МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПЕУБЛИКИ БЕЛАРУСЬ

Учреждение образования   
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Информационных систем и технологий

Специальность 1-40 05 01 Информационные системы и технологии

Направление специальности 1-40 01 02 03 Информационные системы

и технологии (издательско-полиграфический комплекс)

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**КУРСОВОГО ПРОЕКТА:**

по дисциплине «Защита информации и надежность информационных систем»

Тема: «Исследование и реализация блочных шифров TEA и DFC»

|  |  |  |
| --- | --- | --- |
| **Исполнитель** |  | |
| Студент 3 курса группы1 | подпись, дата | Е. К. Шимчёнок  инициалы и фамилия |
|  |  |  |
| **Руководитель** |  |  |
| Ассистент | подпись, дата | М. Г. Савельева  инициалы и фамилия |

|  |  |
| --- | --- |
| **Курсовая работа защищена с оценкой** | **\_\_\_\_\_\_\_\_\_\_** |
| **Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  (подпись) | М. Г. Савельева  инициалы и фамилия |

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПЕУБЛИКИ БЕЛАРУСЬ

Учреждение образования   
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий   
Кафедра информационных систем и технологий

Утверждаю

Заведующий кафедрой

\_\_\_\_\_\_\_\_\_\_\_\_ В.В. Смелов

подпись инициалы и фамилия

“\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2023 г.

**ЗАДАНИЕ**

**к курсовому проектированию по дисциплине**«Защита информации и надежность информационных систем»

Специальность: 1-40 05 01 03 «Информационные системы и технологии (издательство-полиграфический комплекс)»

Курс: 3, группа: 1

Студент: Шимчёнок Елизавета Константиновна

**Тема:** Исследование и реализация блочных шифров TEA и DFC.

**1. Срок сдачи студентом законченной работы: «05» мая 2023г.**

**2. Исходные данные к проекту:**

**2.1. Функционально должны быть выполнены следующие задачи:**

* Изучение блочных шифров, выявление их недостатков
* Разработка программных функций, реализующих алгоритмы шифрования и дешифрования в соответствии с темой
* Создание приложения с графическим интерфейсом, реализующего разработанные функции

**2.2. Требования:**

* Необходимо провести аналитический обзор литературы по теме проекта
* Необходимо описать сферу применимости метода
* Программное средство может быть разработано на любом языке
* Архитектура приложения выбирается разработчиком
* Листинги проекта должны содержать комментарии

**3. Содержание расчетно-пояснительной записки:**

* Введение
* Постановка задачи
* Описание метода
* Описание программного средства
* Тестирование программного средства
* Руководство пользователя
* Заключение
* Список используемых источников
* Приложения

**4. Форма представления выполненного курсового проекта:**

* + Теоретическая часть курсового проекта должна быть представлена в формате MS Word.
  + Оформление записки должно быть согласно правилам.
  + Необходимые схемы, диаграммы и рисунки допускается делать в MS Office Visio или копии экрана (интерфейс).
  + Полные листинги проекта представляются в приложении.
  + К записке необходимо приложить CD (DVD), который должен содержать: пояснительную записку, листинги и файлы базы данных.

**Календарный план**

|  |  |  |  |
| --- | --- | --- | --- |
| №  п/п | Наименование этапов курсового проекта | Срок выполнения этапов проекта | Примечание |
| 1 | Задание курсового проекта. Титульный лист | 10.02.2023 |  |
| 2 | Введение | 24.03.2023 |  |
| 3 | Аналитический обзор литературы по теме проекта | 10.03.2023 |  |
| 4 | Реализация методов | 24.03.2023 |  |
| 5 | Разработка прототипа программного средства | 30.03.2023 |  |
| 6 | Разработка программного средства | 08.04.2023 |  |
| 7 | Тестирование программного средства | 15.04.2023 |  |
| 8 | Написание руководства пользователя | 22.04.2023 |  |
| 9 | Оформление пояснительной записки | 30.04.2023 |  |
| 10 | Сдача проекта | 05.05.2023 |  |

**5. Дата выдачи задания «10» февраля 2023г.**

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ *М. Г. Савельева*

(подпись)

Задание принял к исполнению \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(дата и подпись студента)

Оглавление

[Введение 5](#_Toc134135320)

[1 Аналитический обзор литературы 6](#_Toc134135321)

[1.1 Алгоритм TEA 6](#_Toc134135322)

[1.2 Алгоритм DFC 7](#_Toc134135323)

[1.3 Обзор аналогов 8](#_Toc134135324)

[1.4 Вывод по разделу 10](#_Toc134135325)

[2 Описание методов 11](#_Toc134135326)

[2.1 Алгоритм TEA 11](#_Toc134135327)

[2.1.1 Генерация ключа 11](#_Toc134135328)

[2.1.2 Шифрование 11](#_Toc134135329)

[2.1.3 Расшифрование 12](#_Toc134135330)

[2.2 Алгоритм DFC 12](#_Toc134135331)

[2.2.1 Генерация ключа 12](#_Toc134135332)

[2.2.2 Шифрование 13](#_Toc134135333)

[2.2.3 Расшифрование 15](#_Toc134135334)

[2.3 Исследование блочных методов TEA и DFC 15](#_Toc134135335)

[2.4 Вывод по разделу 18](#_Toc134135336)

[3 Описание программного средства 19](#_Toc134135337)

[3.1 Разработка 19](#_Toc134135338)

[3.2 Тестирование 21](#_Toc134135339)

[3.3 Руководство пользователя 24](#_Toc134135340)

[3.4 Вывод по разделу 26](#_Toc134135341)

[Заключение 27](#_Toc134135342)

[Список используемых источников 28](#_Toc134135343)

[Приложение А 29](#_Toc134135344)

[Приложение Б 40](#_Toc134135345)

[Приложение В 43](#_Toc134135346)

[Приложение Г 48](#_Toc134135347)

# **Введение**

Обеспечение защиты информации всегда было и остается актуальным направлением. В связи с быстрым развитием информационных технологий возникает необходимость в и защите информации от несанкционированного доступа, а также в шифровании данных.

Шифрование – это способ изменения сообщения, обеспечивающее искажение его содержимого с целью сокрытия смысла текста. Оно является лучшей имеющейся в наличии технологией, для защиты данных от злоумышленников. На текущий момент оно достигло такого уровня развития, что при корректном использовании его практически невозможно взломать.

Предметной областью данного курсового проекта является блочный алгоритм шифрования. Его целью является рассмотрение и реализация двух шифров: *DFC* и *TEA*. Для их использования будет разработано соответствующее программное средство, которое будет иметь простой и понятный интерфейс.

Приложение предоставит пользователю возможность шифрования и расшифрования сообщения. При реализации этих операций результат будет отображаться в соответствующих окнах, где пользователь сможет наглядно проанализировать используемый метод.

Чтобы облегчить работу пользователю с программным средством или помочь в случае возникновения затруднительных ситуаций, будет составлено подробное руководство пользователя. Оно будет содержать описание выполнения важнейших операций.

Блочные шифры широко используются в современной криптографии, и многие стандартные алгоритмы, такие как *AES*, *DES* и 3*DES*, являются блочными шифрами. Они обычно используются для шифрования данных в компьютерных сетях, безопасных системах обмена сообщениями и других приложениях, где важны конфиденциальность и безопасность.

Целью этого курсового проекта является изучение и применение двух блочных шифров, а именно *DFC* и *TEA*. Для достижения этой цели был определен ряд задач, которые включают в себя:

– создание программного обеспечения с понятным интерфейсом;

– исследование процесса шифрования для одного раунда каждого алгоритма;

– разработка функций для шифрования/расшифрования сообщения;

– составление подробного руководства пользователя по интерфейсу ПО.

# **Аналитический обзор литературы**

Блочный шифр – это тип алгоритма шифрования с симметричным ключом, который работает с блоками данных фиксированного размера, обычно 64 или 128 бит за раз. В блочном шифре открытый текст делится на блоки также фиксированного размера, а затем шифруется с использованием секретного ключа для получения зашифрованного текста. Тот же ключ используется для расшифровки шифротекста в исходное сообщение.

Блочные шифры работают, применяя ряд математических операций к каждому блоку данных способом, который зависит от секретного ключа. Эти операции могут включать подстановку, перестановку и другие преобразования, предназначенные для того, чтобы злоумышленнику было трудно восстановить открытый текст из зашифрованного текста без ключа.

Одной из важных характеристик блочных шифров является их способность обеспечивать конфиденциальность, целостность и подлинность сообщений. Путем шифрования сообщения с помощью блочного шифра и использования криптографических методов, таких как коды аутентификации сообщений и цифровые подписи, можно гарантировать, что сообщение не было изменено или перехвачено злоумышленником.

## **Алгоритм TEA**

*TEA* (*Tiny Encryption Algorithm*)–блочный алгоритм шифрования типа «Сеть Фейстеля». Алгоритм был разработан на факультете компьютерных наук Кембриджского университета Дэвидом Уилером и Роджером Нидхэмом.

Алгоритм шифрования *TEA* основан на битовых операциях с 64-битным блоком, имеет 128-битный ключ шифрования. Стандартное количество раундов для 64-битного блока равно 32 [1].

*TEA* известен своей простотой и эффективностью, но его также критикуют за безопасность. В алгоритме есть известные недостатки, особенно в отношении атак с использованием связанных ключей, и обычно его не рекомендуется использовать в приложениях с высоким уровнем безопасности.

Преимущества:

– *TEA* прост в реализации и имеет небольшой размер кода, что делает его пригодным для использования во встроенных системах с ограниченными ресурсами.

– шифрование и дешифрование занимают всего несколько циклов на байт на большинстве современных процессоров.

– занимает мало памяти.

Недостатки *TEA*:

– *TEA* имеет размер блока всего 64 бита, что относительно мало по сравнению с другими современными блочными шифрами. Это делает его более уязвимым для определенных типов атак, особенно если он используется для шифрования нескольких сообщений одним и тем же ключом.

– *TEA* подвержен атакам со связанными ключами, когда злоумышленник, знающий взаимосвязь между двумя ключами, может использовать эти знания для взлома шифра.

– не является шифром сети замены-перестановки (*SPN*). Это может сделать его более уязвимым для определенных типов атак. Например, атака методом перебора, атака по выбору открытого текста или атаки на ключи.

– *TEA* поддерживает ключи размером всего 128 бит, чего может быть недостаточно для приложений, требующих более надежной криптографической защиты.

В целом, *TEA* – это простой и эффективный алгоритм шифрования, но он не считается достаточно безопасным для использования в приложениях, требующих надежной криптографической защиты. Он может подходить для использования в приложениях с низким уровнем безопасности или там, где скорость и эффективность важнее безопасности. Однако стоит отметить, что *TEA* не имеет серьезных проблем с криптостойкостью за исключением атак на связанных ключах [2].

## **Алгоритм DFC**

*Decorrelated Fast Cipher* (*DFC*) – это алгоритм блочного шифра, разработанный французской командой в 1998 году под руководством Сержа Воденэ в качестве кандидата на участие в конкурсе *AES* (*Advanced Encryption Standart*), который проводился Национальным институтом стандартов и технологий (*NIST*) в конце 1990-х годов. Общая цель состояла в том, чтобы разработать Федеральный стандарт обработки информации (*FIPS*), определяющий алгоритм шифрования, способный защищать конфиденциальную правительственную информацию даже в 21 веке. Декоррелированный быстрый шифр не прошел в финал [3].

Это шифр с симметричным ключом, что означает, что он использует один и тот же ключ как для шифрования, так и для дешифрования.

*DFC* предназначен для обеспечения высокоскоростного шифрования и дешифрования, а также для надежной защиты. Он работает со 128-битными блоками и поддерживает ключи переменной длины до 256 бит.

Шифр состоит из двух основных компонентов: линейного преобразования и нелинейного преобразования. Линейное преобразование – это матричное умножение, которое служит для смешивания битов входного блока. Нелинейное преобразование представляет собой операцию замены, применяемую к смешанному блоку. *DFC* использует блок подстановки (*S*-блок), устойчивый к дифференциальному и линейному криптоанализу.

Одной из уникальных особенностей *DFC* является его механизм декорреляции, который призван обеспечить дополнительную безопасность, усложняя прогнозирование вывода шифра. Механизм декорреляции работает, применяя случайное вращение к выходу нелинейного преобразования, прежде чем он будет возвращен обратно в линейное преобразование. Это помогает устранить любые шаблоны или корреляции в выводе шифра, делая его более устойчивым к атакам [4].

*DFC* имеет ряд преимуществ и недостатков, которые описаны ниже.

Преимущества:

– высокая скорость шифрования на 64-битных платформах;

– требует очень мало памяти (менее ста байтов оперативной памяти и менее 2 Кбайт энергонезависимой памяти);

– поддержка различных размеров ключей до 256 бит, предоставляя пользователям возможность выбора соответствующего размера ключа в зависимости от их требований к безопасности.

