DOI: 10.55648/1998-6920-2023-17-3-3-11 УДК 004.056.53

Применение многопоточности для защиты программ*

И. В. Нечта

Сибирский гос. унив. телекоммуникаций и информатики (СибГУТИ)

Аннотация: В данной статье представлен метод идентификации копии программы при помощи цифрового водяного знака (ЦВЗ). Известные ранее аналоги использовали изменения графа потока управления программы, и чтение осуществлялось через сложный анализ трассировки программы. Считается, что применение протектора или пакера может сделать невозможным считывание ЦВЗ. В настоящей работе предлагается использовать многопоточное приложение, генерирующее числа с нормальным законом распределения. Внедрение ЦВЗ осуществляется путём изменения дисперсии распределений. Такие случайные величины могут быть найдены по поисковой сигнатуре в памяти процесса и использоваться как внутри самой программы, так и сторонней при чтении ЦВЗ. Проведен экспериментальный подбор оптимальных параметров алгоритма. Оценены накладные расходы, связанные с созданием большого числа потоков.

Ключевые слова: стеганография, цифровые водяные знаки, многопоточность, защита программ.

Для цитирования: Нечта И. В. Применение многопоточности для защиты программ // Вестник СибГУТИ. 2023. Т. 17, № 3. С. 3–11. https://doi.org/10.55648/1998-6920-2023-17-3-3-11.

Статья поступила в редакцию 12.05.2023;
принята к публикации 20.05.2023.

1. Введение

Разработчики программного обеспечения в своей работе сталкиваются с различными рисками. Так, однажды разработанная одним автором библиотека может быть скопирована и использована другим автором без юридических на то оснований. Подобные действия нарушают лицензионное соглашение, в котором автор программы прописывает правила использования его продукта.

Наиболее распространенными рисками являются: использование компонентов программы в другом проекте или дизассемблирование исполняемых модулей с целью раскрытия алгоритмов их работы. Также частым нарушением со стороны злоумышленника является редактирование программ, например, для снятия проверки лицензионного ключа или снятия ограничений функциональности, имеющихся у незарегистрированного пользователя. Для программ, написанных для ОС Android, значимый риск несёт атака типа «repack» – переписывание программы [1]. Например, популярную программу с миллионным рейтингом скачивания можно подредактировать, добавив свою рекламу. В результате пользователь будет скачивать модифицированную версию под видом оригинала. Явной угрозы для пользователя это не несет, но оригинальный автор вряд ли будет получать комиссию от рекламы. Наиболее опасным риском

*Работа выполнена в рамках государственного задания № 071-03-2023-001 от 19.01.2023.

для пользователя является модификация известных программ, например, браузера с целью кражи паролей или иных конфиденциальных данных. В результате возникает острая необходимость принятия мер для противодействия злоумышленнику.

Для решения вышеописанных проблем применяются различные методы, такие как:

- Запутывание (obfuscation) программы [2]. Код программы превращают в трудночитаемый и избыточный, что в крайней степени осложняет его чтение и анализ.
- Построение защиты от изменения (tamper-proofing) [3]. Программа выполняет самосканирование и при обнаружении изменений перестает корректно работать.
- Построение цифрового водяного знака (watermarking) [4]. Специальное сообщение и алгоритмы его обработки, внедрённые в программу, для контроля целостности модулей или выявления факта их использования в составе стороннего проекта.

Рассмотрим известные схемы построения цифровых водяных знаков (ЦВЗ). Статические ЦВЗ представляют собой обычные строки, числовые константы или фрагменты кода программы, которые никогда не выполняются (т.н. «мертвый» код) [5]. Известны методы внедрения, использующие изменение кода программы на эквивалентные ему инструкции. Такой знак может содержать имя автора или название организации производителя и быть продемонстрирован. Таким образом, в ходе судебных разбирательств может быть доказано истинное авторство. Считается, что такой класс ЦВЗ является неустойчивым к удалению или искажению. Например, злоумышленник может применить упаковку программы в протектор [6] и тем самым затруднить считывание ЦВЗ.

