DOI 10.37882/2223-2966.2023.7-2.37

МЕТОДИКА СОЗДАНИЯ И СКРЫТОГО ВЛОЖЕНИЯ ЦИФРОВОГО ВОДЯНОГО ЗНАКА В БАЙТ-КОД CLASS-ФАЙЛА НА ОСНОВЕ НЕ ДЕКЛАРИРОВАННЫХ ВОЗМОЖНОСТЕЙ ВИРТУАЛЬНОЙ МАШИНЫ JAVA

Шариков Павел Иванович
Ассистент, Санкт-Петербургский государственный университет телекоммуникаций им. проф. М.А. Бонч-Бруевича
sharikov.pavel@ro.ru

Аннотация. В данной работе представлена проблема компрометации java-приложений посредством кражи исполняемых class-файлов, а также представлено решение, посредством которого возможно производить создание и вложение цифрового водяного знака в class-файлы java-приложения. Рассмотрены принципы, на которых базируется создание и вложение цифрового водяного знака в class-файлы java-приложений. Произведен анализ class-файлов JRE, сделаны выводы о частоте использования отдельных групп опкодов. Разобраны операционные команды виртуальной машины Java, позволяющие произвести вложение. Произведено редактирование и вложение, проанализированы результаты. Представлена математическая и алгоритмическая модели методики. Сделаны выводы о разработанной методике, ее возможностях и целях применения.

Ключевые слова: байт-код, цифровой водяной знак, цвз, class-файл, bytecode, digital watermark, java, jvm.

A TECHNIQUE FOR CREATING AND COVERTLY EMBEDDING A DIGITAL WATERMARK IN THE BYTECODE OF A CLASS FILE BASED ON UNDECLARED CAPABILITIES OF A JAVA VIRTUAL MACHINE

P. Sharikov

Summary: This paper presents the problem of compromising java applications by stealing executable class files, and also presents a solution by which it is possible to create and embed a digital watermark in java application class files. The principles on which the creation and embedding of a digital watermark in class files of java applications is based are considered. The analysis of JRE class files was carried out, conclusions were drawn about the frequency of use of individual groups of opcodes. The operational commands of the Java virtual machine that allow you to make an attachment have been disassembled. Editing and embedding were performed, the results were analyzed. Mathematical and algorithmic models of the methodology are presented. Conclusions are drawn about the developed methodology, its capabilities and purposes of application.

Keywords: bytecode, digital watermark, cvz, class file, bytecode, digital watermark, java, jvm.

Введение

В современном мире кража программного обеспечения и другой интеллектуальной собственности для использования в своих собственных целях достигает огромных размеров. В данной работе предлагается методика, позволяющая осуществить создание и вложение цифрового водяного знака в байт-код class-файлов java-приложения за счет эквивалентных замен опкодов.

Каждая компания хочет защитить свой продукт от нелегального копирования, использования, а тем самым и от рисков упущенной прибыли. В текущий момент, один из самых востребованных языков программирования, по международному индексу TIOBE — Java [1]. На данном языке программирования написано огромное количество приложений для крупных коммерческих и государственных проектов. Таких как: Jenkins, Web Sphere, Web Logic, JBoss и другие. Java используется в банках, кассовых аппаратах и в других областях, сферах жизни и бизнеса. Скомпилированный в class-файлы программный код Java достаточно легко декомпилируется, что открывает дополнительные возможности для злоумышленников [2, с. 67].

Защита прав интеллектуальной собственности на ПО становится первостепенной задачей, которую необходимо решить. Решением по защите авторских прав на программное обеспечение может стать цифровой водяной знак. Методики для вложения цифровых водяных знаков постоянно разрабатываются и улучшаются для того, чтобы обеспечить максимальным покрытием программные объекты, а также в целях актуализации изменений в логике работы новых версий виртуальной машины Java.

Новизна данного исследования, заключается в том, что на текущий момент существующие методики устарели в силу изменений в работе виртуальной машины Java и не эффективно используют не декларированную возможность эквивалентной замены опкодов.

Причины необходимости разработки методики

Рассматривая проблему пиратства программного обеспечения, разработанного на Java, более пристально, легко понять, что в текущем положении вещей цифровой водяной знак должен удовлетворять целому ряду критериев, для успешного внедрения в class-файл java-программы и устойчивости к атакам [3].

Основные критерии, принятые выдвигать в качестве требований к цифровым водяным знакам:

1. Цифровой водяной знак должен быть надежным, сохраняя устойчивость после программ преобразования
2. Цифровой водяной знак должен быть невидимым для обычных пользователей и злоумышленников.
3. Цифровой водяной знак должен быть устойчивым к перезаписи
4. Цифровой водяной знак должен содержать информацию, подтверждающую утверждение о том, что компания-заявитель имеет все юридические права на украденные class-файлы или само программное обеспечение

Просматривая данный список критериев легко понять, что вложение цифрового водяного знака в class-файл происходит не всегда быстро, с учетом обычного количества файлов в серьезных программных продуктах, а также не всегда выгодно или целесообразно помещать цифровой водяной знак в каждый class-файл программы.