Недостатки:

– низкая скорость шифрования на всех платформах, кроме 64-битных;

– существует несколько классов слабых ключей (использование ключа из одного из этих классов даже приводит к тому, что результат шифрования любого блока эквивалентен исходному, то есть незашифрованному блоку);

– Ларсом Кнудсеном и Винсентом Риджменом было доказано, что теория, лежащая в основе данного алгоритма, не гарантирует безопасности против дифференциального криптоанализа;

– *DFC* – это сложный алгоритм, который может затруднить реализацию и проверку по сравнению с некоторыми другими шифрами.

В целом, *DFC* – ненадежный метод шифрования. Он имеет несколько классов слабых ключей, а также недостатки в процессе расширения ключа. Уникальный метод декорреляции, на котором основан данный шифр, также, как выяснили вышеперечисленные криптологи, не гарантирует безопасность против дифференциального криптоанализа. *DFC* также не считается защищенным от современных атак, и его не следует использовать для новых криптографических приложений [2].

## **Обзор аналогов**

Выявление ряда серьезных уязвимостей и слабых мест в исходном алгоритме *TEA* привело к скорому созданию его расширений. Основными отличиями всех этих алгоритмов являются усовершенствованное расписание ключей, динамическая зависимость ключа от текста, а также другой размер ключа, входного блока и/или количество раундов сети Фейстеля [8].

*XTEA* имеет размер блока, равный 64 битам, размер ключа – 128 битам, количество раундов сети Фейстеля равно 64. Алгоритм был разработан Дэвидом Уилером и Роджером Нидхэмом и опубликован в 1997 году. Главное отличие от исходного алгоритма *TEA* – наличие алгоритма расписания ключей, что позволило устранить критическую уязвимость для «атак на связанных ключах», но привело к ухудшению стойкости к дифференциальному криптоанализу.

В 1998 году было опубликовано следующее расширение алгоритма, получившее название *XXTEA*. Размер ключа – 128 бит. Отличительной особенностью является возможность шифрования любых блоков, длина которых кратна 64 битам, количество раундов равно 52 + 6∙*M* (количество 32-битных слов в блоке) или 52 + 12∙*M* при длине блока 64**∙***M* бит. Практическая эффективность опубликованной анонимно дифференциальной атаки не доказана.

Существует так же альтернативная модификация алгоритма *TEA*, получившая наименование *RTEA*, разработанная в 2007 году. Размер блока – 64 бита; для 128-битного ключа число раундов сети Фейстеля равно 48, для 256-битного – 64. По заявлениям разработчиков этот алгоритм производительнее и более устойчив к криптоанализу, чем *XTEA*, однако и на этот алгоритм уже существует «атака на связанных ключах».

С использованием механизмов генетического программирования в 2006 году командой разработчиков во главе с Хулио Кастро был создан алгоритм *Raiden*, призванный устранить уязвимости шифра *TEA*. Он практически в точности повторяет структуру алгоритма *TEA*, за исключением того, что у алгоритма *Raiden* есть расширенный алгоритм расписания ключей. Стандартное число раундов сети Фейстеля равно 32 (16 циклов). *Raiden* использует ключевое расписание, трансформирует ключ и генерирует подключи для каждого раунда. Шифр успешно проходит тесты *Diehard*, *Sexton* и *ENT*.

В настоящее время, в реальных условиях алгоритм *TEA* на практике используется редко. Одним из немногих программных средств, использующих данный алгоритм является *Suite Agnot StrongBox Security*. Он использует любой из девятнадцати алгоритмов шифрования для защиты вашей информации. В них входят *BlowFish*, *TwoFish*, *CAST*-128, *Cast*-256, *DES*, 3*DES*, *ICE*, *IDEA*, *MARS*, *RC*4, *Rijndael* (*AES*) и, наконец, *TEA*. Уникальные пароли создаются, используя внутренний генератор случайных ключей. Поддерживаются различные размеры паролей, например, длина строки 8, 16, 24, 32 символа.

Альтернативных модификаций алгоритма *DFC* фактически не создавалось. После обнаружения нескольких классов слабых ключей и слабости в процедуре расширения ключа, авторы алгоритма предложили новую версию алгоритма *DFCv*2 с незначительными изменениями в процедуре расширения ключа, устраняющими все проблемы с ключами. Кроме того, была предложена новая реализация алгоритма, еще быстрее работающая на 64-битных платформах, однако на результаты конкурса *AES* это уже никак не повлияло [7].

Также рассмотрим программные средства, использующие другие алгоритмы шифрования.

*AxCrypt* – простая в использовании программа для шифровки данных, которая поддерживает алгоритмы шифрования *AES*-128 и *SHA*-1. Программа интегрируется с *Windows*, для осуществления сжатия и шифрования, дешифрования, удаления, отправки, работы с конкретными файлами.

Программное обеспечение для шифрования данных *CryptoForge* позволяет защитить конфиденциальные данные везде, где они используются с надежным шифрованием *AES*. Он имеет модуль шифрования файлов, который позволяет шифровать файлы и папки любого типа и размера на любых носителях, включая внешние *USB*-накопители, мобильные устройства, флэш-память и сетевые папки. Он также включает в себя уничтожитель файлов для безопасного удаления данных. Шифрование электронной почты также очень простое, поскольку получатели зашифрованных файлов не должны устанавливать программное обеспечение, чтобы иметь возможность расшифровывать файлы. Модуль шифрования текста преобразует все что угодно в зашифрованный текст. Это приложение для шифрования включает в себя четыре алгоритма шифрования: *Rijndael*, *Blowfish*, *TripleDES* и *Gost*.

*CrococryptFile* – это инструмент шифрования файлов, который создает зашифрованные архивы произвольных файлов и папок. Функции шифрования *CrococryptFile* можно сравнить с утилитой *ZIP*, которая использует шифрование *AES ZIP*. Тем не менее, есть значительные различия. *CrococryptFile* шифрует всю информацию о файлах и папках, включая имена файлов, время и дату и информацию о размере файлов, маскирует любую информацию о содержимом архива. Расширение .*croco* исключительно показывает, что это действительно архив *CrococryptFile* и какой криптографический набор (например, шифрование *AES*-256 на основе пароля) используется.

## **Вывод по разделу**

В данном разделе были рассмотрены два блочных шифра, приведены их общие характеристики и выявлены их преимущества и недостатки. Также отдельное внимание было уделено аналогам реализуемых методов и приложения в целом. У каждого метода есть свои плюсы и минусы, на что не стоит закрывать глаза при выборе шифра, поскольку качество выбранного шифра может существенно влиять на степень защиты конфиденциальных данных. Далее будут рассмотрены разработка и тестирование программного средства.

# **Описание методов**

## **Алгоритм TEA**

### **Генерация ключа**

Ключ длиной 128 бит делится на четыре 32-битных подключа *K*0, *K*1, *K*2 и *K*3. Из-за простого расписания ключей каждый ключ имеет 3 эквивалентных ключа. Это означает, что эффективная длина ключа составляет всего 126 бит [5].

### **Шифрование**

Исходный текст разбивается на блоки по 64 бита каждый. При шифровании используются ранее сгенерированные четыре 32-битных подключа *K*0, *K*1, *K*2 и *K*3. Каждый 64-битный блок шифруется на протяжении 32 раундов по нижеприведенному алгоритму.

На вход раунда поступают правая и левая часть блока (*L*, *R*) по 32 бита каждая. На выходе будут левая и правая части (*Ln*+1, *Rn*+1), которые вычисляются по следующим правилам:

Левая часть формулируется так: *Ln*+1 = *Rn*.

Правая часть зависит от четности раунда. Если нечётные раунды, то используется формула 2.1.

*Rn*+1 = *Ln* + (([*Rn* << 4] + *K*0) *XOR* (*Rn*+*i* + *δ*) *XOR* ([*Rn* >> 5] + *K*1). (2.1)

Если чётные раунды, то используется формула 2.2.

*Rn*+1 = *Ln* + (([*Rn* << 4] + *K*2) *XOR* (*Rn*+*i* + *δ) XOR* ([*Rn* >>5] *+ K*3))*.* (2.2)

В процессе преобразования блока исходной информации используются три действия:

– побитовое исключающее или (*XOR*);

– сложение по модулю 232 (+);

– побитовые сдвиги на переменное число бит («<<», «>>»).

Также используется константа *δ* = 2654435769, которая была выведена из Золотого сечения.

Схема шифрования одного блока исходного сообщения показана на рисунке 2.1 [6].

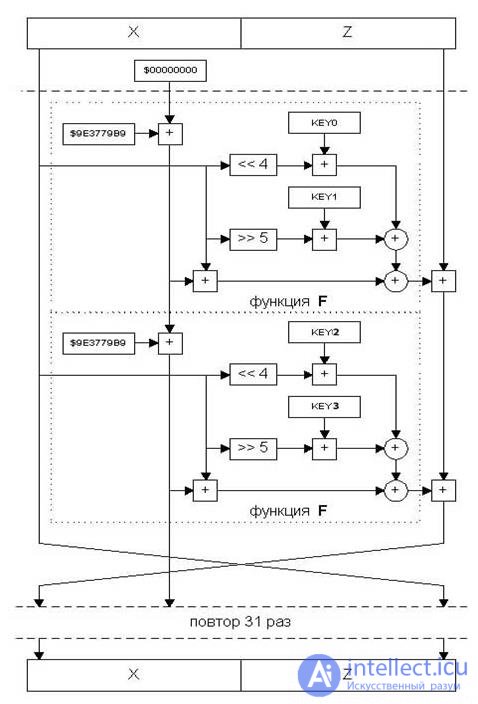


Рисунок 2.1 – Схема шифрования одного блока данных

### **Расшифрование**

Для расшифрования используется аналогичный алгоритм. Его отличие от алгоритма расшифрования является использование точно таких же ключей, но только в обратном порядке [6].

## **Алгоритм DFC**

### **Генерация ключа**

DFC использует восемь различных раундовых ключей: *Ki*, где *i*=1...*n*, *n*=8, по 128 бит, получаемых из одного исходного ключа шифрования *K*.

Для создания последовательности 1, *RK*2, ..., *RK*8 из ключа *K* длиной не более 256 бит, используется следующий процесс. Сначала дополняется ключ *K* предопределенной постоянной цепочкой *KS*, которая усекается до 256 бит, чтобы получить «*Padded Key*» 256(*K*|*KS*). Если *K* имеет длину 128 бит, то только первые 128 бит *K* используются.

Затем *K* разбивается на 8 цепочек по 32 бита *PK*1, ..., *PK*8, таких что 1|...|*PK*8.

Далее определяются 64-битные промежуточные параметры (формулы 2.3 – 2.6):

*OA*1 = *PK*1|*PK*8; (2.3)

*OB*1 = *PK*5|*PK*4; (2.4)

*EA*1 = *PK*2|*PK*7; (2.5)

*EB*1 = *PK*6|*PK*3. (2.6)

Также вычисляются следующие выражения для *i* = 2,3,4:

*OAi* = *OA*1 *XOR* *KAi*; (2.7)

*OBi = OB*1 *XOR KBi*; (2.8)

*EAi = EA*1 *XOR KAi*; (2.9)

*EBi = EB*1 *XOR KBi*, (2.10)

где *KAi* и *KBi* – 64-битные константы.

Конкатенацией определенных величин из вычисленных на предыдущем шаге получаются два временных ключа, соответствующие формулам 2.11 и 2.12:

*Kt*1=*OA*1|*OB*1|...|*OA*4|*OB*4; (2.11)

*Kt*2=*EA*1|*EB*1|…|*EA*4|*EB*4. (2.12)

После этого вычисляются 128-битные ключи раундов (формулы 2.13, 2.14):

*Ki* = *Enc*4(*Ki*-1, *Kt*1) – для нечетных значений *i*; (2.13)

*Ki* = *Enc*4(*Ki*-1, *Kt*2) – для четных значений *i*, (2.14)

где: *Enc*4(*p*, *q*) – функция, выполняющая 4-раундовое шифрование блока данных *p* на ключе *q*; в качестве блока *K*0 берется 128 нулевых битов [2].

### **Шифрование**

Исходный текст разбивается на блоки по 128 бита каждый. При шифровании используется функция шифрования *K*. Каждый блок шифруется на протяжении 8 раундов (рис. 2.2).

