МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН
Некоммерческое акционерное общество
«АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ»
Кафедра систем информационной безопасности

«ДОПУЩЕН К ЗАЩИТЕ»
Зав. кафедрой к.п.н., доцент Р. Ш. Бердибаев
_____ «_____» _____2019 г.
(подпись)

ДИПЛОМНЫЙ ПРОЕКТ

На тему: Система защиты от утечки конфиденциальной информации по стеганографическим каналам.
Специальность: 5В100200 – "Системы информационной безопасности"
Выполнил: Аубакиров Темирлан Серикович Группа СИБ-15-2
Научный руководитель: Покусов Виктор Владимирович

Консультант:
по экономической части:
к. э. н., профессор Аренбаева Ж. Г.
(ученая степень, звание, Ф.И.О)
(подпись) «07» июня 2019 г.

по безопасности жизнедеятельности:
д т.н., ст. преп. Бекбасаров Ш. Ш.
(ученая степень, звание, Ф.И.О)
(подпись) «07» июня 2019 г.

по применению вычислительной техники:
ст. преп Покусов В.В.
(ученая степень, звание, Ф.И.О)
(подпись) «07» июня 2019 г.

Нормоконтролер:
ст.о преподаватель Аскарова Н.Б.
(ученая степень, звание, Ф.И.О)
(подпись) «7» июня 2019 г.

Рецензент:
Тех. директор Буков В. Н.
(ученая степень, звание, Ф.И.О)
(подпись) «6» июня 2019 г.

Алматы 2019

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН
Некоммерческое акционерное общество
«АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ»

Институт систем управления и информационных технологий

Кафедра систем информационной безопасности

Специальность 5В100200 – «Системы информационной безопасности»

ЗАДАНИЕ
на выполнение дипломного проекта

Студенту Аубакирову Темирлану Сериковичу

Тема проекта: Система защиты от утечки конфиденциальной информации по стеганографическим каналам.

Утверждена приказом по университету № 124 от « 26 » октября 2018 г.

Срок сдачи законченного проекта « 2 » июня 2019 г.

Целью дипломной работы является разработка программного обеспечения для скрытия пользовательских данных при помощи стеганографических алгоритмов.

Для достижения указанной цели необходимо решить следующий комплекс задач:

- сбор информации по теме дипломной работы;
- изучение общей структуры информационной системы;
- определение потенциальных источников угроз и уязвимые места;
- изучение и сравнительный анализ современных методов и систем защиты информации;
- внедрение оптимального метода защиты информации.

Перечень вопросов, подлежащих внедрению в дипломном проекте или краткое содержание дипломного проекта: дипломный проект включает в себя 6 глав, разделенные на подглавы, каждая из которых освещает определенную тематику, используемую при организации информационной безопасности.

В первой главе дипломного проекта представлена общая информация про компьютерную стеганографию и методы вложения.

Во второй главе дипломного проекта представлен использование байт-кода в качестве покрывающего сообщения

В третей главе рассматривается вложение в байт-код

В четвертой главе дипломного проекта рассматривается возможности применения

В пятой главе приводится технико-экономическое обоснование проекта.

В шестой главе рассматриваются необходимые условия для удобной работы сотрудников организаций соответствующим общепринятым положениям.

Основная рекомендуемая литература:

1 Быков С.Ф. Алгоритм сжатия JPEG с позиций компьютерной стеганографии // Защита информации. Конфидент. 2000. № 3.

2 Andrey Krasov., Stanislav Shterenberg. Methods for embedding hidden data into executable scripts.KEY ISSUES IN MODERN SCIENCE - 2014, Sofia, Bulgaria, 9 стр.

3 Красов А.В., Верещагин А.С., Цветков А.Ю. Аутентификация программного обеспечения при помощи вложения цифровых водяных знаков в исполняемый код. // М. Телекоммуникации Спецвыпуск 2013, с.27-30

4 Барсуков В.С. Стеганографические технологии защиты документов, авЗторских прав и информации // Обзор специальной техники.- 2000.- №2.-С. 31-40

Конструкции по проекту с указанием относящихся к ним разделов проекта

Раздел	Консультант	Сроки	Подпись
экономика	Арепбаева Ж.Г.	04.03-07.06	
БЖД	Бекбасаров Ш.Ш.	04.03-07.06	

График
подготовки дипломного проекта

Наименование разделов, перечень разрабатываемых вопросов	Сроки представления научному руководителю	Примечание
Компьютерная Стеганография	01.02.19 - 06.02.19	
Выбор покрывающего сообщения	07.02.19 - 10.02.19	
Методы вложения выходных	11.02.19 - 15.02.19	
Методы вложения в исполняемый код	16.02.19 - 22.02.19	
Использование байт-кода в качестве	23.02.19 - 27.02.19	
Язык программирования Java	28.02.19 - 03.03.19	
Виртуальная машина Java	04.03.19 - 07.03.19	
Структура Java кода	08.03.19 - 11.03.19	
Вложение в байт-код	12.03.19 - 15.03.19	
Методы вложения в байт-код	16.03.19 - 21.03.19	
Возможности применения	22.03.19 - 25.03.19	
Оценка вложимого объема	26.03.19 - 29.03.19	
Практические вопросы ПО	01.04.19 - 08.04.19	
Технико-экономическое обоснование	09.04.19 - 13.04.19	
Безопасность жизнедеятельности	15.04.19 - 19.04.19	

Дата выдачи задания «30» октября 2019 г.

Заведующий кафедрой _____ (_____)
(Подпись) (Ф.И.О)

Научный руководитель проекта _____ (Покусов В.В.)
(Подпись) (Ф.И.О)

Задание принял к исполнению студент _____ (Чубакиров П.С.)
(Подпись) (Ф.И.О)

АННОТАЦИЯ

Дипломный проект посвящен системе защиты от утечки конфиденциальной информации про стеганографическим каналам.

В проекте проведен сравнительный анализ криптографических и стеганографическим методов шифрования, рассмотрен метод вложения в байт-код. В спроектированной и реализованной программе имеется модули преобразования текста шифром Вернама и сокрытия его в графическом файле-контейнере. Реализовано тестирование программного продукта.

Выполнено экономическое обоснование затрат разработанного ПО, также представлен расчет мер по безопасности жизнедеятельности.

АҢДАТПА

Дипломдық жоба су арналары туралы құпия ақпараттың ағылуынан қорғау жүйесін қамтыды.

Жобада криптографиялық және стеганографиялық шифрлау әдістеріне салыстырмалы талдау жүргізілді, байт-кодты инвестициялау әдісі қарастырылды.. Бағдарлама мәтіні түрлендіру модуль Вернам шифры әзірленген және графикалық форматта оны жасырынып болады. операциялық терезеде бағдарламасы TXT түрлендіру файл пішімдерінде көзделген деректермен жұмысын жеңілдету мақсатында.

Экономикалық есептеу жүргізіліп, өмір тіршілік қауіпсіздігі бойынша толықтырулар енгізілді.

ANNOTATION

The diploma project is devoted to the system of protection against leakage of confidential information about steganographic channels.

The project conducted a comparative analysis of cryptographic and steganographic encryption methods, considered the method of investing in bytecode. In the projected program there will be a module for converting text by Vernam's cipher and hiding it in a graphical format. In order to facilitate the work with data in the program, the conversion of the txt file formats into a working window of the program is provided.

An economic calculation has been made, and additions have been made for life safety

Содержание

Введение

Использование нелицензионного программного обеспечения наносит ущерб компаниям, занимающихся их разработкой. Для решения данной проблемы используются различные средства, одно из них – это цифровые водяные знаки. Цифровой водяной знак – это специальное сообщение, которое внедряется в каждую копию программу с помощью специальных алгоритмов. Внедренной сообщение позволяет идентифицировать пиратскую копию и найти файл, с которого эта копия была сделана. При внедрении цифрового водяного знака не должна меняться логика работы программы и сам знак не должен быть обнаружимым.

Цифровые водяные знаки являются один из направлений стеганографии. Задачей ЦВЗ является погружение дополнительные сведения (обычно идентификационный код автора) в ПС так, чтобы его нельзя было бы удалить, не ухудшив существенно качество ПС. Факт такого вложения может и обнаруживаться нелегитимными пользователями.

Современная стеганография является цифровой, когда все ПС представляются в цифровой форме, а вложение и извлечение секретной информации производится на ЭВМ.

В данной работе рассматриваются методы сокрытия информации в файлах, исполняемых виртуальной машиной Java. Данная тема выбрана потому, что существует очень мало исследований посвященных вложению информации в исполняемые java-файлы, большинство работ по цифровым водяных знаком посвящены вложения в аудио, видео, графическую информацию, и т.д. К тому большинство работ по цифровых водяных знакам в исполняемых файлах посвящены файлам формата exe.

С точки зрения стеганографии в качестве покрывающего сообщения лучше выбрать файл, обладающий большой избыточностью и разнообразием хранимой информации, что позволяет малозаметно вкладывать большие объёмы информации. С этой точки зрения наиболее выгодными являются графические, мультимедийные и аудиофайлы. В данной работе рассматриваются вложения в исполняемый программный код. Данный тип покрывающего сообщения заведомо известно обладает меньшими возможностями для вложения сообщений, однако целесообразность данного покрывающего сообщения вызвано тем, что вкладываемое сообщение используется совместно с программой, куда оно вкладывается. Примером подобного использования может быть авторские подписи программного кода, счётчики числа установок, данные необходимые для реализации функции защиты от копирования и другая скрытая информация, относящаяся к служебным вопросам функционирования программы.

На сегодняшний день получили широкое распространение системы программирования, которые транслируют программы не в машинный код, а в язык виртуальной машины. Одной из таких виртуальных машин является Java Virtual Machine.

Методы вложения в Java программы, как и любые другие, написанные на языке высокого уровня, можно разделить на две группы: вложения при известном исходном тексте и вложения при неизвестном исходном тексте вложение в исполняемый модуль. В данной работе будут рассматриваться только те методики, которые относятся ко второй группе.

Программы, написанные на языке Java, компилируются в байт-код, выполняемый на виртуальной машине Java (Java Virtual Machine). Компилятор конвертирует исходный текст в Java байт-код, который основан на системе команд виртуальной машины, не зависит от архитектуры конкретного процессора, т. к. исполняется на виртуальной машине.

Исходя из вышеизложенного, следует, что объектом исследования являются исполняемые модули виртуальной машины, так как они отличаются от исполняемых модулей, которые собираются напрямую для использования в конкретной операционной системе.

В данной работе рассматриваются методы скрытого вложения информации в class файлы, исполняемые на виртуальной машине Java. Целью работы является оценка возможного объёма вкладываемой информации, разработка алгоритмов вложения, и их программная реализация.

Практическая значимость данной работы заключается в разработки программного обеспечения, позволяющего организовать скрытое вложение информации в исполняемый код, что позволяет реализовывать защиту авторских прав на программное обеспечение.

1 Компьютерная стеганография

1.1 Выбор покрывающего сообщения

В данном разделе будет описаны причины выбора java-файла в качестве покрывающего сообщения для скрытого вложения информации.

Для начала необходимо, определиться, что такое покрывающее сообщение и из каких соображений его выбирают. В первую очередь выбирается покрывающее сообщение, то есть элемент интерфейса программы, который содержит или может содержать в себе другие элементы интерфейса, а также фрагмент исполняемого файла или исполняемый файл целиком. Покрывающее сообщение объединяет все эти элементы в одно целое, так называемую группу, и отвечает за отображение этих элементов и предоставления возможности по управлению ими. С помощью стеганографических методов, возможно, передавать секретные сообщения, которые встраиваются в покрывающее сообщение, так, что сам факт передачи такого сообщения нельзя обнаружить. Разумеется, при использовании покрывающий сообщений для передачи секретных сообщений и просто для вложения информации (скрытия) от посторонних глаз есть свои преимущества и недостатки. При встраивании сообщений может произойти нарушение естественности покрывающего сообщения и изменение некоторых его свойств, а также потеря полной работоспособности исполняемого файла.

Таким образом, надежность стегосистемы и вероятность обнаружения самого факта передачи скрытого сообщения напрямую зависит от выбора покрывающего сообщения. Известно, что покрывающее сообщение в программировании – это структура, которая позволяет инкапсулировать в себе типы данных различного типа. Наиболее известными покрывающими сообщениями являются те, которые построены на основе шаблонов. Хотя существуют и решения в виде библиотек файлов. До стегокодера – покрывающее сообщение пустое, после – заполненное. Существует два основных типа покрывающих сообщений: фиксированные и потоковые. При использовании потоковых покрывающих сообщений существенную трудность составляет синхронизация начала скрытого сообщения. Также существуют методы, использующиеся для покрывающих сообщений с произвольным доступом. Такие методы предназначены для работы с файлами фиксированной длины (программы, графические, звуковые файлы).

В таком случае размер файла и его содержимое известно заранее.

В данной работе рассматриваются файлы фиксированной длины – исполняемые файлы. Исполняемые файлы содержат код, который запускается при открытии файла. Программы Windows, приложения Mac OS X, скрипты и макросы – все считаются исполняемыми файлами.

Существует два способа вложения информации в исполняемые файлы:

- первый способ – вложение в структуру исполняемого файла и второй;
- второй способ – вложение в исполняемый код.

В первом случае используются имеющиеся у нас знания и документы о форматах исполняемых файлов. Скрываемая информация может вкладываться в участки файла, предназначенные для выравнивания, неиспользуемые поля заголовков файла и другие участки исполняемого файла, изменение которых не влияет на его структуру, а также на работоспособность. При использовании данного способа код не модифицируется, а спецификация форматов исполняемых файлов является общедоступной и открытой. Также файлы генерируются по стандартной схеме, поэтому если использовать данный способ для вложения информации в исполняемые файлы, то он будет легко обнаруживаемым.

Второй способ. В данном случае для вложения используется исполняемый код, содержащийся в файле, а не сам исполняемый файл. То есть изменения будут вноситься в код, который содержит исполняемый файл, не зная его первоначального кода, написанного программистом. Таким образом, приведем пример на исполняемом файле *.jar. JAR – представляет собой zip-архив, в котором содержатся class файлы программы на языке java, а также, чтобы файл функционировал как исполняемый он содержит файл manifest.mf, в котором указан главный класс программы. Именно изменяя class файлы архива, мы получим возможность эффективного и скрытого вложения информации в исполняемый файл.

1.2 Методы вложения в исходный код

Существуют два метода вложения в исполняемые файлы. Первый – это вносить изменения в исходный текст программы, т.е. до компиляции. Второй путь – это внести изменения в уже составленный исполняемый модуль программы так, чтобы структура и логика работы не изменились, или были изменены, но при этом программа работала, как было реализовано автором и выполняла те цели, которые были заложены.

Рассмотрим первый метод. Приведем пример возможного метода вложения. В среду разработки добавляется шифрующий модуль, который заменяет в коде программы операцию сложения тождественной операцией вычитания, и наоборот. Так, например, заменяя операцию сложения некоторого значения со значением числа 2 на операцию вычитания значения -2, можно закодировать один бит информации, принимая сложения за 0, а операцию вычитания – за 1. Подобные замены не повлияют на логику работы программы.

Рассмотрим другой метод, который заключается в том, что ветвления кода программы можно записывать в различном порядке. На рисунке 1.1 приведён пример части алгоритма программы, представленного в виде блоксхемы.

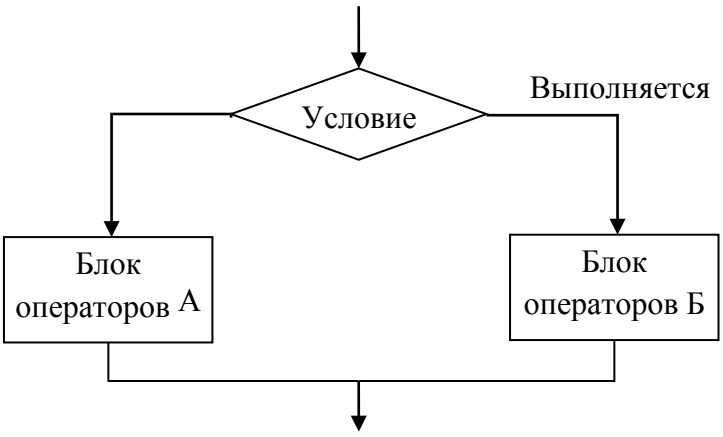


Рисунок 1.1 – Блок-схема ветвления

Если условие выполняется, то произойдёт выполнение «Блока операторов Б», а если не выполняется, то «Блока операторов А». Порядок расположения «Блока операторов Б», и «Блока операторов А», в исходном коде программы может быть различным, но при этом измениться логика работы программы.

Для того что бы сохранить логику ветвления, но при этом иметь возможность изменять расположение «Блоков», достаточно изменять условие на противоположное ему. Например, логическое выражение «a>5» заменить на «a<=5». Взяв выражение со знаком «>» за единицу, а с противоположным ему «<=» за ноль, можно вложить информацию объёмом в один бит. Описанный метод применим как к исходному коду программы, так и к уже собранным исполняемым модулям.

Однако данный метод имеет недостатки. В первых требуется доступ к исходному коду программы, а во-вторых для вложения информации требуется перекомпиляция программы. По этим причинам данный метод не будет использован для реализации вложение цифровых водяных знаков в исполняемые java файлы.

1.3 Методы вложения в исполняемый код

Рассмотрим некоторые существующие работы, касающиеся вложению информации в исполняемые файлы.

Хомяков И.Н. и Красов А.В. в статье «Анализ возможностей скрытого вложения информации в структуру байт-кода» рассматривают варианты с нескольких сторон. Рассматриваются возможность вложения информации с помощью избыточности команд виртуальной машины Java.

Рассматриваются такие методы вложения как:

- Замена существующих ветвлений кода на противоположные, сохраняя при этом логику ветвления.
- Добавление операций присвоения перед существующими.

- Перестановка местами следующих друг за другом операций присвоения, не зависящих друг от друга и очередности присвоения.

И.В. Нечта в статье «Стеганография в файлах формата Portable Executable» рассматривает вложение информации в исполняемые файлы с помощью трех методов:

- перестановка рядом стоящих операций присваивания.
- перестановкаа процедур внутри программы
- перестановки адресов в таблицах импорта, вызывающих функции Windows API.

Приведен сравнительный и экспериментальный анализ всех трех исследуемых методов. Сделан вывод об эффективности использования всех трех методов сразу, так как при экспериментальном анализе такой вариант дал наиболее высокий процент вложения.

В статье «Аутентификация программного обеспечения при помощи вложения цифровых водяных знаков в исполняемый код» авторы Красов А.В., Верещагин А.С. и Цветков А.Ю. Предлагают метод аутентификации исполняемого кода процессоров архитектуры x86 путем вложения цифровой подписи методом замены синонимов. Данный метод может использоваться для проверки целостности отдельных участков исполняемого кода.

Например, А. В. Красов и С. И. Штеренберг в своей статье «Разработка методов защиты от копирования ПО на основе цифровых водяных знаков внедряемых в исполняемые и библиотечные файлы» рассматривают код, как ориентированный граф «… в вершинах которого расположены инструкции, а рёбра соответствуют возможным переходам управления между ними». Также в статье разбираются группы, на которые делятся методы вложения в исполняемые файлы. Предлагается использование замены эквивалентных инструкций, выполняющих одну и ту же операцию и имеющих одинаковую длину.

Выводы

В данной главе был произведен выбор Java файла в качестве покрывающего сообщения, так же был выбран метод вложения информации путем изменения исполняемого файла. Целью данной работы не являлось использование Java-файла, как контейнера для вложения большого количества информации, для этого он не подходит. Цель данной работы в том, чтобы решать задачи совместного использования данной информации с программным обеспечением.

2 Использование байт-кода в качестве покрывающего сообщения.

2.1 Язык программирования Java

Java – объектно-ориентированный язык программирования, разрабатываемый компанией Sun Microsystems. Изначально новый язык программирования разрабатывался для бытовой электроники, но впоследствии стал использоваться для написания апплетов, приложений и серверного программного обеспечения.

Java-апплет – прикладная программа, скомпилированная в байт-код Java. Выполняются Java-апплеты в веб-обозревателе с использованием виртуальной Java машины (JVM), или в Sun's AppletViewer, автономном средстве для испытания апплетов. Используются для предоставления интерактивных возможностей веб-приложений, которые не могут быть предоставлены языком гипертекстовой разметки HTML.

Апплеты были внедрены в первой версии языка. Обычно пишутся на языке программирования Java, но могут быть написаны и на других языках, которые компилируются в байт-код Java, таких, как Jython. Преимущества использования Java-апплетов:

- кроссплатформенность – апплет может работать на «всех» установленных к этому времени версиях Java, а не только с последней версией; однако, если апплет требует последнюю версию JRE, то клиент будет вынужден ждать более длительной загрузки;
- апплет поддерживается большинством браузеров;
- кэшируется в большинстве браузеров, а потому будет быстро загружаться при возвращении на веб-страницу;
- может иметь полный доступ к машине, на которой выполняется, если пользователь согласен на это;
- апплет может улучшить использование: после первого запуска, когда JVM уже выполняется и быстро запускается, преимущественно у постоянных пользователей Java;
- может запуститься с сопоставимой (но обычно медленнее) скоростью на других компилируемых языках, таких как C++, но во много раз быстрее, чем JavaScript;
- может перенести работу с сервера к клиенту, делая интернет вычисления с большим числом пользователей.

Надежность и безопасность Java

Java существенно облегчает создание надежного программного обеспечения. Кроме исчерпывающей проверки на этапе компиляции, система предусматривает анализ на этапе выполнения. Сам язык спроектирован так, чтобы вырабатывать у программиста привычку писать «правильно». Модель работы с памятью, в которой исключено использование указателей, делает невозможным появление целого класса ошибок, характерных для C и C++.

В силу того, что Java предназначена для работы в распределенной среде, безопасность становится чрезвычайно важной проблемой. Требования безопасности определяют многие черты как языка, так и реализации всей системы.

Производительность Java

Схема работы системы и бай-код виртуальной машины Java таковы, что позволяют достичь высокой производительности на этапе выполнения программы:

- анализ кода на соблюдение правил безопасности производится один раз до запуска на выполнение, в момент выполнения таких проверок уже не нужно, и коды выполняются максимально эффективно;
- работа с базовыми типами максимально эффективна, для операций с ними зарезервированы специальные инструкции байт-кода;
- методы в классах не обязательно связываются динамически;
- автоматический сборщик мусора работает отдельным фоновым потоком, не замедляя основную работу программы, но в то же время обеспечивает своевременный возврат свободной памяти в систему;
- стандарт предусматривает возможность написания критических по производительности участков программы в машинных кодах.

Интерпретируемый, многопотоковый и динамический Java

Интерпретируемая природа языка позволяет сделать фазу линкования простой, инкрементальной и, следовательно, быстрой. Это резко сокращает цикл разработки и тестирования программных фрагментов.

Многопотоковость позволяет выполнять в рамках одного приложения несколько задач одновременно. Это становится особенно актуально в современных распределенных приложениях, когда процессы сетевого обмена могут идти одновременно и асинхронно. При этом программа продолжает реагировать на ввод информации пользователем без неприятных задержек.

Многопотоковость поддерживается на уровне языка – часть примитивов синхронизации встроена в систему реального времени, а библиотека содержит базовый класс Thread. К тому же системные библиотеки написаны thread-safe, т.е. все они могут быть использованы в многопотоковых приложениях.

