УДК 003.26

И.В. Нечта, А.Н. Фионов

ЦИФРОВЫЕ ВОДЯНЫЕ ЗНАКИ В ПРОГРАММАХ НА C/C++*

Статья посвящена системам цифровых водяных знаков для программ на языках C и C++. Водяные знаки внедряются путем небольших эквивалентных изменений исходных текстов программ. Предполагается, что такие знаки будут устойчивы к атакам на двоичные исполняемые файлы, при условии, что противник не будет иметь доступа к исходным текстам. Экспериментально получены данные об объеме внедряемой информации для типичных приложений Symbian. Рассматриваются проблемы и направления будущего развития исследований.

Цифровые водяные знаки; стеганография программ; Symbian, ARM.

I.V. Nechta, A.N. Fionov

DIGITAL WATERMARKS FOR C/C++ PROGRAMS

The results of the first stage of the project devoted to development of watermarking system for C/C++ programs are presented. Watermarks are embedded by making slight equivalent variations of program's source codes. It is supposed that such watermarks will be robust with respect to attacks on binary executable files when the adversary has no access to their source codes. The rate of embedding for typical Symbian applications is experimentally obtained. Remaining problems and directions of future research are discussed.

Digital watermarks; software steganography; Symbian; ARM.

Введение. Построение систем цифровых водяных знаков является одним из направлений стеганографии, к целям которого относят сокрытие данных в цифровых средах или файлах таким образом, чтобы внедренный знак было бы трудно удалить или обнаружить. В отличие от классической задачи стеганографии, т.е. проблемы сокрытия факта передачи данных, обычно не требуется скрывать наличие водяного знака в большинстве приложений. Считается, что можно заранее предугадать, содержится ли водяной знак или нет.

Одно из применений цифровых водяных знаков – это защита авторского права. Предположим, что вы создали программный продукт и внедрили в него водяной знак "ТТИ ЮФУ". После некоторого времени вы находите компанию, продающую этот же продукт как свой собственный, возможно с измененными названием или метаданными во внутренних файлах. Но если водяной знак устойчив, то есть противостоит удалению или модификации, то вы будете способны извлечь слово "ТТИ ЮФУ" и доказать ваше авторство в суде. Вероятность того, что водяной знак "ТТИ ЮФУ" возник в файлах программы случайно, ничтожна мала.

Другое, более перспективное применение водяных знаков – защита от нелегального копирования. Идея сводится к следующему: автор внедряет в каждую продаваемую копию программы уникальный водяной знак, который позволил бы отслеживать пользователя, который делает незаконные копии. Проблема здесь возникает, когда несколько пользователей создают коалицию. Они могут сравнить имеющиеся копии программ, найти все различия с целью определения местоположения водяного знака. Знание этих местоположений может помочь им изменить водяной знак таким образом, чтобы создать копию, по которой не может быть прослежен ни один пользователь из коалиции. Для предотвращения подобной возможности существуют так называемые коды снятия отпечатков пальцев, которые используются как водяные знаки. Специальная конструкция и избыточность таких

* Работа выполнена в рамках НИР по Гос. контракту № 02.740.11.0396.

кодов гарантируют, что никакая коалиция размера менее чем некоторое предопределенное значение не может создать непрослеживаемый водяной знак. Одна из проблем в схеме создания водяного знака состоит в обеспечении достаточной его длины для размещения кодов снятия отпечатков пальцев.

Поскольку мы говорим о водяных знаках в программах, то следует отметить, что внедрение данных в программные файлы имеет свою собственную специфику, например, внедрение не может быть сделано прямой модификацией некоторых бит исполняемых файлов, потому что любая такая модификация, с высокой долей вероятности, нарушит алгоритм работы программы. Внедрение должно быть сделано таким путём, который гарантирует правильное функционирование программы. Проблема программных водяных знаков была поставлена и рассматривалась в множестве работ [1], [2], [3], [4], [5], [6], [7]. Существует два основных подхода. Первый предполагает использование специальных дополнительных функций, которые добавляются к коду программы [2], [4], [5]. Код дополнительных функций содержит требуемый водяной знак, например, в виде констант, использующихся в различных вычислениях. Эти добавленные функции не должны выглядеть как “мертвый код” (код, который никогда не будет выполнен), потому что иначе они могут быть легко удалены алгоритмом устранения мертвого кода. То есть они должны быть интегрированы в программу и вызваны в течение выполнения программы. Преимущество этого подхода состоит в фактически неограниченной длине водяного знака. Однако, недостаток – потенциальная деградация алгоритма работы программы. Другим недостатком – является уязвимость к атакам, которые находят некоторые известные образцы функциональных конструкций водяного знака и способов обращения к нему.