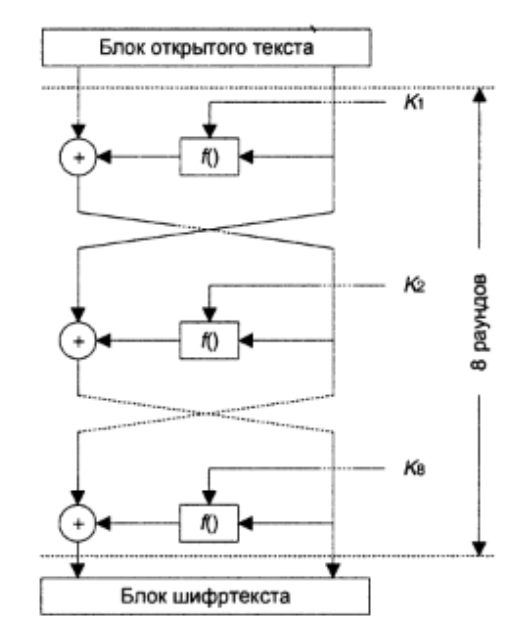


Рисунок 2.2 – Структура алгоритма DFC

В каждом раунде производят некоторые действия, описанные ниже. Упрощенная схема действий одного раунда изображена на рисунке 2.3.

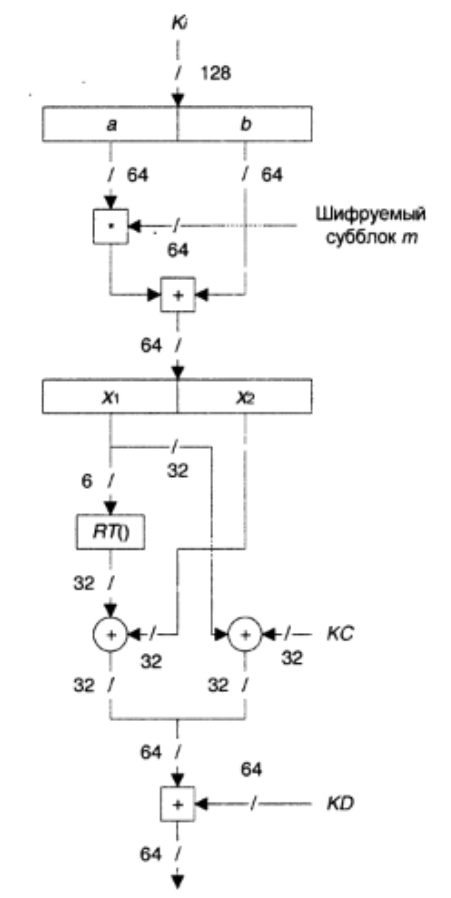


Рисунок 2.3 – Функция раунда алгоритма DFC

Секретный ключ сначала преобразуется в цепочку из 1024 бит («*Expanded Key*») с помощью функции («*Expanding Function*»). Затем выполняется четырехуровневая схема Фейстеля [7]. Процесс генерации уникальных ключей описан в разделе 2.2.1.

В каждом раунде используются 128-битные ключи *Ki*, он разбивается на два 64-битных фрагмента: *a* и *b*.

Далее вычисляется промежуточное значение *x* по формуле 2.15:

*x* = ((¬*a* ∙ (¬*m*) *+* (*¬b*)) *mod* (264+13)) *mod* 264, (2.15)

где ¬*m* – значение 64-битного шифруемого субблока.

Затем 64-битное значение *x* разбивается на два 32-битных фрагмента *x*1 и *x*2.

Над результатом предыдущего шага выполняется операция *CP* (*Confusion Permutation*, перемешивающая перестановка), представляющая собой преобразование по формуле 2.16.

*y* = (¬(*x*2 *XOR RT*¬(*trunc*6(*x*1)) **∙** (*x*1 *XOR KC*))+(¬*KD*)) *mod* 264, (2.16)

где:

**∙ –** операция конкатенации;

*trunc*n – усечение битовой строки до *n* левых битов;

*KC* и *KD* – фиксированные константы.

*RC* – табличная замена, замещающая входное 6-битное значение 32-битным выходом. Таблица является псевдослучайной, в качестве ее заполнения использовалась шестнадцатеричная запись дробной части математической константы *e*. Всего в таблице 64 записи – по количеству возможных 6-битных значений (26) [6].

### **Расшифрование**

Расшифровывание соответствует алгоритму шифрования. Единственное отличие состоит в том, что ключи используются те же (*Ki*, где *i* = 1, 2…8), но в обратном порядке [6].

## **Исследование блочных методов TEA и DFC**

Одним из наиболее важных критериев производительности алгоритмов шифрования является время выполнения. Время выполнения зависит от многих параметров, таких как структура алгоритма, количество раундов и выбранное целевое устройство.

Для оценки производительности блочных шифров *TEA* и *DFC* был проведен эксперимент, в котором измерялось время, необходимое для шифрования и расшифрования блоков данных. Для тестирования использовались данные размера 2 МБ. Ключ шифрования был фиксирован, состоял из 16 символов и был случайно сгенерирован: 4*A*6*B*12*F*6*E*8*D*03*C*98.

Время шифрования составило в среднем 93,4 миллисекунды и 745,3 миллисекунды для *TEA* и *DFC* соответственно. Время расшифрования показало в среднем 72,1 миллисекунды и 931,6 миллисекунды для *TEA* и *DFC* соответственно. Промежуточные значения вычислений изображены на рисунке 2.4.



Рисунок 2.4 – Промежуточные значения скорости шифрования и дешифрования

Если сравнивать *DFC* и *TEA* на основе их алгоритмической сложности, то *TEA* будет работать быстрее, так как он имеет более простой и быстрый алгоритм шифрования. *TEA* является шифром с фиксированным размером блока и длиной ключа, что делает его очень быстрым и эффективным для шифрования небольших объемов данных. С другой стороны, *DFC* имеет более сложную структуру и переменную длину ключа, что делает его более медленным, чем *TEA*. Это подтверждается результатами проведенного эксперимента.

Теперь следует замерить время зашифровки и расшифровки исходного сообщения разной длины, в результате чего можно будет выявить существование закономерности между размером блоков данных и временем процесса шифрования/дешифрования.

Для данного эксперимента было решено взять несколько файлов размерами 1, 10, 100 и 1000 КБ. Ключ использовался тот же, что и в предыдущем эксперименте. Было произведено десять опытов на зашифровку и дешифровку для двух методов шифрования по отдельности. Результаты изображены на рисунке 2.5.

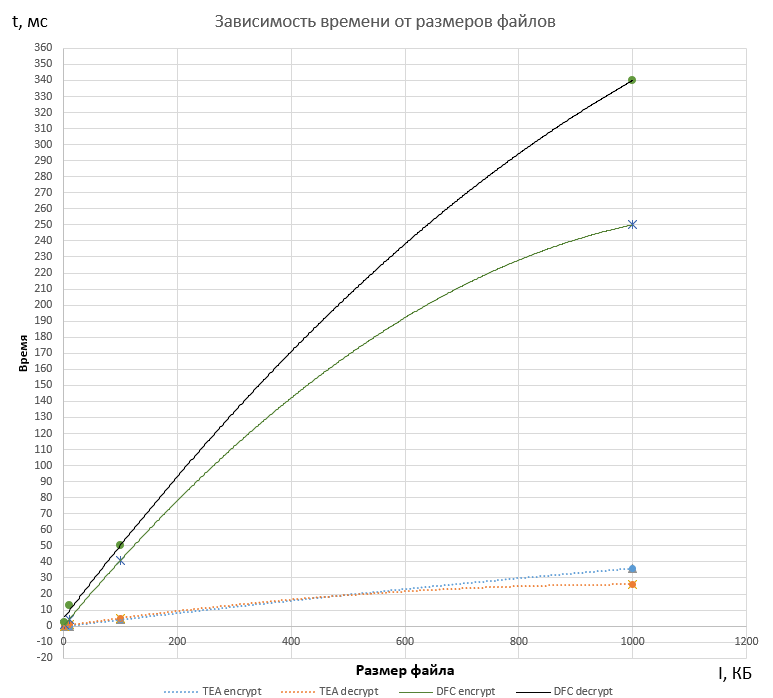


Рисунок 2.5 – Зависимость времени от размеров файлов

Результаты показали, что время шифрования и расшифрования блоков данных возрастает с увеличением размера данных у обоих методов шифрования (рис. 2.5). Это связано с тем, что блочные шифры работают с фиксированным размером блока данных.

Когда размер сообщения превышает размер блока, то необходимо разбить сообщение на блоки и шифровать каждый блок по отдельности. Таким образом, увеличение количества символов в сообщении приводит к увеличению количества блоков, которые необходимо зашифровать.

Каждый блок требует отдельного процесса шифрования и дополнительных операций. Это приводит к увеличению времени, необходимого для выполнения операции шифрования.

Таким образом, при увеличении размера сообщения, количество блоков, которые необходимо зашифровать, растет, а это приводит к увеличению времени зашифровки у блочных шифров.

Также при выборе блочного шифра для конкретного приложения важно учитывать, как он влияет на использование памяти. Если зашифрованный текст занимает значительно больше места, чем исходный текст, это может привести к неэффективному использованию памяти.

Блочные шифры работают с фиксированными размерами блоков данных, которые они шифруют. Если исходный текст не является кратным размеру блока (например, 13 байт), то его необходимо дополнить до размера блока, чтобы его можно было зашифровать. Для этого используются различные методы дополнения данных, такие как дополнение нулевыми байтами.

Когда исходный текст зашифрован, размер зашифрованного блока будет всегда фиксированным и равен размеру блока, даже если исходный текст был дополнен нулевыми байтами или другими данными. Поэтому зашифрованный текст может занимать больше места, чем исходный текст.

Таким образом, был проведен сравнительный анализ размеров файлов с исходным и зашифрованным сообщениями. Использовался тот же ключ и размер исходного файла, что и в предыдущем эксперименте.

Результаты показали, что размер файла с зашифрованным текстом с применением *DFC* остался таким же, когда в случае *TEA* размер файла увеличился на 1330 КБ. Количество символов в зашифрованном тексте показали более низкие значения, нежели количество символов исходного текста. Количество символов зашифрованного текста методом *TEA* равно 1044521 символ, а методом *DFC* – 1188040 символов (рис. 2.6).

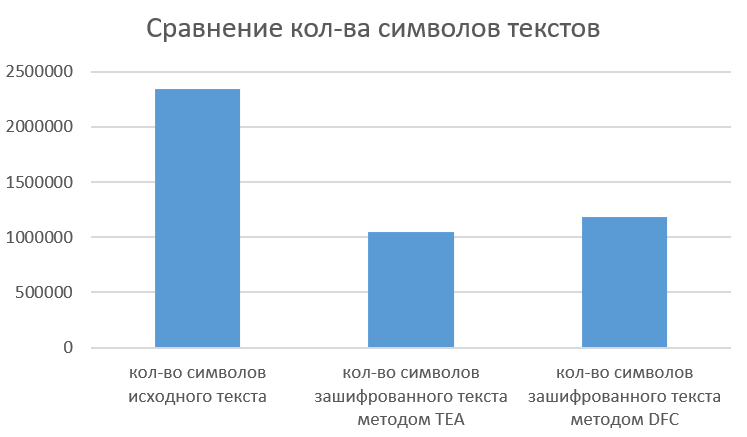


Рисунок 2.6 – Результаты размеров текстов

Таким образом можно сделать вывод, что размер зашифрованного файла зависит от используемого алгоритма шифрования. Оба алгоритма сжимают количество символов зашифрованного текста, что объясняется тем, что блочные шифры работают с блоками фиксированного размера и могут добавлять некоторые дополнительные символы для заполнения последнего блока.

На основе результатов эксперимента можно сказать, что производительность блочных шифров *TEA* и *DFC* зависит от размера данных, которые необходимо зашифровать или расшифровать. С увеличением размера данных время шифрования и расшифрования увеличивается у обоих методов. Однако, в данном эксперименте *TEA* показал лучшую производительность, чем *DFC*, поскольку время шифрования и расшифрования было меньше, а количество символов зашифрованного текста было больше у *DFC*, чем у *TEA*.

## **Вывод по разделу**

В данном разделе были описаны реализуемые методы шифрования: *DFC* и *TEA*. Были представлены алгоритмы генерации ключей, шифрования и дешифрования текста. Было проведено исследование методов, их анализ частотных закономерностей и производительность. В ходе исследования было выяснено, что *TEA* более производителен, нежели *DFC*.

# **Описание программного средства**

## **Разработка**

В ходе курсовой работы было разработано программное средство, основной задачей которого было шифрование и дешифрование текста с помощью двух блочных алгоритмов: *DFC* и *TEA*. Для визуализации алгоритмов шифрования и расшифрования реализовано на объектно-ориентированном языке программирования *С*# на платформе *Microsoft* .*NET* *Framework* версии 4.7 в среде разработки *Microsoft Visual Studio* 2022.

Для реализации программного средства использован *API* – интерфейс *WPF* для создания настольных графических программ, имеющих понятный и интерактивный интерфейс.

Структура приложения выглядит следующим образом (рис. 3.1).