Система обеспечивает динамическую сборку программы. Классы подгружаются по мере необходимости, причем загружены они могут быть с любой точки сети, что позволяет сделать внесение изменений в приложения прозрачным для пользователя. Пользователь может быть уверен, что всегда работает со свежей версией приложения.

Java – универсальный язык, именно поэтому он имеет широкую область применения. Программы, написанные на Java, обладают хорошей переносимостью. Написано однажды будет работать везде не зависимо от того под какой платформой вы работаете Windows, Linux или даже Mcintosh

главное то, что на устройстве должна быть установлена виртуальная машина. Другой важной особенностью языка Java считается то, что на нём пишутся приложения не только для персональных компьютеров, но и для спутников, бытовых устройств и телефонов. И это далеко не все области применения, Java охватывает огромную часть рынка устройств.

2.2 Виртуальная машина Java

Java Virtual Machine (JVM) – виртуальная машина, обрабатывающая байт-код и передающая инструкции оборудованию, как интерпретатор, но с тем отличием, что байт-код, в отличие от исходного текста, обрабатывается значительно быстрее. Виртуальная машина оперирует набором команд (инструкций) байт-кода, набором регистров, стеком, сборщиком мусора и пространством хранения методов.

Типы данных

Типы данных виртуальной машины содержат основные типы данных языка Java:

- byte – 1-байтовое со знаком, представленное в дополнительном обратном коде;
- short – 2-байтовое со знаком, представленное в дополнительном обратном коде;
- int – 4-байтовое со знаком, представленное в дополнительном обратном коде;
- long – 8-байтовое со знаком, представленное в дополнительном обратном коде;
- float – вещественное число одинарной точности стандарта IEEE 754, занимающее 4 байта;
- double – вещественное число двойной точности стандарта IEEE 754, занимающее 8 байт;
- char – символ Unicode, занимающий 2 байта кодировки UTF-16.

Диапазоны значений для целочисленных типов следующие:

- для byte , от -128 до 127 (-2 7 до 2 7 - 1) включительно;
- для short , от -32768 до 32767 (-2 15 до 2 15 - 1) включительно;
- для int , от -2147483648 до 2147483647 (-2 31 до 2 31 - 1) включительно;
- для long , от -9223372036854775808 до 9223372036854775807 (-2 63 до 2 63 - 1) включительно;
- для char , от 0 до 65535 включительно.

Типы с плавающей точкой float и double концептуально связаны с 32битными значениями и операциями одинарной точности и 64-битными в формате двойной точности, указанные в стандарте IEEE для двоичной арифметики с плавающей запятой (ANSI / IEEE Std. 754-1985, Нью-Йорк).

В Java проверка почти всех типов производится во время компиляции. Данные элементарных типов, указанных выше, не требуют аппаратной

поддержки тегов чтобы обеспечить исполнение Java кода. Вместо этого, существуют инструкции байт-кода, которые работают с элементарными значениями и указывают типы операндов, например, каждая из инструкций iadd, ladd, fadd, и dadd складывает два числа имеющих типы int, long, float, и double, соответственно.

Виртуальная машина не содержит отдельных инструкций для типов boolean. Вместо этого, используются инструкции для целых значений, включающие целочисленную инструкцию return, используемую для работы с логическими переменными. Массивы из элементов типа byte используются для массивов из логических элементов, т.е. в байт-коде представляются инструкциями, обрабатывающими значения типа byte.

Иные типы данных виртуальной машины:

- object – ссылка на объект Java, занимающая 4 байта;
- returnAddress – 4 байта, используемые с инструкциями jsr, ret, jsr_w, ret_w.

Стоит обратить внимание на то, что в Java массивы трактуются как объекты.

Программы, представленные байт-кодом виртуальной машины Java, предполагают соблюдение соответствующих правил, относящихся к значениям и операциями над ними вышеописанных типов. Виртуальная машина может отказаться выполнять байт-код программы, если есть вероятность того, что нарушены эти правила.

Регистры

Виртуальная машина Java поддерживает выполнение нескольких потоков одновременно. У каждого потока есть свой собственный PC (program counter). В любой момент времени виртуальная машина выполняет код отдельного метода, и регистр PC содержит адрес следующего байт-кода, который будет выполняться. Для каждого метода выделяется место в памяти для хранения набора локальных переменных, стека операндов и структуры среды выполнения.

Все эти места могут быть отведены сразу, начиная с размера локальных переменных и стека операндов, определяемых во время компиляции, и размера структуры окружающей среды выполнения известного интерпретатору.

Стеки виртуальной машины

У каждого виртуального потока имеется частный Java Virtual Machine stack (далее стек), созданный в то же время, что и поток. Стек хранит фреймы. Стек виртуальной машины Java подобен стеку обычного языка, такого как C: он содержит локальные переменные и промежуточные результаты вычислений, и используется для передачи значений параметров при вызове метода и возврате из него. Например, инструкция iadd складывает два целых. Для этого ей необходимо, чтобы на вершине стека присутствовали

два целых значения, помещенные туда предыдущими инструкциями. Оба целых значения извлекаются из стека, складываются, и их сумма помещается обратно в стек операндов. Хранение промежуточных результатов обеспечивает поддержку вложенных вычислений.

Каждый простейший тип данных имеет специализированные инструкции, которые «знают» как работать с операндами конкретного типа. Каждый операнд требует одной ячейки на стеке, кроме long и double, которым необходимо две ячейки.

Операнды должны использоваться инструкциями, соответствующими их типу. Например, нельзя помещать на стек два целых и потом использовать их как длинный целый. в виртуальной машине это ограничение контролируется верификатором байт-кода. Однако, некоторые операции (коды операций dup и swap) работают с областями данных как со значениями данной ширины, не обращая внимания на тип.

Куча

У виртуальной машины Java есть «куча» (heap), которая совместно используется всеми потоками. «Куча» является областью данных времени выполнения, в которой выделяется память для всех экземпляров класса и массивов.

«Куча» создается при запуске виртуальной машины. Хранение объектов в «куче» управляется автоматической системой управления и хранения (известный как сборщик «мусора»).

Область метода

У виртуальной машины Java есть область метода, доступ к которой имеют все потоки виртуальной машины. Область метода походит на область хранения для скомпилированного кода стандартного языка или аналогичный «текстовому» сегменту в процессе UNIX. Область хранит структуры класса, такие как пул констант этапа выполнения, поле и данные метода, код для методов и конструкторов, включая специальные методы, используемые в классе, инициализации экземпляра, интерфейсной инициализации типа.

Локальные переменные

Каждый фрейм содержит массив переменных, известных как его локальные переменные. Длина массива локальных переменных фрейма определяется во время компиляции и хранится в двоичном представлении класса или интерфейса наряду с кодом для метода, связанного с фреймом.

Единственная локальная переменная может содержать значение типа boolean, byte, char, short, int, float, reference, или returnAddress. Пара локальных переменных может содержать значение типа long или double.

Обращение к локальной переменной осуществляется по её индексу. Индекс первой локальной переменной является нулем.

Значение типа long или double занимает две последовательных локальных переменных. Обращение к подобному значению осуществляется только по меньшему индексу, отведённому под данное значение. Например, значение типа double сохраненное в массиве локальных переменных по индексу n фактически занимает локальные переменные с индексами n и n +1; однако, локальная переменная по индексу n +1 не может быть загружена.

Процесс проверки байт-кода

Несмотря на то, что компилятор гарантирует, что коды не нарушают требований безопасности, если они были получены из других точек сети возникает следующая проблема: коды могут быть созданы не компилятором Java, а другими средствами. Или они могут быть намеренно модифицированы после создания. Поэтому run-time система подвергает полученные коды тщательной проверке. Проверка включает в себя несколько этапов, начиная с контроля целостности формата полученного файла до анализа каждого фрагмента кодов на предмет выполнения следующих правил:

- нет незаконных манипуляций с указателями;
- нет попыток нарушения прав доступа;
- объекты используются в строгом соответствии с их типами, например, объекты класса InputStream используются только как InputStream и никак иначе.

Верификатор байт-кодов

Верификатор байт-кодов (bytecode verifier) сканирует байт-код, извлекает информацию о типах объектов в каждой точке выполнения фрагмента кода. Важно отметить, что загрузчик и верификатор байт-кодов не делают никаких предположений относительно происхождения кодов – получены они с локальной файловой системы или с другого континента. Верификатор гарантирует, что любой код, прошедший проверку, может быть использован интерпретатором без риска повредить его (интерпретатор), а именно:

- не может произойти переполнение или «исчерпание» стека;
- параметры для инструкций имеют нужный тип;
- доступ к полям и методам объектов не нарушает объявленных в классе правил (public, private, protected).

2.3 Структура java кода

Система Java создана на основе классического объектноориентированного языка программирования, техника использования которого близка к общепринятой и обучение которому не требует значительных усилий.

Язык программирования Java является объектно-ориентированным с момента основания. Кроме того программист с самого начала обеспечивается

набором стандартных библиотек, обеспечивающих функциональность от стандартного ввода/вывода и сетевых протоколов до графических пользовательских интерфейсов. Эти библиотеки легко могут быть расширены.

Язык Java является полностью объектно-ориентированным. Это означает, что любая программа, написанная на языке Java, должна поддерживать парадигму объектно-ориентированного программирования. В отличие от традиционного процедурного программирования, объектноориентированные программы подразумевают описание классов и, как правило, создание объектов. На сегодняшний день существует несколько наиболее популярных языков программирования, поддерживающих концепцию ООП. В первую очередь это C++, C# и Java.

Чтобы решить проблему упорядочивания программного кода, было принято решение ввести четкое разграничение данных и методов обработки этих данных. Более того, данные и соответствующие им методы объединили в одну структуру, которая в ООП называется объектом.

Такой на первый взгляд искусственный прием позволяет четко разграничить область применимости методов. Вся программа при этом имеет блочную структуру, что существенно упрощает анализ программного кода. Но даже в таком подходе было бы мало пользы, если бы каждый объект был абсолютно уникальным. Практика же такова, что каждый объект определяется некоторым общим шаблоном, который называется классом. В рамках класса задается общий шаблон, то есть структура, на основе которой затем создаются объекты. Данные, относящиеся к классу, называются полями класса, а программный код для их обработки – методами класса.

В классе описывается, какого типа данные относятся к классу (данные называются полями класса), а также то, какие методы применяются к этим данным. Затем в программе на основе того или иного класса создается экземпляр класса (объект), в котором указываются конкретные значения полей и выполняются необходимые действия над ними. Синтаксис языка Java максимально приближен к синтаксису C++. Это делает язык знакомым широкому кругу программистов. В то же время из языка были удалены многие свойства, которые делают C++ излишне сложным для использования, не являясь абсолютно необходимыми. В результате язык Java получился более простым и органичным, чем C++.

Кодировка в языке

Исключая комментарии, идентификаторы, содержимое символьных и строковых литералов, все элементы ввода в программе на языке Java образуются только из символов ASCII. ASCII (ANSI X3. 4) - это американский стандартный код для обмена информацией (the American Standard Code for Information Interchange). Первые 128 Unicode-символов - символы ASCII.

Лексическая трансляция

Unicode-символы преобразуются в последовательность лексем языка Java, с помощью следующих трех шагов лексической трансляции, которые применяются последовательно:

1) трансляция Unicode-последовательностей в соответствующие Unicodeсимволы. Unicode-последовательность формы \ uxxxx, где xxxx - шестнадцатеричное число, представляет Unicode-символ, чьим кодом является - xxxx. Этот шаг позволяет представить любую программу на языке Java, используя только ASCII-символы;

2) трансляция Unicode-последовательности, являющейся результатом шага 1, в последовательность входных символов и ограничителей

строк;

3) трансляция последовательности входных символов и ограничителей строк, являющейся результатом шага 2, в последовательность элементов ввода языка Java, которая после исключения незначащих символов и комментариев состоит из лексем, которые являются терминальными символами синтаксической грамматики языка Java.

Язык Java всегда использует самую длинную возможную трансляцию на каждом шаге, даже если результат, в конечном счете, не является корректной программой на языке Java, в то время как при другой лексической трансляции, возможно, являлся бы. Таким образом, входные символы a -- b интерпретируются как последовательность a, --, b, которая не является частью любой грамматически правильной Java-программы, хотя последовательность a, -, -, b могла бы быть частью грамматически правильной программы на языке Java.

Ограничители строк

Компилятор языка Java делит последовательность входных Unicodeсимволов на строки, распознавая ограничители строк. Это разделение определяет номера строк, формируемые компилятором языка или другими компонентами Java системы, определяет границы // комментария. ASCII символы окончания строки:

- LF, также называемый «новая строка»;
- CR, также называемый «возврат в начало строки»;
- CR следующий за символом LF.

Строки программы заканчиваются ASCII-символами CR, или LF, или CRLF. Два подряд идущих символа CR и LF являются одним ограничителем строки, а не двумя. Результат - последовательность разделителей строк и входных символов, которые являются терминальными символами третьего шага лексического анализа.

Элементы ввода и лексемы

Входные символы и ограничители строк, которые получились после обработки Unicode-последовательностей и последующего распознавания

строк, преобразуются в последовательности элементов ввода. Элементы ввода, которые не являются незначащими символами или комментариями являются лексемами. Лексемы - это терминальные символы синтаксической грамматики языка Java.

Незначащие символы и комментарии служат для того, чтобы разделять лексемы, которые при слитном написании могли бы интерпретироваться иначе. Например, ASCII-символы - и = могут формировать знак операции –= только в случае, если между ними нет никакого незначащего символа или комментария.

Рассмотрим два символа x и y, стоящие во входной последовательности. Если x предшествует y, то мы говорим, что x стоит слева от y, а y - справа от x. Например, как в этой простом фрагменте кода на языке Java:

```
class Empty {
}
```

мы говорим, что лексема } стоит справа от лексемы {, даже если она появляется ниже и левее лексемы { , как в вышеприведенном примере. Это соглашение относительно использования слов левый и правый позволяет нам говорить, например, о правом операнде двуместной операции или левой части оператора присваивания.

Следующие 37 лексем, составленные из исмволов ASCII, являются знаками операций языка Java:

```
=    >    <    !    ~    ?    :
==   <=   >=   !=   &&   ||   ++   --
+    -    *    /    &    |    ^    %    <<   >>   >>>
+=   -=   *=   /=   &=   |=   ^=   %=   <<=  >>=  >>>=
```

Разделителями (знаки пунктуации) являются следующие 9 знаков:

```
(    )    {    }    [    ]    ;    ,    .
```

Незначащие символы

Незначащие символы определяются как ASCII - символы: пробел (SP), горизонтальная табуляция (HT), символ перевода страницы (FF), а также ограничители строки.

Комментарии

В языке Java выделяют три вида комментариев:

1) /* текст */ – традиционный комментарий: весь текст от ASCII-символов /* до ASCII-символов */ игнорируется (так же как в C и C ++);

2) // текст – однострочный комментарий: весь текст от ASCII-символов // до конца строки игнорируется (как в C ++);

3) /** документация */ – комментарий документации: текст, заключенный в ASCII-символы /** и */ может быть обработан отдельной программой, автоматически формирующей документацию объявления данного класса, интерфейса, конструктора, или члена (метод или поле).

Свойства комментариев:

- комментарии не могут быть вложенными;
- /* и */ не имеют никакого специального значения в комментариях, которые начинаются с //;
- // – не имеет никакого специального значения в комментариях, которые начинаются с /* или /**.

Лексическая грамматика предполагает, что комментарии не действуют в пределах символьных или строковых литералов.

Отметим, что /**/ рассматривается как комментарий документации, в то время как /* */ (с пробелом между звездочками) - традиционный комментарий.

Идентификаторы

Идентификатор - это последовательность неограниченной длины букв и цифр языка Java, на первом месте в этой последовательности должна быть буква. Идентификатор не может иметь ту же самую последовательность Unicode-символов, что и ключевое слово, логический литерал или nullлитерал.

Буква в языке Java - это символ, для которого метод Character.isJavaLetter возвращает значение true. Буква или цифра Java - это символ, для которого метод Character.isJavaLetterOrDigit возвращает значение true.

Ключевые слова

Следующие последовательности символов, сформированные из символов таблицы ASCII, являются ключевыми словами и не могут использоваться как идентификаторы: abstract, default if, private, throw, boolean, do, implements, protected, throws break, double, import, public, transient, byte, else, instanceof, return, try, case, extends, int short, void, catch, final, interface, static, volatile, char, finally, long, super, while, class, float, native, switch, const, for, new, synchronized, continue, goto, package, this. Ключевые слова const и goto зарезервированы языком Java, даже если в настоящее время они не используются. Это позволяет компилятору Java лучше представлять сообщения об ошибке, если эти ключевые слова языка C++ неправильно применяются в программах на зыке Java.

Несмотря на то, что слова true и false могли бы использоваться как ключевые слова, их относят к логическим литералам. Подобно этому, несмотря на то, что слово null могло использоваться как ключевое, оно является null-литералом.

Литералы

Литерал - представление в исходном коде значения простого типа, типа String или типа null:

- IntegerLiteral;
- FloatingPointLiteral;

- BooleanLiteral;
- CharacterLiteral;
- StringLiteral;
- NullLiteral.

Целый литерал может быть выражен десятичным (основание 10), шестнадцатеричным (основание 16) или восьмеричным (основание 8) числом:

- DecimalIntegerLiteral;
- HexIntegerLiteral;
- OctalIntegerLiteral.

Целый литерал имеет тип long, если он заканчивается ASCII-символом L или l; иначе, это литерал типа int. Суффикс L предпочтителен, потому что букву l (эль) часто трудно отличить от цифры 1.

Десятичная запись числа является либо простым ASCII-символом 0, представляющим ноль, либо состоит из ASCII-символов от 1 до 9, за которой могут следовать одна или больше ASCII-символов от 0 до 9, представляющих положительное целое число.

Шестнадцатеричная запись числа состоит из ASCII-символов 0x или 0X, идущих перед одной либо несколькими ASCII-шестнадцатиричными цифрами и может представлять положительное целое число, ноль или отрицательное целое число. Шестнадцатеричные цифры со значениями от 10 до 15 представляются ASCII-символами от A до F в указанном порядке, каждая буква, использующаяся как шестнадцатеричная цифра, может быть из верхнего или нижнего регистра.

Заметим, что восьмеричные числа всегда состоят из двух или более цифр; 0 всегда рассматривается как десятичное число - что не имеет большого значения на практике. Числа 0, 00, и 0x0 представляют одно и то же целое значение.

Самый большой десятичный литерал типа int – 2147483648 (2^{31}). Все десятичные литералы от 0 до 2147483647 могут применяться везде, где может применяться литерал типа int, но литерал 2147483648 может использоваться только как операнд одноместной операции отрицания - .

Самые большие положительные шестнадцатеричный и восьмеричный литералы типа int - 0x7fffffff и 017777777777, соответственно, которые равны 2147483647 (2^{31}-1). Наибольшие отрицательные шестнадцатеричные и восьмеричные литералы типа int 0x80000000 и 020000000000, соответственно, каждый из которых представляет десятичное значение 2147483648 (-2^{31}). Шестнадцатеричный и восьмеричный литералы 0xffffffff и 037777777777, соответственно, представляют десятичное значение -1.

Аналогично представлены значения типа Long, с поправкой на разрядность значений этого типа.

Вещественный литерал имеет следующие части: целая часть, десятичная точка (представлена ASCII-символом «точка»), дробная часть, экспонента, и суффикс типа. Экспонента обозначается ASCII-символом e или E, сопровождаемой целым числом (возможно со знаком).

По крайней мере одна цифра в целой или в дробной части и, либо десятичная точка, либо экспонента, либо плавающий суффикс типа, необходимы.
Все другие части необязательны.

Вещественный литерал имеет тип float, если он заканчивается ASCIIсимволом F или f; иначе он имеет тип double, и может заканчиваться ASCIIсимволом D или d.

Самый большой положительный конечный литерал типа float – 3.40282347e+38f. Самый маленький положительный конечный литерал отличный от нуля типа float – 1.40239846e-45f. Самый большой положительный конечный литерал типа double – 1.79769313486231570e+308. Самый маленький положительный конечный литерал отличный от нуля типа double – 4.94065645841246544e-324.

Тип boolean (логический тип) имеет два значения, представленные литералами true и false, сформированными из ASCII-символов.

Логический литерал всегда имеет тип boolean.

Символьный литерал представляется в виде символа или Unicodeпоследовательности, заключенной в ASCII одиночные кавычки. (Одиночная кавычка, или апостроф, знак \u0027.)

Символьный литерал всегда имеет тип char.

В C и C ++ символьный литерал может содержать более одного символа, но значение такого символьного литерала зависит от реализации. В языке Java символьный литерал всегда представляет ровно один символ.

Строковый литерал состоит из ноля или большего количества символов, заключенных в двойные кавычки. Каждый символ может быть представлен Unicode - последовательностью.

Длинный строковый литерал всегда может быть разбит на более короткие части и написан как выражение, использующее оператор конкатенации строки +.Далее следуют примеры строковых литералов:

- "" – пустая строка
- "\"" – строка, содержащая одну "
- "This is a string" – строка, содержащая 16 символов
- "This is a" + "two-line string" – фактически, строковое константное выражение, сформированное из двух строковых литералов

Типы переменных и значения

Java – строго типизированный язык, это означает, что каждая переменная и каждое выражение имеет тип, который известен во время компиляции. Типы ограничивают значения, которые может принимать переменная, и значения, которые могут быть результатом выражения, ограничивают набор операций, применимых к этим значениям, и определяют смысл операций. Строгая типизация помогает обнаруживать ошибки во время компиляции.

Имеются два вида типов в языке Java: примитивные типы и ссылочные типы. Имеется соответственно два вида значений данных, которые могут быть сохранены в переменных, переданы как параметры, возвращены методами, и вычислены: примитивные значения и значения ссылки.

Значения целочисленных типов - целые числа, лежащие в следующих диапазонах:

- для byte, от -128 до 127, включительно;
- для short, от -32768 до 32767, включительно;
- для int, от -2147483648 до 2147483647, включительно;
- для long, от -9223372036854775808 до 9223372036854775807, включительно;
- для char, от '\u0000' до '\uffff включительно, т.е. от 0 до 65535.

Тип boolean позволяет представить логическую величину, которая может принимать два значения true (истинный) или false (ложный). К типу boolean применимы следующие операции:

- операции отношения (== и !=);
- операция логического отрицания (!);
- логические операции (&, ^, и |);
- операции условное И (&&) и условное ИЛИ (||);
- условная операция (?: или оператор if);
- операция конкатенации строк (+), которая, когда дан операнд типа String и логический операнд, преобразует данный логический операнд в операнд типа String (либо "true", либо "false"), созданная таким образом строка и будет конкатенацией двух строк.

Логические выражения определяют поток управления в некоторых видах операторов:

- В операторе if(?:);
- В операторе while;
- В операторе do;
- В операторе for.

Существует три вида ссылочных типов: классовый тип, интерфейсный тип, и тип массив. Объект это экземпляр класса или массив. Ссылочные значения (часто называемые ссылками) - это указатели на объекты, и специальная ссылка типа null, которая не ссылается ни на какой объект.

Стандартный класс Object – суперкласс всех других классов. Переменная типа Object может содержать ссылку на любой объект, который является экземпляром класса или массив. Все классы и массивы наследуют методы класса Object, которые перечислены далее в приложении А.

