МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г.ЧЕРНЫШЕВСКОГО»

Кафедра информатики и программирования

Оптимизация программного кода: на примере программы, реализующей метод LSB стеганографии

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 4 курса 441 группы
направления 02.03.03 Математическое обеспечение и администрирование информационных систем
факультета компьютерных наук и информационных технологий
Зажарнова Кузьмы Андреевича

Научный руководитель:
Доцент _____ Е.В. Кудрина

подпись, дата

Зав. кафедрой:
к.ф.-м.н., доцент _____ М.В. Огнева

подпись, дата

Саратов 2019

ВВЕДЕНИЕ

В настоящее время практически ни одна сфера жизни человека не обходится без какого-либо программного продукта. Разработка программного обеспечения стала очень прибыльной нишей и, как и в любой сфере с большим оборотом денег, любые ошибки могут повлечь за собой очень неприятные последствия. Очень многие решения нуждаются в дальнейшей поддержке и чем непонятнее будет код, тем сильнее усложнится его поддержка и доработка, тем больше времени и денег придётся тратить на поддержание жизненного цикла приложения. Поэтому создание совершенного кода важно, как никогда ранее.

Проектирование архитектуры приложения на начальном этапе его написания безусловно важно, но всё предусмотреть невозможно, потому необходимо улучшать код на протяжении всего его жизненного цикла. Улучшение структуры кода (рефакторинг) должно происходить на протяжении всего времени его создания. Но понятный код далеко не всегда является оптимальным, хотя его, безусловно, гораздо проще поддерживать и оптимизировать.

Оптимизация – важный момент для множества программ, однако, если рефакторинг, как правило, не имеет особых условий для выполнения, то оптимизация приложения может происходить в разных направлениях и разными способами, в зависимости от аппаратных средств, на которых код будет выполняться. Стоит также помнить, что не стоит слепо пытаться «ускорить» код, ведь помимо быстродействия существует ещё множество других характеристик качества кода, о которых не стоит забывать.

Целью бакалаврской работы – разработка и оптимизации программной реализации метода LSB-стеганографии.

Поставленная цель определила следующие **задачи:**

1. Познакомиться с понятиями «оптимизация» и «рефакторинг» кода.
2. Изучить основные методики оптимизации кода.
3. Познакомиться со средствами профилирования среды разработки Visual Studio.

4. Разработать приложение, реализующее метод LSB стеганографии.
5. Провести оптимизацию приложения, используя методики оптимизации программного кода, обосновав их выбор.
6. Оценить качество оптимизации приложения, используя средства профилирования среды разработки Visual Studio.

Методологические основы оптимизация программного кода представлены в работах Мартина Р.[1], Макконела С.[2], Фаулера М.[3]. Алгоритмические основы стеганографии методом LSB рассмотрены в работе Шелухина О[4]. Руководства про профилированию приложений представлены в технической документации [5-7].

Практическая значимость бакалаврской работы.

В работе описаны основные методики оптимизации программного кода, а также приведены примеры оптимизаций на языке C#. Также рассмотрена работа с профилировщиком в среде разработки Visual Studio 2017 и его применение на практике при оптимизации программы, реализующей метод LSB стеганографии.

Структура и объём работы. Бакалаврская работа состоит из введения, 2 разделов, заключения, списка использованных источников и 4 приложений. Общий объем работы – 82 страницы, из них 49 страниц – основное содержание, включая 15 рисунков и 25 таблиц, список использованных источников информации – 22 наименования.

КРАТКОЕ СОДЕРЖАНИЕ РАБОТЫ

Первый раздел «Теоретические основы оптимизации кода» посвящен описанию оптимизации, рефакторинга, их взаимосвязи и различиям, а также обзору основных методик оптимизации и обзору инструментальных средств оптимизации среды разработки Visual Studio 2017.

Были приведены следующие определения:

Оптимизация программного кода подразумевает внесение небольших изменений в корректный код, которые затрагивают небольшой участок кода. Значительные изменения проекта как правило оптимизацией не считаются.