Второй подход не использует добавление явных кодов, а задействует некоторый незначительный избыток в программных файлах, который позволяет внедрять водяной знак [1], [3], [6], [7]. Подобный подход используется в [8], [9], [10], чтобы скрыть данные непосредственно в исполняемых файлах. Общая особенность этих методов состоит в нахождении некоторого набора эквивалентных способов генерации исполняемого файла и сокрытие данных через выбор одного из них. Методы генерации кода зависят от компилятора и, в частности, от его методов выбора типа команды, планирования инструкций, размещения текста программы, выделения регистров, расстановке переменных и заполнение адресами функций таблиц импорта. Следует обратить внимание на то, что некоторые модификации кода могут быть применены к уже готовому исполняемому файлу, в то время как другие – только во время компиляции и, поэтому, требуется специально разработанный компилятор. Мы кратко проиллюстрируем сущность каждого метода. Примеры во всей статье будут даваться на C / C ++ и языке Ассемблера процессоров ARM. Сегодня процессоры ARM доминируют на рынке мобильной и встраиваемой электроники, составляя аппаратную платформу для мобильных устройств Nokia, Sony Ericsson и др.

Метод выбора команды состоит в нахождении различных последовательностей кода, которые производят то же самое вычисление. Например, инструкция d = d + 1 может быть выполнена путем добавления 1 или, вычитанием -1 от d. Инструкция d = d * 2 может быть выполнена добавлением d к d или левым битовым сдвигом d. Эти варианты могут быть закодированы как 0 и 1 соответственно. Выбор одного из них зависит от внедряемого бита данных. Обратим внимание, что возможность эффективного поиска альтернативной команды зависит от типа процессора и ограничена в RISC типа ARM. В ARM удвоение переменной может быть сделано следующим образом:

add rd, rd, #0 или mov rd, rd, asl #1.

Оба варианта одинаковы по байтовой длине и времени выполнения. Следует учесть, что непосредственное вычитание отрицательного числа в процессорах ARM невозможно.

Метод планирования инструкций находит блоки с независимыми командами процессора. Все перестановки независимых команд упорядочивают лексикографически и рассматриваются как эквивалентные. Двоичное представление номера перестановки команд есть внедренное сообщение. Например, в блоке

```
mov r3, r4,
add r0, r4, #1
```

две операции являются независимыми, следовательно, возможна эквивалентная последовательность

```
add r0, r4, #1,
mov r3, r4.
```

Таким образом, можно закодировать один бит информации.

Метод размещения кода программы основан на информации, полученной в ходе анализа кода программ. Любой машинный код может быть представлен как набор блоков с последовательно выполненными командами и отделенными командами вида “go to” (переходы). Эти блоки кода могут быть помещены в память (в содержимое исполняемого файла) в различном порядке. Аналогично внедренные данные – это двоичное представление номера перестановки блоков кода.

Метод выделения регистров имеет дело с назначением регистра процессора для размещения некоторой переменной. Если в предыдущем примере переменная d может быть помещена в регистр и компилятор имеет несколько свободных регистров, скажем r3 и r4, то дополнительный бит может быть внедрен путем назначения d = r3 или d = r4. Для переменных, которые не могут быть помещены в регистр, может использоваться различный порядок их расположения в памяти. Различные комбинации назначения регистров или расположения переменных в памяти дают возможность скрывать данные.

Мы также можем воспользоваться преимуществом технологии динамического связывания, которая преобладает в современном программировании. Различные перестановки адресов функции в таблицах импорта / экспорта и поддерживающих структурах может быть использовано для скрытия данных.

Все рассмотренные методы могут быть объединены вместе, чтобы увеличить общий объем внедрения информации. Однако существует проблема. Внедрение данных не всегда устойчиво. Это, прежде всего, относится к внедрению, сделанному непосредственно в готовую исполняемую программу. Та же самая программа, которая внедряет данные, может использоваться для стирания или изменения водяного знака. Именно поэтому в нашем проекте мы изучаем методы внедрения, которые применяются к исходным текстам программ. В этой статье мы представляем текущие результаты, проблемы и направления будущих исследований.