Также, необходимо учитывать фактор быстрого устаревания существующих методик создания и вложения цифровых водяных знаков в исполняемые class-файлы java-приложений. В данный момент в открытом доступе нет методик, которые не являлись бы устаревшими для виртуальной машины Java вышедшей в марте 2023. Следовательно, существует потребность в методике менее зависимой от логики работы виртуальной машины Java, которая позволит создать такой цифровой водяной знак, который будет неотъемлемой частью java-приложения или class-файла и разрушение которого будет приводить к полной неработоспособности class-файла или java-приложения.

Таким образом, корпорации вынуждены тратить время и средства на специалистов, которые за какое-то время изучения и анализа проекта смогут сказать в какие class-файлы лучше всего вложить цифровой водяной знак и какие методики использовать. Данное решение не полностью идеально, так как существует риск утечки файлов или бизнес-логики только потому, что часто специалисты такого рода нанимаются со стороны. Идеальным решением будет программное обеспечение, способное в автоматическом режиме произвести анализ всех файлов проекта, выявить максимальный объем вложения в тот или иной файл, дать рекомендации по выбору методик вложения и вывести результат в виде отчета.

Структура class-файла

Class-файл это скомпилированный исходный код Java. Так как Java интерпретируемый язык программирования, то скомпилированный class-файл содержит в себе инструкции для виртуальной машины Java, которая преобразует их в команды к процессору.

Таким образом, на каждое действие в исходном коде существует операционная команда байт-кода, которым оперирует виртуальная машина Java. Class-файл является легко декомпилируемым и редактируемым форматом файлов. Производить редактирование скомпилированного class-файла можно в HEX-редакторе при наличии знаний о структуре байт-кода и его шестнадцатеричном представлении [4, с. 860].

Class-файл представляет собой скомпилированных исходный код на java. Таким образом, исходный код Java интерпретируется в байт-код, состоящий из операцион-

Биты

← 8 → ← 8 → ← 8 → ← 8 → ← 8 →

Формат

1. Операция				
2.Операция	Байт			
3. Операция	SHORT			
4. Операция	Индекс	Константа		
5. Операция	Индекс		Размерность	
6. Операция	Индекс		Параметры	0
7. Операция	Индекс		Константа	
8. Операция	32- битное смещение перехода			

Рис. 1. Формат инструкций виртуальной машины Java

ных кодов виртуальной машины Java, компилируется в class-файлы, которые собираются в java-приложения, которые в свою очередь запускаются посредством инструментов сборки и виртуальной машины Java [5; 6].

Каждый код операции байт-кода имеет размер один байт, формат инструкций наглядно продемонстрирован на рисунке 1.

Class-файл состоит из 10 секций [7; 8]. Секции возможно разбить на группы:

— свойства class-файла
— пул констант, который содержит все наименования методов class-файла и другую вспомогательную информацию для работы
— блок основных свойств class-файла. Информацию о родителях этого class-файла, наследниках, интерфейсах, которые имплементирует данный class-файл, различные флаги доступа
— непосредственно внутреннее наполнение class-файла. Список полей class-файла и его байт-код
— набор атрибутов class-файла.

Подробная структура class-файла продемонстрирована в листинге 1.

Листинг 1. Структура class-файла

```
ClassFile {
    u4              зарезервировано
    u2              младшая часть номера версии
    u2              старшая часть номера версии
    u2              количество константных пулов
    cp_info         константный пул [количество константных пулов — 1]
    u2              флаги доступа
    u2              текущий класс
    u2              предок(суперкласс)
    u2              количество интерфейсов
    u2              интерфейсы [количество интерфейсов]
    u2              количество полей
    field_info      поля [количество полей]
    u2              количество методов
    method_info     методы [количество методов]
    u2              количество атрибутов
    attribute_info  атрибут [количество атрибутов]
    }
```

В разработанной методике предлагается производить вложение посредством редактирования группы, содержащей основную логику работы исполняемого файла, а именно его внутреннее наполнение — байт-код.

Байт-код Java

Самыми распространенными инструкциями являются: операторы условного перехода, операторы управления стеком, преобразование или создание объекта на стеке, логические и арифметические операции, сохранение и загрузка данных.

В таблице 1 продемонстрированы наиболее часто встречаемые инструкции байт-кода, их мнемоническое представление и тип операции.

Таблица 1.

Инструкции байт-кода

Мнемоническое представление инструкции байт-кода	Шестнадцатеричное значение	Операция
astore	0x3a	Сохранить ссылку на объект в локальную переменную
bipush	0x10	Байт расширяется до значения int. Значение помещается в стек операндов
dadd	0x63	Значение типа double помещается в стек операндов
iaload	0x2e	Значение в массиве по индексу извлекается и помещается в стек операндов с типом int
if_icmpeq	0x9f	Оператор условного перехода при равенстве двух значений int
if_icmpne	0xa0	Оператор условного перехода при не равенстве двух значений int
if_icmplt	0xa1	Оператор условного перехода при значении 1 < значения 2. Оба значения типа int
if_icmpgt	0xa3	Оператор условного перехода при int значении 1 > int значения 2
istore	0x36	Сохранить значение int в локальную переменную
return	0xb1	Возврат из метода
ifeq	0x99	Оператор условного перехода при успешном сравнении значения типа int с 0
ifne	0x9a	Оператор условного перехода при значении типа int не равном нулю
ifgt	0x9d	Оператор условного перехода при значении типа int больше нуля

Был произведен анализ 100 class-файлов из пакета java rt.jar, который показал распределение частоты использования тех или иных инструкций байт-кода. На рисунке 2 представлена диаграмма, демонстрирующая частоту появления тех или иных групп опкодов в выборке class-файлов из пакета rt.jar.

