УДК 003.26

Полухрупкие цифровые водяные знаки, базирующиеся на спектре Фурье

И. В. Нечта

Для обнаружения несанкционированных изменений исполняемых файлов вредоносными программами применяются цифровые водяные знаки (ЦВЗ). Свойство хрупкости водяного знака обеспечивает его искажение при незначительных изменениях в программе. Наличие и корректность ЦВЗ контролируется конечным пользователем программы, который заинтересован в использовании целостного (немодифицированного) программного обеспечения. Полухрупкие ЦВЗ разрушаются при изменении программы сверх установленного порога. Согласно предлагаемому методу программа рассматривается как контейнер, содержащий цифровой сигнал, который может быть восстановлен по своим отдельным командам (отсчётам). Изменения в программе меняют спектр сигнала. При анализе модифицированной программы обнаруживаются некоторые искажения исходного сигнала – изменения фаз некоторых гармоник. Предложенный подход позволяет выявлять не только факт изменения программы, но и объём добавленных команд.

Ключевые слова: хрупкие цифровые водяные знаки, стеганография, спектр Фурье.

1. Введение

Цифровые водяные знаки (далее – ЦВЗ) представляют собой двоичную последовательность, которая встраивается в объект данных – так называемый контейнер. В качестве контейнера могут выступать файлы различного формата. ЦВЗ имеют широкую область применения и в зависимости от задачи могут использоваться как для подтверждения авторства или идентификации пользователя, так и для доказательства отсутствия изменения в контейнере.

В одних случаях, например, автор фотографии встраивает ЦВЗ, в котором содержатся его паспортные данные. При обнаружении такой фотографии у постороннего лица, которое выдает указанную фотографию за свою собственную, истинный автор в ходе судебного разбирательства может извлечь и продемонстрировать водяной знак. В других случаях необходимо выявить канал утечки лицензионного продукта. Так, в каждую создаваемую копию программы встраивается ЦВЗ, содержащий данные лицензиата. При обнаружении пиратской копии продукта водяной знак извлекается и определяется его принадлежность к лицензиату.

Еще одной областью применения ЦВЗ является контроль отсутствия изменений в цифровом объекте данных. Например, пользователь скачивает браузер с официального сайта. Согласно требованиям к информационной безопасности пользователь проверяет сертификат сайта и цифровую подпись или значение хэш-функции скачанного дистрибутива. Однако после установки программы возможна её вредоносная модификация компьютерными вирусами с целью кражи паролей. В таких случаях в программу встраивается ЦВЗ, который разрушается при модификации программы. Наличие и корректность водяного знака контролируется самой программой и пользователем.

Исходя из указанных областей применения выдвигаются требования к свойствам ЦВЗ. Свойство `устойчивости` гарантирует, что ЦВЗ не может быть искажен или удален злоумышленником. Данным свойством должен обладать водяной знак, хранящий сведения об авторе

или лицензиате. Свойство `хрупкости` гарантирует, что водяной знак будет разрушен при малейших изменениях в контейнере. Например, такое свойство необходимо для контроля целостности программы.

В ряде случаев программы могут санкционировано (по желанию автора) изменяться. Например, при написании программы часто возникают ошибки в коде. Для устранения ошибок автор распространяет специальные программы, исправляющие указанные ошибки. Для обозначения существенных по объему исправлений применяются термины `software update` и `service pack`. Для незначительных исправлений (порядка нескольких десятков байт) используется термин `patch`. В этой связи возникает свойство `полухрупкости`, которое гарантирует разрушение ЦВЗ, когда доля изменений превышает некоторый заданный порог.