Описание методов внедрения для программ C / C ++. Мы рассматриваем автономную программу, которая используется для того, чтобы внедрить водяной знак в исходные файлы C / C ++ и запускается перед компиляцией программы. Какой же метод из тех, что мы описали во введении, подходит в такой ситуации? Во-первых, очевидно, что трудно влиять на способы заполнения компилятором таблиц импорта / экспорта. Во-вторых, это задача компилятора – выбрать команды для трансляции исходных текстов. Другие три метода вполне могут быть использованы. Планирование инструкций может быть осуществлено путем перестановки некоторых операторов (операций) в исходном тексте программы. Мы также, до некоторой степени, можем контролировать выделение регистров и распределение

памяти в зависимости от порядка объявления переменных. Изменение размещения кода программы также возможно, но представляется наиболее сложным. Кроме того, этот метод защищен патентом [1]. В первой версии нашей программы внедрения водяных знаков мы ограничились упорядочиванием операций присваивания и упорядочиванием локальных переменных.

Рассмотрим пример работы нашей программы. Пусть дана следующая функция:

```
int f (int x)
{
int a, b;
a = x + 1;
b = x – 1;
return a + 2 * b;
}
```

Вычислительная часть функции, откомпилированая GCC:

```
ldr     r3,     [fp, #-16]
add     r3,     r3,     #1
str     r3,     [fp, #-20]
ldr     r3,     [fp, #-16]
sub     r3,     r3,     #1
str     r3,     [fp, #-24]
ldr     r3,     [fp, #-24]
mov     r2,     r3,     asl     #1
ldr     r3,     [fp, #-20]
add     r3,     r2,     r3
mov     r0,     r3
```

Распределение памяти для переменных соответствует порядку их объявления:

x = [fp, #-16], a = [fp, #-20], b = [fp, #-24].

Согласно нашему подходу мы можем внедрить два бита водяного знака в эту функцию путем выбора порядка объявления переменных и порядка следования независимых инструкций:

```
0→   int a, b;          1→   int b, a;

0→   a = x + 1;         1→   b =  x – 1;
     b = x – 1;              a =  x + 1.
```

Так, исходная функция, в которой объявления переменных и инструкции упорядочивают лексикографически, соответствует водяному знаку “00”. Чтобы внедрить водяной знак 11, мы перезаписываем функцию в виде

```
int f (int x)
{
int b, a;
b = x – 1;
a = x + 1;
return a + 2 * b;
}
```

Вычислительная часть функции на языке Ассемблера ARM:

```
ldr     r3,     [fp, #-16]
sub     r3,     r3,     #1
str     r3,     [fp, #-20]
ldr     r3,     [fp, #-16]
add     r3,     r3,     #1
str     r3,     [fp, #-24]
ldr     r3,     [fp, #-20]
mov     r2,     r3,     asl     #1
add     r3,     r2,     r3
mov     r0,     r3
```

Распределение памяти для переменных соответствует порядку их объявления:

x = [fp, #-16], a = [fp, #-24], b = [fp, #-20].

Рассмотрим устойчивость к ошибкам данного метода. Предположим, что мы имеем две исполняемых программы или два объектных модуля, которые содержат нашу функцию с водяными знаками 00 и 11 соответственно. Сравнивая два объектных файла с помощью команды unix shell “cmp –l”, мы обнаружим 4 различия (первый столбец показывает адрес, а другие два столбца – различные байты сравниваемых файлов):

```
79      203     103
91      103     203
97      30      24
105     24      30
```

(те же самые различия, но в других адресах были бы найдены в исполняемых программах). Первые числа (203 и 103)– это различия кодов команд add и sub. Следующие различающиеся байты (30 и 24) возникают вследствие различного распределения памяти для переменных a и b. Мы видим, что нельзя вмешиваться в этот водяной знак, то есть изменять его на 01 или 10, изменяя лишь эти известные различающиеся байты. Действительно, если мы меняем местами 203 и 103, чтобы изменить водяной знак, функция будет повреждена, потому что это добавит 1 к b и вычтет 1 от a,что даст неправильный результат. Та же самая ошибка будет возникать, если мы меняем 30 и 24 (результат будет b + 2*a). Итак, мы можем подвести итог, утверждая, что вообще невозможно вмешаться в водяной знак в двоичном исполняемом файле простой заменой различающихся байт, где обнаружены различия. Эти различия могут только помогать обнаруживать местоположение водяного знака. Конечно, это возможно, только когда пакеты программ содержат различные водяные знаки (отпечатки пальца) и злонамеренные пользователи создают коалиции.