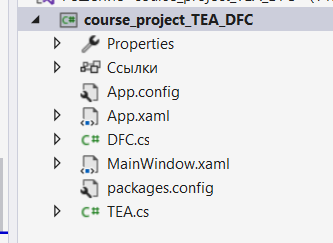


Рисунок 3.1 – Структура проекта

Проект включает в себя одно окно *MainWindow.xaml*, на котором располагаются все элементы управления (кнопки и текстовые поля). Его программную реализацию можно рассмотреть в приложении Г. В нем также было создано два класса *DFC* и *TEA* для реализации и удобного использования методов шифрования. Они содержат основные функции для шифрования, дешифрования текста, записи и чтения данных из текстовых файлов.

Главными методами в классе *TEA* являются: *Encrypt*() для зашифровки данных, в параметры которого передаются сообщение в виде массива байтов и ключ в том же виде, аналогичный *Decrypt*() для расшифровки данных и *CreateKey*(), реализованный для генерации раундовых ключей на основе одного исходного. Также имеются методы *BlockEncrypt*() и *BlockDecrypt*() для реализации шифрования и дешифрования блоков данных размером 64 бита с использованием раундовых ключей.

В классе *DFC* также существуют исходные методы *EncryptText*() и *DecryptText*(), в которых вызываются методы для обработки отдельных блоков по раундовым ключам: *EncryptBlock*() и *DecryptBlock*().Важную роль играет также метод *GenerationRoundKeys*(). Исходя из названия, можно понять, что он используется для генерации раундовых ключей из одного главного ключа, который должен быть введен пользователем в специально отведенное для этого поле.

Программная реализация всех функций классов *DFC* и *TEA* представлена в приложениях А и Б соответственно. Описание класса *MainWindow.xaml.cs* расписано в приложении В.

Программное средство позволяет выполнить следующий ряд задач:

– выбор одного из двух шифров;

– выбор текстового файла с данными, которые нужно зашифровать;

– вставка текста для зашифровки вручную;

– вставка своего ключа для зашифрования;

– зашифровка и расшифровка данных;

– сохранение шифрованного и дешифрованного текста в файлы.

Интерфейс программного средства представлен на рисунке 3.2.

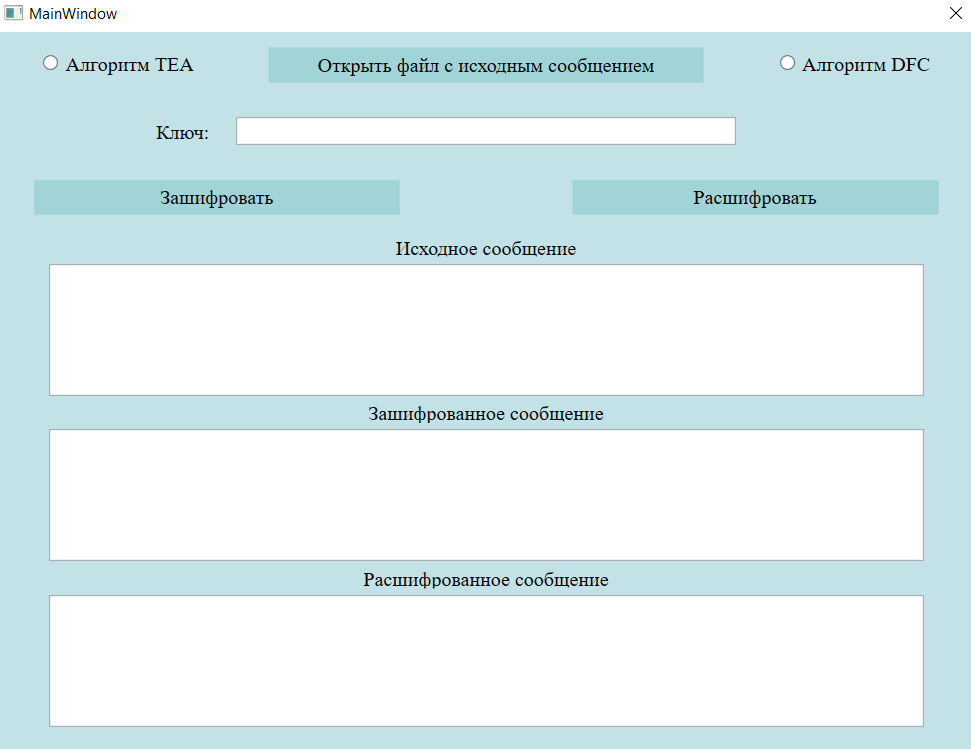


Рисунок 3.2 – Интерфейс программного средства

С помощью кнопки «Открыть файл с исходным сообщением» можно вставить текст из выбранного файла в соответствующее поле для дальнейшей обработки данных. При нажатии на нее все поля очищаются, кроме ключа.

Радиокнопки предназначены для выбора того или иного алгоритма. Выбор именно этого элемента управления позволил реализовать запрет на использование двух шифров одновременно.

Поле с надписью «Ключ» предназначено для ввода ключа одноразового использования. Ключи не запоминаются. Если длина введенной строки меньше требуемого, то она повторяется некоторое количество раз или обрезается для соответствия различным шифрам.

Кнопки «Зашифрование» и «Расшифрование» запускают главные методы в соответствии с их названиями. С помощью них шифруется или дешифруется текст из поля «Исходное сообщение» и в случае, если не было допущено никаких ошибок, остальные поля тоже заполняются.

## **Тестирование**

После создания программного средства с реализацией алгоритмов шифрования требуется его протестировать. Для корректной работы программы было необходимо обеспечить защиту работы пользователя от ошибок и сбоев. Для этого в программы были реализованы конструкции типа «*if*-*else*» и «*try*-*catch*». Они служат для «отлавливания» исключений и их обработкой. Это необходимо, чтобы при вводе пользователем некорректной информации, при вводе ключа неверного формата, изменения данных на непредполагаемые типы данных, программа не получала сбоев.

После запуска приложения можно проследить за потреблением ресурсов компьютера (рис. 3.3).



Рисунок 3.3 – Потребляемые ресурсы компьютера

Ниже приведены основные исключительные ситуации.

Перед началом работы приложения все кнопки и окна приложения находятся в неактивном состоянии, так как данные для шифрования ещё не введены (рис. 3.4).

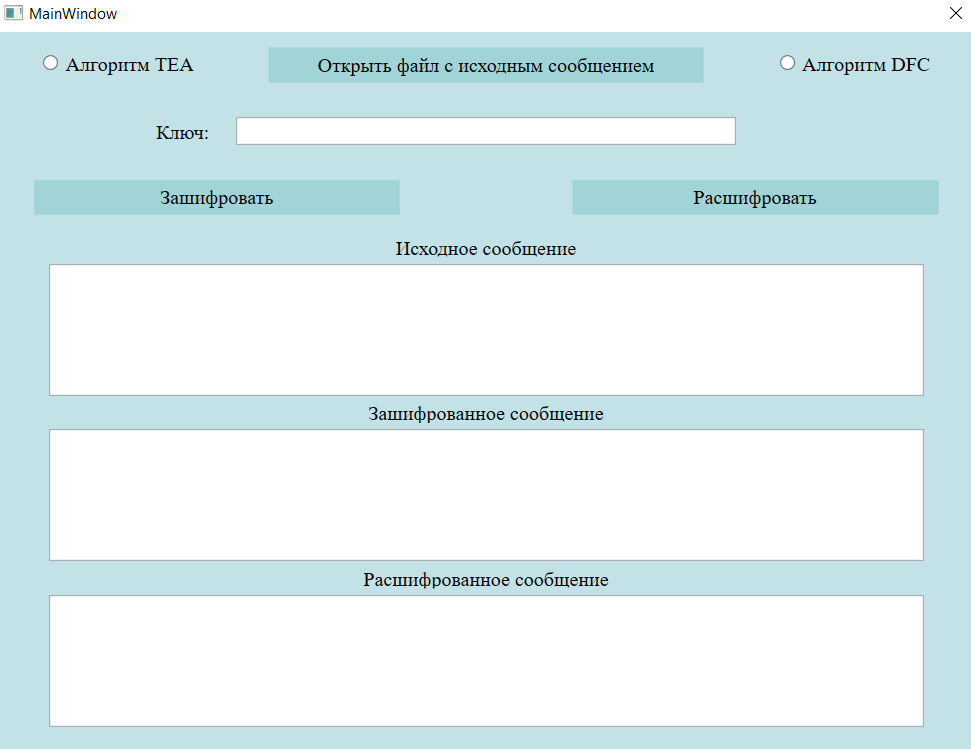


Рисунок 3.4 – Интерфейс программного средства

Для начала пользователю требуется выбрать шифр. Для предотвращения ошибок были использованы радиокнопки. Если сделать активной одну из них, все другие становятся неактивными автоматически. В случае, если пользователь не выберет шифр, после клика по кнопке «Зашифровать» выпадает ошибка, продемонстрированная на рисунке 3.5.

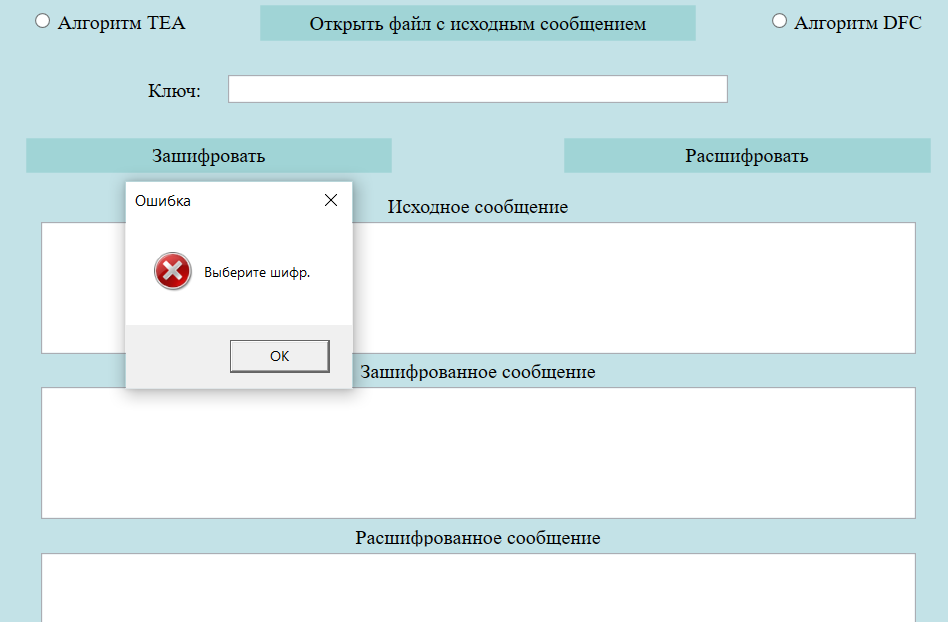


Рисунок 3.5 – Ошибка «Выберите шрифт»

Для шифрования необходимо ввести ключ длиной 16 символов. Для этого необходимо кликнуть по полю «Ключ» и оно станет активным. Если ввести значение длиной больше или меньше шестнадцати символов, то программа сама подгоняет ключ под требующиеся размеры и не выдает никаких ошибок (рис. 3.6).

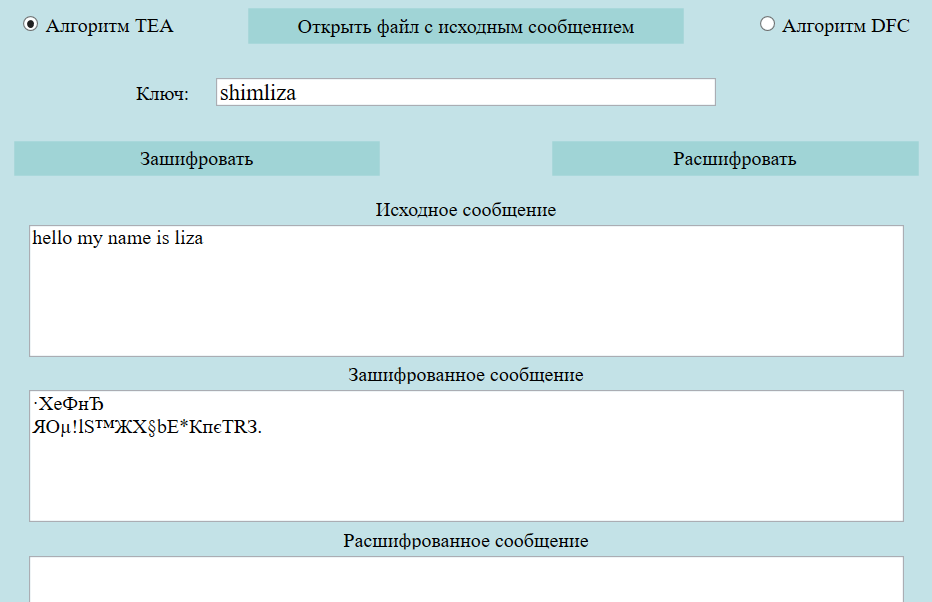


Рисунок 3.6 – Шифрование с ключом длиной 8 символов

Кнопка «Открыть файл с исходным сообщением» позволяет найти в файловой системе требуемый текстовый файл и сообщает об ошибке, когда выбранный файл оказывается пустым (рис. 3.7).