В научной литературе известно множество методов доказательства целостности программ. Ряд методов, например [1–3], базируется на использовании аппаратных средств: Usb-токенах, смарт-картах и аппаратных модулях безопасности. Известны методы, представленные в работе [4], использующие самосканирование выполняемого кода и подсчет его контрольной суммы или хэш-функции. Наибольшей уязвимостью такой защиты обладают места хранения или сравнения подсчитанных хэш-значений с эталонным. В таких случаях применяют методы запутывания программного кода (`obfuscation`) [5–7], что крайне затрудняет анализ программы злоумышленником. В код программы добавляется множество избыточных инструкций, анализ которых может быть произведен только в ручном режиме. Существуют методы самомодификации кода [8, 9], которые направлены на затруднение выявления места, в котором хранится код, хранящий или проверяющий ЦВЗ.

В работах [10, 11] предлагается интеграция проверки ЦВЗ с полезным кодом программы при помощи нейросети. Утверждается, что анализ работы программы эквивалентен извлечению правил работы нейросети, что является NP-трудной задачей. Частичное искажение такого водяного знака сделает программу неработоспособной. Существуют методы, например [12, 13], которые держат под контролем только критически важные участки кода и используют проверку графа переходов между участками кода программы.

Все вышеприведенные методы позволяют обнаружить малое изменение в контролируемом участке кода. Применяемые функции (хэш или контрольная сумма) меняются даже при изменении одной команды программы. Целью данной работы является создание метода контроля целостности программы, который способен срабатывать при заданном пороге изменений. Такой метод может быть использован автором, когда учитывается возможность последующего незначительного изменения программы с помощью патчей.

2. Описание предлагаемого метода

2.1. Извлечение сигнала из программы

В настоящей работе предлагается рассматривать программу, которая является последовательностью инструкций, в качестве формы записи некоторого сигнала. В связи с тем, что малое изменение кода программы может повлечь за собой существенное изменение в её поведении, предполагается использовать трассировщик[1] [14] для получения списка выполненных команд программы. Последовательность команд, полученная трассировщиком, полнее характеризует поведение программы, следовательно, ЦВЗ будет более чувствительным к искажениям.

Согласно предлагаемой идее список команд программы, полученных трассировщиком, есть форма записи значений сигнала в некоторые моменты времени (отсчеты). Считается, что сигнал 2π-периодический на интервале $[-\pi; \pi]$ и является кусочно-непрерывным и кусочно-монотонным. Каждая команда программы соответствует очередному отсчету сигнала. В силу теоремы Дирихле такой сигнал может быть представлен в виде ряда Фурье.

[1] Трассировщик – это специальная программа, которая создаёт список команд, выполненных другой программой.

$$S(t) = \frac{a_0}{2} + \sum_{n=0}^{\infty} \mathrm{A_n} \sin(\mathrm{nx} + \varphi), \quad (1)$$

где $a_0 = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)dx$, A_n– амплитуда гармоники, φ – фазовый сдвиг гармоники.

Разложив сигнал программы в ряд Фурье, мы можем сравнить его с эталонным сигналом, не имеющим искажений. При изменении программы спектр также изменится, что будет обнаружено при сравнении. Спектр сигнала меняется пропорционально внесённым искажениям. Чем больше новых команд в программе, тем сильнее отличается спектр сигнала от эталона. Если изменения окажутся больше, чем предельно установленный порог, то водяной знак считается разрушенным и делается вывод о факте несанкционированного искажения программы.

Работа с предлагаемым водяным знаком строится следующим образом. После создания программы автор при помощи трассировщика получает список команд, выполненных в определенный момент времени работы программы. На основе команд получают сигнал и выполняют дискретное преобразование Фурье, результаты[2] которого затем хранятся либо отдельно, либо внутри файлов программы. На этапе эксплуатации по инициативе пользователя или в рамках периодического самоанализа программы производится аналогичное получение и разложение сигнала в ряд Фурье. По результатам сравнения мнимых частей спектра сигнала с эталонными значениями, сохраненными ранее, пользователю выводится уведомление о состоянии водяного знака (повреждён или не повреждён).