Члены Object следующие:

- Метод getClass возвращает объект Class, который представляет класс объекта. Объект Class существует для каждого ссылочного типа. Это свойство может использоваться, например, для того чтобы обнаружить полностью квалифицированное имя класса, его членов, его

непосредственный суперкласс, и любые интерфейсы, которые им реализованы.

- Метод toString возвращает строковое представление объекта.
- Методы equals и hashCode объявлены специально для перемешанных таблиц таких как java.util.Hashtable.
- Метод equals определяет понятие равенства объекта, которое основывается на сравнении значения, а не ссылки.
- Метод clone используется для создания дубликата объекта.
- Методы wait, notify, и notifyAll используются в параллельном программировании, т.е. при использовании потоков.
- Метод finalize выполняется только до того как объект уничтожается.

Объявления

Объявление вводит объект в программу на языке Java и включает идентификатор, который может быть использован в имени для обращения к этому объекту. Объявленный объект может быть одним из следующих:

a) пакет, указанный в объявлении пакета package;
b) импортированный тип, объявленный в описании одиночного импорта типа, или в описании импорта типа по шаблону;
c) класс, объявленный в описании классового типа • Интерфейс, объявленный в описании интерфейсного типа;
d) член ссылочного типа, один из следующего:
   поле, одно из следующего:
   1) поле объявленное в классовом типе;
   2) константное поле объявленное в интерфейсном типе;
   3) поле length, которое, неявно является членом каждого типа массив ○ метод, один из следующих;
   4) метод (абстрактный или иной) объявленный в классовом типе;
   5) метод (всегда абстрактный) объявленный в интерфейсном типе:
e) Параметр, один из следующего:
   1) параметр метода или конструктора класса ○ параметр абстрактного метода интерфейса
   2) параметр обработчика исключительных ситуаций, объявленный в операторе catch оператора try.
f) Локальная переменная, одна из следующих:
   1) локальная переменная, объявленная в блоке ○ локальная переменная, объявленная в операторе for Конструкторы также вводятся объявлениями, но используют имя того класса, в котором они объявлены, а не новое имя.

Члены и наследование

Пакеты и ссылочные типы содержат члены. Члены пакета, это подпакеты и все классовые и интерфейсные типы, объявленные во всех

модулях компиляции пакета. Члены ссылочного типа - это поля и методы. Члены объявляются в типе, но могут быть унаследованы, потому что они являются доступными членами суперкласса или суперинтерфейса, и не являются, ни скрытыми, ни переопределенными.

Квалифицированные имена и управление доступом

Квалифицированные имена - средства доступа к членам пакетов и ссылочных типов; средствами доступа являются выражения доступа к полю и выражения вызова метода. Все три средства доступа синтаксически похожи тем, что появляется лексема “.”, которой предшествует некоторый указатель на пакет, тип или выражение, имеющее тип, за которой следует Identifier, и этот идентификатор является именем члена пакета или типа. Они все вместе известны как конструкции квалифицированного доступа.

Язык Java предусматривает механизмы управления доступом, устраняющие зависимость пользователей, работающих с пакетами или классами, от деталей реализации этого пакета или класса. Управление доступом применяется к квалифицированному доступу и вызову конструкторов с помощью выражений создания экземпляров класса, явных вызовов конструкторов и метода newInstance класса.

Если доступ разрешен, тогда говорят, что объект является доступным. Определение и свойства доступа:

a) Доступ к пакету определяется базовой системой.
b) Если классовый или интерфейсный тип объявлен с модификатором доступа public, тогда он доступен в любом коде на языке Java, который имеет доступ к пакету, в котором этот тип объявлен. Если классовый или интерфейсный тип не объявлен с модификатором доступа public, тогда к нему можно обратиться только в том пакете, в котором он объявлен.
c) Член (поле или метод) ссылочного типа (класса, интерфейса или массива) или конструктор классового типа доступен только в том случае, если тип общедоступен и к члену или конструктору при объявлении разрешен доступ:
    1) Если член или конструктор объявлен с модификатором доступа public, тогда доступ к нему разрешен. Все члены интерфейсов по умолчанию имеют модификатор доступа public.
    2) Иначе, если член или конструктор объявлен с модификатором доступа protected, тогда доступ разрешается, когда один из следующих пунктов является истинным:
        1) Доступ к члену или конструктору происходит внутри пакета, содержащего класс, в котором объявлен член с модификатором доступа protected.
        2) Доступ происходит в пределах подкласса класса, в котором объявлен член с модификатором доступа protected, и доступ является корректным.

2) Иначе, если член или конструктор объявлен с модификатором доступа private, тогда доступ к нему разрешается только внутри класса, в котором он объявлен. ○ Иначе мы говорим, что имеет место доступ по умолчанию, который разрешен только тогда, когда обращение происходит внутри пакета, в котором объявлен тип.

Член или конструктор класса с модификатором доступа private доступен только внутри тела класса, в котором данный член объявлен, и не наследуется подклассами.

Пакеты

Программы на языке Ява организованы как наборы пакетов. Каждый пакет имеет собственный набор имен для типов, который помогает предотвращать конфликты между именами. Тип доступен вне пакета, в котором он объявлен, только если этот тип объявлен как public.

Структура наименования для пакетов является иерархической. Элементы пакета - это классовые и интерфейсные типы, которые объявлены в модулях компиляции пакета, и подпакетах, которые могут содержать модули компиляции и их собственные подпакеты.

Тело метода

Тело метода является блоком кода, который осуществляет метод или просто точка с запятой, указывающая на отсутствие реализации. Тело метода должно содержать точку с запятой тогда и только тогда, когда метод объявлен abstract или native.

2.4 Структура исполняемого файла

Чтобы приступить к поиску возможных мест вложения в исполняемых файлах виртуальной машины Java, нужно разобраться не только как работает виртуальная машина, но и, естественно, рассмотреть структуру файлов, исполняемых виртуальной машиной.

Скомпилированный код, который будет выполняться виртуальной машиной Java, сохраняется в файле, известный как class файла. Class файл точно определяет представление класса или интерфейса, включая детали, такие как порядок байтов, который мог бы считаться само собой разумеющимся в специфичном для платформы формате объектных файлов.

Class файл состоит из потока 8-разрядных байт. Все 16-разрядные, 32разрядные и 64-разрядные значения создаются при чтении в два, четыре, и восемь последовательных 8-разрядных байт, соответственно. Многобайтовые элементы данных всегда сохранены в порядке с обратным порядком байт, где высокие байты на первом месте. В Java и Java 2 платформы, этот формат поддерживается интерфейсами java.io.DataInput и java.io.DataOutput и классы такой, как java.io.DataInputStream и java.io.DataOutputStream.

Таблица 2.1 – Структура class файла

Количество отведённых байт	**Элемент структуры**
4	идентификатор формата файла класса (магическое число)
2	вспомогательная версия
2	основная версия
2	количество констант в пуле +1
переменное значение	пул констант
2	флаги доступа
2	ссылка на константу с названием класса в пуле
2	ссылка на константу с названием суперкласса
2	количество интерфейсов
переменное значение	массив интерфейсов
2	количество полей
переменное значение	массив полей
2	количество методов
переменное значение	массив методов
2	количество атрибутов
переменное значение	массив атрибутов класса

Версии

Значения вспомогательной и основной версий определяют версию class файла. Если у class файла есть номер основной версии M и номер вспомогательной версии m, обозначаем версию class формат файла как M.m. Таким образом, class версии формата файла могут быть упорядочены лексикографически.

В настоящее время используются версии class файлов 50 и 51. Данные class файлы без проблем интерпретируются виртуальной машиной версии 1.6 и выше. С версиями виртуальных машин более раннего выпуска существуют несоответствия в структуре class файлов, соответствующих виртуальной машине. Данные несоответствия описаны в четвёртой главе спецификации виртуальной машины Java.

Пул констант

Пул констант представлен в виде таблицы структур представления различных строковых констант, имен классов и интерфейсов, имен полей, и других констант, которые упоминаются в пределах class файла. Формат каждого элемента таблицы пула определяется первым байтом.

Флаги доступа

Значение элемента является маской флагов, используемых таким образом, чтобы обозначить права доступа и свойства этого класса или интерфейса. Возможные флаги доступа представлены в таблице 2.2.

Таблица 2.2 – Флаги доступа class файла

Имя флага	Значение	Интерпретация
ACC_PUBLIC	0x0001	Объявлен public – может быть получен доступ из вне
ACC_FINAL	0x0010	Объявлен final – никакие подклассы не допускаются.
ACC_SUPER	0x0020	Методы суперкласса.
ACC_INTERFACE	0x0200	Интерфейс, не класс.
ACC_ABSTRACT	0x0400	Объявлен abstract – не должен быть создан.
ACC_SYNTHETIC	0x1000	Объявлен synthetic – отсутствует в исходном коде.
ACC_ANNOTATION	0x2000	Объявлен в качестве типа аннотации.
ACC_ENUM	0x4000	Объявлен в качестве enum типа.

Интерфейсы

Элемент представляет собой массив интерфейсов. Каждый элемент массива является индексом таблицы пула констант, что указывает на имя суперинтерфейса этого класса.

Поля

Элемент представляет собой таблицу полей. Все элементы данной таблицы имеют общую структуру. Структура дает полное описание поля в этом классе или интерфейсе. Таблица включает только те поля, которые объявленные этим класса или интерфейсом, и не включает элементы, представляющие поля, унаследованные от суперкласса или суперинтерфейса.

Методы

Элемент представляет собой таблицу методов. Каждое значение таблицы имеет общую структуру, дающую полное описание метода в этом классе или интерфейсе. Так же присутствуют инструкции байт-кода, реализующие работу соответствующего метода. Каждый метод может содержать неограниченное количество атрибутов.

Структуры представляют все методы, объявленные данным классом или интерфейсом, за исключением методов, которые наследованы от суперклассов или суперинтерфейсов.

Атрибуты

Элемент представляет собой таблицу атрибутов. Каждому атрибуту соответствует своя структура. Наборы атрибутов в различных версиях виртуальных машин могут отличаться.

Если реализация виртуальной машины Java распознаёт class файлы, версии 49.0 или выше, это означат что присутствует поддержка атрибутов: Signature, RuntimeVisibleAnnotations и RuntimeInvisibleAnnotations, то есть данные атрибуты могут присутствовать в class файлах данной версии.

Если же реализация виртуальной машины Java распознаёт class файлы, версии номер 51.0 или выше, это означат что поддерживается атрибут BootstrapMethods.

2.5 Структура методов исполняемого файла

Каждый метод, в том числе и методы инициализации экземпляра класса или интерфейса, описывается структурой method_info. Имя каждого метода является оригинальным. Существование второго метода с тем же именем в одном class файле невозможно.

Структура method_info имеет следующий формат:

- 2 байта – флаги доступа;
- 2 байта – индекс имени метода;
- 2 байта – индекс дескриптора;
- 2 байта – количество атрибутов метода;
- переменной длины массив атрибутов, структуры attribute_info.

Флаги доступа

Элемент флаги доступа является маской флагов, используемых для обозначения прав доступа к свойствам данного метода.

Таблица 2.3 – Представление флагов доступа

Имя флага	Значение
ACC_PUBLIC	0x0001
ACC_PRIVATE	0x0002
ACC_PROTECTED	0x0004
ACC_STATIC	0x0008
ACC_FINAL	0x0010
ACC_SYNCHRONIZED	0x0020
ACC_BRIDGE	0x0040
ACC_VARARGS	0x0080
ACC_NATIVE	0x0100
ACC_ABSTRACT	0x0400

ACC_STRICT	0x0800
ACC_SYNTHETIC	0x1000

Индекс имени

Значение индекса является действительным индексом пула констант. Запись пула констант по данному индексу является символьной строкой, не представляющей один из специальных имен методов “init” или “clinit”.

Индекс дескриптора

Значение индекса является действительным индексом пула констант. Запись пула констант по данному индексу является символьной строкой, представляющей действительный дескриптор метода.

Атрибуты

Каждый элемент массива атрибутов имеет соответствующую имени атрибута структуру. Метод может содержать любое количество дополнительных атрибутов, связанных с ним.

Атрибуты, на данный момент определенные в спецификации, которые могут присутствовать в методе следующие: Code, Exceptions, Synthetic, Signature, Deprecated, RuntimeVisibleAnnotations, RuntimeInvisibleAnnotations, RuntimeVisibleParameterAnnotations, RuntimeInvisibleParameterAnnotations, и AnnotationDefault.

Виртуальная машина распознаёт и соответственно спецификации интерпретирует атрибуты Code и Exceptions, найденные в атрибутах таблице структуры method_info. Реализация виртуальной машины Java, поддерживающая class файлы версии номер 49.0 или выше, поддерживает атрибуты: Signature, RuntimeVisibleAnnotations, RuntimeInvisibleAnnotations, RuntimeVisibleParameterAnnotations, RuntimeInvisibleParameterAnnotations и AnnotationDefault.

Виртуальная машина игнорирует любые атрибуты таблицы атрибутов method_info структуры, которые не поддерживает. Атрибуты, отсутствующие в спецификации, допускаются только для того, чтобы обеспечить дополнительную описательную информацию.

2.6 Атрибуты

Атрибуты используются в структуре полей, method_info и атрибутах атрибута Code. Все атрибуты имеют следующий формат attribute_info:

- 2 байт – индекс имени атрибута; • 4 байта – длина атрибута в байтах;
- 1 байт – информация.

Индекс имени атрибута является беззнаковым целым числом. Два байта, выделенные на индекс, являются номером строковой константы в пуле констант, представляющие имя атрибута.
Значение длины атрибута указывает на последующую длину информации в байтах. Длина не включает в себя первые шесть байт индекса имени и длины атрибута.

Таблица 2.4 – Стандартные атрибуты class файла

Атрибут	**Java SE**	**Версия class файла**
ConstantValue	1.0.2	45.3
Code	1.0.2	45.3
StackMapTable	6	50.0
Exceptions	1.0.2	45.3
InnerClasses	1.1	45.3
EnclosingMethod	5.0	49.0
Synthetic	1.1	45.3
Signature	5.0	49.0
SourceFile	1.0.2	45.3
SourceDebugExtension	5.0	49.0
LineNumberTable	1.0.2	45.3
LocalVariableTable	1.0.2	45.3
LocalVariableTypeTable	5.0	49.0
Deprecated	1.1	45.3
RuntimeVisibleAnnotations	5.0	49.0
RuntimeInvisibleAnnotations	5.0	49.0
RuntimeVisibleParameterAnnotations	5.0	49.0
RuntimeInvisibleParameterAnnotations	5.0	49.0
AnnotationDefault	5.0	49.0
BootstrapMethods	7	51.0

Оригинальной фирменной (официальной версии компании ORACLE) спецификаций допускается использование в class файлах дополнительных атрибутов, например, идентификаторов, описания специфичной (неоригинальной) виртуальной машиной. Однако любой подобный атрибут, присутствующий в class файле, не должен влиять на семантику типов класса или интерфейса.

Атрибут Code

Code является атрибутом переменной длины. Атрибут содержит инструкции виртуальной машины Java и вспомогательную информацию для конкретного метода, инициализации экземпляра метода, или метода инициализации класса или интерфейса. Любая виртуальная машина Java распознает данный атрибут. Если метод является native или abstract , его структура не содержит атрибут Code. Любой метод может содержать не более одного данного атрибута.

Структура атрибута следующая:

- 2 байта – индекс имени атрибута;
- 4 байта – длина атрибута;
- 2 байта – максимальный размер стека;
- 2 байта – максимальное количество локальных переменных;
- 4 байта – длина кода;
- Код метода;
- 2 байта – длина таблицы исключений в байтах;
- таблица исключений;
- 2 байта – количество атрибутов;
- таблица атрибутов.

Индекс имени атрибута. Значение индекса имени атрибута является действительным индексом пула констант. Элемент, соответствующий индексу пула констант, является символьной строкой "Code".

Длина атрибута. Указывает на длину атрибута за исключением первых шести байт.

Максимальный размер стека. Значение указывает на максимальную глубину стека операндов этого метода в любой точке во время его выполнения.

Максимальное количество локальных переменных. Значение указывает на число локальных переменных, используемых при вызове этого метода, в том числе локальных переменных, используемых для передачи параметров метода при его вызов.

Наибольший индекс локальной переменной для значений типа long или double является количество переменных - 2. Наибольший индекс локальной переменной для значения любого другого типа количество переменных - 1.

Длина кода. Значение элемента указывает на число байт в массиве инструкций для этого метода.

Код метода. Массив, содержащий фактически байт-код виртуальной машины, который реализует метод.

Таблица исключений. Каждая строка таблицы описывает один обработчик исключений массива кода.

Таблица атрибутов. Атрибут Code может содержать неограниченное количество своих атрибутов. Стандартные атрибуты, использующиеся атрибутом Code: LineNumberTable, LocalVariableTable,

LocalVariableTypeTable, StackMapTable. Виртуальная машины Java, поддерживающая class файлы версии номер 50.0 или выше, распознаёт и интерпретирует атрибут StackMapTable. Любые нестандартные атрибуты игнорируются. Под нестандартными атрибутами понимается, атрибуты не входящие в оригинальную JVM. Так же не допускаются нестандартные атрибуты, влияющие на семантику класс файла. Подобные атрибуты допускаются только в том случае, когда они обеспечивают дополнительную описательную информацию.

2.7 Набор инструкций виртуальной машины

Совокупность инструкций конкретного атрибута Code является байткодом соответствующего метода. Байт-код в свою очередь описывает логику выполнения соответствующего метода.

Инструкция виртуальной машины Java состоит из кода операции, определяющего работу, которая будет выполняться, сопровождается нулем или большим количеством операндов. Структура форматов большинства инструкций представлены на рисунке 2.1[Архитектура компьютера 4-е издание] за исключением двух инструкций – это lookupswitch и tableswitch, которые не имеют постоянной длины. Длина данных инструкций зависит от количества возможных ситуаций, описанных в теле оператора switch, описанного на языке Java. Каждая инструкция байт-кода начинается с кода операции длиной не более одного байта. В зависимости от команды используются следующие форматы:

- ф.1 – для команд, состоящих только из кода операции;
- ф.2 – для команд, во втором байте которых может содержаться индекс (как в команде ILOAD), константа (как в команде BIPUSH) или указатель типа данных (как в команде NEWARRAY);
- ф.3 – отличается от второго формата только тем, что вместо 8-битной константы присутствует 16-битная константа (как в команде GOTO);
- ф.4 – используется только для команды IINC (инкремент);
- ф.5 – используется только для команды MULTINEWARRAY (создает многомерный массив «в heap»);
- ф.6 – только для команды INVOKEINTERFACE, которая вызывает процедуру при определенных обстоятельствах;
- ф.7 – только для команды WIDE IINC, чтобы обеспечить 16-битный индекс и 16-битную константу, которая прибавляется к выбранной переменной;
- ф.8 – только для команд WIDE GOTO и WIDE JSR, чтобы осуществлять переходы на большие расстояния в памяти и вызовы определенных процедур.

В виртуальной машине языка Java операции производятся над элементами стека, и результаты помещаются обратно в стек. Более подробное описание, объясняющее функции инструкций и указывающее все

исключения, которые могли бы быть сгенерированы при выполнении, рассмотрим далее.

Загрузка констант в стек iconst_i.

Загрузка в стек целочисленной константы i. Инструкция состоит из одного байта. Код операции зависит от значения константы так, например, iconst_0 будет иметь значение 0x03, а iconst_5 – 0x08. Значение лежит в промежутке [0;5]. Отрицательные значения от -2 до -5 загружаются в стек при помощи инструкции bipush и iconst_null (0x02) – для значения -1.

bipush. Загрузка в стек однобайтового целого со знаком. Формат инструкции следующий:

- 1 байт – код операции 0x10;
- 1 байт – 8-битное целое со знаком, лежащее в промежутках [6;127] или [-128;-2]. sipush. Загрузка в стек двухбайтового целого со знаком. Формат инструкции следующий:
- 1 байт – код операции 0x11;
- 2 байта – 16-битное целое со знаком, лежащее в промежутке [128;32767] или [-32768;-129].

fconst_i и dconst_i. Инструкции fconst_0 (0x0B), fconst_1 (0x0C), fconst_2 (0x0D), dconst_0 (0x0E), dconst_1(0x0F) загружают вещественные значения

0,1,2,0,1 соответственно. Иные значения вещественного типа загружаются напрямую из пула констант соответствующей инструкцией.

lconst_i. Инструкции lconst_0(0x09), lconst_1(0x0A) загружают значения типа long 0 и 1 соотвественно. Иные значения типа long загружаются из пула констант.

ldc. Загрузка в стек элемента из пула констант. Формат инструкции следующий:

- 1 байт – код операции 0x12;
- 1 байт – 8-битный беззнаковый индекс пула констант.

Существуют так же инструкции ldc_w (код операции 0x13) и ldc2_w (код операции 0x14). Первая аналогична ldc, но имеет два байта на индекс пула констант. Вторая используется только для элементов пула констант типа long или double.

Загрузка локальных переменных в стек iload. Загрузка целого значения из локальной переменной. Формат инструкции следующий:

- 1 байт – код операции 0x15;
- 1 байт – номер переменной.

Значение локальной переменной с соответствующим номером в текущем методе помещается в стек операндов. iload_n. Загрузка целого значения из локальной переменной. Инструкция состоит только из кода операции, который может принимать одно из следующих значений: iload_0 (0x1A), iload_1 (0x1B), iload_2 (0x1C), iload_3

(0x1D). Значение локальной переменной с номером n в текущем методе помещается в стек операндов.

lload. Загрузка целого значения типа long из локальной переменной. Формат инструкции следующий:

- 1 байт – код операции 0x16;
- 1 байт – номер переменной.

Значения локальных переменных с соответствующим номером и следующим за ним в текущем методе помещается в стек операндов. lload_n. Загрузка целого значения типа long из локальной переменной.

Инструкция состоит только из кода операции, который может принимать одно из следующих значений: lload_0 (0x1E), lload_1 (0x1E), lload_2 (0x20), lload_3 (0x21). Значения локальных переменных с номером n и n+1 в текущем методе помещается в стек операндов.

fload. Загрузка вещественного значения из локальной переменной. Формат инструкции следующий:

- 1 байт – код операции 0x17;
- 1 байт – номер переменной.

Значение локальной переменной с соответствующим номером в текущем методе помещается в стек операндов. fload_n. Загрузка вещественного значения из локальной переменной.

Инструкция состоит только из кода операции, который может принимать одно из следующих значений: fload_0 (0x22), fload_1 (0x23), fload_2 (0x24), fload_3 (0x25). Значение локальной переменной с номером n в текущем методе помещается в стек операндов.

dload. Загрузка вещественного значения типа double из локальной переменной. Формат инструкции следующий:

- 1 байт – код операции 0x18;
- 1 байт – номер переменной.

Значения локальных переменных с соответствующим номером и следующим за ним в текущем методе помещается в стек операндов. dload_n. Загрузка вещественного значения типа double из локальной переменной. Инструкция состоит только из кода операции, который может принимать одно из следующих значений: dload_0 (0x26), dload_1 (0x27), dload_2 (0x28), dload_3 (0x29). Значения локальных переменных с номером n и n+1 в текущем методе помещается в стек операндов. Сохранение значений стека в локальных переменных istore. Сохранение целого значения в локальной переменной. Формат инструкции следующий:

- 1 байт – код операции 0x36;
- 1 байт – номер переменной, в которой сохраняется значение. istore_n. Сохранение целого значения в локальной переменной. Аналог инструкции istore (0x36), за исключением того, что состоит из одного байта и может иметь одно из четырёх значений: istore_0 (0x3B), istore_1 (0x3C), istore_2 (0x3D), istore_3 (0x3E). Значение n

соответствует индексу переменной, с которой работает инструкция в тот или иной момент.

lstore. Сохранение целого значения типа long в локальной переменной.