В отличие от статических, динамические ЦВЗ представляют собой создаваемые и заполняемые в памяти процесса структуры данных. Например, на определенном этапе работы программы создается граф, который можно найти по сигнатуре, известной только истинному автору. Очевидно, что в данном случае применение протекторов не затруднит считывание ЦВЗ.

Последний класс методов активно развивается. Известны работы по созданию хрупких ЦВЗ [7], в которых водяной знак демонстрируется пользователю. При искажении программы также искажается ЦВЗ, о чем сразу же уведомляется пользователь. В работе [8] описан алгоритм полухрупкого ЦВЗ, который разрушается при превышении заданного порога изменений кода.

Известны методы, представляющие водяной знак в виде графа пути выполнения программы (Control Flow Graph, CFG). Чтение такого знака выполняется через пошаговое выполнение программы (трассировки) с фиксацией адресов перехода. Различные пути выполнения кода определяют различные данные в ЦВЗ. Так, в работе [9] предлагалось внедрять данные в форме конкурентного выполнения потоков программы. Фрагменты кода, где программа выполнялась в одном потоке, заменяют на многопоточные. Порядок взаимного выполнения и блокировок потоков определяет встроенное сообщение. Для чтения ЦВЗ программе на вход подается специальный набор параметров и анализируется трассировка.

Трассировка может быть искусственно усложнена с целью запутывания программы. Например, обычный переход (`goto`) может быть заменен на цикл с условным переходом. Два значения сравниваются, и по этим результатам выполняется либо переход, либо продолжается работа цикла. Условие можно подобрать так, что оно будет гарантированно выполнено. Но на этапе анализа программы злоумышленником работа такого условия будет неочевидной. Так, в работе [10] используется гипотеза Коллатца, которая производит ряд чисел (последовательность чисел-градин), гарантированно оканчивающихся на число 1. Значит, цикл будет выполняться и завершится с заранее задаваемыми значениями переменных. В результате можно строить произвольное количество запутывающих злоумышленника циклов и получать различные трассировки программы, которые и определяют ЦВЗ.

В настоящей работе предлагается метод построения цифрового водяного знака, который базируется на многопоточности. В программе часть потоков генерируют числа с нормальным законом распределения, а другая часть перехватывает эти числа и анализирует их дисперсию. Внедрение ЦВЗ осуществляется путём изменения дисперсий случайных величин. Сгенерированные числа могут быть как найдены по сигнатуре в памяти процесса и использованы внутри

программы, так и считаны сторонней. Преимущество данного подхода состоит в том, что не требуется трудоемкий анализ трассировки программы. Подобный ЦВЗ может быть реализован даже в программах, которые подвергнутся упаковке протектором.

2. Описание предлагаемого метода

Для того чтобы было невозможно прочитать ЦВЗ, злоумышленник может воспользоваться упаковщиком программ [11]. Оригинальная программа будет сжата и зашифрована. При запуске программы упаковщик поэтапно её расшифровывает и выполняет. В результате в трассировку оригинальной программы добавляются фрагменты выполнения упаковщика, что усложняет извлечение ЦВЗ, базирующегося на анализе графа потока управления (CFG). Такая же проблема возникает у программ, использующих статические ЦВЗ.

В настоящей статье предлагается использовать динамический ЦВЗ, заполнение которого идет в непрерывном режиме. На рис. 1. представлен упрощенный вариант предлагаемой схемы.