Исходя из таблицы 1 возможно сделать вывод о том, что инструкции байт-кода, относящиеся к группе операторов условного перехода, часто встречаются в class-файлах и имеют зеркальные или эквивалентные инструкции исходя из результатов, представленных в таблице 1.

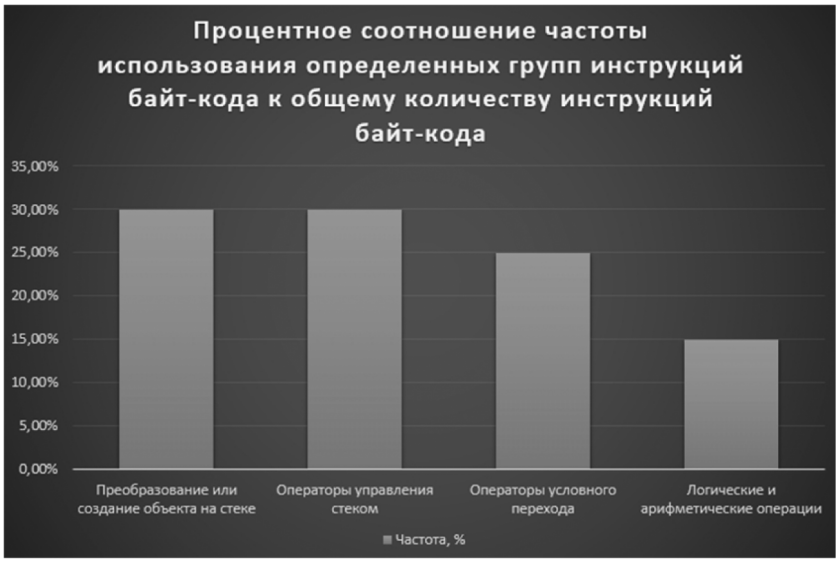

Рис. 2. Частота возникновения в байт-коде class-файла различных групп инструкций байт-кода в соотношении к общему количеству инструкций байт-кода в class-файле

Рис. 3. Пример исходного кода, скомпилированного в class-файл с демонстрацией сопоставления исходного кода и инструкций байт-кода соответствующих ему

На рисунке 3 продемонстрирован пример работы байт-кода java, как исходный код интерпретируется в байт-код, и какая инструкция байт-кода отвечает за действия, описанные в исходном коде.

Эквивалентность инструкций байт-кода Java

Ранее был сделан вывод о том, что в виртуальной машине Java существуют инструкции байт-кода, которые по своим действиям являются зеркальными или эквивалентными. Таким образом необходимо произвести эксперимент. В данном эксперименте будут выявлен набор инструкций, который может быть пригоден для эквивалентных замен. После этого будет выбран демонстрационный class-файл с минимальным количеством логики, но содержащий инструкции из определенного набора байт-кодов. Без перекомпиляции или декомпиляции class-файла, используя редактор class-файлов Bytecode Editor будет произведено редактирование с заменой одной инструкции на другую, после чего сделаны выводы о результате [9, с. 548; 10].

Одним из простых решений выбора инструкций для эквивалентных замен являются инструкции байт-кода ответственные за операторы условного перехода [11, с. 260]. В связи с тем, что при изменении инструкции достаточно изменить направление ветвей оператора или осуществить перестановку сравниваемых переменных на стеке перед использованием оператора условного перехода. Возможные варианты эквивалентных замен представлены на рисунке 4.

Замена инструкции байт-кода в скомпилированном class-файле

Предлагаемый метод вложения информации в байт-код class-файл основан на замене существующих ветвлений кода. Любой ветвление, написанное на языке Java, всегда можно заменить аналогичным, но с противоположным логическим выражением. В таком случае логика ветвления измениться. Чтобы этого не допустить, также