Для получения ряда Фурье необходимо определить функцию $f(x)$, задающую сигнал контейнера (программы). В нашем случае $f(x)$: $x \in N$, $f(x) \in R$, где x – шаг работы программы, в котором была выполнена одна команда[3]. В рамках настоящей работы рассматривается два варианта функции $f(x)$. В связи с тем, что существует множество различных команд, которые могут влиять на сигнал взаимно компенсирующим образом, предлагается выбрать одну ключевую команду Cmd_{key}, которая будет влиять на сигнал. Остальные команды не будут учитываться.

Рассмотрим первый вариант функции $f_1(x)$, который вычисляется по алгоритму F1.

Алгоритм F1.

вход (x, cmd_x, Cmd_{key}) // x – номер команды в списке, полученном трассировщиком.
// cmd_x – выполненная на шаге x команда.
выход $(Value_{sig})$ // $Value_{sig}$ – значение сигнала в отсчете x.
начало
если $x < 1$ **то** $Value_{sig} = 0$ **возврат** $Value_{sig}$ **конец если**
если $cmd_x = Cmd_{key}$
то $Value_{sig} = 1$
иначе $Value_{sig} = max(F1(x-1, cmd_{x-1}, Cmd_{key}) - 0.1, 0)$
конец если
возврат $Value_{sig}$
конец

Рассмотрим второй вариант $f_2(x)$, который вычисляется по алгоритму F2.

Алгоритм F2.

вход (x, cmd_x, Cmd_{key}) // x – номер команды в списке, полученном трассировщиком,
// cmd_x – выполненная на шаге x команда.
выход $(Value_{sig})$ // $Value_{sig}$ – значение сигнала в отсчете x.
начало
если $x < 1$ **то** $Value_{sig} = 0$ **возврат** $Value_{sig}$ **конец если**
если $cmd_x = Cmd_{key}$
то $Value_{sig} = 1$

[2] В настоящей работе используются только мнимые части спектра сигнала.
[3] Согласно алгоритму используются команды без учёта их операндов (параметров).

иначе $Value_{sig} = 0$
конец если
возврат $Value_{sig}$
конец

Для большей наглядности представим возможный вид сигнала некоторой программы на рис. 1 и 2. Как правило, ключевые команды отстоят друг от друга на значительном расстоянии (более 10 команд).

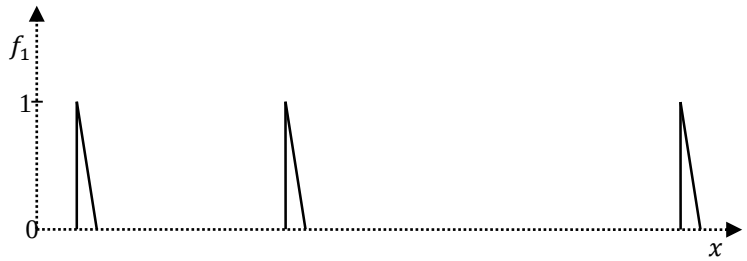

Рис. 1. Сигнал, хранимый в программе, при $f = f_1$

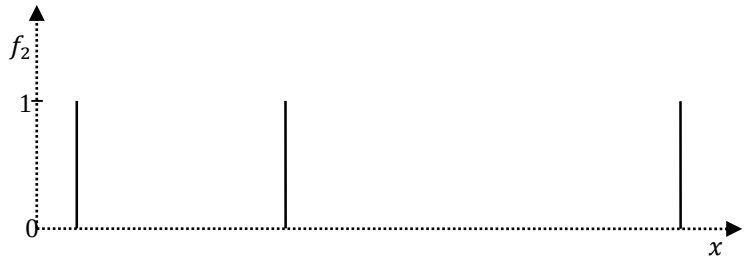

Рис. 2. Сигнал, хранимый в программе, при $f = f_2$

2.2. Сравнение спектров сигналов

Одним из важных моментов работы предлагаемого метода является сравнение спектров сигналов до и после изменения программы. Полученный спектр представляет собой множество действительных и мнимых частей сигнала. В силу теоремы Бенедикса [15] сигнал может быть конечен либо только по времени, либо только по спектру, соответственно, мы получим бесконечный спектр. Следовательно, для сравнения спектров сигналов нам необходимо избавиться от части членов ряда Фурье, получая при этом некоторое (желательно несущественное) искажение анализируемых сигналов.