Более серьезное нападение может быть осуществлено, если мы дизассемблируем исполняемую программу. Хотя дизассемблирование большой исполняемой программы является достаточно трудной задачей, все же имеется ряд дополнительных трудностей, связанных с внедрением в исходный код C / C ++. Рассмотрим код нашей функции. Во-первых, мы можем видеть, что нет никаких последовательных и независимых команд, которые могут быть переставлены для того, чтобы модифицировать водяной знак. Это получается потому, что независимые операторы C / C ++ не компилируются в отдельные независимые команды, а скорее в независимые группы команд. Так что мы нуждаемся в дополнительном уровне декомпиляции для определения групп команд, соответствующих выше упомянутым операторам языка.

Что касается назначения переменных, то на первый взгляд это кажется простой задачей – взаимно заменить смещения (-20 и -24 в нашем примере), чтобы вмешаться в некоторые биты водяного знака. Но иногда (не в нашем примере) необходим адрес переменной. В этом случае смещение будет использоваться как обычная константа и может потребоваться более детальный анализ кода, чтобы отличить это смещение от некоторой другой константы.

Фактически, чтобы осуществить эффективную атаку, мы нуждаемся в своего роде декомпиляторе, нежели чем в дизассемблере. Вполне очевидно, что задача осуществления декомпиляции с целью последующего вмешательства в водяной знак является лучшим способом нападения, который можно себе представить: если иметь исходные тексты, то можно внедрять любой водяной знак, используя ту же самую внедряющую программу.

Описание процесса внедрения. Когда мы говорим об эффективности внедрения, мы учитываем число битов, которые могут быть внедрены в исходные файлы. Исходя из особенностей описываемого метода емкость определяется двумя вещами: сколько переменных объявлены в каждой функции и сколько групп независимых инструкций мы можем найти. Однако существуют ограничения, уменьшающие объем внедряемой информации. Ниже приводятся наиболее существенные из них.

1) Допускается перестановка переменных, объявленных только в начале блока. Это согласуется с общепринятой практикой программирования на C++: прежде чем использовать переменные, их необходимо объявить. Хотя некоторые переменные могут быть объявлены по ходу выполнения программы, их позиции в исходном тексте нельзя менять, так как это нарушит их область видимости и может вести к различного рода ошибкам.

2) Мы не можем переставлять инструкции, которые выглядят независимыми, если они содержат любой вызов функции. Это делается потому, что мы не можем гарантировать, что функция не имеет никаких побочных эффектов. Если она их имеет, то замена позиции такой инструкции может привести к ошибкам.

3) Запрещается переставлять кажущиеся независимыми инструкции, если они содержат любую косвенную ссылку (указатель, массив). Проследить содержимое указателей по ее имени, чтобы гарантировать независимость инструкции, не представляется возможным. Например, операции

```
*p = 5;
*q = 4;
```

могут быть зависимы, когда указатели p и q ссылаются на одну и ту же область памяти.

Как показывают эксперименты, эти ограничения очень сильно уменьшают объем внедряемой информации. Здесь уместно разделять два вида емкости: фактическая емкость, которая была достигнута с учетом всех ограничений, и теоретическая, которая может быть достигнута, если программист будет сам непосредственно указывать независимые переменные (в таком случае возможно отказаться от вышеописанных ограничений). Для практической проверки внедрения информации были использованы 33 программы на C / C ++ для платформы Symbian. Проекты с исходными текстами являются общедоступными и были взяты из сети Internet. В табл. 1 приводятся результаты провёденного эксперимента для некоторых проектов.

Таблица 1

Результаты внедрения

Имя	Число файлов c/c++	Размер файлов c/c++	Длина (фактическая) водяного знака	Длина (теоретическая) водяного знака
HView v1 13beta source	10	58K	27	32
SymTorrent 1.30 source	2007	104680K	40	131
DosBox	107	2066K	123	535
putty src 1.5.1	227	3650K	1018	1360
Vim	395	3980K	207	261
OpenVideoHub	405	6853K	2205	2635

Остальная часть проектов была малопригодна для встраивания, позволяя внедрить всего от 0 до 10 битов. Анализ причины такого низкого объема внедрения показывает, что, в дополнение к вышеописанным ограничениям, широко распространенное использование объектно-ориентированной технологии программирования не подходит к нашему методу сокрытия данных. Почти любая операция применяет методы класса, и трудно определить, безопасно ли изменять порядок операций из-за возможных побочных эффектов на свойства класса. Для объектно-ориентированных программ следует разрабатывать новые идеи.