Формат инструкции следующий:

- 1 байт – код операции 0x37;
- 1 байт – номер переменной.

Значение сохраняется в двух переменных. Индекс первой переменной соответствует заданному в параметре, а индекс второй переменной соответствует следующему.

lstore_n. Сохранение целого значения типа long в локальной переменной. Аналог инструкции lstore (0x37), за исключением того, что состоит из одного байта и может иметь одно из четырёх значений: lstore_0 (0x3F), lstore_1 (0x40), lstore_2 (0x41), lstore_3 (0x42). Значения n и n+1 соответствует индексу первой и второй переменных, с которыми работает инструкция в тот или иной момент.

fstore. Сохранение вещественного значения в локальной переменной.

Формат инструкции следующий:

- 1 байт – код операции 0x38;
- 1 байт – номер переменной.

fstore_n. Сохранение вещественного значения в локальной переменной. Аналог инструкции fstore (0x38), за исключением того, что состоит из одного байта и может иметь одно из четырёх значений: fstore_0 (0x43), fstore_1 (0x44), fstore_2 (0x45), fstore_3 (0x46). Значение n соответствует индексу переменной, с которой работает инструкция в тот или иной момент.

dstore. Сохранение вещественного значения типа double в локальной переменной. Формат инструкции следующий:

- 1 байт – код операции 0x39;
- 1 байт – номер переменной.

Значение сохраняется в двух переменных. Индекс первой переменной соответствует заданному в параметре, а индекс второй переменной соответствует следующему.

dstore_n. Сохранение вещественного значения типа double в локальной переменной. Аналог инструкции dstore (0x39), за исключением того, что состоит из одного байта и может иметь одно из четырёх значений: dstore_0 (0x47), dstore_1 (0x48), dstore_2 (0x49), dstore_3 (0x4A). Значения n и n+1 соответствует индексу первой и второй переменных, с которыми работает инструкция в тот или иной момент.

iinc. Увеличение локальной переменной на константу. Формат инструкции следующий:

- 1 байт – код инструкции 0x84;
- 1 байт – индекс локальной переменной;
- 1 байт – 8-битное целое значение со знаком.

При выполнении данной инструкции стек остаётся без изменений. Расширенный индекс wide. Расширение индекса для доступа к локальным

переменным инструкций загрузки, сохранения и приращения. Инструкция имеет два формата. Первый формат предназначен для расширения индекса переменной следующих инструкций: iload, fload, aload, lload, dload, istore, fstore, astore, lstore, dstore, ret, а второй формат только для инструкции iinc. Первый формат инструкции следующий:

- 1 байт – код операции 0xC4;
- 1 байт – код операции изменяемой инструкции;
- 2 байта – расширенный индекс переменной.

Второй формат инструкции следующий:

- 1 байт – код операции 0xC4;
- 1 байт – код операции 0x84;
- 2 байта – расширенный индекс переменной;
- 2 байта – беззнаковое целое значение.

Данная инструкция предшествует одной из следующих инструкций: iload, lload, fload, dload, aload, istore, lstore, fstore, dstore, astore, iinc. Индекс переменной перечисленных инструкций расширяется до 16-битного беззнакового целого значения, а для инструкции iinc расширяется еще и значение константы. Следующая инструкция исполняется как обычно за исключением того, что использует расширенный индекс переменной и значение константы (в случае использования формата 2).

Управление массивами

newarray. Создание нового массива. Формат инструкции следующий:

- 1 байт – код операции 0xBC;
- 1 байт – внутренний код, указывающий на тип массива.

Значение стека к моменту использования инструкции представляет собой число элементов в новом массиве, значение должно быть целого типа.

Существуют и другие инструкции управления массивом: anewarray (0xBD), arraylength (0xBE), multianewarray (0xC5), iaload (0x2E), laload (0x2F), faload (0x30), daload (0x31), aaload (0x32), baload (0x33), caload (0x34), saload (0x35), iastore (0x4F), lastore (0x50), fastore (0x51), dastore (0x52), aastore (0x53), bastore (0x54), castore (0x55), sastore (0x56). Подробное описание и формат перечисленных инструкций представлены в спецификации виртуальной машины Java. Инструкции передачи управления ifeq. Пропуск байта осуществляется, если значение равно 0. Формат инструкции следующий:

- 1 байт – код операции 0x99;
- 2 байта – количество байт.

Двухбайтовый параметр инструкции используется для построения 16битного смещения со знаком. Инструкция сравнивает целое значение стека с 0. Смещение осуществляется относительно адреса данной инструкции и измеряется в байтах. Если условие инструкции не выполняется, то осуществляется переход к следующей за данной инструкцией. Значение аргумента является количеством байт смещения ifne (0x9A). Пропуск байта

осуществляется, если значение не равно 0. Формат и логика работы инструкции аналогичны инструкции ifeq (0x99).
iflt (0x9B). Пропуск байта осуществляется, если значение меньше 0. Формат и логика работы инструкции аналогичны инструкции ifeq (0x99).
ifge (0x9C). Пропуск байта осуществляется, если значение больше или равно 0. Формат и логика работы инструкции аналогичны инструкции ifeq (0x99).
ifgt (0x9D). Пропуск байта осуществляется, если значение больше 0. Формат и логика работы инструкции аналогичны инструкции ifeq (0x99).
ifle (0x9E). Пропуск байта осуществляется, если значение меньше или равно 0. Формат и логика работы инструкции аналогичны инструкции ifeq (0x99).

Для сравнения двух значений типов: long, float или double существуют дополнительные инструкции: lcmp, fcmpl, fcmpg, dcmpl, dcmpg, результат работы которых обрабатывают соответствующие управляющие инструкции. Описание и коды данных дополнительных инструкций представлены далее.
lcmp (0x94). Сравнение двух значений типа long, лежащих на стеке. Если первое значение больше чем второе, тогда целое значение «1» помещается на стек. Если значения совпадают, на стек помещается значение «0». Если первое значение меньше чем второе, на стек помещается значение «-1».
Сравниваемые значения со стека удаляются.
fcmpl (0x95). Сравнение двух вещественных значений одинарной точности. Логика работы инструкции аналогична lcmp (0x94), за исключением типа обрабатываемых значений и четвёртым дополнительным результатом сравнения. Если одно из значений является NaN (Not-a-Number), то на стек помещается целое значение -1. fcmpg (0x96). Инструкция аналогична инструкции fcmpl (0x95), за исключением только ситуации, когда одно из значений является NaN. В данном случае вместо целого значения «-1» на стек помещается значение «1».

Логика работы инструкций dcmpl (0x97) и dcmpg (0x98) аналогичная fcmpl (0x95) и fcmpg (0x96) соответственно, но в качестве сравниваемых значений используются значения типа double.

Для сравнения пар значений типа int применяются соответствующие управляющие инструкции (if_icmp*) , представленные далее. if_icmpeq (0x9F). Пропуск байт осуществляется, если значения равны. Формат аналогичен инструкции ifeq (0x99). Двухбайтовый параметр инструкции используется для построения 16-битного смещения со знаком. Смещение осуществляется относительно адреса данной инструкции и измеряется в байтах. Инструкция сравнивает два целых значения стека. Если условие инструкции не выполняется, то осуществляется переход к следующей за данной инструкцией. if_icmpne (0xA0). Пропуск байт осуществляется, если значения не равны.
Формат и логика работы инструкции аналогичны инструкции if_icmpeq (0x9F). if_icmplt (0xA1). Пропуск байт осуществляется, если первое целое значение

меньше второго. Формат и логика работы инструкции аналогичны инструкции if_icmpeq (0x9F). if_icmpge (0xA2). Пропуск байт осуществляется, если первое целое значение больше или равно второму. Формат и логика работы инструкции аналогичны инструкции if_icmpeq (0x9F). if_icmpgt (0xA3). Пропуск байт осуществляется, если первое целое значение
больше второго. Формат и логика работы инструкции аналогичны инструкции if_icmpeq (0x9F). if_icmple (0xA4). Пропуск байт осуществляется, если первое целое значение меньше или равно второму. Формат и логика работы инструкции аналогичны инструкции if_icmpeq (0x9F). if_acmpeq (0xA5). Пропуск байт осуществляется, если значения ссылок на объекты равны. Формат аналогичен инструкции ifeq (0x99). Двухбайтовый параметр инструкции используется для построения 16-битного смещения со знаком. Смещение осуществляется относительно адреса данной инструкции и измеряется в байтах. Инструкция сравнивает два значения стека, содержащих ссылки на объекты. Если условие инструкции не выполняется, то осуществляется переход к следующей инструкции за данной. if_acmpne (0xA6). Пропуск байт осуществляется, если значения ссылок на объекты не равны. Формат аналогичен инструкции ifeq (0x99). Двухбайтовый параметр инструкции используется для построения 16-битного смещения со знаком. Смещение осуществляется относительно адреса данной инструкции и измеряется в байтах. Инструкция сравнивает два значения стека, содержащих ссылки на объекты. Если условие инструкции не выполняется, то осуществляется переход к следующей инструкции за данной. goto (0xA7). Безусловный пропуск байт осуществляется. Формат аналогичен инструкции ifeq (0x99). Двухбайтовый параметр инструкции используется для построения 16-битного смещения со знаком. Смещение осуществляется относительно адреса данной инструкции и измеряется в байтах.

Помимо инструкций безусловного и условного переходов (пропуска определённого количества байт) существуют инструкции, используемые для перехода к подпрограммам: jsr (0xA8), jsr_w (0xC9), ret (0xA9).

ifnull (0xC6). Пропуск, если значение является пустым указателем. Формат и логика работы инструкции аналогичны инструкции ifeq (0x99).

ifnonnull (0xC7). Пропуск, если значение содержит указатель. Формат и логика работы инструкции аналогичны инструкции ifeq (0x99).

goto_w (0xC8). Инструкция полностью аналогична goto (0xA7), за исключением размера параметра смещения, который состоит из 32 бит.

Переходы по таблице

tableswitch. Доступ к таблице перехода по индексу и переход. Формат инструкции следующий:

- 1 байт – код операции 0xAA;
- 0-3 байта – «подушка» из нулей;
- 4 байта – смещение «по-умолчанию»;

- 4 байта – минимальное значение промежутка; • 4 байта – максимальное значение промежутка;
- Перечисление 4-байтовых смещений.

Данная инструкция имеет переменную длину. Между кодом операции и смещением «по-умолчанию» может присутствовать «подушка» от 0 до 3-х байт. Размер «подушки» подбирается таким образом, что бы следующий за ней байт начинался с адреса кратного 4.

Смещение «по-умолчанию» используется аналогично логике оператора языка Java switch, то есть если значение стека больше чем максимальное значение промежутка или меньше минимального значения промежутка, то значение смещения «по-умолчанию» добавляется к адресу этой инструкции. В противном случае вычисляется соответствующее смещение следующим образом: выбирается смещение, соответствующее индексу, который в свою очередь определяется как разность значения стека и минимального значения промежутка. После чего выполняется смещение в соответствии с полученным значением. Все смещения в данной инструкции ведут отсчёт от начала инструкции, то есть от начала кода операции, то есть для определения количества пропускаемых байт вычисляется разность между текущим смещением и полученным. lookupswitch. Доступ к таблице перехода в соответствии с ключом и переход.

Формат инструкции следующий:

- 1 байт – код операции 0xAB;
- 0-3 байта – «подушка» из нулей;
- 4 байта – смещение «по-умолчанию»;
- 4 байта – количество пар смещений;
- по 8 байт – соответствующие пары смещений.

Данная инструкция имеет переменную длину. Между кодом операции и смещением «по-умолчанию» может присутствовать «подушка» от 0 до 3-х байт. Размер «подушки» подбирается таким образом, что бы следующий за ней байт начинался с адреса кратного 4.

Каждая пара смещений состоит из значения, которое сравнивается с значением из стека, и собственно смещения. Целое значение стека сравнивается с каждым значением пар. Как только найдено значение, равное значению стека вычисляется значение суммы смещения из соответствующей пары и смещения начала инструкции. Если значение стека не соответствует не одному из значений пар, значение итогового смещения вычисляется как сумма значений смещения «по-умолчанию» и смещения начала инструкции. После чего выполняется смещение в соответствии с полученным значением. Все смещения в данной инструкции ведут отсчёт от начала инструкции, то есть от начала кода операции, то есть для определения количества пропускаемых байт вычисляется разность между текущим смещением и полученным.

3 Вложение в байт-код

3.1 Методы вложения в байт-код

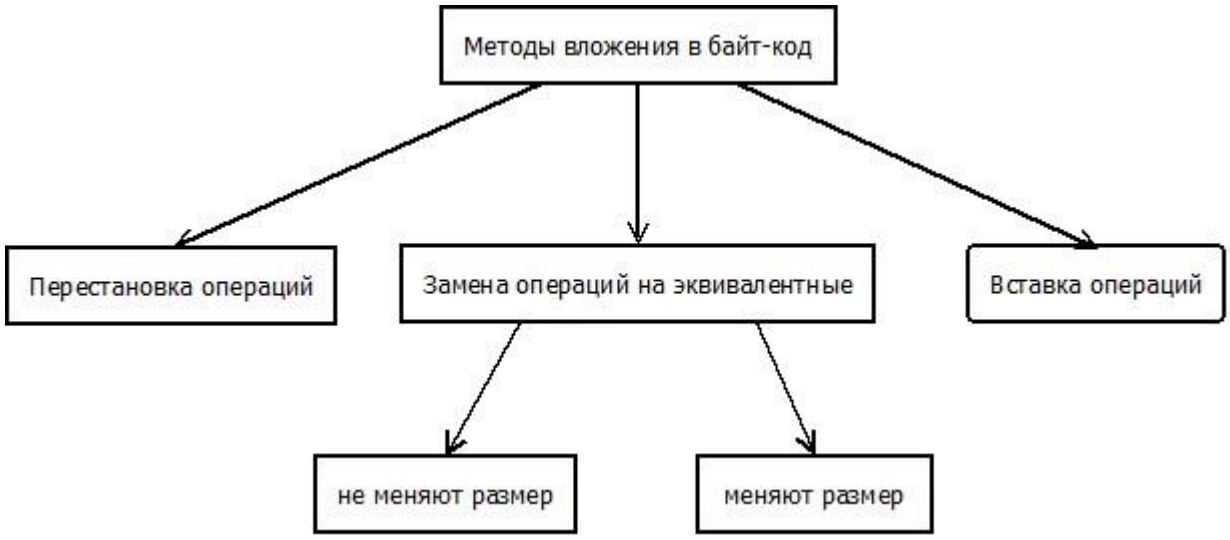


Рисунок 3.1 – Методы вложения в байт-код

Рассмотрим возможные методы вложения в байт-код:

1) Перестановка операций
2) Замена операций на эквивалентные:
   a) Меняющие размер байт-кода (замена на эквивалентные математические операции)
   b) Не меняющие размер байт-кода (замена ветвлений кода в условиях)
3) Вставка операций

Таблица 3.1 – сравнение методов вложения.

Метод вложения	Перестановка операций	Замена операций на эквивалентные		Вставка операций
		замена в условиях	математические операции	
Увеличивают размер кода	Нет	Нет	Да	Да
Увеличивают время выполнения программы	Нет	Нет	Да	Да
Размер вложения	Ограничен	Ограничен	Ограничен	Неограничен

Для вложения цифровых водяных знаков метод должен удовлетворять следующим условиями:

- не должен быть обнаруживаем
- не должен менять размера файла • не должен увеличивать время выполнения программы.

Для вложения информации в исполняемый файл был выбран первый метод, так как он не меняет размер исполняемого файла и не меняет скорость его выполнения. Данный метод будет рассмотрен подробнее в следующем разделе.

Второй метод – Замена операций на эквивалентные. Один из вариантов – замен ветвлений веток кода в условиях. Данный метод основан на том, что ветви условного оператора можно использовать в различном порядке, при условии, что логическое выражение будет изменено на противоположное (табл. 3.1).

Таблица 3.1 – Замена операций на эквивалентные.

Исходный код	Байткод	Команды	Вкладываемый бит

if (a > b)
{
c = 0;
}
else
{
c = 1;
}
	0x1b
	0x1c
	0xa4
	0x03
	0x3e
	0xa7
	0x04
	0x3e
		iload_1
		iload_2
		if_icmple 14
		iconst_0
		istore_3 goto
		16 iconst_1
		istore_3
			0

if (a < b)
{
c = 1;
} else
{
c = 0;
}
	0x1b
	0x1c
	0xa2
	0x04
	0x3e
	0xa7
	0x03
	0x3e
		iload_1 iload_2
		if_icmpge 14
		iconst_1
		istore_3
		goto 16
		iconst_0 istore_3
			1

Один из вариантов второй из вариантов – замена математических операций на эквивалентные. Пример в табл. 3.2 основан на замене сложения на вычитание. Данные метод не подходит для вложения цифровых водяных знаков, т.к. он меняет размер исполняемого файла, и к тому же такое

сочетание команд **a -(-b)** не было обнаружено при исследование class файлов, так что данный метод легко обнаруживается при помощи статистического анализа.

Таблица 3.2 – Замена математических операций на эквивалентные.

Исходный код	Байткод	Команды	Вкладываемый бит
int c = a + b;	0x1b 0x1c 0x60 0x3e	iload_1 iload_2 iadd istore_3	0
int c = a -(-b);	0x1b 0x1c 0x74 0x64 0x3e	iload_1 iload_2 ineg isub istore_3	1

Третий метод – вставка операций, не меняющих логику работы программы. Пример в табл. 3.3,основан на том, что прибавление нуля не меняет результат. В данном случае при вложении единицы, добавляется две дополнительные команды. Несмотря на то, что данный метод позволяет вложить неограниченное количество информации в исполняемый файл, он не подходит для вложения цифровых водяных знаков, поскольку уменьшает скорость выполнения программы. Кроме того вставка дополнительных операций легче обнаруживаема статистическим анализом, чем другие методы вложения.

Таблица 3.3 – Вставка операций.

Исходный код	Байткод	Команды	Вкладываемый бит
int a = 2; int b = a + 3;	0x15 0x06 0x60 0x3d	iload_1 iconst_3 iadd istore_2	0
int a = 2; int b = a + 3 + 0;	0x15 0x06 0x60 0x03 0x60 0x3d	iload_1 iconst_3 iadd iconst_0 iadd istore_2	1

3.2 Описание выбранного метода вложения

Компилятор преобразует исходный текст java в байт код, который выполняется на виртуальной Java машине (JVM) и не зависит от архитектуры процессора. Виртуальная Java машина реализована в качестве стековой виртуальной машины. В качестве структуры данных, куда помещаются операнды, используется стек. Операции получают данные из стека, обрабатывают их и заносят в стек результат по правилу LIFO (последний пришел, первый ушел).

Рассмотрим подробнее как происходит вычисления при помощи стека значение переменной **c**(рис 3.2):

1) Добавление константы **5** в стек.
2) Добавление переменной **a(3)** в стек.
3) Операция вычитания извлекает два элемента из стека, и результат операции**(2)** кладёт на вершину стека.
4) Добавление переменной **b(1)** в стек.
5) Добавление константы **5** в стек.
6) Операция сложения извлекает два элемента из стека, и результат операции **(3)** кладёт на вершину стека.
7) Операция умножения**(6)** извлекает два элемента из стека, и результат операции кладёт на вершину стека
8) Результат умножения **(6)** извлекается из стека и записывается в переменную **c**, после этого стек пуст.

Таблица 3.4 – Исходный и байт-код выражения.

Исходный код	Байт код

int a = 3; int
b = 1;
int c = (5-a)*(b+2);
	……
	1) iconst_5
	2) iload_1
	3) isub
	4) iload_2
	5) iconst_2
	6) iadd
	7) imul
	8) istore_3

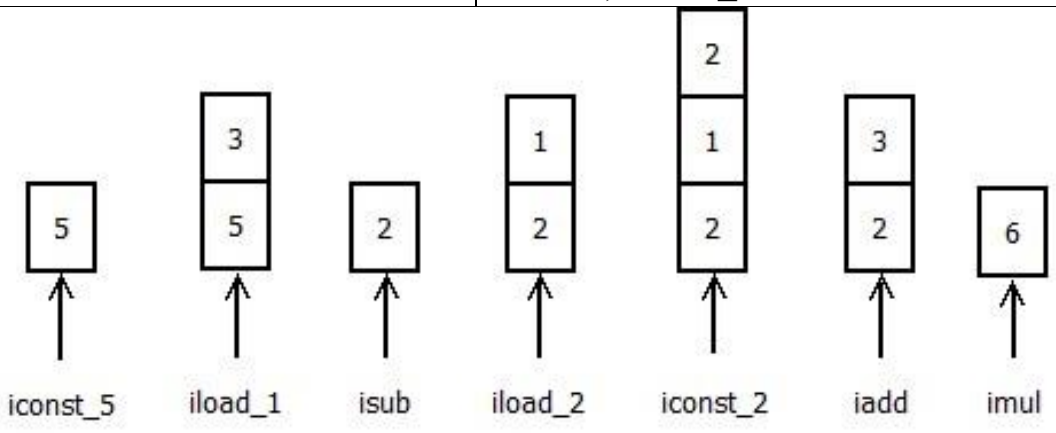


Рисунок 3.2 – Вычисления выражения в стековой машине.

Выбранная методика вложения информации в исполняемый файл основана на том, что перед некоторыми командами, которые берут из стека два аргумента, аргументы можно переставить местами и результат вычисления не поменяется. К таким командам относятся:

- Сложение: *iadd, ladd, fadd, dadd*;
- Умножение: *imul*, *lmul*, *fmul*, *dmul;*
- Логическое ИЛИ: *ior*, *lor;*
- Логическое. И: *iand*, *land;*
- Исключающее ИЛИ: *ixor*, *lxor;*
- Сравнения чисел: *lcmp, fcmpl, fcmpg, dcmpl, dcmpg;*
- Оператор If для целых чисел (равно, не равно): *if_icmpeq, if_icmpne.*

Данные команды в качестве аргументов могут принимать:

- Переменные (*iload,lload,dload*);
- Константы (*sipush, bipush, ldc*);
- поля классов (*getfield,getstatic*);
- результаты вычислений методов (*invokevirtual, invokestatic, invokeinterface*);
- результаты вычислений операций (*iadd, imul, iand, ior,ixor*).

Количество бит, которые можно вложить в исполняемый файл равно:

$$K_b = K_1 - K_2$$

где K_1 – количество команд, перед которыми можно переставить местами аргументы, K_2 – количество команд, перед которыми можно переставить местами аргументы, при условии, что аргументы одинаковые. В таблице 3.5 показан пример вложения 1 бита. Один бит вкладывается за счет перестановки аргументов перед сложением, в качестве аргументов – две переменные.

Таблица 3.5 – Пример вложения одного бита.

Исходный код	Байт код	Команды	Вкладываемый бит
int c = b+a;	0x1c 0x1b 0x60 0x3e	iload_2 iload_1 iadd istore_3	0
int c = a+b;	0x1b 0x1c 0x60 0x3e	iload_1 iload_2 iadd istore_3	1

В таблице 3.6 показан пример вложения двух битов. Первый бит вкладывается за счет перестановки аргументов перед сложением, в качестве аргументов – результат умножения и переменная. Второй бит вкладывается за счет перестановки аргументов перед умножением, в качестве аргументов – константа и переменная.