Согласно предлагаемому алгоритму оставляются низкочастотные гармоники сигнала по следующим причинам:

1. Как и в случае с изображением формата jpg, высокочастотные гармоники отвечают за «мелкие детали» сигнала, низкочастотные – за «грубые». Так как малые (легальные) изменения в программе допускаются при помощи патчей, то высокочастотные гармоники сигнала будут меняться, следовательно, являются бесполезными для нашего алгоритма.

2. Так как алгоритм может быть применён к коротким фрагментам программы, то число команд и, следовательно, отсчётов сигнала может быть невелико. Тогда в силу теоремы Котельникова $F_{\text{отсч}} \geq 2 * F_{max}$ мы не сможем восстановить высокочастотные компоненты сигнала.

В результате для сравнения спектров используются мнимые части первых 20 гармоник сигнала $Im = A_i\ sin(\varphi)$. Более того, из оставшихся низкочастотные гармоники считаются более значимыми, чем высокочастотные, поэтому используются весовые коэффициенты, взятые из распределения Ципфа:

$$Zipf(i) = \frac{1}{i} * \frac{1}{\sum_1^{20}\left[\frac{1}{i}\right]}. \quad (2)$$

Сравнение спектров сигналов осуществляется с помощью меры $Comp_{spec}(IM_1, IM_2)$, где IM – множество мнимых частей сигнала.

$$Comp_{spec}(IM_1, IM_2) = \sum_{i=1}^{19} Zipf(i) * \big||im_{1i} - im_{1i+1}| - |im_{2i} - im_{2i+1}|\big|. \quad (3)$$

Очевидно, что если мера равна нулю, то это означает отсутствие изменений в сигнале, следовательно, целостность программы не нарушена.

3. Экспериментальный анализ эффективности

Для проведения экспериментальной оценки эффективности предлагаемого метода были отобраны 30 файлов из операционной системы Windows 7 (консольные утилиты). Программы запускались в режиме трассировки и их выполненные команды записывались в trace-файлы. Из trace-файлов извлекался эталонный спектр, затем в них добавлялись случайные команды или удалялись блоки команд, имитирующие несанкционированное искажение. Имитировались следующие ситуации:

- добавление команд в начало trace-файла, имитируя работу вредоносных программ, меняющих стартовую точку входа в программу, например, вирус Virus.Win9x.CIH;
- добавление команд в случайную позицию trace-файла, имитируя искажение текущего функционала, например, проверку лицензионного ключа;
- добавление команд в конец trace-файла, имитируя вредоносные программы, дополняющие функционал программы;
- удаление блока команд из случайной позиции trace-файла, имитируя вредоносные программы, частично отключающие функционал программы.

В ходе анализа было установлено, что выбор ключевой команды из 10 самых часто встречающихся не существенно влияет на итоговый результат сравнения. С одной стороны, редко встречающаяся команда может быть неравномерно распределена по коду, что, как показано в эксперименте, создает угрозу модификации программы без искажения водяного знака. С другой стороны, если команда встречается слишком часто, то удельный вклад одного её вхождения уменьшается, что также создает возможность несанкционированной модификации кода без разрушения ЦВЗ. Среднестатистическое распределение команд в файле, рассчитанное для 30 файлов, показано в табл. 1.