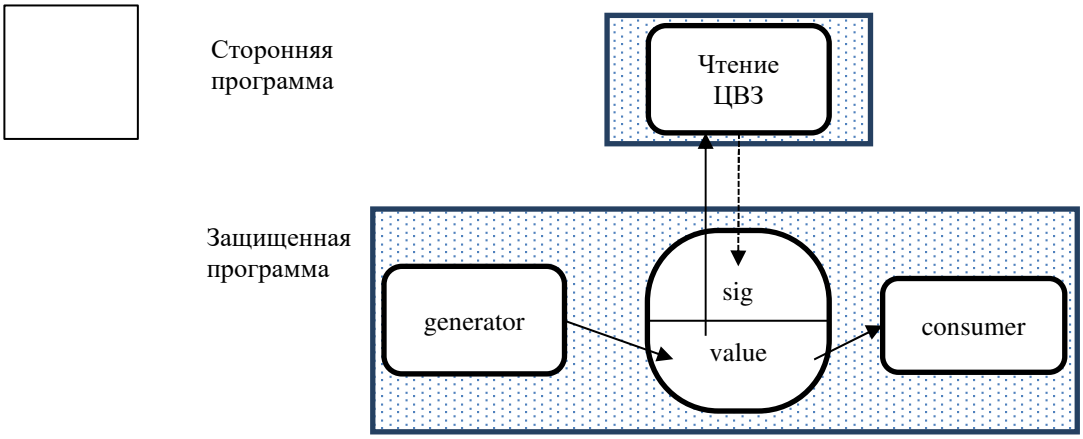

Рис. 1. Предлагаемая схема ЦВЗ

Рассмотрим схему более подробно. Имеется генератор чисел (`generator`), запущенный в отдельном потоке. Генератор непрерывно порождает число $x \in [-5;5]$ с нормальным законом распределения $N(0,D)$, где D – заданная дисперсия. Число записывается в структуру данных, состоящую из поисковой сигнатуры (`sig`) и непосредственно ячейки памяти (`value`) для хранения числа. Поток потребителя (`consumer`) непрерывно считывает число и записывает его в свой массив, формируя аналогичное генератору распределение. Здесь важно отметить, что при чтении и записи `value` нам не требуется использование механизмов синхронизации потоков.

Для интеграции ЦВЗ в исходный алгоритм программы полученное потребителем распределение сравнивается (измеряется расхождение) с другим фиксированным или также генерируемым распределением. Здесь для сравнения используется дивергенция Кульбака–Лейблера:

$$D_{KL}(P,Q) = \sum_{i=1}^{n} p_i \log_2 \frac{p_i}{q_i}, \quad (1)$$

где P и Q – вероятностные распределения размером n элементов.

Полученная дивергенция (число) может быть использовано в «полезной» части программы. Такая интеграция противодействует удалению или нарушению нормальной работы потоков. Если поток будет удален злоумышленником, то программа не сможет получить и использовать правильное значение дивергенции, что нарушит алгоритм её работы.

Предполагается, что в реальной программе количество потоков порядка 10–600. При реализации водяного знака вполне достаточно всего 10 чисел (т.е. потоков), например, чтобы записать серию и номер паспорта автора. Часть потоков может рассчитывать дивергенцию, но

далее её не использовать. Считается, что анализ многопоточного приложения сам по себе затруднен, к тому же интегрированные значения представлены не в явном виде, а продолжительно меняются во времени. Для снижения нагрузки на процессор любой поток должен иметь в своем коде функцию ожидания, например, равного 1 мс.

Пример потока на языке программирования C++:

```
void SomeThread(){ // генератор
    Generator gen(dispersion,mean);
    while(isProgrammWorking==true){
        gen.generate();
        usleep(1000);// 1 мсек
    }
}
```

Очевидно, что распределение вероятностей в самом начале запуска программы еще не обладает необходимыми статистическими свойствами. Время стабилизации значения дивергенции мы определим экспериментально в следующей главе. При практической реализации ЦВЗ потоки рекомендуется запускать в самом начале работы программы, однако использование полезного значения дивергенции нужно отложить до того момента, когда оно стабилизируется и не будет отличаться на значимую величину. Например, это значение может использоваться при запуске настроек приложения. Во-первых, пользователь потратит некоторое время на вход в нужный пункт меню, а, во-вторых, нагрузка на процессор в этот момент не будет заметна пользователю.

Для чтения ЦВЗ запускаются две программы: защищенная и сторонняя, которая анализирует содержимое оперативной памяти. Анализатор находит по сигнатуре (`sig`) расположение переменной (`value`), строит распределение, вычисляет дивергенцию и получает из каждого потока по одному внедренному числу. Преобразование дивергенции в дисперсию выполняется согласно табл. 1, а из дисперсии в число ЦВЗ – по табл. 2. Порядок чтения чисел определяется правильно подобранными сигнатурами, обсуждение которых выходит за рамки данной статьи.

3. Экспериментальный выбор параметров работы алгоритма

В предыдущей главе мы рассматривали общее описание метода без конкретизации параметров. В текущем разделе будет проведен экспериментальный подбор оптимальных параметров работы алгоритма и оценка накладных расходов. Для эксперимента было создано специальное программное обеспечение, которое имитировало работу потоков.