Таблица 3.6 – Пример вложения двух битов.

Исходный код	Байт код	Команды	Вкладываемые биты
int c = a*2+b;	0x1b 0x05 0x68 0x1c 0x60 0x3e	iload_1 iconst_2 imul iload_2 iadd istore_3	00
int c = 2*a+b;	0x05 0x1b 0x68 0x1c 0x60 0x3e	iconst_2 iload_1 imul iload_2 iadd istore_3	01
int c = b+a*2;	0x1c 0x1b 0x05 0x68 0x60 0x3e	iload_2 iload_1 iconst_2 imul iadd istore_3	10
int c = b+2*a;	0x1c 0x05 0x1b 0x68 0x60 0x3e	iload_2 iconst_2 iload_1 imul iadd istore_3	11

3.3 Описание алгоритмов

Общий алгоритм

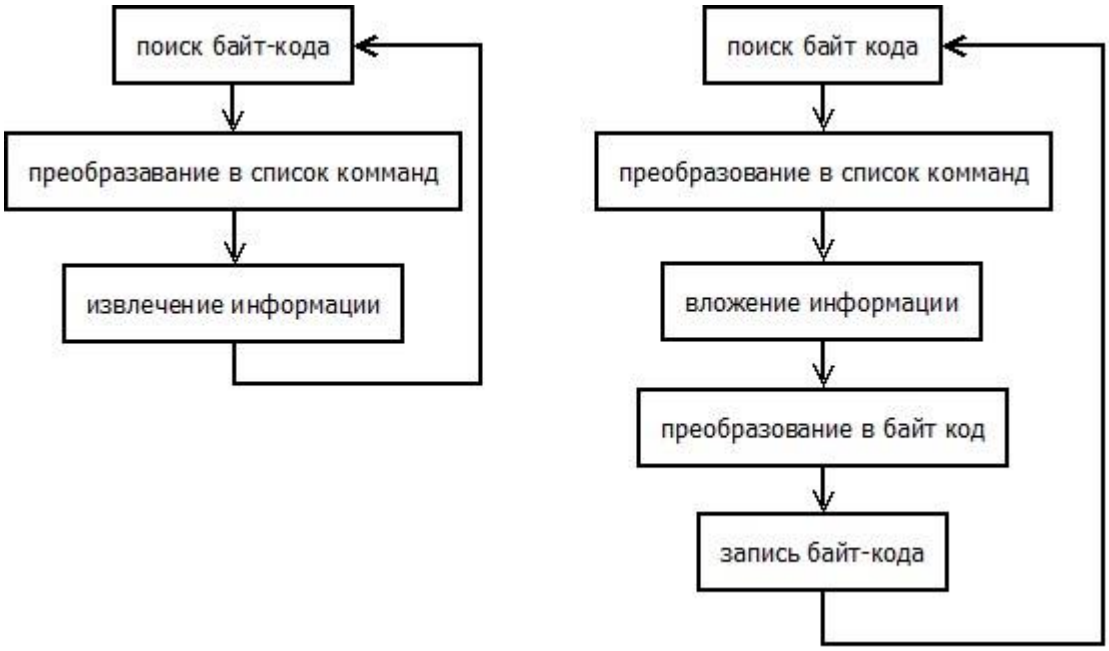


Рисунок 3.3 – Общие алгоритмы извлечения и вложения.

Общий алгоритм извлечения информации из class файла:

1. Находим байт код и его копируем.
2. Преобразуем байт-код в список команд.
3. Извлекаем информацию.
4. Переходим к следующему участку с байт-кодом.

Общий алгоритм вложения информации в class файл:

1. Находим байт код и его копируем.
2. Преобразуем байт-код в список команд.
3. Вкладываем информацию
4. Преобразуем список команд в байт код.
5. Записываем измененный байт код в class файл.
6. Переходим к следующему участку с байт-кодом.

Реализация поиска места вложения

На рисунке 3.4 показан алгоритм поиска. Рассмотрим его подробнее:

1. Открываем class файл для чтения.
2. Копируем пул констант.
3. Читаем количество методов и читаем каждый по отдельности.
4. У каждого отдельного метода читаем количество атрибутов.

5. Читаем все атрибуты до тех пор пока не находи атрибут Code, если мы его не находим (если метод является абстрактным) переходим к следующему методу.
6. Переходим к атрибуту Code, читаем длину байт кода.
7. Копируем байт-код и производим с ним нужные операции.
8. После того как прочитаны все методы и их атрибуты закрываем файл.

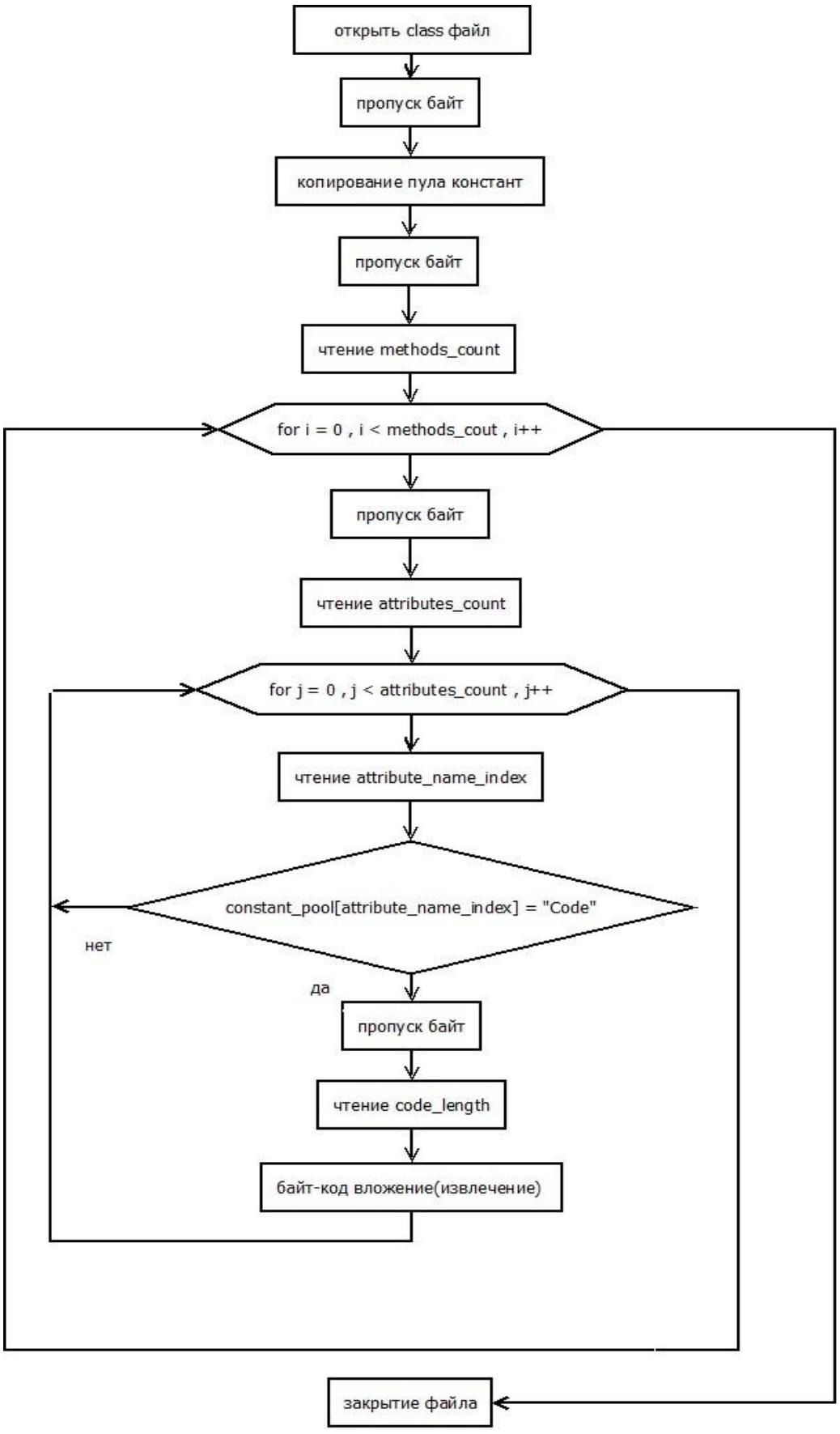


Рисунок 3.4 – Алгоритм поиск места вложения.

Реализация преобразования

Java байт-код представляем собой массив байт. Даная структура данных неудобна для выполнения перестановок, и поэтому данная структура преобразуется в список структур (код операции и массив параметров). Рассмотрим подробнее алгоритм:

1. Создаём пустой список
2. Создаем цикл для чтения массива.
3. Создаем структуру (номер команды, массив параметров).
4. Читаем номер команды и по ней определяем размер массива параметров N.
5. Читаем N байтов (массив параметров), увеличиваем счетчик цикла на N.
6. Добавляем структуру в список.
7. Переходим к следующей команде, а так до тех пор пока не прочитаем весь массив байтов.

Рисунок 3.5 – Алгоритм преобразования.

Рассмотрим обратного преобразования:

1. Создаем переменную, счетчик индекса массива **k** 2.
Создаем массив байтов (длина байт-кода известна)
3. читаем в цикл список структур.
4. читаем номер команды и записываем её в массив, увеличиваем **k** на единицу.
5. читаем массив параметров и записываем его в массив, увеличиваем **k** на длину массива.

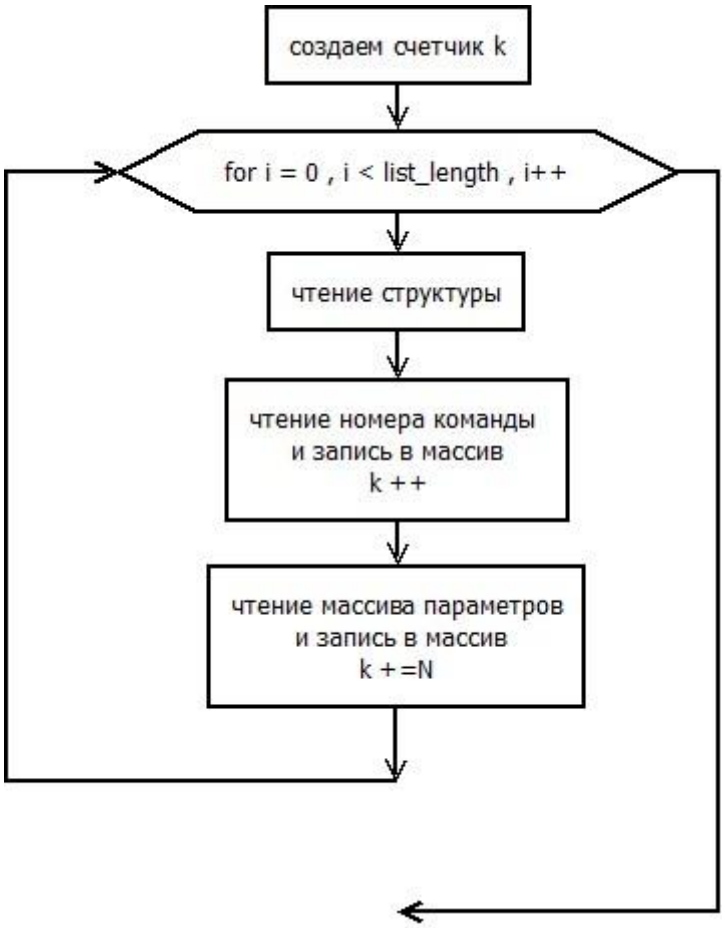


Рисунок 3.6 – Алгоритм обратного преобразования.

Реализация алгоритма извлечения

На рисунке 3.7 показан Алгоритм извлечения:

1. Копируем байт-код.
2. Преобразовываем байт-код в список команд.
3. Читаем список с конца, из-за того используется стековая машина, то есть операция расположена после параметров.
4. Читаем номер команды, определяем, является ли она командой для извлечения, если нет, переходим к следующей.

5. Копируем два участка кода, перед этой командой.
6. Вычисляем сумму байт каждого участка кода **sum1 sum2**
7. Если **sum1** равна **sum2,** то ничего не извлекаем и переходим к следующей команде
8. если **sum1** < **sum2** извлекаем ноль
9. если **sum1** > **sum2** извлекаем единицу

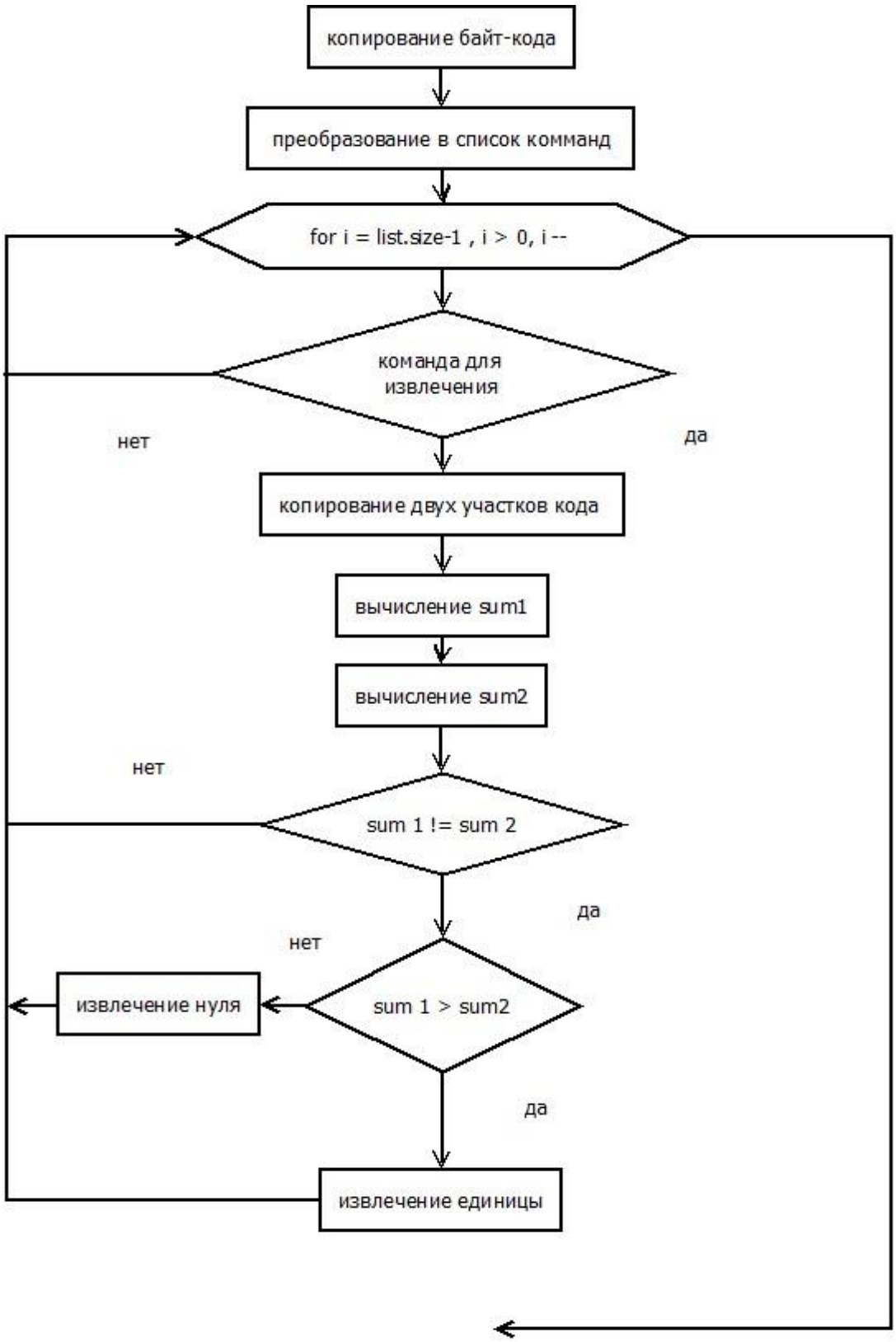


Рисунок 3.7 – Алгоритм извлечения.

Реализация алгоритм вложения

Алгоритм извлечения:

1. Копируем байт-код.
2. Преобразовываем байт-код в список команд.
3. Читаем список с конца, из-за того используется стековая машина, то есть операция расположена после параметров.
4. Читаем номер команды, определяем, является ли она командой для вложения, если нет, переходим к следующей.
5. Копируем два участка кода, перед этой командой, которые могут быть переставлены.
6. Вычисляем сумму байт каждого участка кода **sum1 sum2**
7. Если **sum1** равна **sum2,** то ничего не вкладываем и переходим к следующей команде
8. Выполняем вложение согласно таблице 3.7

Таблица 3.7 Правило вложения.

	Вложение **0**	Вложение **1**
sum1 > sum 2	Переставляем	Не переставляем
sum1 < sum 2	Не переставляем	Переставляем

9. После того как весь лист команд прочитан, преобразуем его обратно в байт код.
10. Записываем измененный байт-код в class файл.

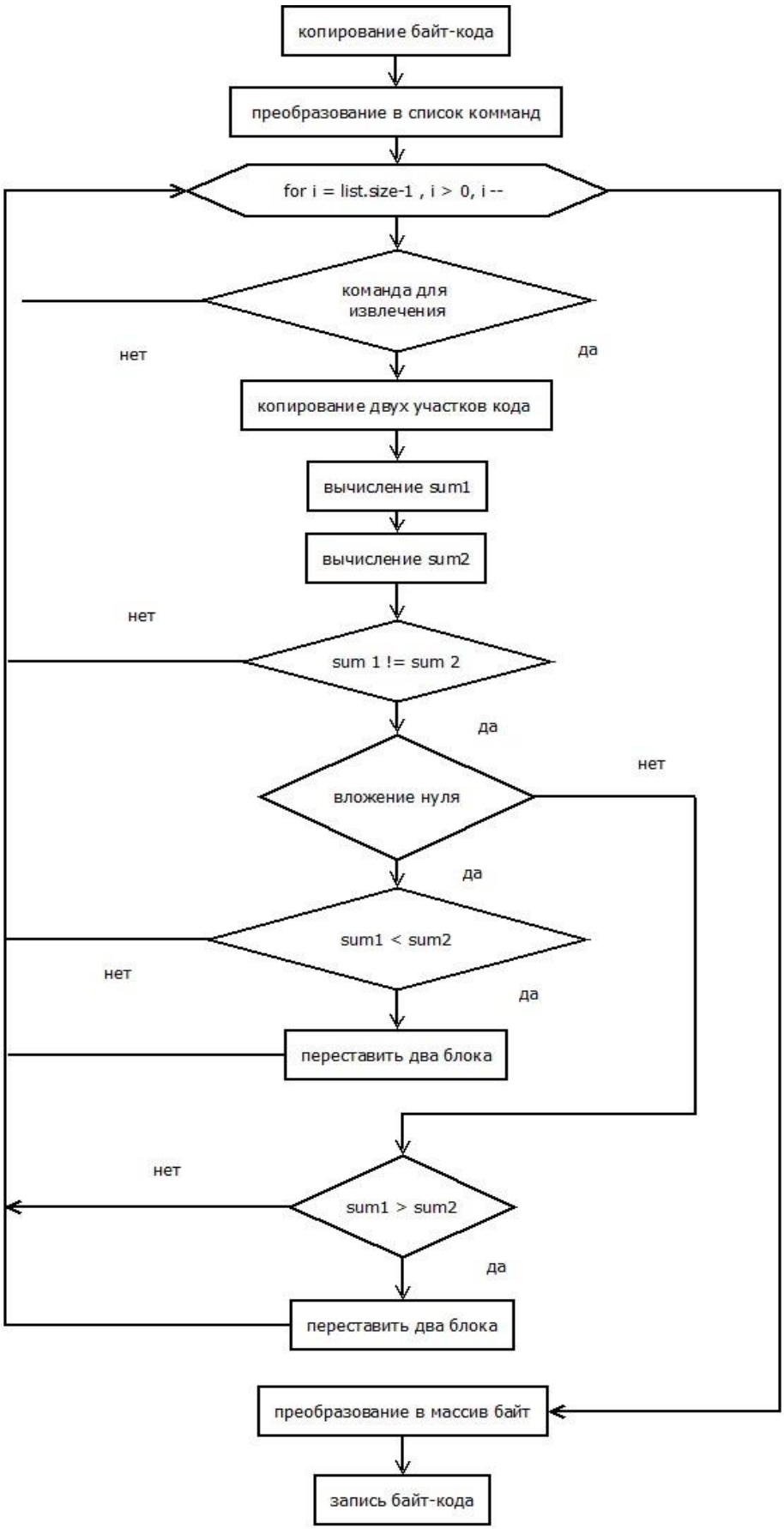


Рисунок 3.8 – Алгоритм вложения.

4 Возможности применения

4.1 Оценка возможного объема вложения

В таблице 4.1 показан результат экспериментальной проверки данного метода вложения информации на выборке файлов.

Таблица 4.1 – Исследование объема вложения.

Выборка	Количество class файлов	Размер файлов	Объем вложения	Скорость вложения
tools.jar	4145	14,5 МБ	4145бит	1бит на 4.6 кбайт
rt.jar	373	0,98 МБ	21бит	1 бит на 47 кбайт
jfxrt.jar	5770	14,4 МБ	93 бит	1бит на 159 кбайт
По всей выборке	155282	630 МБ	85860 бит	1 бит на 7.5 кбайт

Для исследования данного метода вложения информации были взяты около 155 тысяч class- файлов. Средний размер class-файла составляет 4.15 кбайт. В среднем в каждые 7.5 кбайт можно вложить 1 бит информации.

Как мы видим, что java файлы позволяют вложить сравнительно небольшой объем информации. Но как было написано в предыдущих главах в данном случае нас интересует не объем информации, а возможность использования данной информации с программным обеспечением. В следующем разделе будет дано описание применения данного метода.

4.2 Описание применения

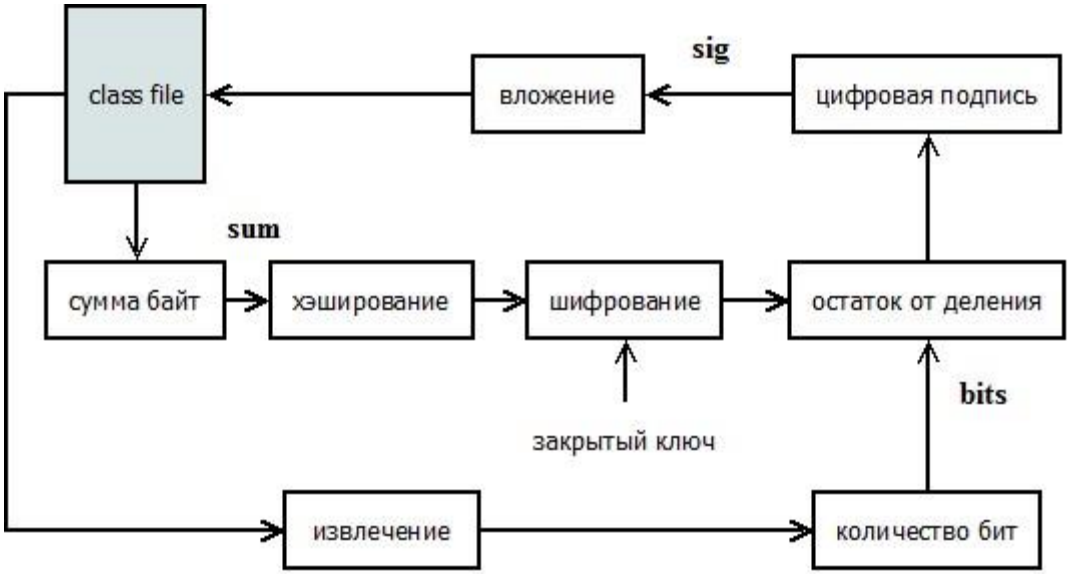


Рисунок 4.1. Алгоритм вложения цифровой подписи

На рисунке 4.1 показан алгоритм вложения цифровой подпись в class файл. Рассмотрим, как происходит вложения:

1. Извлекается последовательность из файла и подсчитывается её длину, это будет количество бит, которые можно вложить.
2. Подсчитываем сумму байт файла. Производится подсчёт суммы, из-за того что, при использовании данного алгоритма вложения сумма байтов не меняется при вложении любой последовательности битов.
3. Вычисляется хеш от суммы байт.
4. Вычисленный хеш шифруется закрытым ключом, которым имеется у владельца программы.
5. Вычисляется остаток от деления на $2^{\mathbf{bits}}$, поскольку количество вкладываемых бит ограничено.
6. В конце производится вложения битовой последовательности в файл.