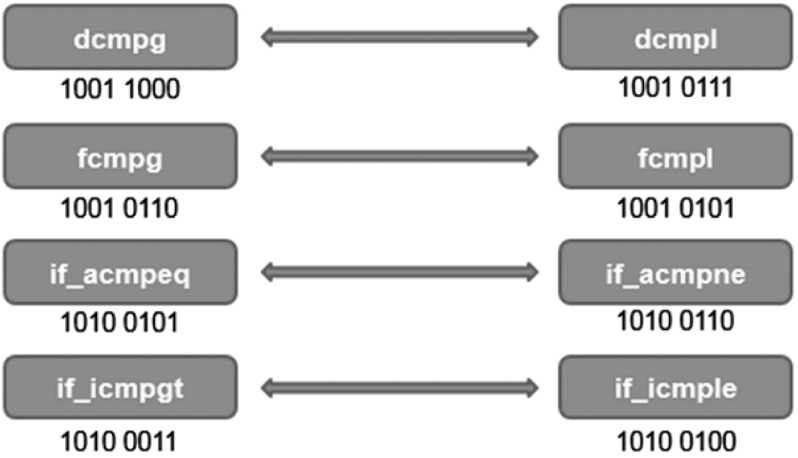

Рис. 4. Пример эквивалентности опкодов операторов условного перехода

```
C:\Users\TonyStark\Documents\Кафедра\Статьи\Код>javac Zss1.java

C:\Users\TonyStark\Documents\Кафедра\Статьи\Код>javap Zss1.class
Compiled from "Zss1.java"
public class Zss1 {
  public Zss1();
  public static void main(java.lang.String[]);
}

C:\Users\TonyStark\Documents\Кафедра\Статьи\Код>javap -c Zss1.class
Compiled from "Zss1.java"
public class Zss1 {
  public Zss1();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":
()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: iconst_1
       1: istore_1
       2: iconst_2
       3: istore_2
       4: iload_1
       5: iload_2
       6: if_icmpge     20
       9: getstatic     #2                  // Field java/lang/System.out:Ljava/
io/PrintStream;
      12: ldc           #3                  // String a < b; a = 1; b = 2
      14: invokevirtual #4                  // Method java/io/PrintStream.printl
n:(Ljava/lang/String;)V
      17: goto          28
      20: getstatic     #2                  // Field java/lang/System.out:Ljava/
io/PrintStream;
      23: ldc           #5                  // String a > b; a = 1; b = 2
      25: invokevirtual #4                  // Method java/io/PrintStream.printl
n:(Ljava/lang/String;)V
      28: return
}

C:\Users\TonyStark\Documents\Кафедра\Статьи\Код>_
```

Рис. 5. Результат компиляции исходного кода и просмотр его байт-кода в командной строке

следует при замене логического выражения заменить и сами ветви условия местами. Таким образом, логика работы не меняется, но меняются местами наборы команд. Рассмотрим пример исходного кода, продемонстрированного в листинге 2.