Таблица 1. Распределение команд в программе

Команда	Вероятн. %	Команда	Вероятн. %	Команда	Вероятн. %	Команда	Вероятн. %
mov	18.64	pop	6.96	jb	2.20	jmp	1.11
cmp	12.76	test	4.32	lea	2.03	add	1.09
push	12.29	jz	3.05	dec	1.59	sub	0.43
inc	11.88	call	2.87	xor	1.46	jnb	0.42
jnz	10.59	movzx	2.52	retn	1.24	div	0.38

Результаты сравнения эталонных и изменённых файлов при последовательном добавлении команд для команды `pop` (извлечение данных из стека) показаны в табл. 2 и 3.

Таблица 2. Средние значения меры $Comp_{spec}$ при различном числе добавленных команд для $f = f_1$

Кол-во добавленных команд	Значение $Comp_{spec}$	
	Добавление в начало программы	
	ср. знач.	дисперсия
0	0.000	0.000
1	45.831	590.785
2	45.852	590.067
3	45.874	589.514
4	45.859	588.930
5	45.916	588.321
6	45.936	587.691
7	45.956	587.038
8	45.978	586.266
9	46.001	585.485
10	46.025	584.888

Таблица 3. Средние значения меры $Comp_{spec}$ при различном числе добавленных команд для $f = f_2$

Кол-во добавленных команд	Значение $Comp_{spec}$							
	Добавление в начало программы		Добавление в случайное место программы		Добавление в конец программы		Удаление из случайного места программы	
	ср. знач.	дисперсия	ср. знач.	дисперсия	ср. знач.	дисперсия	ср. знач.	дисперсия
0	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
1	0.069	0.029	0.045	0.002	0.000	0.000	29.699	622.835
2	0.138	0.012	0.090	0.005	0.000	0.000	29.701	623.875
3	0.207	0.026	0.141	0.016	0.000	0.000	29.641	622.465
4	0.275	0.047	0.184	0.024	0.000	0.000	29.639	623.332
5	0.344	0.073	0.229	0.037	0.000	0.000	29.657	624.086
6	0.412	0.105	0.272	0.055	0.000	0.000	29.651	624.776
7	0.479	0.143	0.319	0.078	0.000	0.000	29.663	626.462
8	0.547	0.186	0.354	0.090	0.000	0.000	29.550	623.013
9	0.615	0.235	0.397	0.124	0.000	0.000	29.585	625.410
10	0.683	0.290	0.439	0.149	0.000	0.000	29.629	625.734

Из полученных результатов видно, что меньшую дисперсию даёт функция f_2, следовательно, она рекомендуется к использованию в данном методе для извлечения водяного знака. Вариант функции f_1 также может быть использован, но значения меры $Comp_{spec}$ сильно зависят от исходной программы. Тем не менее вариант f_2 можно считать более универсальным.

Нулевые значения $Comp_{spec}$ при добавлении в конец файла объясняются тем, что случайно добавленные команды были либо отличны от ключевой, либо воздействовали на высокочастотную часть спектра, которая не учитывается. Удаление даже незначительной доли команд существенно искажает спектр сигнала, что следует учитывать при санкционированных изменениях в программе.

На основании данных, полученных в результате эксперимента, можно утверждать следующее. Во-первых, значение $Comp_{spec}$ монотонно возрастает с увеличением добавляемых команд. Данная особенность позволяет определять объём новых команд, привнесённых в исходную программу. Следовательно, мы можем задавать порог допустимых изменений для случая санкционированной модификации программы авторскими патчами. Во-вторых, ЦВЗ защищает только тот участок программы, который находится перед последним вхождением ключевой команды. Следовательно, для эффективной защиты целесообразно использовать несколько водяных знаков, соответствующих исходному и обратному порядку команд программы.

4. Практическая реализация

В данной главе будут описаны ключевые моменты программной реализации предлагаемого метода. Будем рассматривать программу как совокупность непересекающихся блоков кода $B_{all} = \{b_1, \dots, b_n\}$[4]. Пусть имеется фрагмент программы $B_{prot} = \{b_i, b_{i+1}, b_{i+2}, \dots, b_{i+k}\}$, защищаемый ЦВЗ, такой что $B_{prot} \subseteq B_{all}$. Указанный фрагмент запускается, когда на вход программы подаются специальные параметры. Предлагаемый метод будет анализировать ЦВЗ, полученный только от фрагмента B_{prot}. Остальные фрагменты считаются малозначимыми и их модификация не влияет на целостность программы. Автор программы самостоятельно определяет специальные входные параметры и фрагмент B_{prot}.