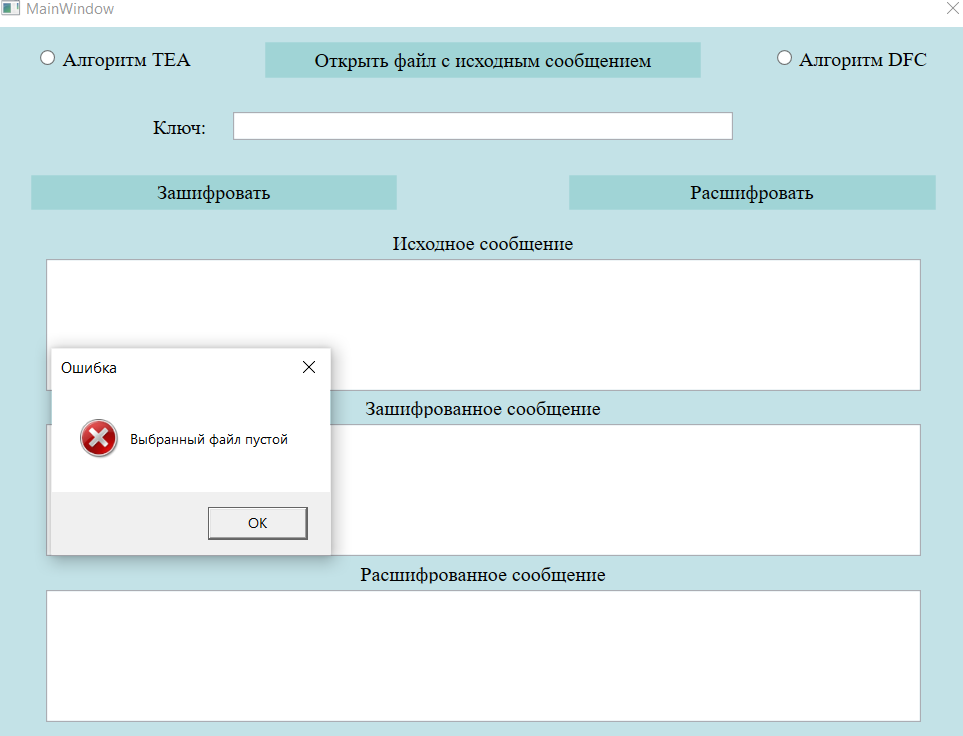


Рисунок 3.7 – Ошибка «Выбранный файл пустой»

Кнопка зашифровки не будет работать, пока поле «Зашифрованное сообщение» не будет заполнено (рис. 3.8). Также и для кнопки «Расшифровать».

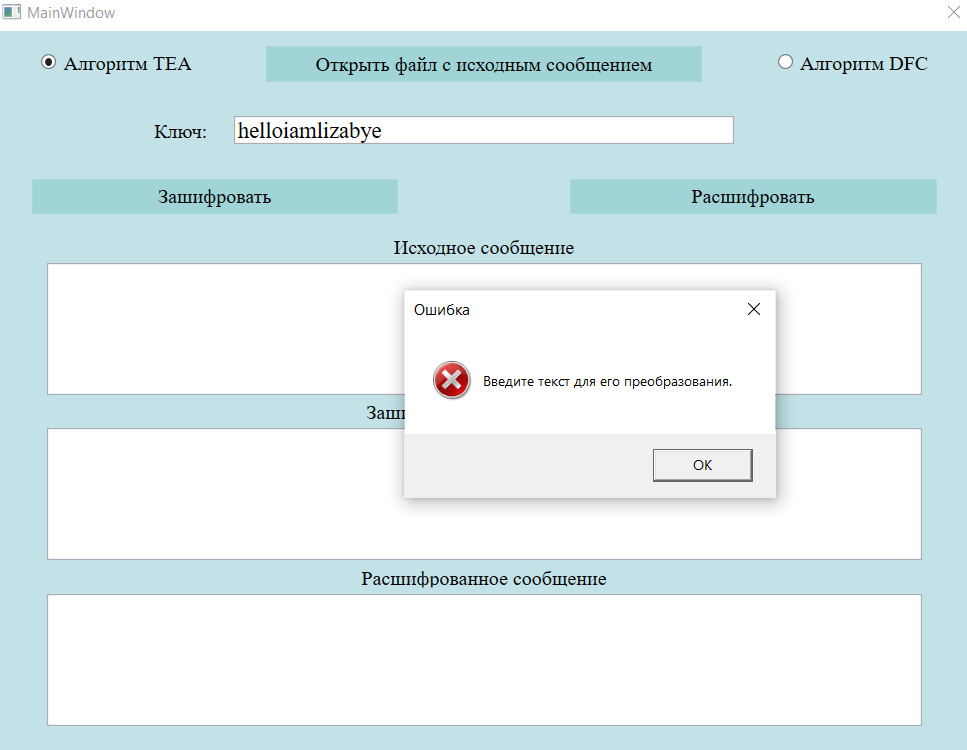


Рисунок 3.8 – Ошибка «Введите текст для его преобразования»

Если шрифт выбран, ключ введен и файл с текстом для зашифровки выбран, то кнопки «Зашифровка» и «Расшифровка» работают и программа может обрабатывать текст (рис. 3.9).

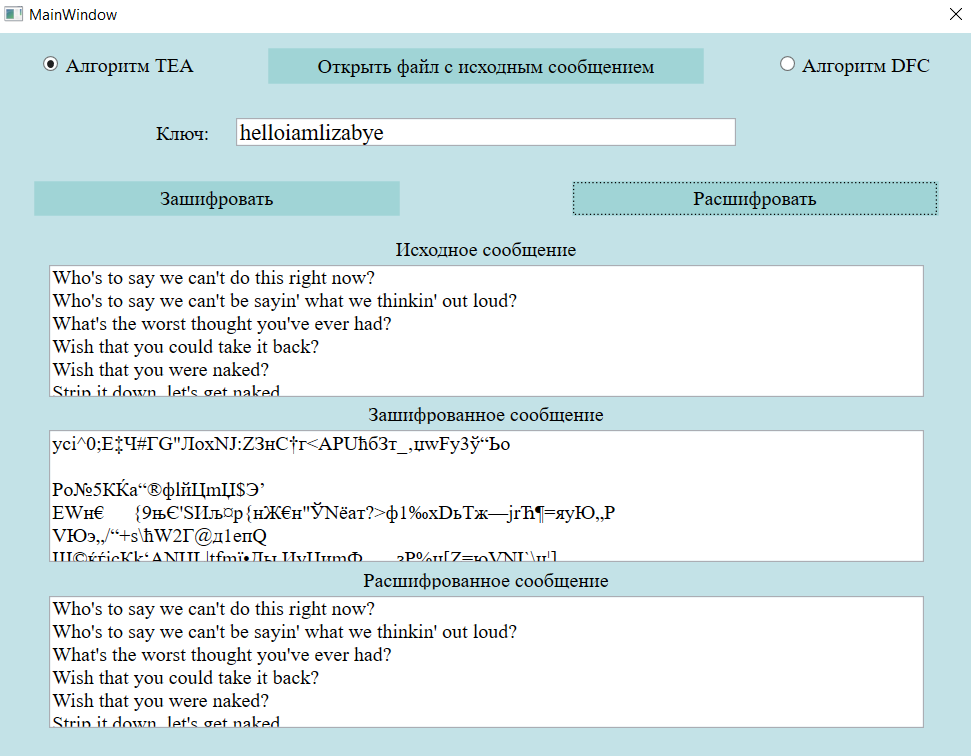


Рисунок 3.9 – Безошибочное шифрование и дешифрование

Следует отметить, что приложение разработано таким образом, чтобы минимизировать возможность появления пользовательских ошибок.

## **Руководство пользователя**

Чтобы запустить приложение нужно запустить *exe*-файл – *course\_project\_TEA\_DFC*.*exe*. При запуске открывается окно приложения. Окно содержит окна для ввода и вывода данных и кнопки для совершения операций шифрования и расшифрования.

После запуска приложения необходимо выбрать метод для шифрования: *DFC* или *TEA*. Это делается при помощи радиокнопок в верхней части приложения (рис. 3.10).



Рисунок 3.10 – Радиокнопки для выбора шифра

На рисунке 3.10 также изображена кнопка «Открыть файл с исходным текстом». При клике на нее откроется проводник, где можно будет выбрать текстовый файл с входным сообщением для зашифровки (рис. 3.11).

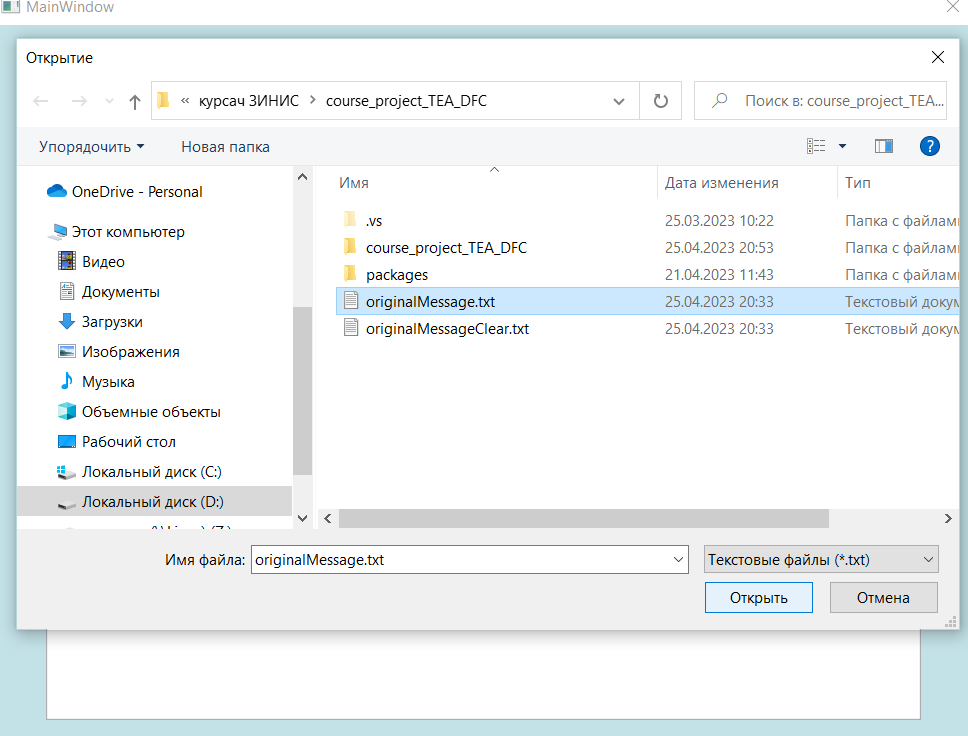


Рисунок 3.11 – Проводник

После того, как в диалоговом окне будет нажата кнопка «Открыть», текст из файла будет помещен в поле для исходного сообщения (рис. 3.12).

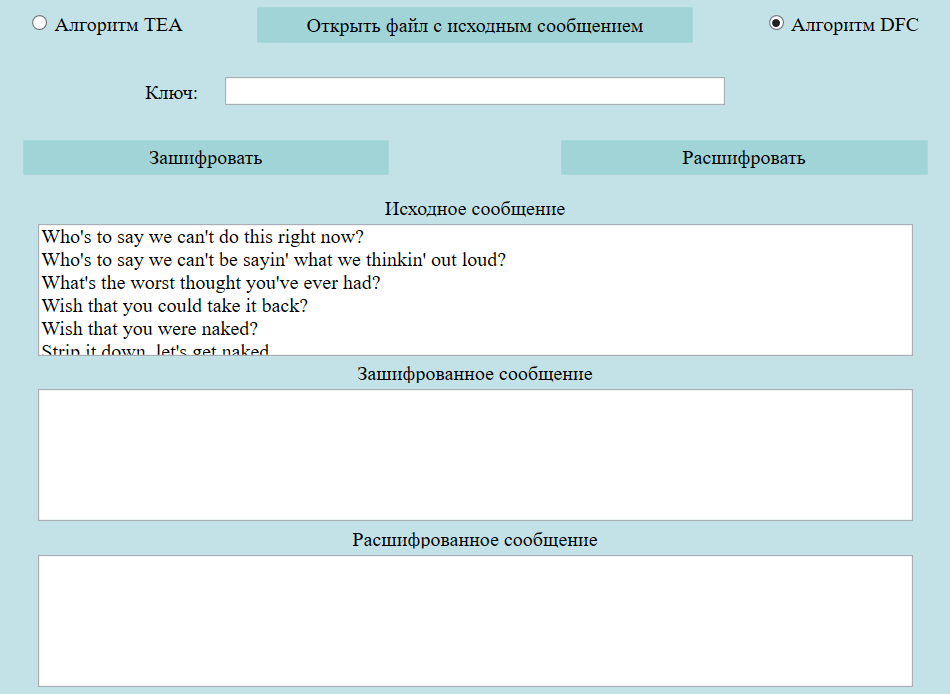


Рисунок 3.12 – Поле «Исходное сообщение» с текстом из файла

Для зашифрования в поле для ключа требуется ввести значение (рис. 3.13).