Целями оптимизации являются: уменьшение объема кода; уменьшение объема используемой программой оперативной памяти; ускорение работы программы; уменьшение количества операций ввода и вывода.

Рефакторинг кода – это процесс изменения внутренней структуры программы, не затрагивающий внешнего поведения самой программы и имеющий целью облегчить понимание ее работы.

Виды оптимизации, рассмотренные в работе:

1. Логические выражения
    - Прекращение сразу после получения проверки
    - Упорядочение проверок по частоте
    - Сравнение быстродействия похожих структур логики
    - Замена сложных выражений на обращения к таблице
    - Отложенные вычисления
2. Циклы
    - Размыкание цикла
    - Объединение циклов
    - Развёртывание цикла
    - Минимизация объёма работы внутри цикла
3. Изменения типов данных
    - Использование целых чисел с плавающей запятой
    - Использование массивов с минимальным числом измерений
    - Минимизация количества обращений к массивам
    - Использование дополнительных индексов
4. Использование выражений
    - Алгебраические тождества
    - Снижение стоимости операций

- Инициализация во время компиляции
- Замена некоторых системных методов

Второй раздел «Профилирование программного кода с целью оптимизации» посвящен разработке приложения, реализующего метод LSB стеганографии, его профилированию и оптимизации.

В рамках данного проекта в качестве контейнера были выбраны изображения форматов BMP и PNG, в них каждый пиксель занимает 3-4 байта, где каждый байт отвечает за свой цвет: красный, зелёный и синий, а 4 байт, если он есть, за прозрачность.

Основной идеей программы является кодирование сообщения в картинку, используя информацию о цветах пикселей. В самом начале из первого пикселя с координатами (0; 0) зашифровывается символ, который говорит о наличии сообщения в изображении. Далее, сразу после символа-индикатора, шифруется размер сообщения, который не должен превышать 8 разрядов. Такое разрядность числа выбрана исходя из предположения, что на данный момент широко распространены камеры с разрешение от 10 до 20 мегапикселей. После количества символов в сообщении кодируется уже само сообщение.

Идея состоит в том, чтобы уместить 8 бит, которыми кодируется 1 символ, в пиксель. В каждом пикселе извлекается значение красного, зелёного и синего цветов из объекта структуры Color, который возвращает метод GetPixel(int x, int y) класса Bitmap. В красном цвете заменяются первые 2 бита, в зелёном и синем цвете заменяются по 3 младших бита. После чего все 3 цвета собираются вместе и заменяют исходный пиксель при помощи метода SetPixel(int x, int y, System.Drawing.Color color).

Программа написана в среде разработки Visual Studio на языке программирования C. Графическая часть приложения представлена средствами Windows Forms.

Визуальная составляющая программы представляет собой небольшое окно с двумя вкладками. Первая вкладка Encode представляет из себя 3 элемента CheckBox и 3 кнопки. Каждый элемент CheckBox запрещён к изменению

пользователем и служит для индикации успешности указания пути к изображению и тексту, а также для индикации об успешности выполнения кодировки текста в изображение.

Вторая вкладка Decode позволяет извлечь сообщение из картинки, если оно там зашифровано при помощи описываемой программы. Также есть 2 элемента CheckBox, один из которых говорит об успешности указания пути до картинки, а второй об успешности извлечения сообщения. При этом, если указать путь до картинки в первой вкладке, то индикация об указании пути до картинки во второй вкладке будет отключена, поскольку за эту информацию в программе отвечает одно поле.

Описанная выше программа оптимизировалась при помощи средств профилирования среды разработки Visual Studio 2017.