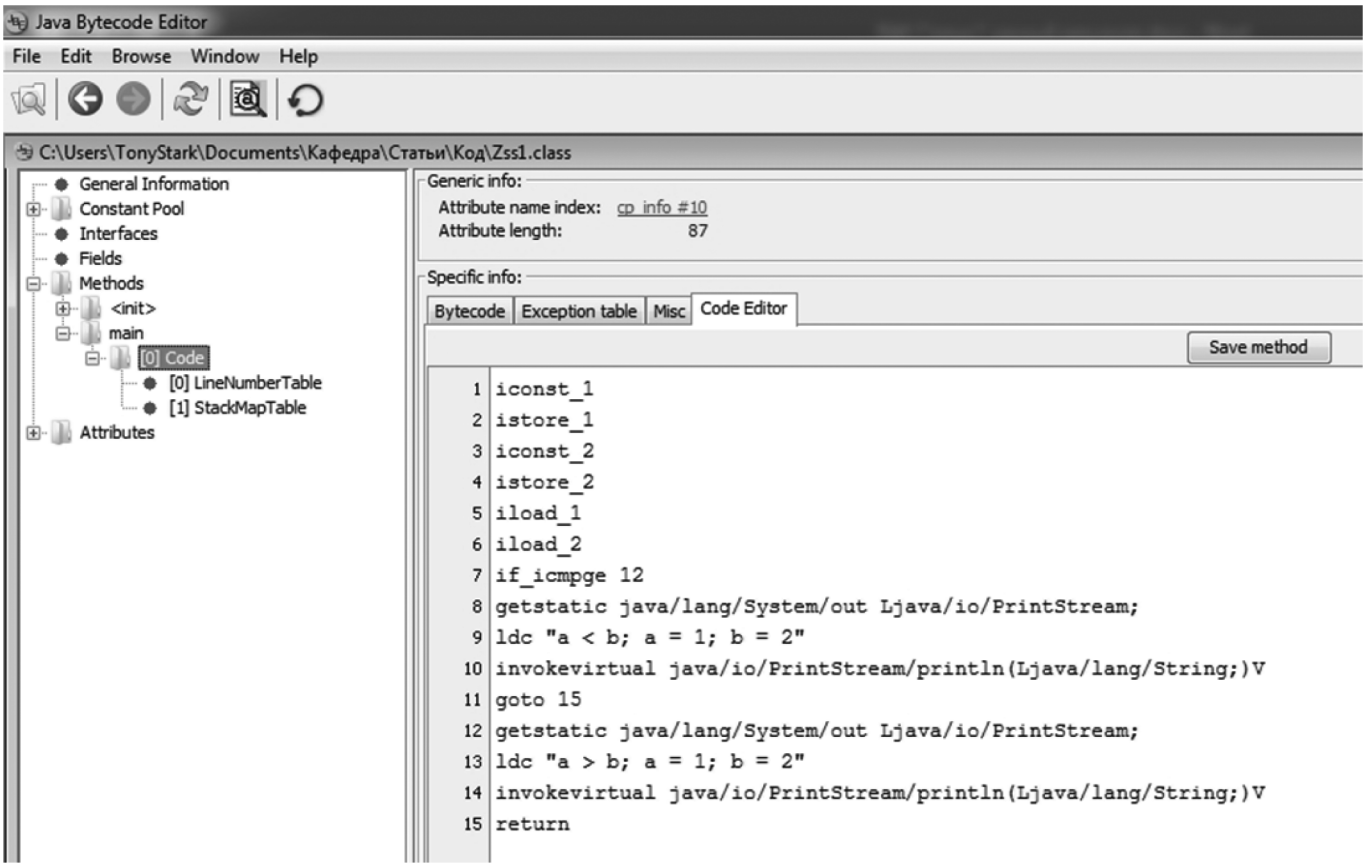

Рис. 6. Демонстрация байт-кода скомпилированного class-файла в редакторе байт-коде

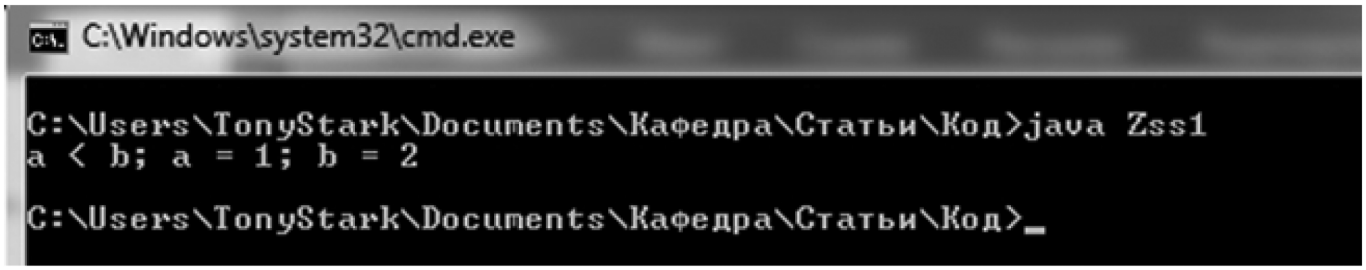

Рис. 7. Результат исполнения class-файла.

Листинг 2. Исходный код простейшей программы с ветвлением

```
public class Zss1 {
 public static void main(String[] args) {
 int a = 1;
 int b = 2;
 if (a < b) {
 System.out.println("a < b; a = 1; b = 2");
 } else {
 System.out.println("a > b; a = 1; b = 2");
 }
 }
}
```

На рисунке 5 изображена командная строка с результатами компиляции данного кода и вывод его байт-кода, а на рисунке 6 продемонстрировано, что в ПО ByteCode Editor байт-код данного class-файла выглядит точно также.

Как продемонстрировано на рисунках 5-6 байт-код скомпилированного class-файла не отличается вне зависимости от ПО с помощью которого осуществляется просмотр. Рисунок 7 демонстрирует результат исполнения class-файла.

На рисунке 8 продемонстрирован объем занимаемый class-файлом на жестком диске равный 512 байт.

Теперь произведем эквивалентную замену инструкции из определенного ранее набора опкодов. Для упрощения эксперимента, ветви блока условного перехода меняться не будут. Соответственно, после изменения

бай-кода программа будет выводить строку, содержащуюся в блоке else.

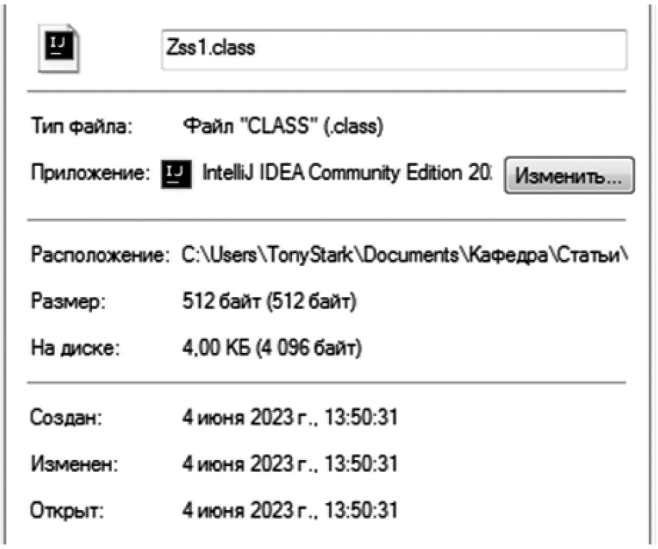

Рис. 8. Занимаемый class-файлом объем

Как продемонстрировано на рисунках 5–6 оператор условного перехода в байт-коде заменяется инструкцией if_icmpge = 162 (0xa2), следовательно, для эквивалентной замены необходимо воспользоваться инструкцией if_icmplt = 161 (0xa1). Создадим каталог «edit_class_file» и скопируем туда скомпилированный class-файл Zss1.class, после чего произведем редактирование с помощью ПО ByteCode Editor. Рисунок 9 демонстрирует результат редактирования class-файла — байт-код изменен.