Рисунок 3.13 – Пример значения ключа для ввода

После этого можно шифровать с помощью кнопки «Зашифровать». Результат будет отображен в поле «Зашифрованное сообщение» и в файле «*EncryptMessage*» в папке с *exe*-файлом (рис. 3.14).

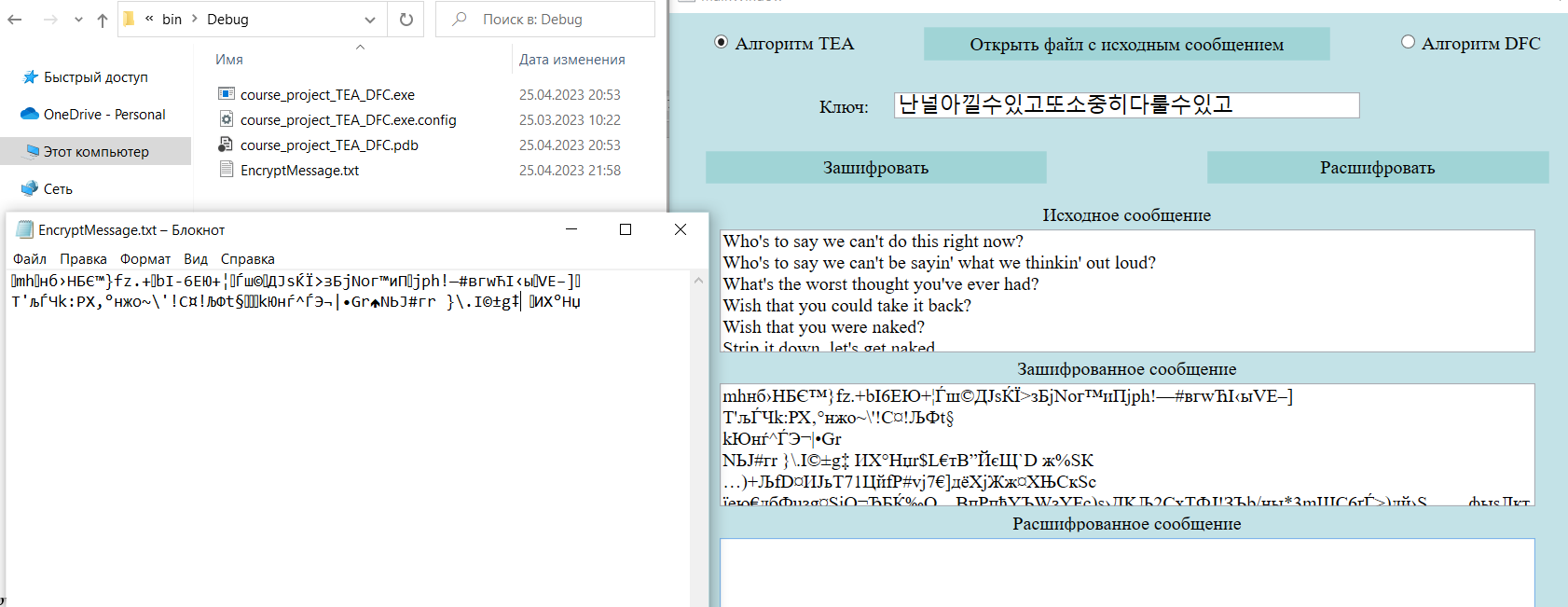


Рисунок 3.14 – Зашифрованное сообщение и файл «*EncryptMessage*»

После того как получили зашифрованное сообщение, расшифруем его, нажатием на кнопку «Расшифровать». Расшифрованное сообщение и файл с ним представлены на рисунке 3.15.

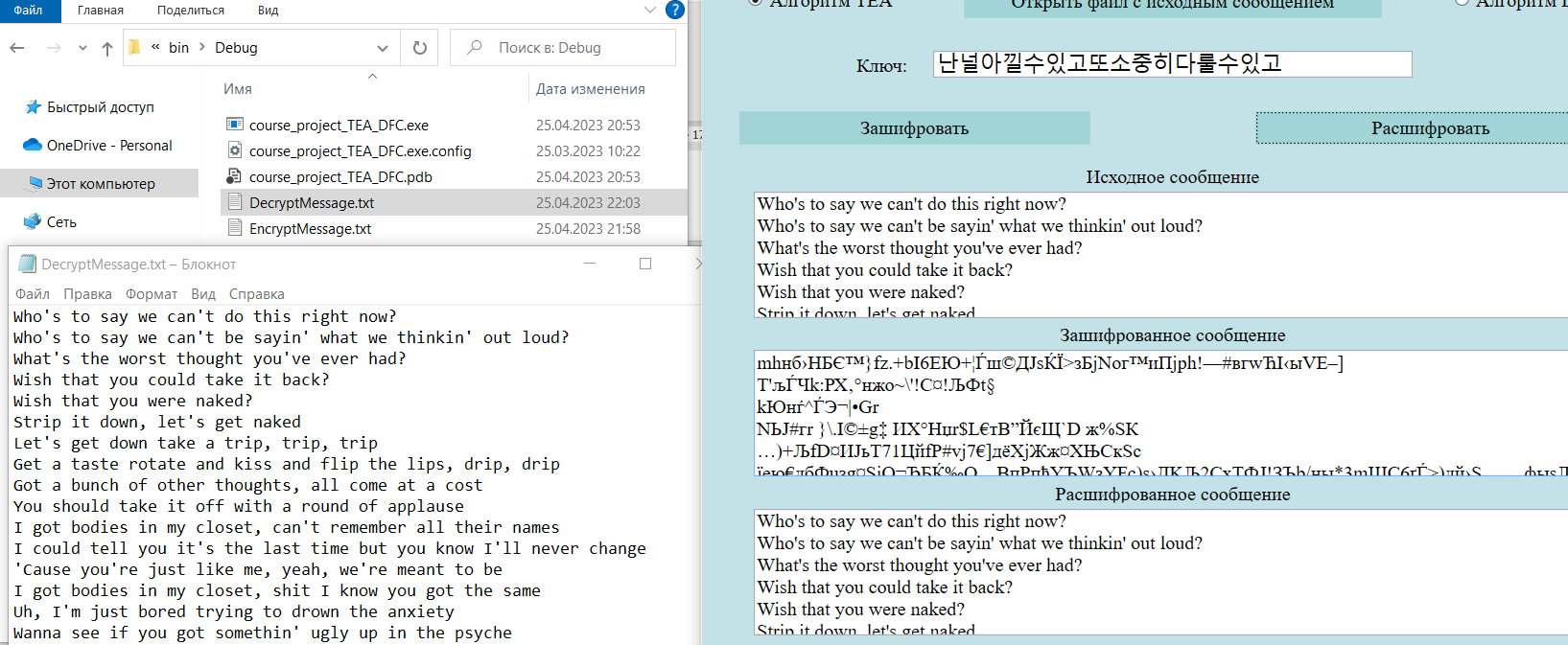


Рисунок 3.15 – Расшифрованное сообщение и файл «*DecryptMessage*»

Данное руководство пользователя поможет новым пользователям быстрее овладеть программным обеспечением, предоставляя необходимую информацию о его функциях, таких как выбор исходного файла, шифрование и дешифрование. Это руководство пользователя также поможет более эффективно использовать программное обеспечение, позволяя им получить более полное понимание его функций и возможностей. Это поможет пользователям сэкономить время и уменьшить количество ошибок при работе с программным обеспечением.

## **Вывод по разделу**

В данном разделе представлена информация о программном средстве, которое разработано для демонстрации работы нескольких блочных шифров визуально. Описаны технологии, примененные при разработке, а также процесс разработки и тестирования. Проведенное тестирование подтвердило эффективность и корректность работы программного средства. Также было составлено руководство пользователя, в котором описаны все необходимые шаги для работы с программным средством. Благодаря этому руководству пользователи могут использовать программное средство более просто и удобно.

# **Заключение**

Этот проект был разработан для демонстрации работы блочных алгоритмов *TEA* и *DFC*. Он представляет собой функциональное приложение с простым и удобным интерфейсом. Для этого был использован *API*-интерфейс *WPF* и среда разработки *Visual Studio* 2022.

В процессе разработки были изучены различные модификации алгоритма и проанализированы аналоги программного обеспечения для шифрования данных. Также было уделено особое внимание обработке ошибок и написанию руководства пользователя для более удобного использования программы.

Выбранный алгоритм *TEA* обладает рядом преимуществ, таких как простота реализации, небольшой размер кода и высокая скорость выполнения, что делает его оптимальным для использования на стандартных 32-битных процессорах. Однако, алгоритм уязвим к атакам на связанных ключах, что может уменьшить его эффективность и безопасность.

Второй выбранный алгоритм *DFC* имеет высокую скорость шифрования на 64-битных платформах, требует очень мало памяти и способен шифровать и дешифровать с использованием ключей различной длины до 256 бит. Алгоритм сложный, уязвим к атакам и имеет несколько классов слабых ключей. Он не стандартизирован и не используется в какой-либо сфере на практике.

После разработки приложения было также произведено короткое исследование шифров и даже их сравнение. В результате чего был сделан вывод, что *TEA* имеет лучшую производительность.

# **Список используемых источников**

1. Брюс Шнайер. Прикладная криптография / Брюс Шнайер. – М.: 2016. – 610с.
2. Сергей Понасенко. Алгоритмы шифрования. Специальный справочник – БХВ-Петербург, 2009. – 195с.
3. Википедия [Электронный ресурс]. – Режим доступа: https://www.wiki3.ru-ru.nina.az/DFC.html – Дата доступа: 09.04.2023.
4. EPFL. Statistical Cryptanalysis of Block Ciphers / Emre Telatar, Serge Vaudenay and others. – Lausanne, 2005. – 170c.
5. Введение в криптографию // Под общ. ред. В.В. Ященко. – 3-е изд., доп. – М.: 2000. – 288с.
6. Файловый архив студентов [Ressource électronique]. – Mode d'accès: https://studfile.net/preview/2262041/page:5/ – Date d'accès: 11.03.2023.
7. DFC - un Algorithme de Chiffrement [Electronic resource]. – France, 1999. – Access mode: https://lasec.epfl.ch/people/vaudenay/xtc/pi\_dfc2.html – Access date: 18.04.2023.
8. Энциклопедия [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/TEA – Дата доступа: 15.03.2023.

# **Приложение А**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Numerics;

using System.Windows;

namespace course\_project\_TEA\_DFC

{

internal class DFC

{

/// <summary>

/// Таблица с 32-битовыми константами

/// </summary>

private static UInt32[] E ={

0xb7e15162, 0x8aed2a6a, 0xbf715880, 0x9cf4f3c7,

0x62e7160f, 0x38b4da56, 0xa784d904, 0x5190cfef,

0x324e7738, 0x926cfbe5, 0xf4bf8d8d, 0x8c31d763,

0xda06c80a, 0xbb1185eb, 0x4f7c7b57, 0x57f59584,

0x90cfd47d, 0x7c19bb42, 0x158d9554, 0xf7b46bce,

0xd55c4d79, 0xfd5f24d6, 0x613c31c3, 0x839a2ddf,

0x8a9a276b, 0xcfbfa1c8, 0x77c56284, 0xdab79cd4,

0xc2b3293d, 0x20e9e5ea, 0xf02ac60a, 0xcc93ed87,

0x4422a52e, 0xcb238fee, 0xe5ab6add, 0x835fd1a0,

0x753d0a8f, 0x78e537d2, 0xb95bb79d, 0x8dcaec64,

0x2c1e9f23, 0xb829b5c2, 0x780bf387, 0x37df8bb3,

0x00d01334, 0xa0d0bd86, 0x45cbfa73, 0xa6160ffe,

0x393c48cb, 0xbbca060f, 0x0ff8ec6d, 0x31beb5cc,

0xeed7f2f0, 0xbb088017, 0x163bc60d, 0xf45a0ecb,

0x1bcd289b, 0x06cbbfea, 0x21ad08e1, 0x847f3f73,

0x78d56ced, 0x94640d6e, 0xf0d3d37b, 0xe67008e1,

0x86d1bf27, 0x5b9b241d, 0xeb64749a, 0x47dfdfb9,

0x6632c3eb, 0x061b6472, 0xbbf84c26, 0x144e49c2 };

/// <summary>

/// 32-битовые константы E[0...63]