Полученные в ходе профилирования результаты показали, что стоит обратить внимание на метод BitToByte. В нём можно увидеть метод возведения в степень Math.Pow. Вызов метода возведения в степень будет вызываться в худшем случае 8 раз, каждый раз производя возведения в степень одного и того же числа заново. Помимо этого, в методе присутствует приведение типа double к типу byte, что также тратит ресурсы.

```
public static byte BitToByte(BitArray src)
{
    byte num = 0;
    for (int i = 0; i < src.Count; i++)
    {
        if (src[i] == true)
            num += (byte)Math.Pow(2, i);
    }
    return num;
}
```

В данной ситуации отличным решением кажется создание таблицы, которая будет содержать в себе степени двойки от 0 до 7 для обращения из

метода по индексу. Также не стоит забывать о приведении типов. В таком случае код примет вид:

```
private const byte[] powOf2 = {1, 2, 4, 8, 16, 32, 64, 128};
  …
public static byte BitToByte(BitArray src)
{
  byte num = 0;
  for (int i = 0; i < src.Count; i++)
    if (src[i] == true)
      num += powOf2[i];
  return num;
}
```

Результаты профилирования и замеры времени выполнения участка кода, приведённые в таблице 1, показали улучшение производительности на 19%. Также данная оптимизация не ухудшила читаемость кода, а значит её стоит оставить.

Дальнейшая оптимизация касалась главных циклов методов Encode и Decode, структура которых схожа. Потому результаты оптимизации цикла были рассмотрены только на примере метода Encode, главный цикл которого выглядит следующим образом.

Таблица 1 – Время выполнения метода Encode в мс до оптимизации и после оптимизации метода BitToByte

Кол-во символов (шт)	До оптимизации					После оптимизации				
48	1	0	0	0	0	1	0	0	0	0
	0,2					0,2				
15515	39	38	38	39	39	40	31	32	31	31
	38,6					33				
310338	792	780	779	782	780	636	638	642	643	631
	782,6					638				
1862038	4726	4713	4751	4752	4709	3868	3860	3803	3796	3831
	4730,2					3831,6				

```
int index = 0;
bool st = false;
for (int j = 1; j < img.Height; j++)
{
   for (int i = 0; i < img.Width; i++)
   {
      if (index == textLength)
      {
         st = true;
         break;
      }
      WritePixel(img, i, j, list[index]);
      index++;
   }
   if (st)
   {
      break;
   }
}
```

В данной конструкции сразу бросаются в глаза 2 временные переменные, которые проверяются каждую итерацию цикла. Избавление от этих переменных улучшит читаемость кода, а также уберёт проверки каждую итерацию, что видно на примере кода ниже.

```
for (int i = MessageLength + 1; i < textLength + MessageLength + 1; i++)
{
   WritePixel(img, i % width, i / width, list[i - MessageLength - 1]);
}
```

Пожалуй, главной проблемой такого решения является то, что каждую итерацию цикла будут вычисляться координаты пикселя, что может даже ухудшить время выполнения цикла. Для оценки успешности оптимизации были проведены замеры времени выполнения, приведённые в таблице 2.

Таблица 2 – Время выполнения метода Encode в мс до и после оптимизации цикла в теле метода

Кол-во символов (шт)	До оптимизации					После оптимизации				
48	1	0	0	0	0	1	0	0	0	0
	0,2					0,2				
15515	40	31	32	31	31	30	30	30	29	30
	33					29,8				
310338	636	638	642	643	631	614	607	617	609	605
	638					610,4				
1862038	3868	3860	3803	3796	3831	3673	3639	3632	3643	3668
	3831,6					3651				

По полученным результатам видно, что оптимизация принесла примерно 5% увеличение быстродействия, а учитывая то, что в результате данной оптимизации код стал гораздо более удобным для чтения и понимая, оптимизацию однозначно стоит оставить в данном виде.

Далее была предпринята попытка распараллелить цикл при помощи метода Parallel.For().

```
Parallel.For (MessageLength + 1, textLength + MessageLength + 1,
(i) =>
 {
    WritePixel(img, i % width, i / width, list[i - MessageLength - 1]);
 });
```

Время выполнения участка кода до и после распараллеливания приведено в таблице 3.

По результатам из таблицы видно, что оптимизация прошла неудачно. Это можно объяснить тем, что метод WritePixel, который вызывается в цикле, содержит методы GetPixel() и SetPixel(), которые медленно работают и ко всему прочему не являются потокобезопасными, из-за чего приходится применять оператор lock. Этот нюанс в паре с затраченным на распараллеливание цикла временем не дал никакого выигрыша во времени, а даже наоборот, поэтому от этой оптимизации однозначно нужно отказаться.