Формула вычисления цифровой подписи

$\boldsymbol{sig} = \boldsymbol{encryption}\ (\ \boldsymbol{hash}(\mathbf{sum}), \mathbf{K_{private}})\ \boldsymbol{mod}\ \mathbf{2}^{\boldsymbol{bits}}$, где

$\boldsymbol{sum}$– сумма байт файла, $\mathbf{K_{private}}$ – закрытый ключ, $\boldsymbol{bits}$ – количество бит, доступных для вложения

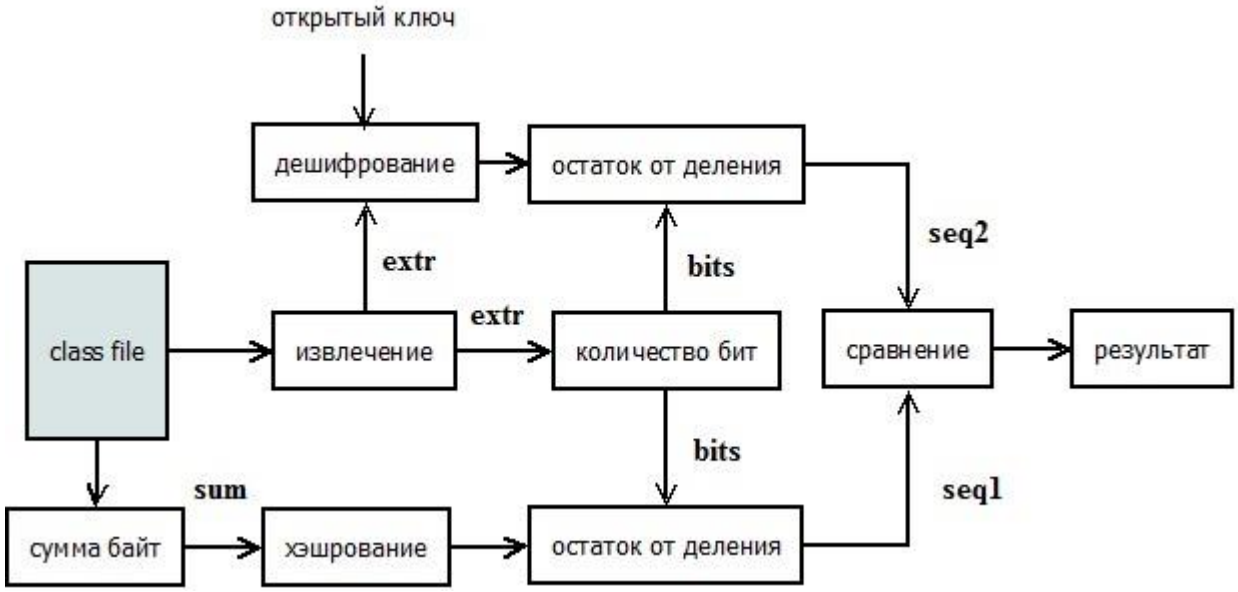


Рисунок 4.2 – Алгоритм верификации цифровой подписи.

На рисунке 4.2 показан алгоритм верификации цифровой подпись. Рассмотрим, как он происходит:

1. Извлекается последовательность из файла и дешифровывается закрытым ключом, и получается последовательность **seq1**.
2. Подсчитывается сумма байт файла и вычисляется хеш, и получается последовательность **seq2**.
3. Далее сравниваются две последовательности **seq1** и **seq2**, если они равны, то цифровая подпись правильна.

$$\boldsymbol{seq1 = decryption(extr, \quad K_{public}) mod\ 2^{bits}}$$

$$\boldsymbol{seq2 = hash(sum)\ mod\ 2^{bits}}, \text{ где}$$

$\boldsymbol{extr}$ – извлеченная последовательность, $\boldsymbol{sum}$ – сумма байт файла,

$\mathbf{K_{private}}$ – закрытый ключ, $\boldsymbol{bits}$ – количество бит, доступных для вложения

4.3 Практические вопросы тестирования ПО

В главном окне программы ProStega 2.0 мы вводим текст в окно «Текст шифрования», либо же выбираем файл в формате .txt нажав кнопку «Открыть текстовый файл»

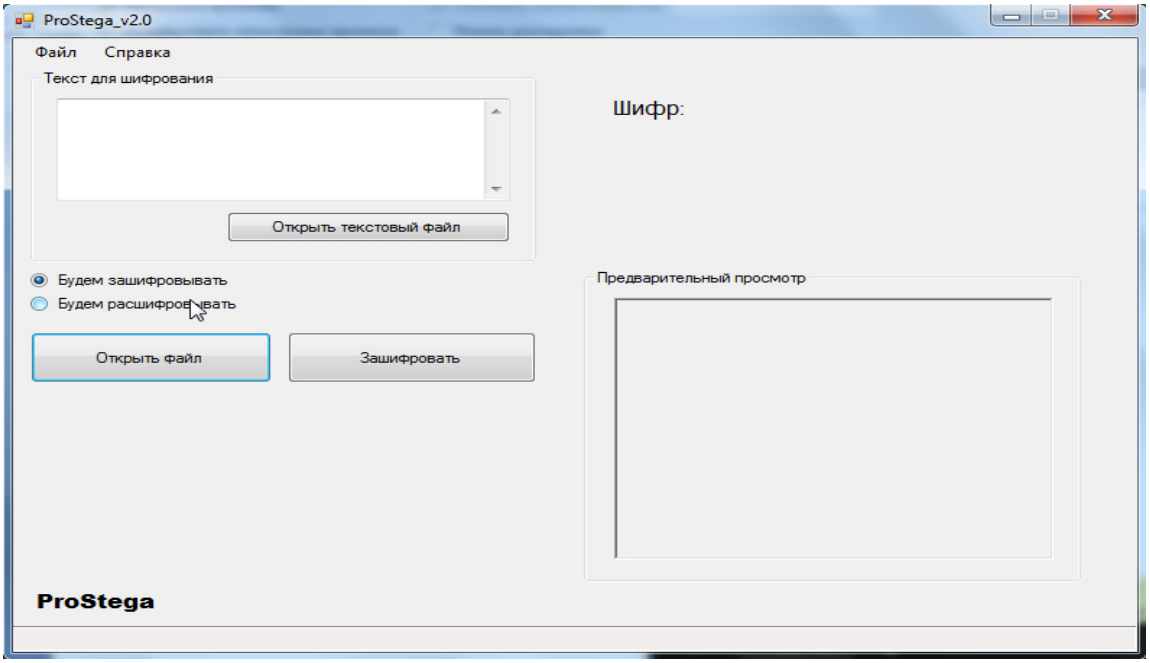


Рисунок 3.9 – Главное окно программы

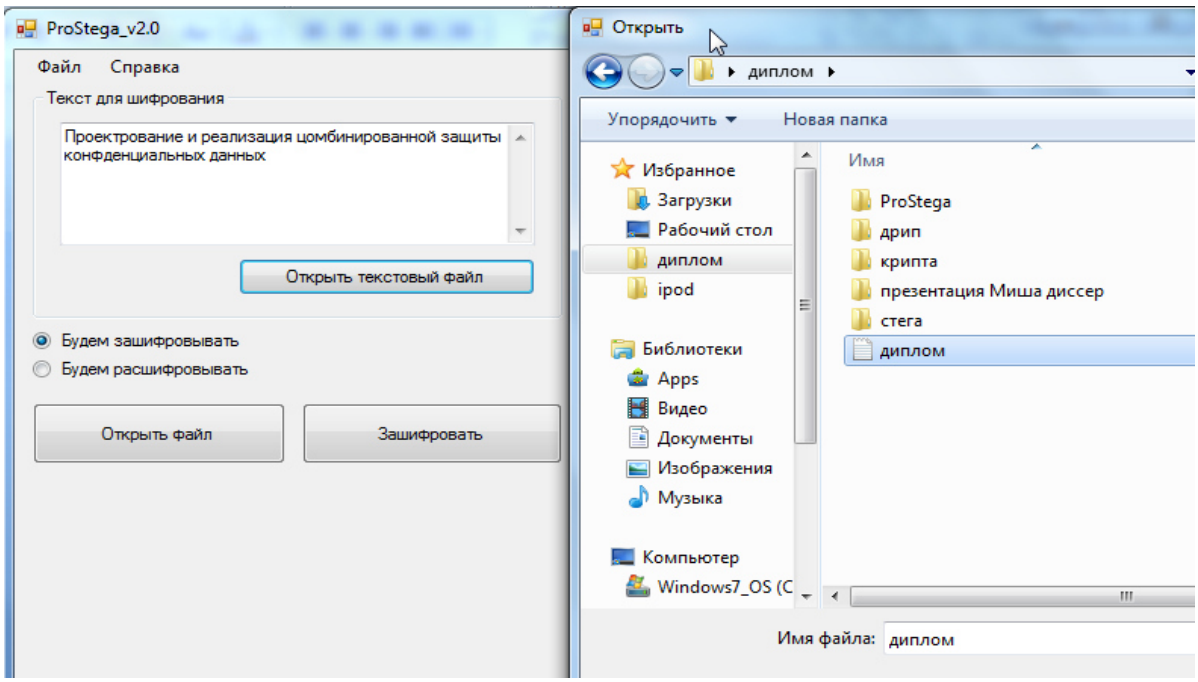


Рисунок 3.10 – Добавление информации для шифрования

Для начала шифрования мы выбираем функцию «Будем зашифровывать» и открываем файл – контейнер, куда и будет «вшиваться» наша конфиденциальная информация. Программа видит только файлы формата BMP.

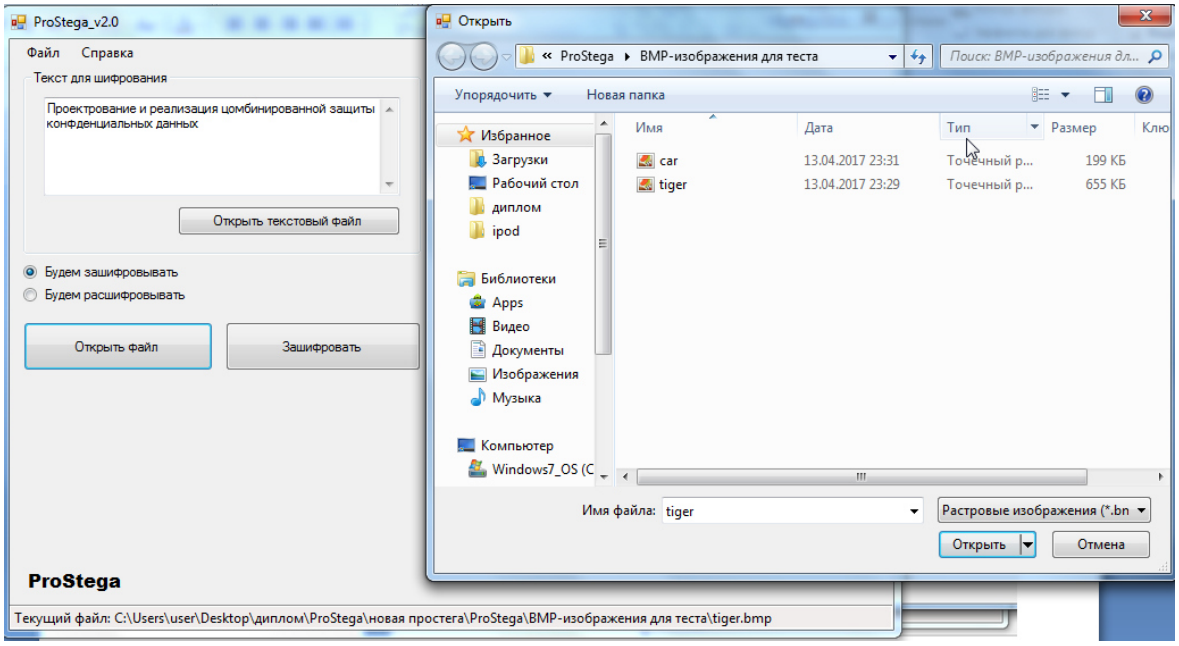


Рисунок 3.11 – Добавление контейнера

Далее при нажатии кнопки «Зашифровать», в папке где был взят исходный файл, появляется наш зашифрованный файл с названием «tiger1».

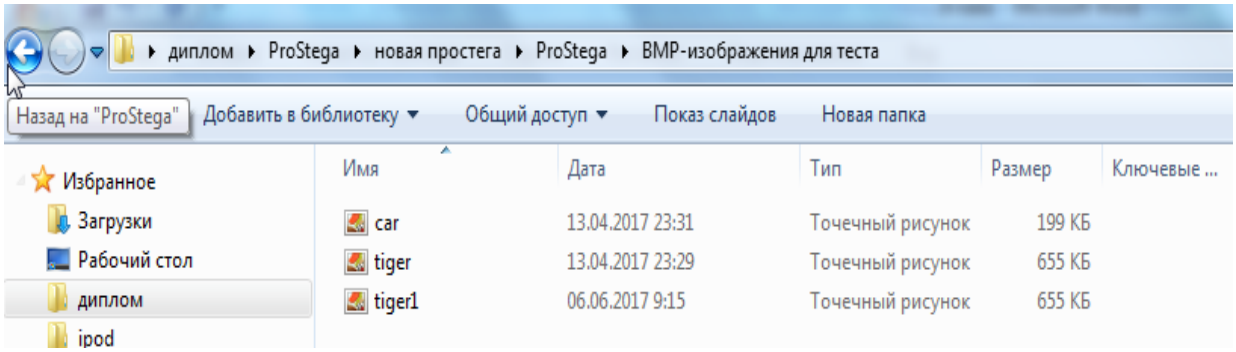


Рисунок 3.12 –Зашифрованное изображение

Проверяем размер файла и визуальные изменения.

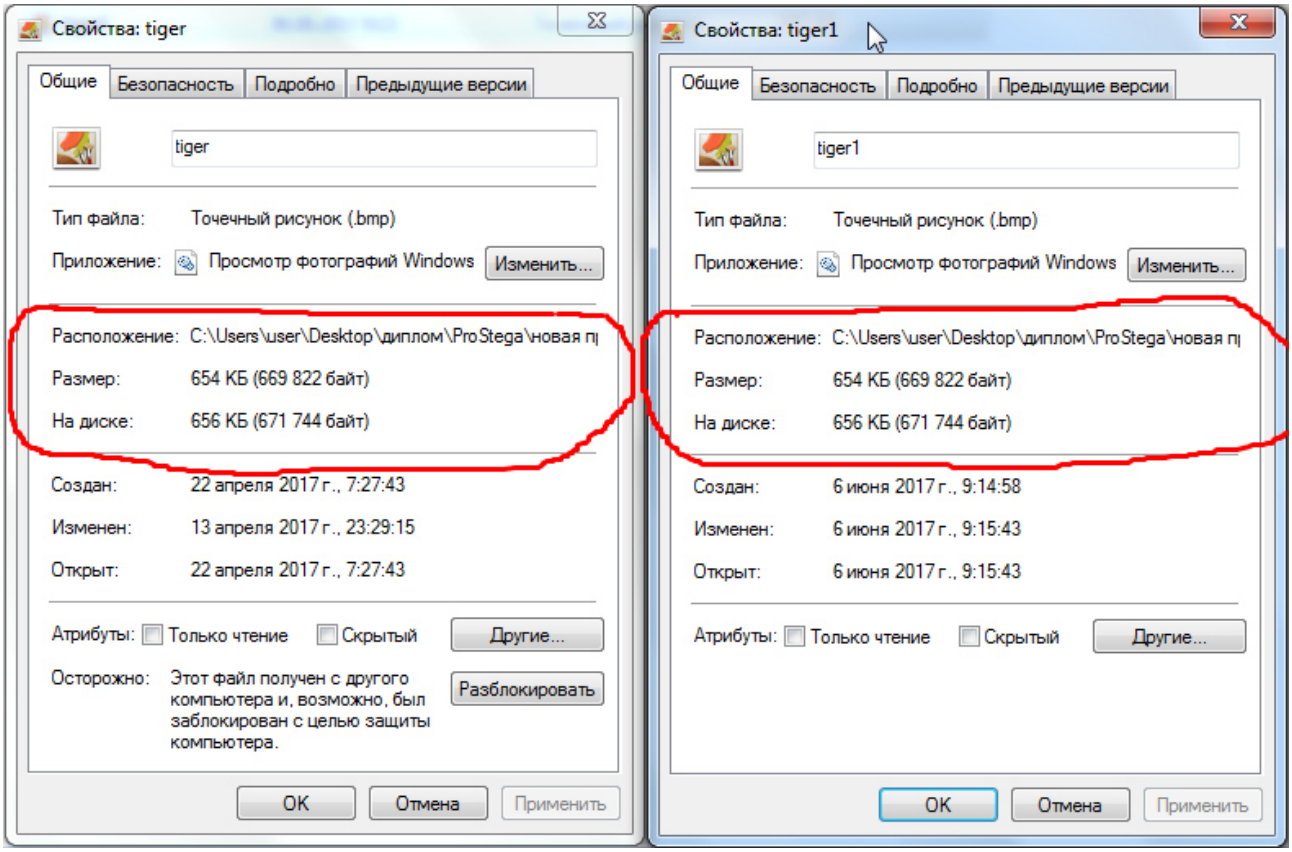


Рисунок 3.13 –Размер исходного и зашифрованного изображения

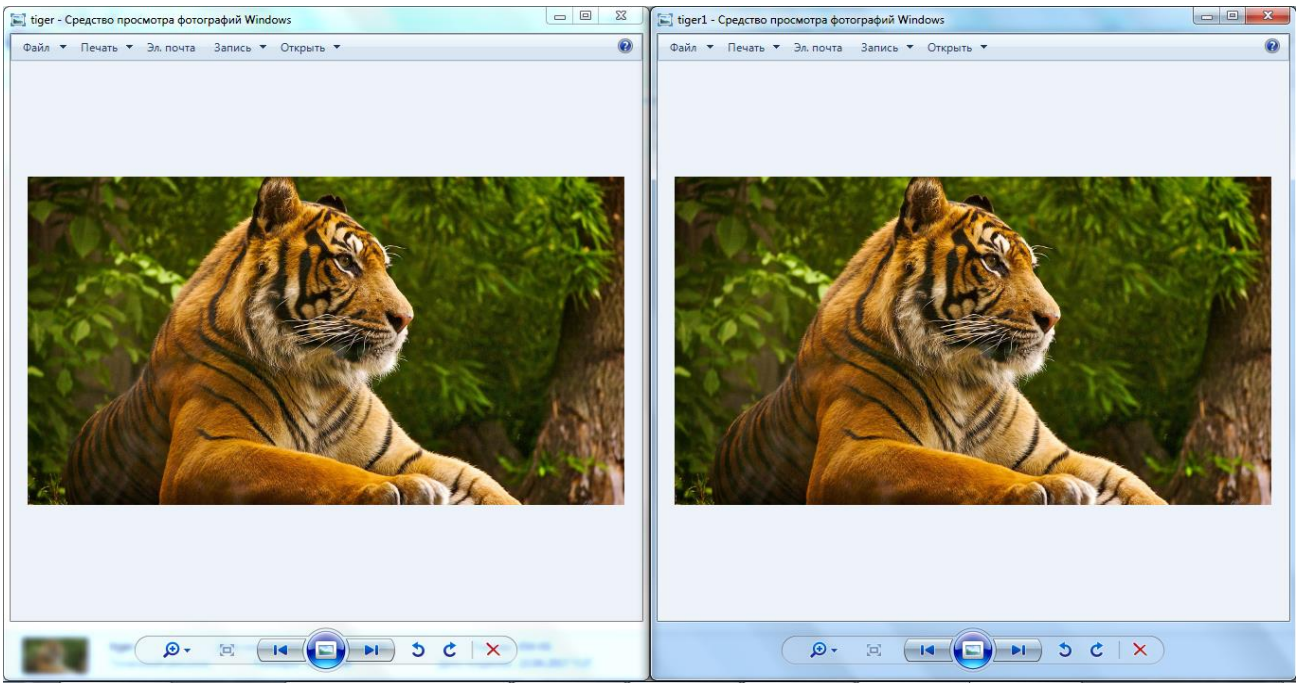


Рисунок 3.14 –Визуальные изменения исходного и зашифрованного изображения

Чтобы извлечь исходную информацию нужно выбрать функцию «Будем расшифровывать» и открываем файл с зашифрованным сообщением.

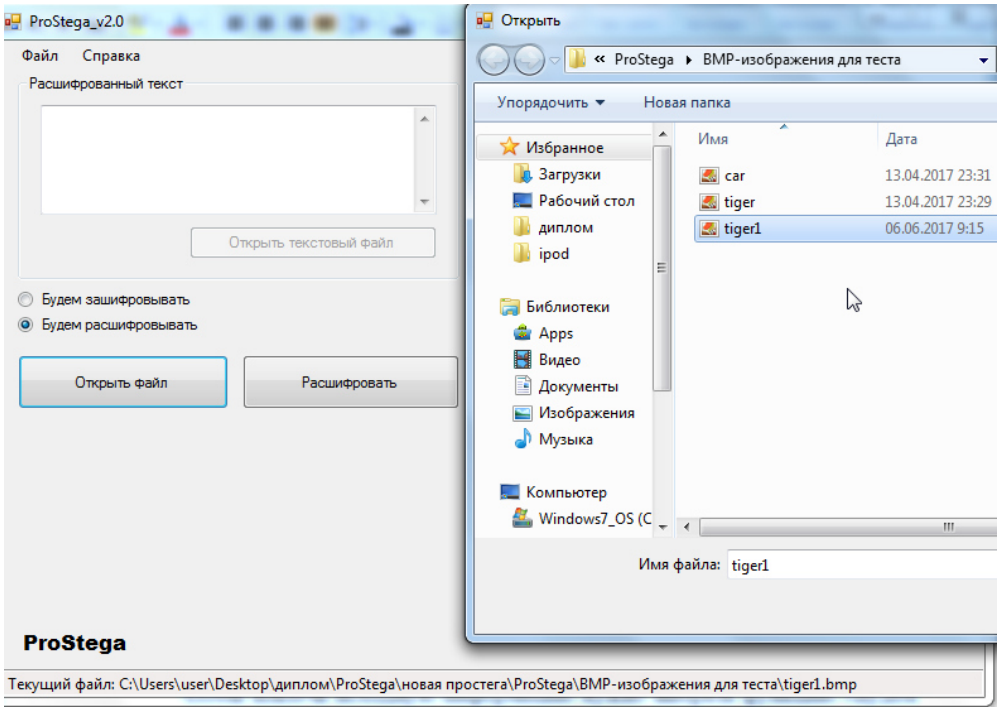


Рисунок 3.15 – Извлечение информации из контейнера и расшифровка

При нажатии «Расшифровать» в окне появляется исходный текст сообщения.