3.1. Оценка погрешности и скорости накопления статистики

Потоки генератора и потребителя запускаются и останавливаются независимо друг от друга. Последовательность их работы трудно прогнозируема и определяется операционной системой. В результате часть сгенерированных чисел не будет перезаписываться генератором, не успев обработаться потоком потребителя. Как следствие, возникает погрешность передачи распределения от генератора к потребителю. Учитывая, что дивергенция Кульбака–Лейблера не обладает симметрией, т.е. зависит от порядка аргументов, то здесь и далее первым аргументом будет передаваться эталонное распределение (генератора).

На рис. 2 представлен график зависимости дивергенций распределений от времени. Кривая с круглым маркером соответствует зависимым распределениям – генератор-потребитель. В идеальном случае обе кривые должны совпадать. Для сравнения приведена дивергенция между двумя произвольными независимыми нормальными распределениями $N(0,1)$.

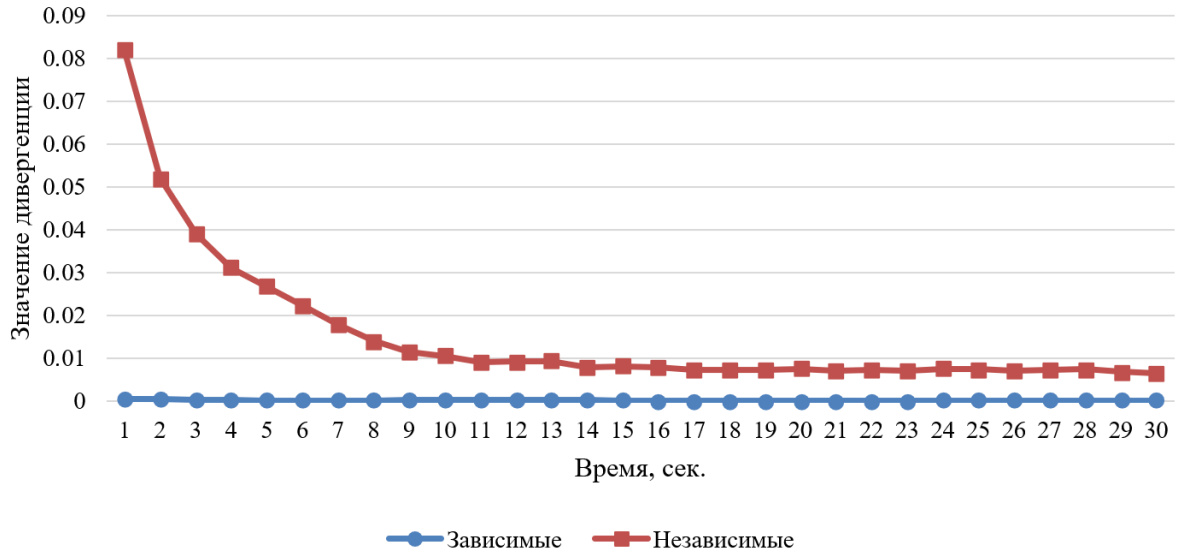

Рис. 2. Значение дивергенции у зависимых и независимых распределений

Из графика видно, что по мере накопления статистики кривые приближаются к своему пределу последовательности, равному 0.00011 и 0.0065 соответственно. Таким образом, передача распределения через переменную, характерная для нашей схемы, выполняется с высокой точностью. Учитывая, что дивергенция сходится к пределу постепенно, то следует создавать задержку между началом генерации и фактическим использованием значения в программе значения. Т.е. должно сгенерироваться достаточно чисел для получения адекватной статистики по распределению. В практической реализации мы можем запускать генерацию при старте всей программы, а использовать её, например, после захода в некоторый раздел настроек программы, что обеспечит требуемую задержку. Другим вариантом может быть использование таймера.

На рис. 3 показаны различия скоростей накопления статистики для потоков с разными задержками. Видно, что задержка практически не влияет на скорость накопления, что может объясняться работой планировщика, который во время остановки одних потоков обрабатывает другие.