Описываемая ниже реализация аналогична классической проверке контрольной суммы файла с помощью хэш-функций. Отличие настоящей реализации заключается в используемой функции и способе получения её аргументов. Предполагается, что процесс проверки ЦВЗ производится отдельной программой (модулем) и выполняется без участия пользователя.

Рассмотрим простейший вариант реализации. Для проверки целостности фрагмента B_{prot} последовательно выполняются шаги: получение списка команд фрагмента B_{prot}, получение спектра, сравнение спектра с эталонным значением. Для получения списка команд программа, содержащая ЦВЗ, запускается в режиме отладки с помощью функции `CreateProcess` с флагом `DEBUG_ONLY_THIS_PROCESS`. В память по адресу начала выполняемого кода (`Original Entry Point`) записывается команда прерывания для отладки (`int 3`), имеющаяя опкод 0xCC, как показано далее в примере.

```
ReadProcessMemory(pi.hProcess, (void*)pStartAddress,
&OldInstruction, 1, &dwReadBytes);
NewInstruction= 0xCC;
WriteProcessMemory(pi.hProcess, (void*)pStartAddress,
&NewInstruction, 1, &dwWriteBytes);
FlushInstructionCache(pi.hProcess,(void*)pStartAddress,1);
```

Данная команда прерывания позволит перехватить управление анализатору спектра после того как анализируемая программа загрузится операционной системой в память и статические библиотеки выполнят инициализацию функциями `DllMain`, но перед началом выполнения B_{prot}. Как только произойдет перехват, необходимо восстановить оригинальную перезаписанную команду (в примере хранящуюся в `OldInstruction`) и уменьшить регистр EIP на единицу (т.е. вернуться на начало выполняемого кода). Далее меняем регистр флагов для установления режима пошагового выполнения инструкций и передаем управление программе.

```
CONTEXT lcContext;
lcContext.ContextFlags = CONTEXT_FULL;
GetThreadContext(pi.hThread, &lcContext);
```

[4] Такое утверждение правомерно, так как применяется принцип модульного программирования.

```
   lcContext.EFlags |= 0x100;
   lcContext.Eip --;
   SetThreadContext(pi.hThread, &lcContext);
   ContinueDebugEvent(debug_event.dwProcessId,    debug_event.dwThreadId,
dwContinueStatus);
```

На каждом шаге программы считываем байты выполняемой команды, адрес начала которой находится в регистре `eip` и декодируем команду согласно таблице [16]. Таким образом, мы получим список команд защищаемого фрагмента B_{prot}. Подробнее работа трассировщика описана в [17].

Последующие процессы получения спектра сигнала и сравнения его с эталоном были описаны во второй главе. Как и в случае с проверкой контрольной хэш-суммы, считается, что анализирующая программа и места хранения эталонного значения спектра не подвергаются атаке злоумышленником. Допускается, что анализ целостности ЦВЗ может выполняться по расписанию. При создании более сложных схем, например, когда проверка осуществляется в самом защищаемом модуле (программе), необходимо реализовывать самоанализ вне защищаемого фрагмента кода. Другими словами, если фрагмент программы b_j выполняет анализ ЦВЗ, то необходимо выполнение условия: $b_j \notin B_{prot}$.

