каждого конкретного программиста значения пределов параметров программы будут различными. График на рис. 5 и соответствующие графики для других языков показали, что средняя длина имен идентификаторов равна 7–8 букв, т. е. это та длина слов, которая достаточно много может сказать о назначении идентификатора, в то же время не загромождая текст программы. Видно некоторое несоответствие между языками в величинах средней длины всех остальных слов (кроме идентификаторов), между процедурно-ориентированными (C/C++) и объектно-ориентированными (C#, Java) языками. Для первой

группы средняя длина слов достаточно велика (9–10 букв), для второй – существенно меньше (6–7 букв). Объяснить это можно тем, что процедура должна достаточно полно раскрывать свое назначение: в ней приходится кодировать и объект воздействия, и само действие; в методах объектно-ориентированного программирования достаточно указать только действие, объект кодировать не нужно [12].

5. Система кодирования Obfuscation Studio

В предыдущем разделе было показано, что часть кода, которую можно заменить и запутать, представляет собой достаточно большую его часть, достигающую 55%. Косвенно эти результаты говорят и об эффективности исследуемого метода запутывающего кодирования, ведь чем большую часть программы можно запутать, тем сложнее ее понять. Для использования этого метода был создан программный комплекс Obfuscation Studio, он состоит из визуальной оболочки, к которой могут подключаться dll-библиотеки, реализующие тот или иной метод запутывающего кодирования. Для комплекса написана библиотека, позволяющая использовать метод замены имен идентификаторов для языков C, C++, C# и Java. Программа работает в три этапа.

Создается карта идентификаторов. Для этого текст разбивается на слова, последовательность которых затем анализируется. Для нахождения новых идентификаторов программа использует некоторое количество шаблонов инициализации:

– <тип> <имя переменной>;

– <тип объекта = имя класса > <имя объекта>;

– <имя объекта = имя класса > = new <конструктор объекта = имя класса >(<параметры конструктора класса >);

– <тип объекта = имя класса> <имя объекта> = new <конструктор класса = имя класса>(<параметры конструктора класса>);

– class <имя класса>

– <модификатор доступа> <имя конструктора = имя класса>(< конструктора класса >)

Данные шаблоны и некоторые другие позволяют находить все идентификаторы, объявленные разработчиком в программе. При нахождении очередного идентификатора вызывается функция добавления идентификатора. Так строится дерево, где корень – пространство имен, затем классы; их методы и свойства составляют узлы, а листья – поля классов и локальные переменные методов и свойств.

Создается еще одна карта имен идентификаторов, которая представляет собой точную копию оригинальной, но имена в ней закодированы. Программа-кодер может работать в нескольких режимах запутывания имен идентификаторов:

– замена имен идентификаторов на длинные имена, составленные из буквы l и цифры 1, которые во многих шрифтах практически идентичны, что весьма затрудняет восприятие слов;

– замена на длинные слова, составленные из буквы O и цифры 0, которые также не всегда просто отличить;

– замена на короткие идентификаторы типа _n, где n – число, равное номеру метода, свойства, поля, класса или пространства имен. Этот метод визуально выглядит предпочтительнее двух предыдущих, поэтому по отношению к ним его можно считать методом декодирования.

Новая карта используется для создания закодированного файла.

В перспективе планируется применять и другие правила именования переменных, а также создавать модули, которые будут реализовывать не только лексический метод, исследуемый в данной работе, но и семантический.

Заключение

Исследование, проведенное в настоящей работе, позволило сделать определенные выводы о возможности и целесообразности использования данного метода запутывающего кодирования (замены имен идентификаторов) для защиты программы. Количественно это выражается

следующим образом: от 15 до 55% слов программы (что примерно соответствует той же доле кода программы) поддается изменению. Однако выводы этим утверждением не ограничиваются: указанную долю программы можно использовать и для запутывающего кодирования, и для внесения в код цифровых подписей, водяных знаков и т. д. Примечательно, что различные языки программирования показывают близкую эффективность исследуемого метода кодирования. На основе исследования был создан программный комплекс Obfuscation Studio, который реализует метод запутывающего кодирования заменой имен идентификаторов.

Список литературы

1. Reverse Engineering Wizard // Microsoft .NET Framework SDK Tool Developer's Documentation. – Microsoft Corporation, 2001.
2. The Dotfuscator Solution // Obfuscation for .NET – Dotfuscator by PreEmptive Solutions. PreEmptive Solutions, 2002.
3. Goals of Obfuscation // Obfuscation for .NET – Dotfuscator by PreEmptive Solutions. PreEmptive Solutions, 2002.
4. Simonyi C. Hungarian notation // Visual Studio (General) Technical Articles. – Microsoft Corporation, 2004.
5. Троелсен Э. C# и платформа .NET. Библиотека программиста. – СПб.: Питер, 2003. – 800 с.
6. Робисон У. C# без лишних слов: Пер. с англ. – М.: ДМК Пресс, 2002. – 323 с.
7. Common Language Infrastructure (CLI) // Microsoft .NET Framework SDK Tool Developer's Documentation. – Microsoft Corporation, 2001.
8. CIL Assembler (ilasm.exe/ilasm) // Microsoft .NET Framework SDK Tool Developer's Documentation. – Microsoft Corporation, 2002.
9. Ildasm.exe Tutorial. .NET Framework Tutorials // Microsoft .NET Framework SDK Tool Developer's Documentation. – Microsoft Corporation, 2002.
10. Рихтер Дж. Программирование на платформе Microsoft .NET Framework: Пер. с англ. – М.: Русская редакция, 2002. – 512 с.
11. Петрик С.Ю., Ярмолик В.Н. Исследование эффективности обфускации исходных текстов программ // Докл. БГУИР. – 2004. – № 5. – С. 27.
12. Петрик С.Ю., Ярмолик В.Н. Использование обфускации для защиты интеллектуальной собственности // Известия Белорусской инженерной академии. – №1(17)/2. – 2004. – С. 168-171.

Поступила 29.07.04

Белорусский государственный университет
информатики и радиоэлектроники,
Минск, П. Бровки, 6
E-mail: ciceron@tut.by, yarmolik@bsuir.unibel.by

S.Y. Petrik, V.N. Yarmolik

LEXICAL APPROACH INVESTIGATION FOR SOURCE CODE OBFUSCATION

A novel approach to protect software code against reverse engineering has been investigated. This solution can be regarded as the measure for the software application copywrite protection for the .NET platform. The main part of the paper is dealing with the statistical analyses of the software code to get the numerical estimations for the obfuscation efficiency. As have been shown up to the 55% of the original software code can be obfuscated to increase the complexity of reverse engineering. The last part of the paper presents the obfuscation tool “OBFUSCATION STUDIO”, which has been developed by the authors and describes the main characteristics of this tool.