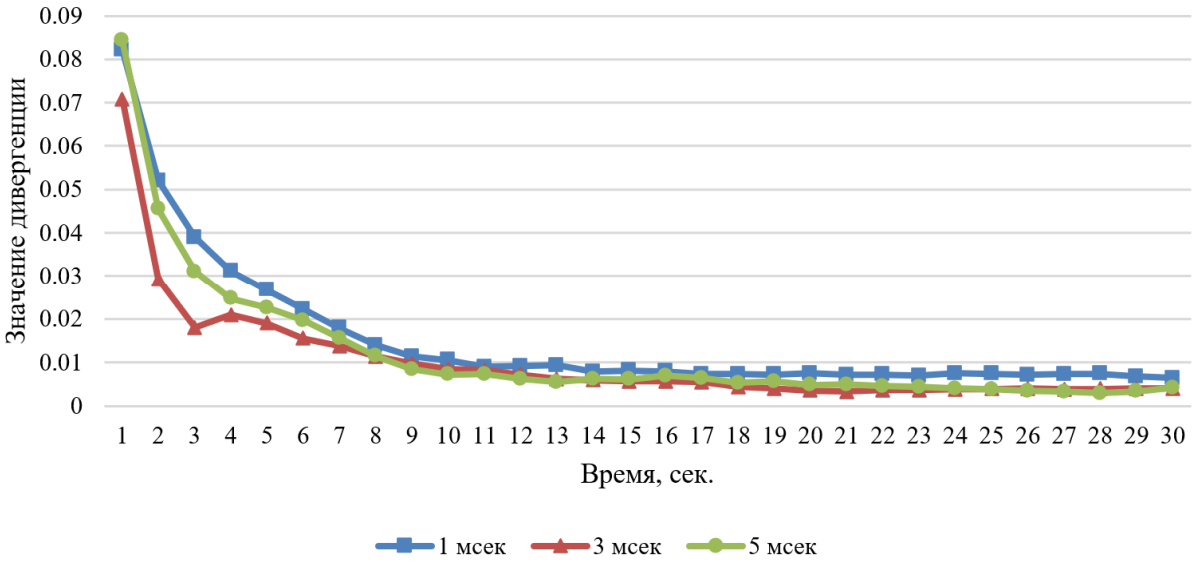

Рис. 3. Влияние времени задержки на скорость накопления статистики

3.2. Анализ различимости двух распределений

Для записи ЦВЗ в программу предполагается менять дисперсию случайной величины у генератора. Здесь возникает вопрос о разрешающей способности нашего метода: какое минимальное различие между дисперсиями должно быть? Для ответа на этот вопрос проведён следующий эксперимент. Было запущено три потока-генератора нормального распределения и один поток-потребитель. Обозначим их распределения как $Distr_1$, $Distr_3$, $Distr_4$ – для генераторов, а $Distr_2$ – для потребителя. Распределения 1 и 2 зависимы. Соответствующие им дисперсии: $D_1 = D_3 = 1$, $D_2 \approx 1$, а D_4 выбирается произвольно в диапазоне [1.05; 10]. Согласно схеме при работе программы будет проводиться расчёт дивергенции между $Distr_4$ и $Distr_2$, а при чтении ЦВЗ $Distr_3$ и $Distr_2$. Для демонстрации корректной работы нашей схемы нужно показать, что распределение $Distr_2$ значимо отличается от $Distr_4$ и не отличается от $Distr_3$. Это будет означать, что совпадающие по дисперсиям распределения не различаются значимо по дивергенции и наоборот. Значит, мы сможем достоверно различить дисперсии и успешно прочитать ЦВЗ.

Из графика видно, что со временем пределы последовательностей значимо различаются. Более полно пределы последовательностей представлены в табл. 1.