Проблемы и будущее исследование. Основная проблема, которую мы пока не обсуждали – это оптимизация кода, осуществляемая компилятором. Любая модификация исходного текста может “пострадать” от оптимизации. Методы, описанные выше, работают только, если оптимизация выключена. Рассмотрим два варианта функции, соответствующие водяным знакам 00 и 11, откомпилированные GCC в следующие инструкции:

```
add     r3,     r0,     #1
sub     r0,     r0,     #1
add     r3,     r3,     r0,     asl #1
mov     r0,     r3
```

и

```
sub     r3,     r0,     #1
add     r0,     r0,     #1
add     r0,     r0,     r3,     asl #1
```

соответственно. Различия здесь только в порядке инструкций и не зависят от порядка объявлений переменных (GCC всегда распределяет первый свободный регистр (r3) для переменной, которая используется первой). Положительный эффект оптимизации для нанесения водяных знаков заключается в переупорядочивании инструкций, что ведет не только к изменению размещения текста программы, но также и к изменениям в длине кода. Как следствие, весь остальной код будет смещен в памяти и, благодаря перекрестным ссылкам на другие части кода программы, создаст достаточно много различий по всему файлу. Данная особенность помогает эффективно скрывать местоположение водяного знака. Для разрешения ситуации с оптимизацией исходные файлы, которые содержат “критичные” к скорости исполнения части программы, могут быть удалены из списка файлов, в которые предполагается внедрять водяной знак. Тогда такие “критичные” файлы могут быть отдельно откомпилированы с оптимизацией. Более перспективный подход предполагает полностью перепрограммировать генератор объектного кода

компилятора, преобразовав его в “stego генератор объектного кода”. Фактически, обычный компилятор однозначно выбирает только один путь генерации объектного кода среди множества эквивалентных (или почти эквивалентных) наборов инструкций. Компилятор со stego генератором объектного кода мог бы выбирать один из многих наборов инструкций, исходя из значения водяного знака, который будет внедрен. Вероятно, это – лучший способ осуществления внедрения водяного знака.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Davidson R.L., Myhrvold N.* Method and system for generating and auditing a signature for a computer program // US Patent 5559884. – 1996.
2. *Collberg C., Thomborson C.* Software watermarking: Models and dynamic embeddings // Annual Symposium «Principles of Programming Languages». – San Antonio: ACM Press, 1999. – P. 311-324.
3. *Stern J., Hachez G., Koeune F., Quisquater J.* Robust object watermarking: Application to code // III International Workshop «Information Hiding 1999». Vol. 1768. – Berlin: Springer, 1999. – P. 368-378.
4. *Venkatesan R., Vazirani V., Sinha S.* A graph theoretic approach to software watermarking // IV International Workshop «Information Hiding 2001». V. 2137. – Pittsburgh: Springer, 2001. – P. 157-168.
5. *Collberg C., Thomborson C., Townsend G.* Dynamic graph-based software watermarking // Technical report, Dept. of Computer Science. – Univ. of Arizona. – 2004.
6. *Curran D., Cinneide M.O., Hurley N., Silvestre G.* Dependency in software watermarking // I International Conference «Information and Communication Technologies: from Theory to Applications». – Damascus, 2004. – P. 569-570.
7. *Sahoo T.R., Collberg C.* Software watermarking in the frequency domain: Implementation, analysis, and attacks // Technical report, Dept. of Computer Science. – Univ. of Arizona. – 2004.
8. *El-Khalil R., Keromytis A.* Hydan: hiding information in program binaries // VI International Conference «Information and Communications Security». V. 3269. – Berlin: Springer, 2004. – P. 187-199.
9. *Anckaert B., De Sutter B., Chanet D., De Bosschere K.* Steganography for executables and code transformation signatures // VII International Conference «Information Security and Cryptology».V. 3506. – Seoul: Springer, 2005. – P. 431-445.
10. *Nechta I., Ryabko B., Fionov A.* Stealthy steganographic methods for executable files // XII International Symposium «Problems of Redundancy». – St.-Petersburg, 2009 . – P. 191-195.

Нечта Иван Васильевич
Сибирский государственный университет телекоммуникации и информатики.
E-mail: www@inbox.ru.
630102, г. Новосибирск, ул. Кирова, 86.
Тел.: 889231138992.

Фионов Андрей Николаевич
E-mail: a.fionov@ieee.org.
Тел.: 83832698272.

Nechta Ivan Vasil’evich
Siberian State University of Telecommunications and Informatics.
E-mail: www@inbox.ru.
86, Kirova street, Novosibirsk, 630102, Russia.
Phone: +789231138992.

Fionov Andrej Nikolaevich
E-mail: a.fionov@ieee.org.
Phone: +73832698272.