На рисунке 10 продемонстрирован результат исполнения class-файла. Как было описано ранее, в силу того, что не была произведена замена веток условия местами, теперь из-за измененного (зеркального) условия в блоке if() программа выдает неверный результат, а именно то, что «a > b».

Рисунок 11 демонстрирует, что после редактирования инструкции байт-кода на эквивалентную, размер class-файла изменился и стал равным 480 байт, что на 6,25 % меньше объема исходного class-файла [12,

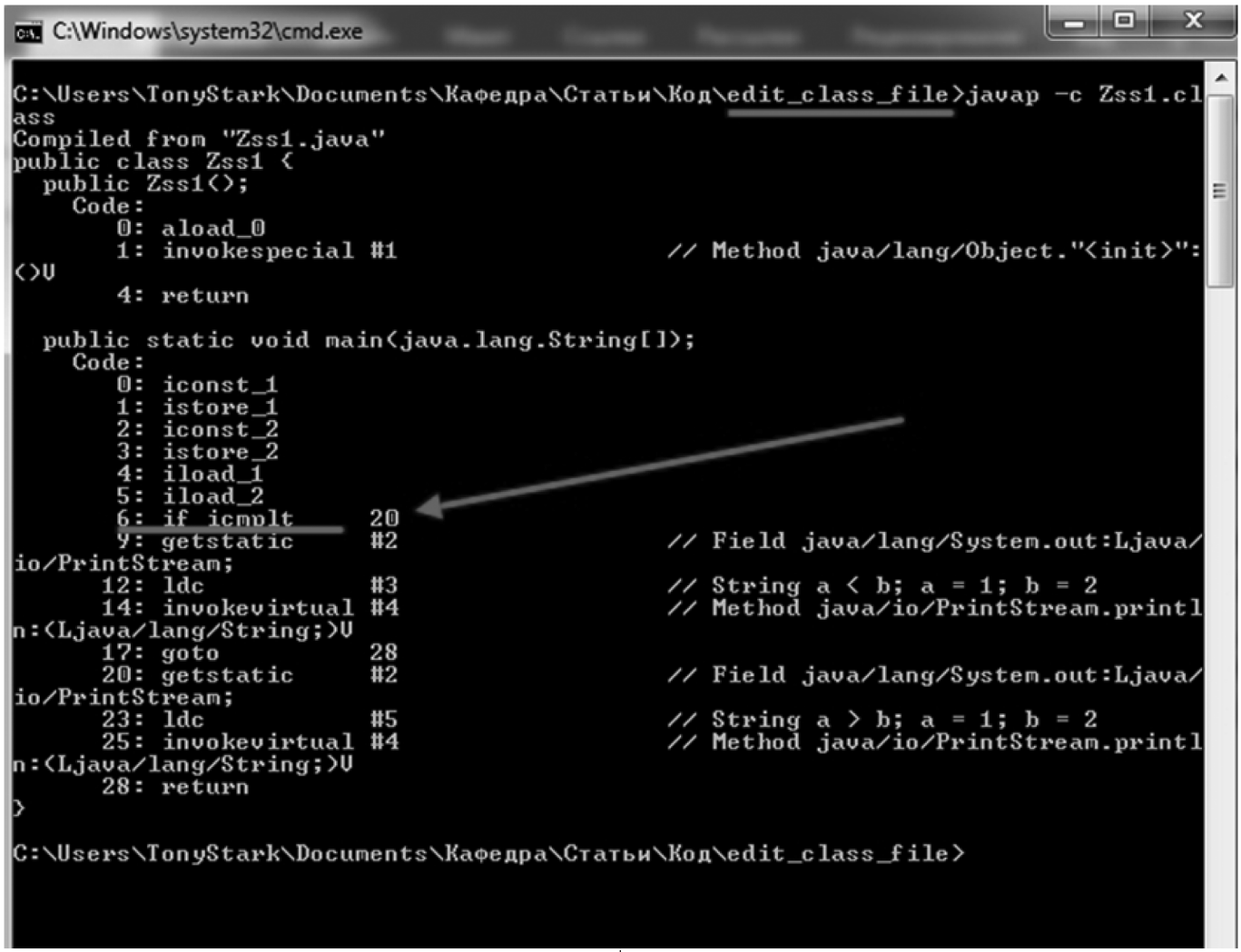

Рис. 9. Демонстрация успешного редактирования байт-кода class-файла без перекомпиляции