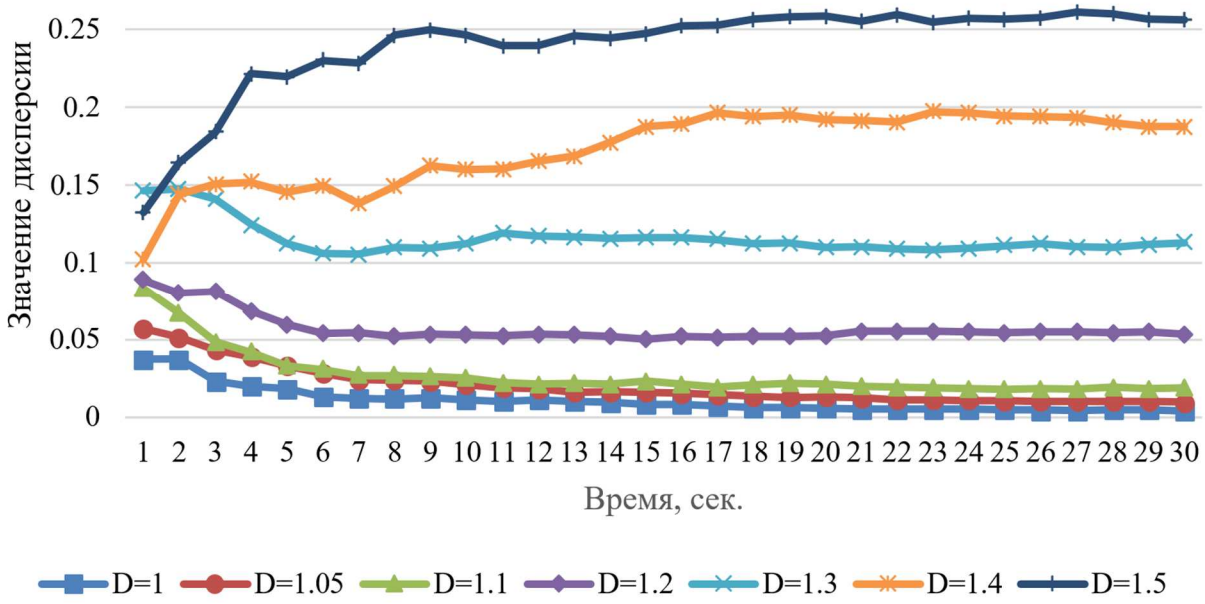

Рис. 4. Влияние дисперсии на различие распределений

Таблица 1. Соответствие между предельными значениями дивергенций и дисперсиями

Дисперсия	1.0	1.05	1.1	1.2	1.3	1.4	1.5	1.7	2.0
Дивергенция	0.0028	0.0077	0.0170	0.0520	0.1040	0.1800	0.2630	0.4580	0.7690
Дисперсия	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	–
Дивергенция	1.5412	2.0636	2.2370	2.3246	2.3292	2.5714	2.3922	2.4347	–

Из табл. 1 видно, что значения отстоят друг от друга неравномерно: первые 10 имеют большее удельное расстояние (на единицу дисперсии) между соседними распределениями, чем последние. Такая неравномерность связана с тем, что при реализации программы мы ограничили массив, хранящий дискретное распределение, всего 100 значениями. В результате распределения с большими дисперсиями почти совпадают и, следовательно, хуже различаются.

Очевидно, что для уменьшения вероятности ошибки следует выбирать такие значения дисперсий, которые дают максимально удалённые друг от друга значения дивергенции. На основании данных из табл. 1. выберем первые 10 значений дисперсий для использования при построении ЦВЗ. Вычисленная дисперсия определяет хранимое число ЦВЗ согласно табл. 2.

Таблица 2. Соответствие между дисперсией и хранимым сообщением ЦВЗ

Дисперсия	1.0	1.05	1.1	1.2	1.3
Сообщение	0	1	2	3	4
Дисперсия	1.4	1.5	1.7	2.0	3.0
Сообщение	5	6	7	8	9

3.3. Анализ накладных расходов

При разработке программ стремятся оптимизировать работу алгоритма так, чтобы время выполнения было минимальным. Даже была разработана широко известная теория сложности вычислительных процессов и структур. Однако в ряде случаев повышение трудоёмкости алгоритма является одной из задач. В работе [12] демонстрируется пример такого случая. Допустим, что алгоритм проверяет серийный ключ лицензионной программы за 0.001 с. Тогда для усложнения отладки в алгоритм добавляется специальная вычислительна сложная задача, которая увеличивает время выполнения до 1–2 с. Такое «подвисание» программы будет допустимым и редким событием. Если программа будет запущена злоумышленником в режиме отладки или трассировки, то фактическое время её выполнения увеличится на несколько порядков, например, до нескольких часов или суток, что неприемлемо для анализа.