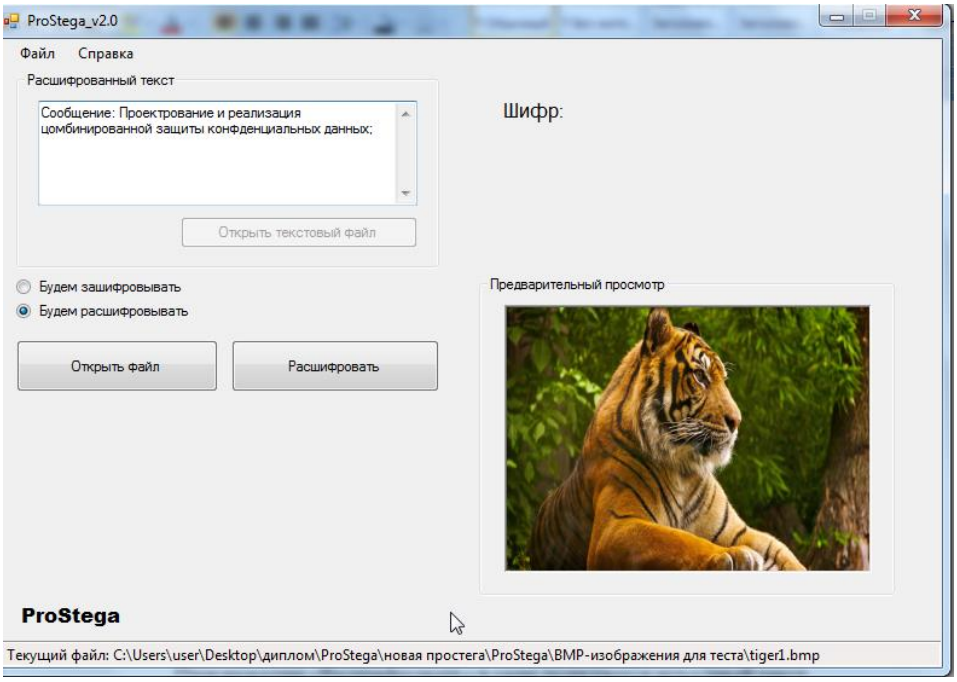


Рисунок 3.16 – Расшифрованная информация

В результате проведенного тестирования, можно сделать вывод: созданная мною программа в качестве дипломного проекта работает исправно, выполняет функции шифрования и расшифрования текста, также помещает шифрованный текст в контейнер и извлекает его правильно без каких-либо изменений.

5 Технико-экономическое обоснование

Дипломная работа посвящена системе защиты от утечки конфиденциальной информации по стеганографическим каналам. Написанная программа служит для скрытия информации. Программа была полностью протестирована в соответствии с поставленными задачами. Капитал знаний в сфере информационных технологий позволял осуществить запланированный проект.

Технико-экономическое обоснование содержит следующие пункты:

- определение трудности построения системы защиты;
- расчет затрат на внедрение системы защиты;
- определение ценности проекта;
- оценка результатов работы системы защиты.

5.1 Определение трудоёмкости построения системы защиты

Следует раздробить задачи на этапы для определения трудоемкости защиты системы, следствие этого разделение — это более удобное и продуктивное использование. Этот метод поможет ускорить процесс обработки подзадачи и станет более доступным для понимания. Модель

распределения сложности системы защиты и стадии проектирования представлены в таблице 5.1.

Таблица 5.1 – Этапы построения системы защиты

Этапы построения системы защиты	Вид работы	Трудоемкость, чел. час.
Этап 1	Разработка и утверждение ТЗ на проектирование системы защиты	25
Этап 2	Изучение литературы	25
Этап 3	Разработка программы для проекта	40
Этап 4	Отладка программы	45
Этап 5	Тестирование программы	20
Этап 6	Корректировка проекта	30
Этап 7	Внедрение проекта в производство	25
Итого:		220

Продолжительность рабочего дня равна 8 часам. В результате для реализации системы защиты необходимо 27 рабочих дней. (220:8)

5.2 Расчет затрат на проектирование системы

Определение затрат необходимых для построения системы защиты по стеганографическим каналам производится на основе имеющейся сметы, которая включает следующие элементы:

- материальные затраты;
- затраты на оплату труда;
- социальный налог;
- амортизация основных фондов;
- прочие затраты.

Материальные затраты делятся на основные и вспомогательные затраты на материалы, энергию и другие затраты необходимые для построения системы защиты с помощью стеганографических каналов. Расчет материальных затрат происходит по форме, предоставленной в таблице 5.2.

Таблица 5.2 – Затраты на материальные ресурсы

Наименование материала	Марка	Ед. измерения	Количество	Цена за ед. в тенге	Сумма в тенге
Бумага для офиса	Снегурочка	Упаковка	1	1 315	1 315
Тетрадь (96 листов)	Fruit time	Штук	2	250	500
Блокнот	Bullet journal	Штук	1	3501	3501
Ручки	Pilot	Штук	2	100	200
Компьютерная мышь	Logitech	Штук	2	2990	5980
Итого:					11 496

Для выполнения проекта необходимо использовать ноутбук Asus Vivobook с операционной системой Windows 10 и приложение Java. Ноутбук имеет все для выполнения проекта и программы. Он не требуется в дополнительном обновлении или покупки различных программ.

Общую сумму, необходимую на материальные средства ($З_м$) можно рассчитать по следующей формуле:

$$З_м = \sum P_i * Ц_i, \quad (5.1)$$

где P_i - расход i-го вида материального ресурса, натуральные единицы;
$Ц_i$ - цена за единицу i-го вида материального ресурса, тг;
i - вид материального ресурса;
n - количество видов материальных ресурсов.

Расчет затрат на необходимое оборудование и программное обеспечение производится по форме, приведенной в таблице 5.3.

Таблица 5.3 – Расчет затрат на оборудование, необходимого для проекта

Наименование материала	Марка	Ед. измерения	Количество	Цена за ед. в тенге	Сумма в тенге
Ноутбук	Asus Vivobook	Штук	1	250 000	250 000
Принтер	Samsung SL-M2020/XEV/FEV	Штук	1	38 580	38 580

Хостинг	hostgator.com	Штук	2	3 350	6 700
Модем	TP-Link TD-W8901N-R	Штук	1	7 400	7 400
Домен	hostgator.com	Штук	1	4 448	4 448
Итого:					307 128

Зм = 11 496 + 307 128 = 318 624 (тг)

Затраты на оборудование и материальные ресурсы необходима данная сумма 318 624 тенге.

5.3 Расчет затрат на электроэнергию

В выполнении проекта, целью которой является проектирование системы защиты на уровне операционной системы необходимо использование электроэнергии.

Согласно таблице 4.1 для проектирования системы защиты необходимо 220 часов, теперь необходимо рассчитать стоимость электроэнергии, которая будет потрачена в течении 220 часов. Для принтера расчет будет проводиться для периода в 35 часа, так как нет необходимости постоянно использовать принтер.

$$Э = З_{эл.эн.обор.} + З_{доп.нужды.}, \quad (5.2)$$

где $З_{эл.эн.обор.}$ – затраты на электроэнергию оборудования;

$З_{доп.нужды.}$ – затраты электроэнергии на дополнительные нужды.

Расчет электроэнергии, которая необходима для оборудования определяется по следующей формуле:

$$З_{эл.эн.обор.} = \sum W * К_{исц} * S * T, \quad (5.3)$$

где W – потребляемая мощность, Вт;

$К_{исц}$ – коэффициент использования ($К_{исц}$ = 0,7..0,9);

T – время работы;

S – тариф (1кВт/ч = 18,32 тг).

Итоги по расчетам стоимости затрачиваемой электроэнергии представлены в таблице 5.4.

Таблица 5.4 – Затраты на электроэнергию

Наименование приборов	Паспортная мощность, кВт	Коэффициент мощности	Время работы оборудования, ч	Цена ЭЭ тг/кВтч	Сумма, тг.
Ноутбук	0,7	0,7	220	18,32	1974,89
Модем	0,07	0,9	220	18,32	253,91
Принтер	0,4	0,9	35	18,32	230,83
Кондиционер	0,9	0,9	180	18,32	2671,05
Освещение	0,4	0,7	220	18,32	1128,51
Итого:					6259,19

$$З_{эл.эн.обор.} = 6259{,}19(\text{тенге})$$

На дополнительные потребности расходы подсчитываются на основе повышенного показателя в объеме 5% от расходов на электроэнергию:

$$З_{доп.нужды} = 5\% * З_{эл.эн.обор.}, \quad (5.4)$$

Определим затраты на дополнительные потребности согласно формуле (5.4):

$$З_{доп.нужды} = 0.05 * 6259{,}19 = 312{,}959 \text{ (тенге)}$$

Исходя из всех расчетов, полные расходы на электроэнергию составляют:

$$Э = 312{,}959 + 6259{,}19 = 6572{,}149 \text{ (тенге)}$$

5.4 Расчет затрат на оплату труда

Для проектирования системы защиты, необходимо два работника:
- руководитель проекта – управление рабочим временем, корректировка рабочих процессов, координация, изучение предметной области;
- проектировщик системы защиты, тестирование и сопровождение.

Сумму расходов на оплату труда можно рассчитать по следующей формуле:

$$З_{тр} = \sum ЧС_i * T_i \quad (5.5)$$

где $ЧС_i$ - часовая ставка i-го работника, тг;

T_i - трудоемкость разработки модели, чел.×ч; i - категория работника;

n - количество работников, занятых разработкой ПП.

Во время реализации проекта рабочее время участников не равномерно, поэтому имеет смысл установить часовую ставку каждого работника и общий объем заработной платы.

Часовую ставку сотрудника можно рассчитать по следующей формуле:

$$ЧС_i = \frac{ЗП_i}{ФРВ_i} \quad (5.6)$$

где $ЗП_i$ - месячная заработная плата i-го работника, тг;
$ФРВ_i$ - месячный фонд рабочего времени i-го работника, час.

Месячная заработная плата руководителя равняется 210 000 тенге и месячная заработная плата проектировщика равняется 190 000 тенге. Рассчитаем часовую ставку каждого работника согласно формуле (5.6):

$$ЧС_{руководитель} = \frac{210\,000}{22 * 8} = 1\,193{,}18 \text{ тг/ч}$$
$$ЧС_{проектировщик} = \frac{190\,000}{22 * 8} = 1079{,}54 \text{ тг/ч}$$

Часовая ставка руководителя составляет 1 022,72 (тг/ч), трудоемкость проектирования равняется 110 часам. Часовая ставка проектировщика составляет 880,68 тг (тг/ч), трудоемкость разработки равняется 220 часам. Согласно формуле (5.5) можно рассчитать сумму расходов на заработную плату работников:

$$З_{тр} = 1193{,}18 * 110 + 1079{,}54 * 220 = 131249 + 237\,498 = 368\,747$$

Расчеты затрат по оплате труда показаны в таблице (5.5).

Таблица 5.5. – Расчет заработной платы

Категория работника	Квалификация	Трудоемкость разработки ПП, час.	Часовая ставка, тг/ч	Сумма, тг.
Руководитель проекта	Инженер-проектировщик	110	1 193,18	131 249,00
Проектировщик	IT специалист	220	1079,54	237 498,00
Итого:				368 747,00

5.5 Расчет затрат по социальному налогу

Согласно Налоговому кодексу Республики, Казахстан социальный налог составляет 9,5% от фонда оплаты труда. Социальный налог можно рассчитать по следующей формуле:

$$С_н = (ФОТ - ПО) * 0{,}095 \quad (5.7)$$

где ПО - отчисления в пенсионный фонд, они составляют 10% от ФОТ.

$$ПО = 368\,747 * 0{,}1 = 36\,874{,}7 \text{ тенге}$$
$$С_н = (368\,747 - 36\,874{,}7) * 0{,}095 = 31\,527{,}8 \text{ тенге}$$

Результаты расчетов представлены в таблице (5,6):

Таблица 5.6 – Начисление социального налога

Категория работника	Количество человек	Заработная плата, тг	Пенсионные отчисления, тг	Социальный налог, тг
Руководитель проекта	1	131 249	13 124	11 221,87
Проектировщик	1	237 498	23 749	20 306,1
Итого:				31 527,97

5.6 Амортизация основных фондов и прочие затраты

Нормы амортизации ОФ необходимо определить в соответствии с налоговым кодексом РК. Амортизацию ОФ можно определить по следующей формуле:

$$A_г = \frac{С_{об} * Н_а}{100} \quad (5.8)$$

где, $С_{об}$ – стоимость оборудования;

$Н_а$ – норма амортизации (норма амортизация = 25);

Формула (5.8) позволяет рассчитать нужную сумму для амортизационных отчислений за год для ноутбука:

$$A_г = \frac{250\,000 * 25}{100} = 62\,500 \text{ тенге}$$

Теперь необходимо рассчитать норму амортизации за период проектирования:

$$A_г = \frac{62500 * 27}{365} = 4\,623{,}28 \text{ тенге}$$

Подобным образом необходимо рассчитать норму амортизации для всего оборудования. Результаты расчетов приведены в таблице (5.7).

Таблица 5.7 – Амортизация ОФ

Наименование оборудования и ПО	Стоимость оборудования и ПО, тг	Годовая норма амортизации, %	Сумма амортизации за год, тг	Сумма амортизации за время проектирования, тг
Ноутбук	250 000	25	62 500	4 280,82
Принтер	38 580	25	9 645	660,61
Модем	3 350	20	670	45,8
Хостинг	7 400	20	1 480	101,36
Домен	4 448	15	667	45,68
Итого:			74 962	5134

Смета расходов на разработку ПО.

На основе всех представленных расчетов необходимо оформить смету расходов на проектирование системы защиты согласно форме, которая приведена в таблице (5.8). На рисунке (4) продемонстрирована диаграмма рабочих расходов.

Таблица 5.8 – Смета затрат на проектирование системы защиты

Статьи затрат	Сумма, тг
Затраты на оборудование	307 128,00
Затраты на оплату труда	368 747,00
Социальные налоги	31 527,80
Затраты на электроэнергию	6572,15
Амортизация основных фондов	5134,00
Затраты на материальные ресурсы	11 496,00
Итого по смете:	730 604,95

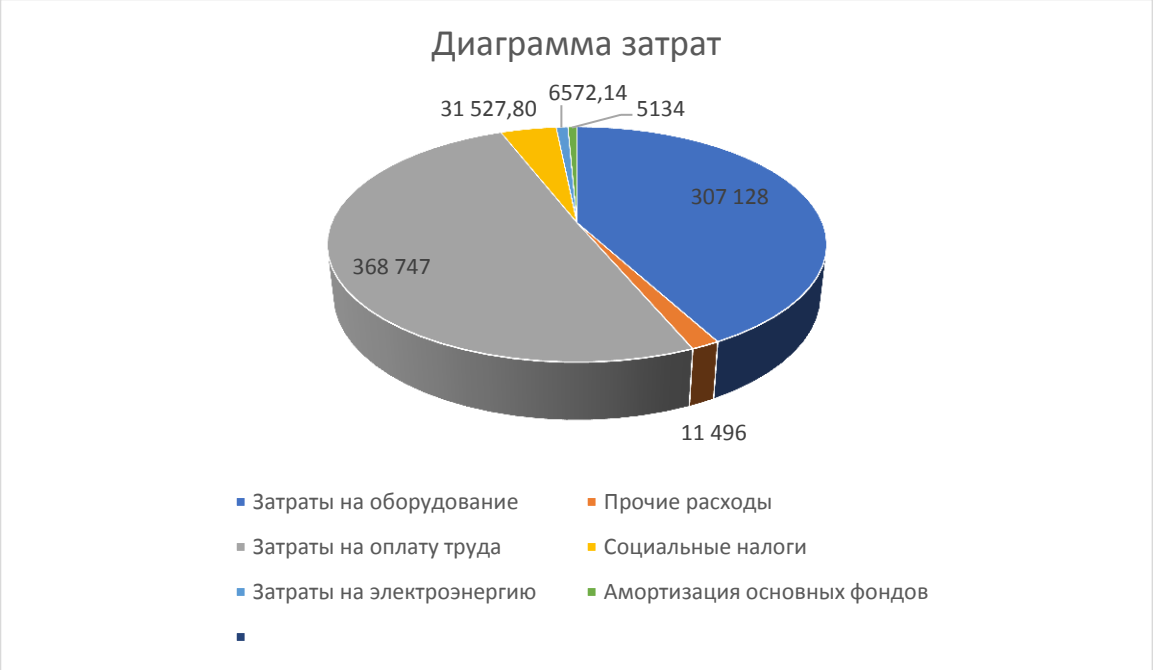


Рисунок 4 – Диаграмма затрат

5.7 Определение возможной (договорной) цены системы защиты

Стоимость системы защиты определяется на основе качества разработанной системы, сроков его проектирования и производительности. Стоимость $Ц_Д$ для системы защиты можно рассчитать по следующей формуле(5:

$$Ц_Д = З_{нир}\left(1 + \frac{P}{100}\right), \quad (5.9)$$

где $З_{нир}$ – затраты на проектирование системы защиты, тг;

Р – средний уровень рентабельности, (%). Данный параметр принят равным 25%.

Прибыль = 730 604,95* 0,25 = 182 651,24 тенге

$$Ц_Д = 730\,604{,}95\left(1 + \frac{25}{100}\right) = 730\,604{,}95 + 730\,604{,}95 * 0{,}25$$
$$= 913\,256{,}19 \text{ тенге}$$

Далее необходимо определить стоимость реализации с учетом НДС, ставка НДС устанавливается законодательством РК. На 2019 года ставка НДС составляет 12%. Стоимость реализации учитывая НДС можно рассчитать по следующей формуле(5.10):

$$Ц_р = Ц_д + Ц_д * НДС, \quad (5.10)$$

$$Ц_р = 913\,256{,}19 + 913\,256{,}19 * 0{,}12 = 1\,022\,846{,}94 \text{ тенге}$$

Данную цену можно округлить до 1 022 847,00 тенге.
Результатом расчетов является: прибыль равна 182 651,24 тенге , стоимость затрат на проектирование системы защиты равна 730 604,95 тенге, стоимость реализации с учетом НДС 1 022 847,00 тенге.

6 Безопасность жизнедеятельности

6.1 Анализ условий труда

В дипломной работе я проектирую систему шифрования. В ее эксплуатации необходим особо секретный объект , в котором будет работать 4 человека, сотрудник охраны и люди обладающие навыками в области информационной безопасностью . В помещении есть несколько рабочих мест с установленным на нём компьютером и мониторами и аппаратным устройством для шифрования. В данном помещение присутствует современная техника, которая не издает шума и хорошая вентиляция. В разделе БЖД задаюсь целью рассчитать естественное и искусственное освещение. В таблице 6.1 приведены исходные данные:

Таблица 6.1 – Исходные данные

Тип помещения	Параметры помещения				Разряд зрит. работ	$\rho_{пот}$	$\rho_{стен}$	$\rho_{пол}$	$h_{нок}$, м	Световой пояс	Нзд	Расст. до рядом стоящего здания, Р
	L, м	В, м	Н, м	$h_{ок}$, м								
Помещения для шифрования	15	10	4	3	III,б	50	50	10	1	Алматы	18	10

6.2 Расчетная часть

Расчет естественного освещения

Изначально в помещение, в котором мы рассматривали было 2 лампы и окно 25м^2

Расчет площади световых проемов

Площадь боковых проемов при боковом освещении определяется из следующей формулы(6.1):

$$100 \cdot \frac{S_0}{S_n} = \frac{e_N \cdot K_з \cdot \eta_0}{\tau_0 \cdot r_1} \cdot K_{зд},$$

где S_0- площадь световых проемов при боковом освещении, м^2;

S_n– площадь пола помещения, м^2;

e_N – нормируемое значение КЕО;

$K_з$ –коэффициент запаса;

η_0– световая характеристика окон;

τ_0– общий коэффициент светопропускания;

r_1– коэффициент, учитывающий повышение КЕО при боковом освещении, благодаря свету, отраженному от поверхности помещения и подстилающего слоя, примыкающего к заданию;

$K_{зд}$– коэффициент, учитывающий затемнение окон противостоящими зданиями.

Определим площадь пола помещения:

$$S_n = L \cdot B$$

$$S_n = 15 \cdot 10 = 150 \text{ м}^2$$

Нормируемое значение КЕО, e_N, для заданий, располагаемых в различных районах определять по формуле:

$$e_N = e_H \cdot m_N$$

где m_N – коэффициент светового климата, определяемый из таблицы

Учитывая заданный световой пояс (г.Алматы) адм. район 8, приняв ориентацию световых проемов **З , В** определим:

m_N= 0.8

e_H – значение КЕО.

По таблице13 , учитывая III б разряд зрительных работ, найдем:

e_H=1.2

Следовательно:

$$e_N = 1.2 \cdot 0.8 = 0{,}96$$

Учитывая тип помещения, найдем коэффициент запаса с помощью таблицы 10. каб.учебн. помещение, лаб. раб.

$K_з$=1.2 при ЕО вертикально.

Для определения световой характеристики, η_0, необходимо рассчитать отношение длины помещения к его глубине $\frac{L}{l}$, отношение ширины помещения к расчетной высоте $\frac{l}{h_{расч}}$.

$$l = B - 1 = 10 - 1 = 9 \text{ м}$$

$$\frac{L}{l} = \frac{15}{9} = 1.6$$

Найдем $h_{расч}$:

$$h_{расч} = h_{0k} + h_{н.ok.} - h_{р.п.}$$

$$h_{расч} = 3 + 1 - 0.8 = 3.2 \text{ м}$$

$$\frac{l}{h_{pac}} = \frac{9}{3.2} = 2.8 \approx 3$$

$$\frac{l}{B} = \frac{9}{10} = 0.9 \approx 1$$

Учитывая найденные отношения примем световую характеристику, $\eta_0 = 15$, по таблице 2 .