Литература

1. *Dyer J. G., et al.* Building the IBM 4758 secure coprocessor // IEEE Computer. 2001. V. 34, № 10. P. 57–66.
2. *Smith S. W., Weingart S.* Building a high-performance, programmable secure coprocessor // Computer Networks. 1999. V. 31, № 8. P. 831–860.
3. *Rajan H., Hosamani M.* Tisa: Toward trustworthy services in a service-oriented architecture // IEEE Transactions on Services Computing. 2008. V. 1, № 4. P. 201–213.
4. *Qiu J., et al.* Identifying and Understanding Self-Checksumming // Defenses in Software. 2015. P. 207–218.
5. *Junod P., et al.* Obfuscator-LLVM software protection for the masses // 2015 IEEE/ACM 1st International Workshop on Software Protection. 2015. P. 3–9.
6. *Gautam P., Saini H.* A Novel Software Protection Approach for Code Obfuscation to Enhance Software Security // International Journal of Mobile Computing and Multimedia Communications (IJMCMC). 2017. V. 8, № 1. С. 34–47.
7. *Schrittwieser S. et al.* Protecting software through obfuscation: Can it keep pace with progress in code analysis? // ACM Computing Surveys (CSUR). 2016. V. 49, № 1. P. 4.
8. Официальный сайт программы UPX [Электронный ресурс]. URL: `https://upx.github.io/` (дата обращения: 24.03.2019).
9. *Touili T., Ye X.* Reachability Analysis of Self Modifying Code // 2017 22nd International Conference on Engineering of Complex Computer Systems (ICECCS). IEEE. 2017. P. 120–127.
10. *Chen Z., Wang Z., Jia C.* Semantic-integrated software watermarking with tamper-proofing // Multimedia Tools and Applications. 2018. V. 77, № 9. P. 11159–11178.
11. *Balachandran V., Emmanuel S.* Potent and Stealthy Control Flow Obfuscation by Stack Based Self-Modifying Code // IEEE Transactions on Information Forensics and Security. 2013. V. 8, № 4. P. 669–681.
12. *Chen Y., et al.* Oblivious hashing: A stealthy software integrity verification primitive // International Workshop on Information Hiding. Springer. 2002. P. 400–414.
13. *Jacob M., Jakubowski M. H., Venkatesan R.* Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings // Proceedings of the 9th Workshop on Multimedia & security. ACM. 2007. P. 129–140.

14. Официальный сайт программы IDA Pro Free [Электронный ресурс]. URL: https://www.hex-rays.com/products/ida/index.shtml (дата обращения: 24.03.2019).
15. *Benedicks M.* On Fourier transforms of functions supported on a set of finite Lebesgue measure // Journal of Mathematical Analysis and Applications. V. 106. 1985. P. 180–183.
16. X86 Opcode and Instruction Reference [Электронный ресурс]. URL: http://ref.x86asm.net/coder32.html(дата обращения: 06.05.2019).
17. *Vijayvargiya A.* Writing Windows Debugger – Part 2 [Электронный ресурс]. URL: https://www.codeproject.com/Articles/132742/Writing-Windows-Debugger-Part-3 (дата обращения: 06.05.2019).

Статья поступила в редакцию 22.05.2019.

Нечта Иван Васильевич
к.т.н., доцент, доцент кафедры прикладной математики и кибернетики СибГУТИ (630102, Новосибирск, ул. Кирова, 86), тел. (383) 2-698-359, e-mail: ivannechta@gmail.com.

Semi-fragile digital watermarks based on Fourier spectrum

I. V. Nechta

Digital watermarks are used for detection unauthorized edition of executable files by malicious programs. A property called “fragile” provides its distortion after minor changes in a program. The watermark presence and correctness controlled by endpoint user, which is obviously interested in software integrity. Semi-fragile DWM distorts when changing the program more than the set threshold. According to proposed method a program is considered as a container with digital signal that can be recovered from its samples (program commands). Any changes in the program code inevitably change the signal spectrum. During the analysis of modified program it was founded some distortion of the original signal: changing phases and amplitudes of some harmonics. The proposed approach allows to reveal not only the fact of modification, but also an amount of added code.

Keywords: fragile digital watermarks, steganography, Fourier spectrum.