Рис. 10. Результат исполнения class-файла с отредактированным байт-кодом

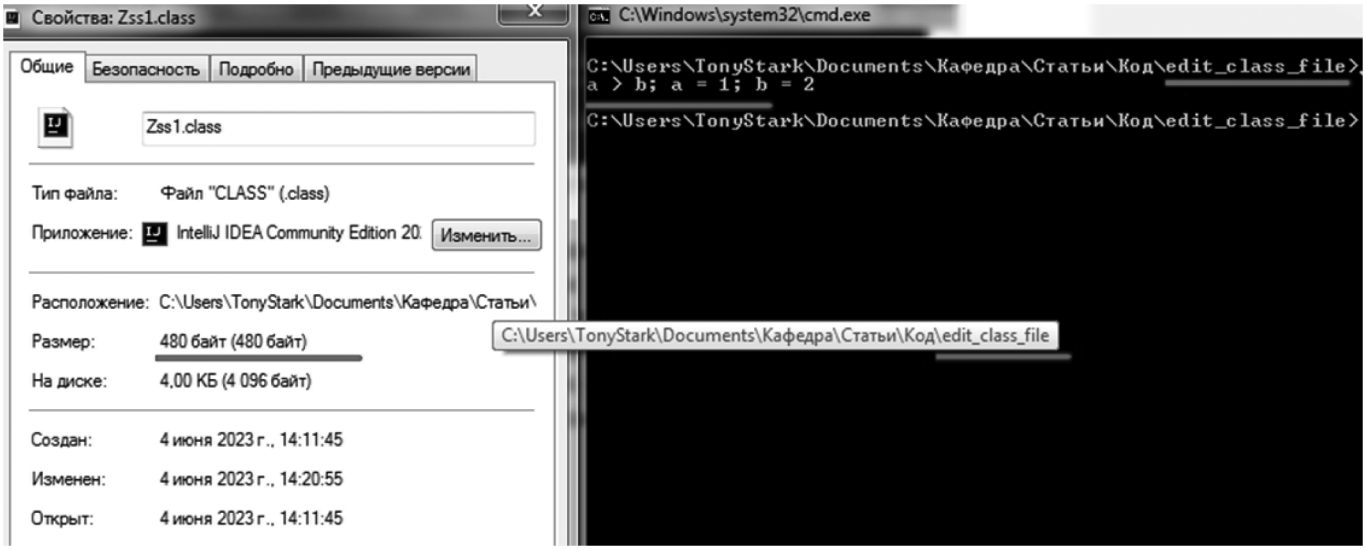

Рис. 11. Размер class-файла с редактированным байт-кодом

с. 1035]. Данный эффект оптимизации связан с тем, что замена инструкции байт-кода была произведена с нарушениями. Как было описано ранее, для того чтобы размер class-файла остался не именным необходимо также произвести зеркальную замену блоков условия. Блок if(), исполняемый при верности условия поместить в блок else() и наоборот.

Математическая модель разработанной методики

Исходя из результатов проведенного эксперимента можно сделать вывод о том, что размер возможного вложения зависит от объема class-файла и содержащихся в нем инструкций байт-кода, которые пригодны для эквивалентных замен. Таким образом, количество возможных к эквивалентным заменам опкодов для java-приложения будет высчитываться по формуле:

$$\sum_{i=1}^{n} C_i,$$

где C_i — это количество пригодных к эквивалентным заменам опкодов в class-файле, а n — количество class-файлов в java-приложении.

Таким образом, необходимо осуществлять анализ java-приложения на составляющие части и работу с каждым отдельным class-файлом, как с самостоятельной частью. Тогда число возможных замен инструкций байт-кода в class-файле будет равно $N!$, где N — число операционных кодов JVM пригодных для эквивалентных замен, присутствующих в байт-коде данного class-файла. Следовательно, количество информации доступной к вложению для одного class-файла в байтах (1 байт-код = 1 байт) будет вычисляться по формуле:

$$\log_2(N!) = N\log_2 N.$$

Алгоритмическая модель разработанной методики

Таким образом алгоритмическая модель разработанной методики будет представлять собой анализ всего java-приложения, которая продемонстрирована на рисунке 12.

Выводы

Разработанная методика позволяет создавать и производить вложение цифрового водяного знака в байт-код class-файлов java-приложений. С помощью данной методики производится создание и вложение цифрового водяного знака, который является частью class-файла, его удаление повлечет нарушение работы class-файла или всего java-приложения. Было доказано, что с помощью редактирования эквивалентных по логике работы инструкций байт-кода в скомпилированном class-файле без перекомпиляции не приводит к ошибкам исполнения class-файла. Следовательно, разработанная методи-