Общий коэффициент светопропускания, τ_0, рассчитывают по формуле:

$$\tau_0 = \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4,$$

где τ_1 – коэффициент светопропускания материала, принимаемый по таблице 4. Так как в качестве светопропускающего материала используется стекло листовое двойное, то:

$\tau_1 = 0.8$

τ_2 – коэффициент, учитывающий потери света в переплетах светопроема. Определяется с помощью таблицы 5 с учетом использования стальных двойных глухих переплетов:

$\tau_2 = 0.8$

τ_3 – коэффициент, учитывающий потери света несущих конструкциях, при ,боковом освещении:

$\tau_3 = 1$

τ_4 – коэффициент, учитывающий потери света в солнцезащитных устройствах, принимается по таблице 7. Выбираем убирающиеся регулируемые жалюзи и шторы (межстекольные внутренние, наружные)

$\tau_4 = 1$

Следовательно:

$$\tau_0 = \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4 = 0.8 \cdot 0.8 \cdot 1 \cdot 1 = 0.64$$

$$\rho_{ср} = \frac{\rho_{пот}\rho_{стен}\rho_{пол}}{3}\% = \frac{50 + 50 + 10}{3} = 0{,}36 \approx 0{,}3$$

r_1=1,7

Учитывая $Н_{зд}$=18 и Р=10 м (расстояние до рядом стоящего здания) из таблицы 9 найдем коэффициент, учитывающий затемнение окон противостоящими зданиями, $К_{зд}$:

$$\frac{Р}{Н_{зд}} = \frac{10}{18} = 0.55 => К_{зд} = 1.7$$

Зная значение всех параметров, рассчитываем площадь боковых проемов при естественном освещении по следующей формуле:

$$S_0 = \frac{S_n \cdot e_N \cdot К_з \cdot \eta_0}{100 \cdot \tau_0 \cdot r_1} \cdot К_{зд}$$

Где S_n– площадь пола помещения, м^2;

e_N – нормируемое значение КЕО;

$К_З$ –коэффициент запаса;

η_0– световая характеристика окон;

τ_0– общий коэффициент светопропускания;

r_1– коэффициент, учитывающий повышение КЕО при боковом освещении, благодаря свету, отраженному от поверхности помещения и подстилающего слоя, примыкающего к заданию;

$К_{зд}$– коэффициент, учитывающий затемнение окон противостоящими зданиями.

$$S_0 = \frac{150 \cdot 0{,}96 \cdot 1.2 \cdot 15 \cdot 1{,}7}{100 \cdot 0.64 \cdot 1{,}7} = 40{,}5 \text{ м}^2$$

Таким образом данных расчётов естественное освещение не удовлетворяет рассчитанному нормативному значения.

6.3

Тип помещения	Параметры помещения				Разряд	$\rho_{пот}$	$\rho_{стен}$	$\rho_{пол}$	W	Ф
ния	L, м	B, м	H, м	$h_{ок}$, м	зрит. работ					
Помещение для шифрования	15	10	4	3	III,б	50	50	10	36	5000 лм

Расчет искусственного освещения

Для расчета искусственного освещения используют один из трех методов: по коэффициенту использования светового потока, точечный и метод удельной мощности.

При рас чете общего равномерного освещения основным является метод использования светового потока, создаваемого источником света, и с учетом отражения от стен, потолка, пола.

Расчет освещения начинают с выбора типа светильника, который принимается в зависимости от условий среды и класса помещений по взрыво-пожароопасности.

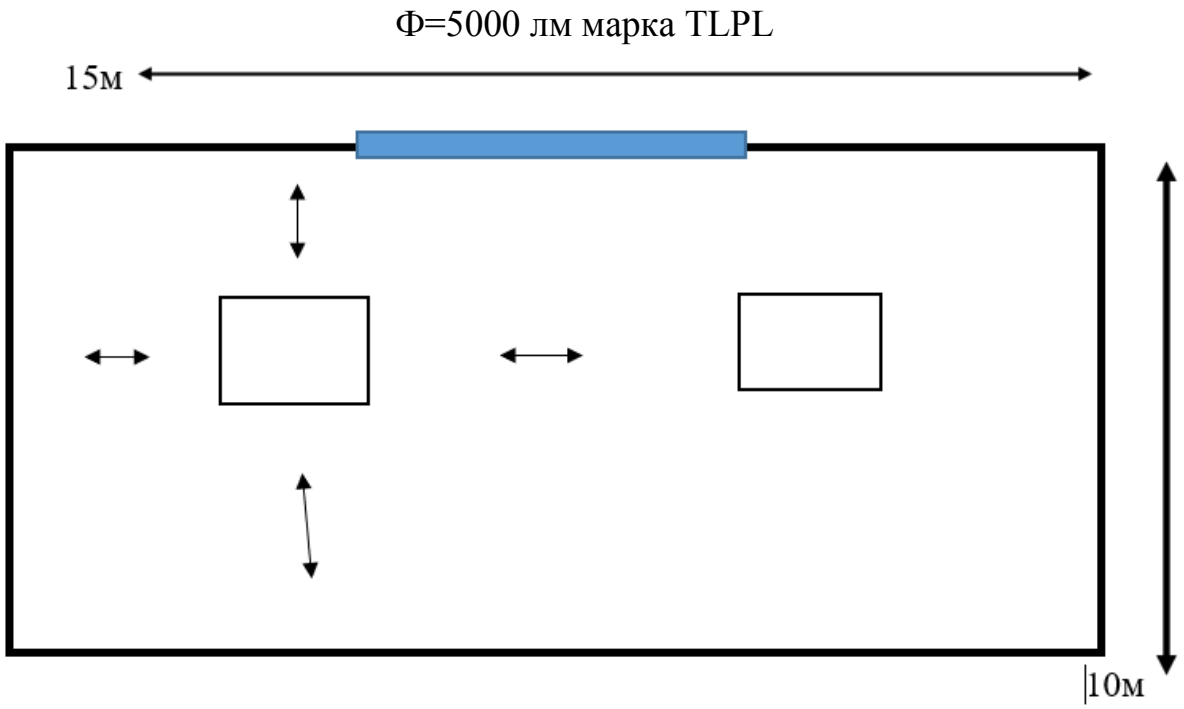


Рисунок 1

	L, м	B, м	H, м	$h_{ок}$, м						
Помещение для шифрования	15	10	4	3	III,б	50	50	10	36	5000 лм

Таблица 6,1- Исходные данные

6.4 Метод коэффициента использования светового потока.

$$E_\tau = \frac{Nn\,\phi\cdot\mu}{K\cdot S\,V} = \frac{2*2*5000*0{,}55}{1.5*150*1.1} = 44{,}4$$

Сначала нужно рассчитать заданное номинальное значение оно должно быть больше 300

Получилась у нас 158<300 что не удовлетворяет условному значению

Разряд зрительной работы III, б, поэтому нормируемая освещенность по таблице Ен= 300 лк (при системе общего освещения).
Определение расчетной высоты подвеса:

$$h_{расч} = H_{помещения} - H_{свеса} - H_{р.п.}, \qquad (6.1)$$

где $H_{свеса}$ = 0,5 - высота свеса ламп, м;
$H_{р.п.}$ = 0,8 - расстояние рабочей поверхности над полом, м;
$H_{помещения} = 4$ - высота помещения, м.

$$h_{расч}=4-0{,}5-0{,}8=2{,}7 \text{ м};$$

В практике расчетов значения коэффициентов η находятся из таблиц, связывающих геометрические параметры помещения (индекс помещения) с их оптическими характеристиками.

Индекс помещения определяется по формуле:

$$i = \frac{A*B}{h_{расч}*(A+B)} = \frac{15*10}{2{,}7(15+10)} = \frac{150}{405} = 0{,}37 \qquad (6.2)$$

где А- длина помещения , м;
В- ширина помещения , м;
$h_{расч}$ - расчетная высота, м.

По таблице 15 для светильника типа TLPL228.2x36 находим η = 0,55. Таким образом, количество светильников равно:

$$N = \frac{E_\tau \cdot K_З \cdot S \cdot V}{n\,ф \cdot \mu} = \frac{300 \cdot 1.7 \cdot 150 \cdot 1.1}{2 * 5000 * 0{,}55} = 15{,}3$$

$E_н$ = 300 лк - заданное номинальное освещение.

S = 150м^2 – площадь помещения.

Z = 1,1- коэффициент неравномерности освещения.

n- количество ламп в светильнике.

ф = 5000 лм

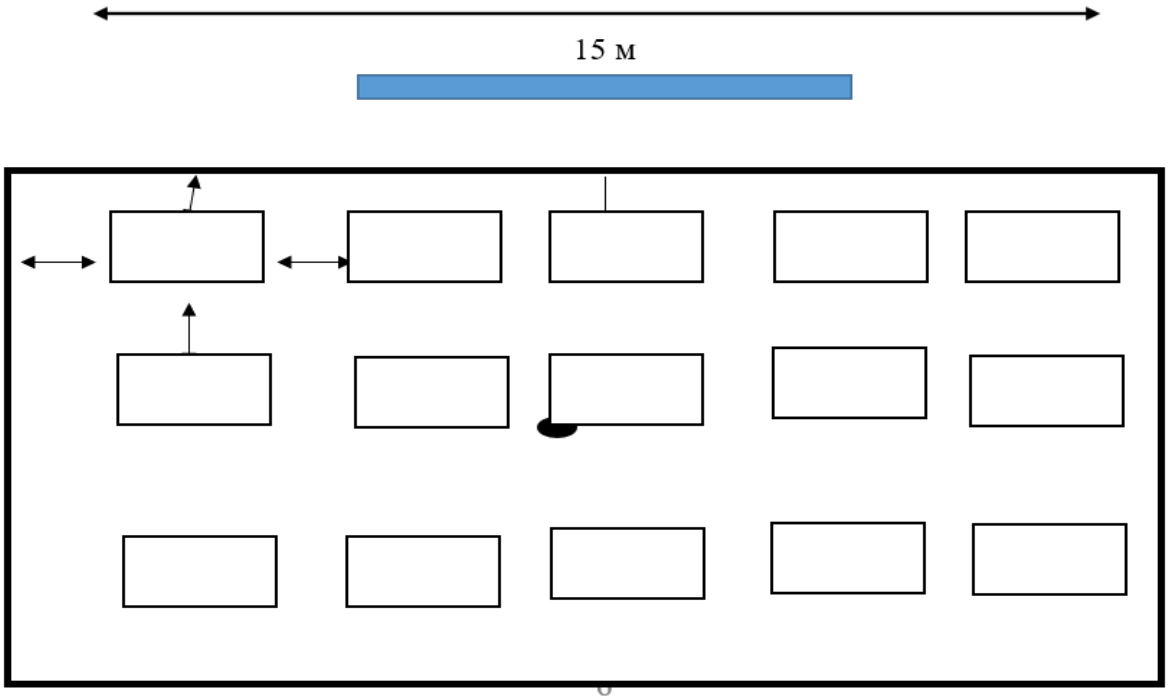
$$= \frac{200 * 1{,}5 * 242 * 1{,}1}{0.44 * 2 * 3120} N \approx 15 \text{ шт}$$

6.5 Расчет освещенности точечным методом

Определим расчетную высоту подвеса.

$$h_р = H_п - h_{свеса} - h_{р.пов.}$$

Принимаем $h_{свеса}$ = 0,5 м $h_{р.пов.}$ = 0,8 м



$$H_р = 6 - 0{,}5 - 0{,}8 = 4{,}7 \text{ м}$$

Найдем расстояние между светильниками, учитывая λ=0.6÷1.5.

$$L_A = \lambda \cdot h_р = 1{,}216 \cdot 2{,}7 = 3{,}2 \text{ м}$$

$$L_B = L_A - (0.3 \div 0.5) = 3,2 - 0,3 = 2,9 \text{ м}$$

$$l_a = l_b = (L_a/3) = 3,2/1,45 = 2,2$$
$$= l_b = (L_a/3) = 3,2/1,5 = 2,1$$

Для расчета намечаем контрольную точку А. Необходимо найти $d_{3,18}$; $d_{1,5,16,20}$; $d_{8,13}$; $d_{2,4,17,19}$; $d_{6,10,11,15}$; $d_{7,9,12,14}$; – проекции расстояния на потолок между точкой А и соответствующим светильником:

$$d_{1,5,16,20} = \sqrt{6^2 + 9^2} = 10,82 \text{ м};$$
$$d_{2,4,17,19} = \sqrt{4,5^2 + 6^2} = 7,5 \text{ м};$$
$$d_{6,10,11,15} = \sqrt{2 + 9^2} = 9,22 \text{ м}$$
$$d_{7,9,12,14} = \sqrt{4,5^2 + 2^2} = 4,9 \text{м}$$
$$d_{3,18} = 6 \text{ м};$$
$$d_{8,13} = 2 \text{ м};$$

Далее определяем угол между высотой потолка и соответствующим отрезком d:

$$\text{tg}\alpha_1 = \frac{d_{1,5,16,20}}{h\text{расч}} = \frac{10,82}{2,7} = 8,12 \rightarrow \alpha_1 = 71°;$$
$$\cos^3 \alpha_1 = 0,0345$$
$$\text{tg}\alpha_2 = \frac{d_{2,4,17,19}}{h_{\text{расч}}} = \frac{7,5}{2,7} = 2,7 \rightarrow \alpha_2 = 64 °;$$
$$\cos^3 \alpha_2 = 0,084$$
$$\text{tg}\alpha_3 = \frac{d_{6,10,11,15}}{h\text{расч}} = \frac{9,22}{2,7} = 3,4 \rightarrow \alpha_1 = 68 °;$$
$$\cos^3 \alpha_3 = 0,053$$
$$\text{tg}\alpha_4 = \frac{d_{7,9,12,14}}{h\text{расч}} = \frac{4,9}{2,7} = 1,8 \rightarrow \alpha_1 = 53°;$$
$$\cos^3 \alpha_4 = 0,218$$
$$\text{tg}\alpha_5 = \frac{d_{3,18}}{h\text{расч}} = \frac{6}{2,7} = 2,2 \rightarrow \alpha_1 = 58 °;$$
$$\cos^3 \alpha_5 = 0,149$$
$$\text{tg}\alpha_6 = \frac{d_{8,13}}{h\text{расч}} = \frac{2}{2,7} = 0,7 \rightarrow \alpha_1 = 29 °;$$
$$\cos^3 \alpha_6 = 0,669$$

Выбираем тип светильника ПВЛМ (с 2 лампами ЛБР)

Таблица – 6.1 Светотехнические характеристики светильника

Тип светильника	Освещеность I_α, кд при угле α									
	0	15	25	35	45	55	65	75	85	90
ПВЛМ	175	165	148	130	110	70	60	30	20	–

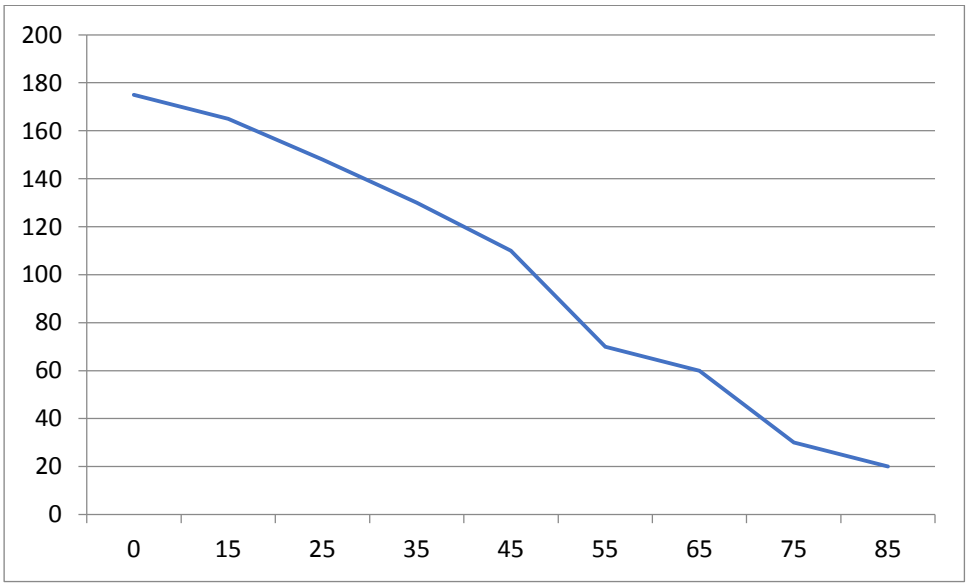


Рисунок – 2 Зависимость α=f(I_α)

По этому углу находим силу света от каждого источника по рисунку 1

$I_{\alpha1(1,5,16,20)}$=45 кд
$I_{\alpha2(2,4,17,19)}$=61 кд
$I_{\alpha3(6,10,11,15)}$=55 кд
$I_{\alpha4(7,9,12,14)}$=77 кд
$I_{\alpha5(3,18)}$=70 кд
$I_{\alpha6(8,13)}$=140 кд

Освещенность помещения относительно контрольной точки от каждого источника:

$$e_{АГ} = \frac{n \cdot I_{\alpha} \cos^3 \alpha}{h_p}$$

$$e_{АГ1} = \frac{4 \cdot 45 \cdot 0{,}0345}{4{,}7^2} = 0{,}45\text{лк};$$

$$e_{АГ2} = \frac{4 \cdot 61 \cdot 0{,}084}{4{,}7^2} = 1{,}5\text{лк};$$

$$e_{АГ3} = \frac{4 \cdot 55 \cdot 0{,}053}{4{,}7^2} = 0{,}85\text{лк};$$

$$e_{АГ4} = \frac{4 \cdot 77 \cdot 0{,}218}{4{,}7^2} = 5\text{лл};$$

$$e_{АГ5} = \frac{2 \cdot 70 \cdot 0{,}149}{4{,}7^2} = 1{,}6\text{лк};$$

$$e_{АГ6} = \frac{2 \cdot 140 \cdot 0.669}{4{,}7^2} = 13{,}9\text{лк};$$

$$\sum_{i=1}^{15} e_{АГi} = e_{АГ1} + e_{АГ2} + e_{АГ3} + e_{АГ4} + e_{АГ5} + e_{АГ6} = 0{,}45+1{,}5+0{,}85+5+1{,}6+13{,}9=23{,}3\text{лк}$$

Суммарная освещенность:

$$E = \frac{\mu \cdot \Phi_л \cdot n}{1000 \cdot K_з} \cdot \sum_{i=1}^{15} e_{АГi}$$

где μ – коэффициент, учитывающий действие «удаленных» светильников (1,1 ÷ 1,25). Световой поток выбранной лампы ЛБР(ЛХБР80) 4160 лм.

$$E_{АГ} = \frac{1.2 \cdot 4160 \cdot 40}{1000 \cdot 1.5} \cdot 23{,}3 = 464{,}521\text{лк}$$

E_{min}=300 лк берем из таблицы 3.12 (см. список литературы первая МУ)
$E_{АГ} \geq E_{min}$(т.к. освещенность незначительно больше нормированного освещения нужно увеличить количетво светильников до 28 шт).

Заключение

В Дипломной работы был проведен расчет естественного освещения. При проверки естественного освещения помещений необходимо определить площадь световых проемов, обеспечивающих нормированное значение КЕО. В особо секретном помещении для шифрования для обеспечения

нормированного значения КЕО, e_N= 0.96, при разряде IIIб окна 25 м^2 не обеспечивает нормирования значение зрительных работ требуется площадь световых проемов равная 40,5 м^2.

Далее был проведен расчет искусственного освещения. Расчет освещенности точечным методом показал, что заданного числа светильников было меньше необходимого для обеспечения достаточной освещенности помещения. Для обеспечения необходимой освещенности помещения необходимо увеличить количество светильников до 15 штуки. Чтобы обеспечить нормальное освещения для человеческого глаза

Заключение

Целью работы являлась реализация скрытого вложения информации в исполняемые Java файлы и также подсчет возможного объема вложения. Было принято вкладывать информацию непосредственно байт-код java посредством перестановки операторов. Достоинство данного метода в том, что он не изменяет размера исполняемого файла и времени его выполнения.

Результатом работы является следующее:

- Разработан и реализован алгоритм вложения в байт-код java посредством перестановки операторов.
- Также разработана программа для вложения с графическим интерфейсом.
- Проведена оценка возможного объёма данной методикой.

Как было сказано ранее данный алгоритм можно использовать для следующих целей:

- вложение электронной цифровой подписи;
- вложение счетчика контроля копирования;
- вложение номера лицензии;
- вложение информации об авторских правах;
- вложение информации о целостности самой программы или ее отдельных частей; В дальнейшем планируется:
- комбинировать данный метод вложения с другими методами вложения с целью увеличения объема вложения.
- реализовать алгоритм вложения и проверки цифровой подписи.

Список литературы

1. Быков С.Ф. Алгоритм сжатия JPEG с позиций компьютерной стеганографии // Защита информации. Конфидент. 2000. № 3.

2. Andrey Krasov., Stanislav Shterenberg. Methods for embedding hidden data into executable scripts.KEY ISSUES IN MODERN SCIENCE - 2014, Sofia, Bulgaria, 9 стр.
3. Красов А.В., Верещагин А.С., Цветков А.Ю. Аутентификация программного обеспечения при помощи вложения цифровых водяных знаков в исполняемый код. // М. Телекоммуникации Спецвыпуск 2013, с.27-30

4. Andrey Krasov, Yuriy Tregubov, Stanislav Shterenberg. Research of copy protection methods software based on embed of digital watermarks into executable and library files. Cambridge Journal of Education and Science, 2015, № 2(14), (July-December). Volume VI. “Cambridge University Press”, 2015. - 642 p. Proceedings of the Journal are located in the Databases Scopus. Source Normalized Impact per Paper (SNIP): 5.275 SCImago Journal Rank (SJR): 5.347. С. 565-573

5. Shterenberg S.I., Krasov A.V., Ushakov I.A..ANALYSIS OF USING EQUIVALENT INSTRUCTIONS AT THE HIDDEN EMBEDDING OF INFORMATION INTO THE EXECUTABLE FILESJournal of Theoretical and Applied Information Technology. 2015. Т. 80. № 1. С. 28-34.

6. Красов А.В., Верещагин А.С., Абатуров В.С., МЕТОДЫ СКРЫТОГО ВЛОЖЕНИЯ ИНФОРМАЦИИ В ИСПОЛНЯЕМЫЕ ФАЙЛЫ, Известия Санкт-Петербургского государственного электротехнического университета ЛЭТИ. 2012.№ 8. С. 51-55.

7. Хомяков И.Н., Красов А.В., АНАЛИЗ ВОЗМОЖНОСТЕЙ СКРЫТОГО ВЛОЖЕНИЯ ИНФОРМАЦИИ В СТРУКТУРУ БАЙТ-КОДА JAVA, В сборнике: Актуальные проблемы инфотелекоммуникаций в науке и образовании II Международная научно-техническая и научно-методическая конференция. 2013. С. 859-861.

8. Барсуков В.С. Стеганографические технологии защиты документов, авЗторских прав и информации // Обзор специальной техники.- 2000.- №2.-С. 31-40.

9. Красов А.В., Штеренберг С.И., РАЗРАБОТКА МЕТОДОВ ЗАЩИТЫ ОТ КОПИРОВАНИЯ ПО НА ОСНОВЕ ЦИФРОВЫХ ВОДЯНЫХ ЗНАКОВ ВНЕДРЯЕМЫХ В ИСПОЛНЯЕМЫЕ И БИБЛИОТЕЧНЫЕ ФАЙЛЫ, В сборнике: Актуальные проблемы инфотелекоммуникаций в науке и образовании II Международная научно-техническая и научно-методическая конференция. 2013. С. 847-852.

Приложение А

Листинг 2.1. Пример исходного кода

```
package java.lang; public class Object {        public final
Class getClass() { . . . }          public String toString() { . . .
}          public boolean equals(Object obj) { . . . }      public
int hashCode() { . . . }            protected Object clone()
throws CloneNotSupportedException { . . . }           public
final void wait()                   throws
IllegalMonitorStateException,
InterruptedException { . . . }
          public final void wait(long millis)
throws IllegalMonitorStateException,
InterruptedException { . . . }  public final void
wait(long millis, int nanos) { . . . }

  throws IllegalMonitorStateException,
InterruptedException { . . . }
          public final void notify() { . . . }
                    throws IllegalMonitorStateException
          public final void notifyAll() { . . . }
                    throws IllegalMonitorStateException
protected void finalize()                   throws
Throwable { . . . }
}
```