Во многих вышеупомянутых работах, например в [12], применяются подобные системы антиотладочных механизмов, в т.ч. запутывания программы, реализованные через ЦВЗ. Считается, что критически важные к скорости выполнения кода участки программы должны быть неизменными (оптимизированными). В нашем методе ожидается увеличение трудоёмкости алгоритма за счёт длительной генерации случайных чисел.

Для оценки накладных расходов было измерено среднее время выполнения фиксированной задачи, выполняемой 50 раз для различного числа потоков-генераторов. Для удобства результаты представлены в табл. 3 в относительных единицах (за единицу взято время выполнения 10-поточной программы). Создание большого числа потоков, например 20000 и более, может спровоцировать появление ошибки. Исходя из анализа полученных данных рекомендуется запускать не более 500–600 потоков, если программа критична к временным задержкам.

Таблица 3. Увеличение времени выполнения программы

Число потоков	10	100	300	500	600	700	800	1000	10000
Увеличение времени, раз	1.000	1.003	0.999	1.045	5.059	7.083	13.257	22.290	230.138

4. Заключение

В результате исследования был предложен метод построения цифрового водяного знака, базирующийся на многопоточности. Основным преимуществом данного метода является отсутствие необходимости выполнять трассировку программы и устойчивость к программным упаковщикам. В отличие от классических динамических водяных знаков, в которых полезное сообщение фиксировано, новый метод предполагает непрерывное изменение случайной величины для повышения защищенности. Большое количество ложных потоков маскирует ЦВЗ и обеспечивает антиотладочную защиту программы.

Литература

1. *Jung J. H., Kim J. Y., Lee H. C., Yi J. H.* Repackaging attack on android banking applications and its countermeasures // Wireless Personal Communications. 2013. V. 73. P. 1421–1437.
2. *Balakrishnan A., Schulze C.* Code obfuscation literature survey // CS701 Construction of compilers. 2005. V. 19. P. 31.
3. *Thomborson C., Nagra J., Somaraju R., He C.* Tamper-proofing software watermarks // Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation. 2004. V. 32. P. 27–36.
4. *Collberg C., Thomborson C.* Software watermarking: Models and dynamic embeddings // Proc. 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1999. P. 311–324.
5. *Hamilton J., Danicic S.* A survey of static software watermarking // IEEE World Congress on Internet Security (WorldCIS-2011). 2011. P. 100–107.
6. *Wang Y., Gong D., Lu B., Xiang F., Liu F.* Exception handling-based dynamic software watermarking // IEEE Access. 2018. V. 6. P. 8882–8889.
7. *Lin E. T., Podilchuk C. I., Delp III E. J.* Detection of image alterations using semifragile watermarks // Security and Watermarking of Multimedia Contents II. SPIE. 2000. V. 3971. P. 152–163.
8. *Нечта И. В.* Полухрупкие цифровые водяные знаки, базирующиеся на спектре Фурье // Вестник СибГУТИ. 2019. № 4. С. 33–41.
9. *Nagra J., Thomborson C.* Threading software watermarks // Information Hiding: 6th International Workshop, IH 2004, Toronto, Canada, May 23-25, 2004, Revised Selected Papers 6. Springer Berlin Heidelberg, 2005. P. 208–223.
10. *Ma H., Jia C., Li S., Zheng W., Wu D.* Xmark: dynamic software watermarking using Collatz conjecture // IEEE Transactions on Information Forensics and Security. 2019. V. 14, № 11. P. 2859–2874.
11. Официальный сайт программы UPX [Электронный ресурс]. URL: `https://upx.github.io` (дата обращения: 08.05.2023).
12. *Нечта И. В.* О методе считывания цифрового водяного знака в исполняемых файлах. Вестник СибГУТИ. 2020. № 1. С. 3–10.

Нечта Иван Васильевич

д.т.н., заведующий кафедрой прикладной математики и кибернетики, Сибирский государственный университет телекоммуникаций и информатики (СибГУТИ, 630102, Новосибирск, ул. Кирова, д. 86), e-mail: `ivannechta@gmail.com`, ORCID ID: 0000-0003-0361-2742.