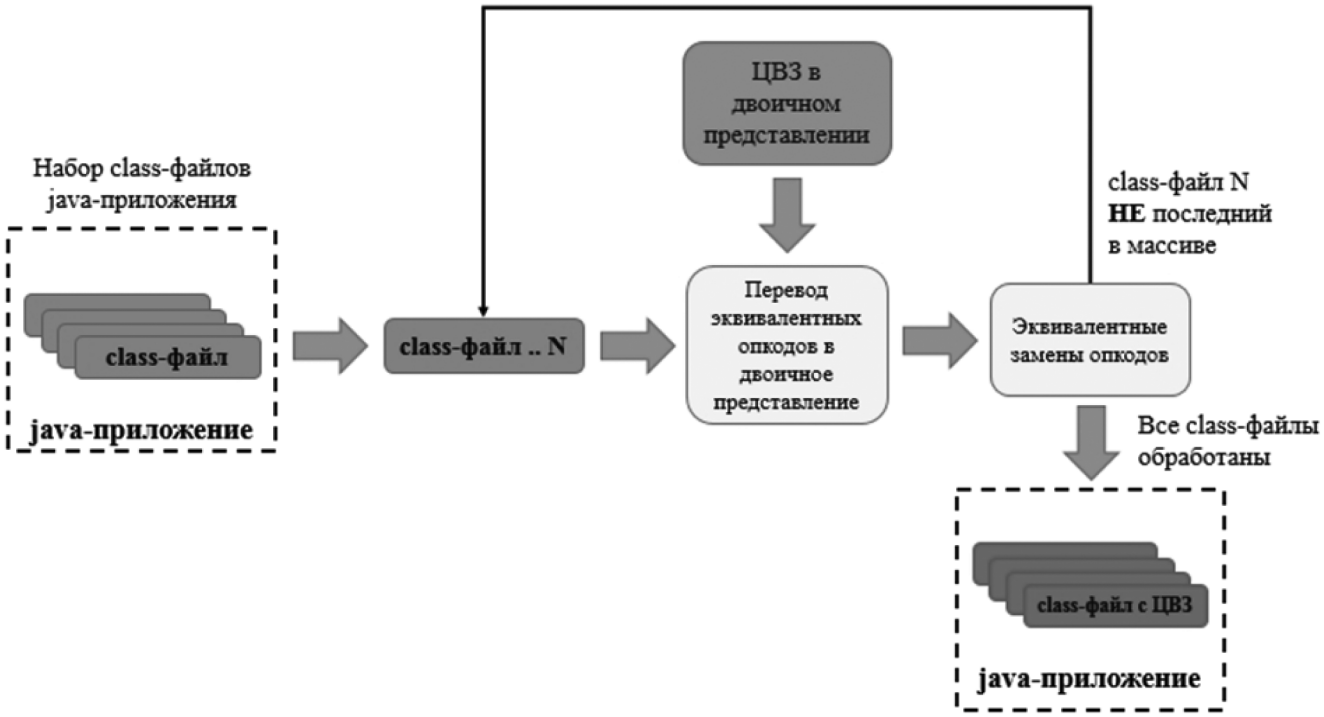

Рис. 12. Алгоритмическая модель разработанной методики

ка подходит для реализации вложения цифрового водяного знака.

Таким образом, данная методика позволяет создать и осуществить скрытое вложение цифрового водяного знака. Цифровой водяной знак, который является частью исполняемого файла может применяться в рамках юридического подтверждения прав собственности результат интеллектуальной деятельности — исходный код. Также, с помощью цифрового водяного знака вложенного в class-файлы java-приложения возможно отслеживать контроль целостности приложения.

Продолжением исследований данной работы могут быть такие шаги, как:

— Исследование возможности использования расширенного набора инструкций байт-кода для произведения вложения цифрового водяного знака
— Исследование устойчивости цифрового водяного знака вложенного разработанной методикой к различного рода атакам
— Исследование эффективной проверки наличия цифрового водяного знака в байт-коде
— Исследование возможности алгоритмизации и доведения до реализации программы на ЭВМ разработанной методики
— Расчет усредненных показателей методики в сравнении с другими на выборке class-файлов большого объема
— Исследование возможности вложения дробной части единого цифрового водяного знака в байт-код различных class-файлов информационной системы.

ЛИТЕРАТУРА

1. TIOBE. URL: https://tiobe.com/tiobe-index/java/ (дата обращения: 03.06.2023)
2. Шариков П.И., Красов А.В., Штеренберг С.И. Методика создания и вложения цифрового водяного знака в исполняемые java файлы на основе замен опкодов // T-Comm: Телекоммуникации и транспорт. 2017. Т. 11. №3. С. 66–70.
3. Sharikov P.I., Krasov A.V., Volkogonov V.N. A study of the correctness of the execution of a class file with an embedded digital watermark in different environments // IOP Conference Series: Materials Science and Engineering. 2020. С. 52052–52052.
4. Хомяков И.Н., Красов А.В. Анализ возможностей скрытого вложения информации в структуру байт-кода java //Актуальные проблемы инфотелекоммуникаций в науке и образовании. 2013. С. 859–861.
5. Hamilton J., Danicic S. An evaluation of static java bytecode watermarking //Proceedings of the International Conference on Computer Science and Applications (ICCSA'10), The World Congress on Engineering and Computer Science (WCECS'10), San Francisco. 2010.
6. Shi J. C. C. W. Q., Lv G. Implementation of bytecode-based software watermarking for java programs.
7. DOCS.ORACLE. URL: https://docs.oracle.com/javase/specs/jvms/se11/html/jvms-4.html (дата обращения: 03.06.2023)

8. DOCS.ORACLE. URL: https://docs.oracle.com/javase/specs/jvms/se20/html/index.html (дата обращения: 04.06.2023)
9. Calvagna A., Tramontana E. Automated conformance testing of Java virtual machines // 2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems. IEEE, 2013. С. 547–552.
10. Java ByteCode Editor. URL: https://set.ee/jbe/ (дата обращения: 04.06.2023)
11. Andrey K., Pavel S. A Technique for Analyzing Bytecode in a Java Project for the Purpose of an Automated Assessment of the Possibility and Effectiveness of the Hidden Investment of Information and its Volumes in a Java Project // 2020 12th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT). IEEE, 2020. С. 258–263.
12. Clausen L.R. A Java bytecode optimizer using side-effect analysis // Concurrency: Practice and Experience. 1997. Т. 9. №. 11. С. 1031–1045.