Таблица 3 - Время выполнения метода Encode в мс до и после распараллеливания

Кол-во символов (шт)	До оптимизации					После оптимизации				
48	1	0	0	0	0	4	0	0	0	0
	0,2					0,8				
15515	30	30	30	29	30	46	45	45	45	45
	29,8					45,2				
310338	614	607	617	609	605	905	891	895	893	905
	610,4					897,8				
1862038	3673	3639	3632	3643	3668	5433	5510	5481	5478	5488
	3651					5478				

После всех оптимизаций, описанных выше, не осталось тех методов, которые можно оптимизировать, однако осталась одна оптимизация, к которой, пожалуй, не стоит прибегать никогда – инлайнинг. Суть данной оптимизации заключается в том, что вызываемый метод замещается кодом, содержащимся в этом методе. В результате этого не тратится время на обращение к таблице методов. Конечно, такую оптимизацию имеет смысл проводить только в методах, которые вызываются очень часто, а потому в цикл, который уже рассматривался выше, будет заинлайнен метод WritePixel, который в ходе тестирования программы вызывался до 1 862 038 раз. Результаты данной оптимизации приведены в таблице 4.

Таблица 4 – Время выполнения метода Encode в мс до и после инлайнинга метода WritePixel

Кол-во символов (шт)	До оптимизации					После оптимизации				
48	1	0	0	0	0	1	0	0	0	0
	0,2					0,2				
15515	30	30	30	29	30	30	30	29	29	29
	29,8					29,4				
310338	614	607	617	609	605	614	611	611	601	594
	610,4					606,2				
1862038	3673	3639	3632	3643	3668	3660	3647	3623	3654	3629
	3651					3642,6				

В результате данной оптимизации время выполнения кода возросло на 0,2%, что с учётом ухудшения читаемости кода и увеличения его объёма говорит о необходимости отказа от данной оптимизации из-за её низкой результативности и затруднения поддержки кода в дальнейшем.

Таким образом, класс SteganographyLSB удалось оптимизировать, время работы библиотеки сократилось на 23%.

ЗАКЛЮЧЕНИЕ

В ходе данной работы были решены все поставленные задачи: изучены методики оптимизации кода, средства профилирования Visual Studio 2017 Community, а также продемонстрирована оптимизация кода на примере программы, реализующей метод LSB стеганографии. Результативность оптимизации отслеживались средствами профилирования. Таким образом, цель работы достигнута.

Действительно не существует универсального средства оптимизации кода и очень многое зависит как от языка, так и от конкретной задачи. Однако оптимизацию гораздо проще проводить с использованием средств профилирования, в чём мы убедились в данной работе.

Основные источники информации:

1. Мартин Р. Чистый код: создание, анализ и рефакторинг. Библиотека программиста. – СПб.: Питер, 2013. – 464 с.: ил.
2. Макконел С. Совершенный код. Мастер - класс / Пер. с англ - М.: Издательство "Русская редакция", 2010. – 895 с., ил.
3. Фаулер М. Рефакторинг. Улучшение существующего кода. / Пер. с англ. - Спб: Символ-Плюс, 2003. – 432 с., ил.
4. Шелухин О. Стеганография. Алгоритмы и программная реализация. – Горячая линия – Телеком, 2017. – 592 с.: ил.
5. Краткое руководство. Первое знакомство со средствами профилирования. [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/visualstudio/profiling/profiling-feature-tour?view=vs-2017 (дата обращения: 28.05.2019). Загл. с экрана. Яз. рус.
6. Общие сведения о сеансе анализа производительности. [Электронный ресурс]. URL: https://docs.microsoft.com/ru-

ru/visualstudio/profiling/performance-session-overview?view=vs-2017 (дата обращения: 28.05.2019). Загл. с экрана. Яз. рус.

7. Краткое руководство. Общие сведения о методах сбора данных о производительности. [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/visualstudio/profiling/understanding-performance-collection-methods?view=vs-2019 (дата обращения: 28.05.2019). Загл. с экрана. Яз. рус.