/// </summary>

private static UInt32[] RT = {

0xb7e15162, 0x8aed2a6a, 0xbf715880, 0x9cf4f3c7,

0x62e7160f, 0x38b4da56, 0xa784d904, 0x5190cfef,

0x324e7738, 0x926cfbe5, 0xf4bf8d8d, 0x8c31d763,

0xda06c80a, 0xbb1185eb, 0x4f7c7b57, 0x57f59584,

0x90cfd47d, 0x7c19bb42, 0x158d9554, 0xf7b46bce,

0xd55c4d79, 0xfd5f24d6, 0x613c31c3, 0x839a2ddf,

0x8a9a276b, 0xcfbfa1c8, 0x77c56284, 0xdab79cd4,

0xc2b3293d, 0x20e9e5ea, 0xf02ac60a, 0xcc93ed87,

0x4422a52e, 0xcb238fee, 0xe5ab6add, 0x835fd1a0,

0x753d0a8f, 0x78e537d2, 0xb95bb79d, 0x8dcaec64,

0x2c1e9f23, 0xb829b5c2, 0x780bf387, 0x37df8bb3,

0x00d01334, 0xa0d0bd86, 0x45cbfa73, 0xa6160ffe,

0x393c48cb, 0xbbca060f, 0x0ff8ec6d, 0x31beb5cc,

0xeed7f2f0, 0xbb088017, 0x163bc60d, 0xf45a0ecb,

0x1bcd289b, 0x06cbbfea, 0x21ad08e1, 0x847f3f73,

0x78d56ced, 0x94640d6e, 0xf0d3d37b, 0xe67008e1};

/// <summary>

/// 64-битовая константа E[64]||E[65]

/// </summary>

private static UInt64 KD = 0x86d1bf275b9b241d;

/// <summary>

/// 32-битовая константа E[66]

/// </summary>

private static UInt32 KC = 0xeb64749a;

/// <summary>

/// 256-битовая константа

/// </summary>

private static UInt64[] KS = {

0x86d1bf275b9b241d, 0xeb64749a47dfdfb9,

0x6632c3eb061b6472, 0xbbf84c26144e49c2};

/// <summary>

/// (будет вычисляться при вызове конструктора)

/// </summary>

private static List<UInt64[]> KAB;

/// <summary>

/// Промежуточный результат шифрования (128 бит)

/// </summary>

private byte[] state = new byte[16];

/// <summary>

/// Непосредственно ключ

/// </summary>

private static UInt64[] key = new UInt64[2];

/// <summary>

/// Шифруемый текст

/// </summary>

public byte[] text;

/// <summary>

/// Раундовые подключи

/// </summary>

public List<UInt64[]> RK;

/// <summary>

/// Конструктор

/// </summary>

public DFC(byte[] k)

{

KAB = new List<ulong[]>();

for (int i = 0; i <= 15; i++)

{

UInt64 kab0 = E[4 \* i], kab1 = E[4 \* i + 1], kab2 = E[4 \* i + 2], kab3 = E[4 \* i + 3];

kab0 = kab0 << 32; kab2 = kab2 << 32;

kab0 = kab0 ^ kab1; kab2 = kab2 ^ kab3;

UInt64[] arr = { kab0, kab2 };

KAB.Add(arr);

}

//Переводим байтовое представление ключа в ulong(для удобства работы)

key = KeyCheckAndRebuild(k);

RK = GenerationRoundKeys(key);

}

public DFC()

{

}

/// <summary>

/// Подготовка исходного текста к шифрованию

/// </summary>

/// <param name="sourceText">Исходный текст</param>

/// <returns>Готовый к шифрованию текст</returns>

public static byte[] TextPrepare(byte[] sourceText)

{

//Узнаём сколько нужно дописать в конец блока

int lenghtZero = sourceText.Length % 16;

byte[] buf = sourceText;

if (lenghtZero > 0)

{

//Копируем все исходные байты в расширенный массив

Array.Resize(ref buf, sourceText.Length + (16 - lenghtZero));

buf[sourceText.Length] = 0x80;

for (int i = 1; i < lenghtZero; i++)

{

//остаток дополняем нулями

buf[sourceText.Length + i] = 0x00;

}

}

else//если длина текста кратна 16, дописываем целый блок

{

Array.Resize(ref buf, sourceText.Length + 16);

buf[sourceText.Length] = 0x80;

for (int i = 1; i < 16; i++)

{

buf[sourceText.Length + i] = 0x00;

}

}

return buf;

}

/// <summary>

/// Проверка и укарачивание ключа

/// </summary>

/// <param name="k">Ключ</param>

/// <returns>Обрезанный либо null, если произошла ошибка или ключ слишком короткий</returns>

public static UInt64[] KeyCheckAndRebuild(byte[] k)

{

try

{

if (k.Length < 16)

{

System.Windows.MessageBox.Show("Длина ключа меньше минимальной!\nПопробуйте ввести другой ключ.", "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

return null;

}

else

{

UInt64[] keybuf = new UInt64[2];

keybuf[0] = keybuf[1] = 0;

for (int i = 0; i < 8; i++)

{

keybuf[0] <<= 8; keybuf[0] ^= k[i];

keybuf[1] <<= 8; keybuf[1] ^= k[i + 8];

}

return keybuf;

}

}

catch (Exception ex)

{

System.Windows.MessageBox.Show(ex.Message, "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

return null;

}

}

/// <summary>

/// Раундовая функция

/// </summary>

/// <param name="X"></param>

/// <param name="K"></param>

/// <returns></returns>

private static UInt64 RF(UInt64 X, UInt64[] K)

{

BigInteger PowerTwo64 = new BigInteger(Math.Pow(2, 64));

//UInt64 PowerTwo64 = (UInt64)Math.Pow(2, 64);

UInt64 buf = (UInt64)((K[0] \* X + K[1]) % (PowerTwo64 + 13));

buf = (UInt64)(buf % PowerTwo64);

CP(buf);

return buf;

}

/// <summary>

///

/// </summary>

/// <param name="y"></param>

/// <returns></returns>

private static UInt64 CP(UInt64 y)

{

UInt64 buf = 0;

//UInt64 PowerTwo64 = (UInt64)Math.Pow(2, 64);

BigInteger PowerTwo64 = new BigInteger(Math.Pow(2, 64));

UInt32 yL = (UInt32)(y >> 32), yR = (UInt32)(y);

yR = yR ^ RT[trunc(yR)]; yL = yL ^ KC;

buf = (UInt64)((((UInt64)yR ^ yL) + KD) % PowerTwo64);

return buf;

}

/// <summary>

/// Числовое значение которое определяется 6 левыми битами(старшими) блока данных

/// </summary>

/// <param name="yL">32-битовый блок</param>

/// <returns>Индекс</returns>

private static UInt32 trunc(UInt32 yL)

{

return yL >> 26;

}

/// <summary>

/// Генерация раундовых ключей

/// </summary>

/// <param name="k">Ключ(главный)</param>

/// <returns>Список раундовых ключей</returns>

private static List<UInt64[]> GenerationRoundKeys(UInt64[] k)

{

UInt64[] arr = new UInt64[2], IRK0 = new UInt64[2], RK0 = new UInt64[2];

List<UInt64[]> IRK = new List<UInt64[]>();

for (int i = 0; i < 33; i++)

{

arr[0] = arr[1] = 0;

IRK.Add(arr);

}

UInt64[] PK = { key[0], key[1], KS[0], KS[1] };

IRK0[0] = PK[0]; IRK0[1] = PK[1]; RK0[0] = PK[2]; RK0[1] = PK[3];

IRK[0] = IRK0;

int s = 0;

for (int i = 1; i <= 15; i++)

{

s = (int)(RT[i] % 16);

IRK[i][0] = IRK[i - 1][0] ^ KAB[s][0];

IRK[i][1] = IRK[i - 1][1] ^ KAB[s][1];

}

List<UInt64[]> RoundKey = new List<ulong[]>();

RoundKey.Add(RK0);

for (int i = 1; i <= 8; i++)

{

UInt64 X = RoundKey[i - 1][0], Y = RoundKey[i - 1][1];

for (int j = 1; j <= 4; j++)

{

Y = Y ^ RF(X, IRK[4 \* (i - 1) + 1]);

UInt64 buf = X;

X = Y;

Y = buf;

}

arr[0] = Y; arr[1] = X;

RoundKey.Add(arr);

}

return RoundKey;

}

/// <summary>

/// Шифрование блока по раундовым подключам

/// </summary>

/// <param name="block">Шифруемый блок</param>

/// <param name="round">Количество раундов(рекомендовано 8)</param>

/// <param name="roundkeys">Раундовые ключи</param>

/// <returns>Зашифрованный блок</returns>

private UInt64[] EncryptBlock(UInt64[] block, List<UInt64[]> roundkeys, int round)

{

UInt64 X = block[0];//левая часть блока

UInt64 Y = block[1];//правая часть блока

UInt64 tmp = 0;

for (int i = 1; i <= round; i++)

{

Y = Y ^ RF(X, RK[i]);

//меняем местами X Y

tmp = Y;

Y = X;

X = tmp;

}

UInt64[] CT = new UInt64[2];

CT[0] = Y; CT[1] = X;

return CT;

}

/// <summary>

/// Расшифрование блока по раундовым подключам

/// </summary>

/// <param name="block">Блок данных</param>

/// <param name="roundkeys">Раундовые ключи</param>

/// <param name="round">Количество раундов</param>

/// <returns>Расшифрованный блок</returns>

private UInt64[] DecryptBlock(UInt64[] block, List<UInt64[]> roundkeys, int round)

{

UInt64 X = block[0];//левая чатсть блока

UInt64 Y = block[1];//правая часть блока

UInt64 tmp = 0;

for (int i = round; i >= 1; i--)

{

Y = Y ^ RF(X, RK[i]);

//меняем местами X Y

tmp = Y;

Y = X;

X = tmp;

}

UInt64[] CT = new UInt64[2];

CT[0] = Y; CT[1] = X;

return CT;

}

/// <summary>

/// Зашифрование текста

/// </summary>

/// <param name="txt">Шифруемый текст</param>

/// <returns>Зашифрованный текст</returns>

public byte[] EncryptText(byte[] txt)

{

byte[] buf = txt;

if (txt == null || txt.Length == 0)

{

System.Windows.MessageBox.Show("Текст шифруемого сообщения отсутствует", "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

return buf;

}

try

{

//Проверяем, нужно ли дописывать 0x00 в конец,

//чтобы дополнить блок

int lenghtZero = txt.Length % 16;

if (lenghtZero > 0)

{

Array.Resize(ref buf, txt.Length + (16 - lenghtZero));

buf[txt.Length] = 0x80;

for (int i = 1; i < 16 - lenghtZero; i++)

{

buf[txt.Length + i] = 0x00;

}

}

else//если длина текста кратна 16 дописываем целый блок

{

Array.Resize(ref buf, txt.Length + 16);

buf[txt.Length] = 0x80;

for (int i = 1; i < 16; i++)

{

buf[txt.Length + i] = 0x00;

}

}

UInt64[] block = new UInt64[2];

for (int i = 0; i < buf.Length; i = i + 16)

{

block[0] = block[1] = 0;

for (int j = 0; j < 8; j++)

{

block[0] <<= 8; block[0] ^= buf[i + j];

block[1] <<= 8; block[1] ^= buf[i + j + 8];

}

block = EncryptBlock(block, RK, 8);

buf[i + 7] = (byte)block[0]; buf[i + 15] = (byte)block[1];

for (int j = 6; j >= 0; j--)

{

block[0] >>= 8; block[1] >>= 8;

buf[i + j] = (byte)block[0];

buf[i + j + 8] = (byte)block[1];

}

}

}

catch (Exception ex)

{

System.Windows.MessageBox.Show(ex.Message, "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

}

return buf;

}

/// <summary>

/// Расшифрование текста

/// </summary>

/// <param name="txt">Зашифрованный текст</param>

/// <returns>Расшифрованный текст</returns>

public byte[] DecryptText(byte[] txt)

{

byte[] buf = new byte[txt.Length];

Array.Copy(txt, buf, txt.Length);

if (txt == null || txt.Length == 0)

{

System.Windows.MessageBox.Show("Текст дешефруемого сообщения отсутствует", "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

return buf;

}

try

{

UInt64[] block = new UInt64[2];

for (int i = 0; i < buf.Length; i = i + 16)

{

block[0] = block[1] = 0;

for (int j = 0; j < 8; j++)

{

block[0] <<= 8; block[0] ^= buf[i + j];

block[1] <<= 8; block[1] ^= buf[i + j + 8];

}

block = DecryptBlock(block, RK, 8);

buf[i + 7] = (byte)block[0]; buf[i + 15] = (byte)block[1];

for (int j = 6; j >= 0; j--)

{

block[0] >>= 8; block[1] >>= 8;

buf[i + j] = (byte)block[0];

buf[i + j + 8] = (byte)block[1];

}

}

//"обламываем" лишний конец сообщения

int k = buf.Length;

while (buf[k - 1] != 0x80)

{

k--;

}

Array.Resize(ref buf, k - 1);

}

catch (Exception ex)