Автор прочитал и одобрил окончательный вариант рукописи.
Автор заявляет об отсутствии конфликта интересов.

Application of Multithreading for Protection Programs

Ivan V. Nechta

Siberian State University of Telecommunications and Information Science (SibSUTIS)

Abstract: A method for identifying a program copy by means of digital watermark is presented in this article. Previously known analogues for constructing a watermark used modification of program's control flow graph, and reading was made via complex analysis of the program's trace. It is known that applying a protector or packer can neutralize such a watermark. In this paper, it is proposed to use a multi-threaded application that generates numbers with a normal

distribution law. The embedding of the digital watermark is carried out by changing the dispersion of distributions. Such random variable can be found via a signature in the process memory and is used both within the program itself and by a third-party when reading the watermark. An experimental selection of the optimal parameters of the algorithm has been carried out. Overhead expenses associated with the creation of a large number of threads are estimated.

Keywords: steganography, digital watermarks, multithreading, program protection.

For citation: Nechta I. V. Application of multithreading for protection programs (in Russian). *Vestnik SibGUTI*, 2023, vol. 17, no. 3, pp. 3-11. https://doi.org/10.55648/1998-6920-2023-17-3-3-11.

The article was submitted: 12.05.2023;
accepted for publication 20.05.2023.

References

1. Jung J. H., Kim J. Y., Lee H. C., Yi J. H. Repackaging attack on android banking applications and its countermeasures. *Wireless Personal Communications*, 2013, vol. 73, pp. 1421-1437.
2. Balakrishnan A., Schulze C. Code obfuscation literature survey. *CS701 Construction of compilers*, 2005, vol. 19, pp. 31.
3. Thomborson C., Nagra J., Somaraju R., He C. Tamper-proofing software watermarks. *Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, 2004, vol. 32, pp. 27-36.
4. Collberg C., Thomborson C. Software watermarking: Models and dynamic embeddings. *Proc. of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999, pp. 311-324.
5. Hamilton J., Danicic S. A survey of static software watermarking. *IEEE World Congress on Internet Security (WorldCIS-2011)*, 2011, pp. 100-107.
6. Wang Y., Gong D., Lu B., Xiang F., Liu F. Exception handling-based dynamic software watermarking. *IEEE Access*, 2018, vol. 6, pp. 8882-8889.
7. Lin E. T., Podilchuk C. I., Delp III E. J. Detection of image alterations using semifragile watermarks. *Security and Watermarking of Multimedia Contents II. SPIE*, 2000, vol. 3971, pp. 152-163.
8. Nechta I. V. Poluhrupkie cifrovye vodyanye znaki, baziruyushchiesya na spektre Fur'e [Semi-fragile digital watermarks based on Fourier spectrum]. *Vestnik SibGUTI*, 2019, no 4. pp. 33-41.
9. Nagra J., Thomborson C. Threading software watermarks. *Information Hiding: 6th International Workshop, IH 2004*, Toronto, Canada, 23-25 May, 2004, Revised Selected Papers 6. Springer Berlin Heidelberg, 2005, pp. 208-223.
10. Ma H., Jia C., Li S., Zheng W., Wu D. Xmark: dynamic software watermarking using Collatz conjecture. *IEEE Transactions on Information Forensics and Security*, 2019, vol. 14, no. 11, pp. 2859–2874.
11. Oficial'nyj sajt programmy UPX [Official website of the UPX program], available at: https://upx.github.io (accessed: 08.05.2023).
12. Nechta I. V. O metode schityvaniya cifrovogo vodyanogo znaka v ispolnyaemyh fajlah [On the method of reading a digital watermark in executable file]. *Vestnik SibGUTI*, 2020, no. 1. pp. 3-10.

Ivan V. Nechta

Dr. of Sci. (Engineering), Associate Professor; Chief of the Department of Applied Mathematics and Cybernetics, Siberian State University of Telecommunications and Information Science (SibSUTIS, Russia, 630102, Novosibirsk, Kirov St. 86), phone: +7 383 269 82 72, e-mail: ivannechta@gmail.com, ORCID ID: 0000-0003-0361-2742.