{

System.Windows.MessageBox.Show(ex.Message, "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

}

return buf;

}

/// <summary>

/// Бинарное чтение из файла

/// </summary>

/// <param name="fileName">Имя файла</param>

/// <returns></returns>

public byte[] ReadByteArrayFromFile(string fileName)

{

byte[] buf = null;

try

{

FileStream file = new FileStream(fileName, FileMode.Open, FileAccess.Read);

BinaryReader binary = new BinaryReader(file);

long numBytes = new FileInfo(fileName).Length;

buf = binary.ReadBytes((int)numBytes);

binary.Close();

file.Close();

}

catch (Exception ex)

{

System.Windows.MessageBox.Show(ex.Message, "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

}

return buf;

}

public byte[] ReadByteArrayFromString(string text)

{

byte[] buf = null;

try

{

MemoryStream stream = new MemoryStream(Encoding.ASCII.GetBytes(text));

BinaryReader binary = new BinaryReader(stream);

long numBytes = text.Length;

buf = binary.ReadBytes((int)numBytes);

binary.Close();

stream.Close();

}

catch (Exception ex)

{

System.Windows.MessageBox.Show(ex.Message, "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

}

return buf;

}

/// <summary>

/// Бинарная запись в файл

/// </summary>

/// <param name="buf">Массив, который необходимо записать</param>

/// <param name="fileName">Имя файла</param>

public void WriteByteArrayToFile(byte[] buf, string fileName)

{

try

{

FileStream file = new FileStream(fileName, FileMode.Create, FileAccess.ReadWrite);

BinaryWriter binary = new BinaryWriter(file);

for (int i = 0; i < buf.Length; i++)

{

binary.Write(buf[i]);

}

binary.Close();

file.Close();

}

catch (Exception ex)

{

System.Windows.MessageBox.Show(ex.Message, "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

}

}

}

}

# **Приложение Б**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace course\_project\_TEA\_DFC

{

internal class TEA

{

public static string Encrypt(string data, string key) => Encoding.Default.GetString(

Encrypt(Encoding.Default.GetBytes(data), Encoding.Default.GetBytes(key)));

public static string Decrypt(string data, string key) => Encoding.Default.GetString(Decrypt(

Encoding.Default.GetBytes(data), Encoding.Default.GetBytes(key)));

public static byte[] Encrypt(byte[] data, byte[] key)

{

uint[] key1 = CreateKey(key);

uint[] v = new uint[2];

byte[] buffer = new byte[NextMultipleOf8(data.Length + 4)];

byte[] bytes = BitConverter.GetBytes(data.Length);

Array.Copy((Array)bytes, (Array)buffer, bytes.Length);

Array.Copy((Array)data, 0, (Array)buffer, bytes.Length, data.Length);

using (MemoryStream memoryStream = new MemoryStream(buffer))

{

using (BinaryWriter binaryWriter = new BinaryWriter((Stream)memoryStream))

{

for (int startIndex = 0; startIndex < buffer.Length; startIndex += 8)

{

v[0] = BitConverter.ToUInt32(buffer, startIndex);

v[1] = BitConverter.ToUInt32(buffer, startIndex + 4);

BlockEncrypt(v, key1);

binaryWriter.Write(v[0]);

binaryWriter.Write(v[1]);

}

}

}

return buffer;

}

public static byte[] Decrypt(byte[] numArray1, byte[] key)

{

if ((uint)(numArray1.Length % 8) > 0U)

throw new ArgumentException("Длина зашифрованных данных должна быть кратна 8 байтам");

uint[] key1 = CreateKey(key);

uint[] v = new uint[2];

byte[] buffer = new byte[numArray1.Length];

Array.Copy((Array)numArray1, (Array)buffer, numArray1.Length);

using (MemoryStream memoryStream = new MemoryStream(buffer))

{

using (BinaryWriter binaryWriter = new BinaryWriter((Stream)memoryStream))

{

for (int startIndex = 0; startIndex < buffer.Length; startIndex += 8)

{

v[0] = BitConverter.ToUInt32(buffer, startIndex);

v[1] = BitConverter.ToUInt32(buffer, startIndex + 4);

BlockDecrypt(v, key1);

binaryWriter.Write(v[0]);

binaryWriter.Write(v[1]);

}

}

}

uint uint32 = BitConverter.ToUInt32(buffer, 0);

if ((long)uint32 > (long)(buffer.Length - 4))

throw new ArgumentException("Зашифрованные данные разрушены");

byte[] numArray2 = new byte[(int)uint32];

Array.Copy((Array)buffer, 4L, (Array)numArray2, 0L, (long)uint32);

return numArray2;

}

public static uint[] CreateKey(byte[] key)

{

byte[] numArray1 = new byte[16];

byte[] numArray2 = key;

return new uint[4]

{

BitConverter.ToUInt32(numArray2, 0),

BitConverter.ToUInt32(numArray2, 4),

BitConverter.ToUInt32(numArray2, 8),

BitConverter.ToUInt32(numArray2, 12)

};

}

public static void BlockEncrypt(uint[] v, uint[] key)

{

uint num1 = v[0];

uint num2 = v[1];

uint num3 = 0;

uint num4 = key[0];

uint num5 = key[1];

uint num6 = key[2];

uint num7 = key[3];

for (uint index = 0; index < 32U; ++index)

{

num3 += 2654435769U;

num1 += (uint)(((int)num2 << 4) + (int)num4 ^ (int)num2 + (int)num3 ^ (int)(num2 >> 5) + (int)num5);

num2 += (uint)(((int)num1 << 4) + (int)num6 ^ (int)num1 + (int)num3 ^ (int)(num1 >> 5) + (int)num7);

}

v[0] = num1;

v[1] = num2;

}

private static void BlockDecrypt(uint[] v, uint[] key)

{

uint num1 = v[0];

uint num2 = v[1];

uint num3 = 3337565984;

uint num4 = key[0];

uint num5 = key[1];

uint num6 = key[2];

uint num7 = key[3];

for (uint index = 0; index < 32U; ++index)

{

num2 -= (uint)(((int)num1 << 4) + (int)num6 ^ (int)num1 + (int)num3 ^ (int)(num1 >> 5) + (int)num7);

num1 -= (uint)(((int)num2 << 4) + (int)num4 ^ (int)num2 + (int)num3 ^ (int)(num2 >> 5) + (int)num5);

num3 -= 2654435769U;

}

v[0] = num1;

v[1] = num2;

}

private static int NextMultipleOf8(int length) => (length + 7) / 8 \* 8;

}

}

# **Приложение В**

using System;

using System.Collections.Generic;

using System.Data;

using System.Diagnostics;

using System.IO;

using System.Linq;

using System.Text;

using System.Windows;

using System.Windows.Forms;

using System.Windows.Media;

using Microsoft.Win32;

using static System.Net.WebRequestMethods;

using File = System.IO.File;

using OpenFileDialog = Microsoft.Win32.OpenFileDialog;

namespace course\_project\_TEA\_DFC

{

/// <summary>

/// Логика взаимодействия для MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

string \_InputMessageFile = "";

string \_EncryptMessageFile = "EncryptMessage.txt";

string \_DecryptMessageFile = "DecryptMessage.txt";

private void LoadTextFromFile\_Click(object sender, EventArgs e)

{

try

{

tbOriginalMessage.Text = "";

tbEncryptMessage.Text = "";

tbDecryptMessage.Text = "";

string fileText;

OpenFileDialog openFileDialog = new OpenFileDialog

{

Filter = "Текстовые файлы (\*.txt)|\*.txt"

};

bool? result = openFileDialog.ShowDialog();

if (result == true)

{

tbOriginalMessage.Text = "";

fileText = File.ReadAllText(openFileDialog.FileName);

\_InputMessageFile = openFileDialog.FileName;

tbOriginalMessage.Text = fileText;

if (string.IsNullOrEmpty(fileText))

{

System.Windows.MessageBox.Show("Выбранный файл пустой", "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

return;

}

}

}

catch (Exception ex)

{

System.Windows.MessageBox.Show(ex.Message, "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

}

}

private void btEncrypt\_Click(object sender, RoutedEventArgs e)

{

var watch = new Stopwatch();

if ((bool)rbTEA.IsChecked)

{

try

{

if (tbKey.Text.Length == 0)

System.Windows.MessageBox.Show("Введите ключ.", "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

else if (tbKey.Text.Length != 16)

tbKey.Text = RepeatOrTruncate(tbKey.Text);

if (tbOriginalMessage.Text.Length == 0)

System.Windows.MessageBox.Show("Введите текст для его преобразования.",

"Ошибка", MessageBoxButton.OK, MessageBoxImage.Error);

else

{

var dfc = new DFC();

watch.Start();

tbEncryptMessage.Text = TEA.Encrypt(tbOriginalMessage.Text, tbKey.Text);

watch.Stop();

dfc.WriteByteArrayToFile(Encoding.UTF8.GetBytes(tbEncryptMessage.Text),

"TEA" + \_EncryptMessageFile);

System.Windows.MessageBox.Show($"Время зашифрования: {watch.Elapsed}",

"Уведомление", MessageBoxButton.OK, MessageBoxImage.Information); ;

}

}

catch (Exception ex)

{

System.Windows.MessageBox.Show(ex.Message, "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

}

}

else if ((bool)rbDFC.IsChecked)

{

if (tbKey.Text.Length == 0)

System.Windows.MessageBox.Show("Введите ключ.", "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

else

{

try

{

if (tbKey.Text.Length < 16)

tbKey.Text = RepeatOrTruncate(tbKey.Text);

if(tbOriginalMessage.Text.Length == 0)

System.Windows.MessageBox.Show("Введите текст для его преобразования.",

"Ошибка", MessageBoxButton.OK, MessageBoxImage.Error);

else {

var dfc = new DFC(Encoding.UTF8.GetBytes(tbKey.Text));

dfc.text = dfc.ReadByteArrayFromString(tbOriginalMessage.Text);

watch.Start();

dfc.WriteByteArrayToFile(dfc.EncryptText(dfc.text), "DFC" + \_EncryptMessageFile);

watch.Stop();

tbEncryptMessage.Text = File.ReadAllText("DFC" + \_EncryptMessageFile);

System.Windows.MessageBox.Show($"Время зашифрования: {watch.Elapsed}",

"Уведомление", MessageBoxButton.OK, MessageBoxImage.Information);

}

}

catch (Exception ex)

{

System.Windows.MessageBox.Show(ex.Message, "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

}

}

}

else

{

System.Windows.MessageBox.Show("Выберите шифр.", "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

}

}

private void btDecrypt\_Click(object sender, RoutedEventArgs e)

{

var watch = new Stopwatch();

if ((bool)rbTEA.IsChecked)

{

try

{

if (tbKey.Text.Length == 0)

System.Windows.MessageBox.Show("Введите ключ.", "Ошибка", MessageBoxButton.OK, MessageBoxImage.Error);

else if (tbKey.Text.Length != 16)

tbKey.Text = RepeatOrTruncate(tbKey.Text);

else if (tbEncryptMessage.Text.Length == 0)

System.Windows.MessageBox.Show("Введите текст для его преобразования.", "Ошибка", MessageBoxButton.OK, MessageBoxImage.Error);

else

{

var dfc = new DFC();

watch.Start();

tbDecryptMessage.Text = TEA.Decrypt(tbEncryptMessage.Text, tbKey.Text);

watch.Stop();

dfc.WriteByteArrayToFile(Encoding.UTF8.GetBytes(tbDecryptMessage.Text),

"TEA" + \_DecryptMessageFile);

System.Windows.MessageBox.Show($"Время расшифрования: {watch.Elapsed}",

"Уведомление", MessageBoxButton.OK, MessageBoxImage.Information);

}

}

catch (Exception ex)

{

System.Windows.MessageBox.Show(ex.Message, "Ошибка", MessageBoxButton.OK, MessageBoxImage.Error);

}

}

else if ((bool)rbDFC.IsChecked)

{

if (tbKey.Text.Length < 16)

tbKey.Text = RepeatOrTruncate(tbKey.Text);

if (tbEncryptMessage.Text.Length == 0)

{

System.Windows.MessageBox.Show("Введите текст для его преобразования.",

"Ошибка", MessageBoxButton.OK, MessageBoxImage.Error);

}

else

{

var dfc = new DFC(Encoding.UTF8.GetBytes(tbKey.Text));

dfc.text = dfc.ReadByteArrayFromFile("DFC" + \_EncryptMessageFile);

watch.Start();

dfc.WriteByteArrayToFile(dfc.DecryptText(dfc.text), "DFC" + \_DecryptMessageFile);

tbDecryptMessage.Text = File.ReadAllText("DFC" + \_DecryptMessageFile);

watch.Stop();

System.Windows.MessageBox.Show($"Время расшифрования: {watch.Elapsed}",

"Уведомление", MessageBoxButton.OK, MessageBoxImage.Information);

}

}

else

{

System.Windows.MessageBox.Show("Выберите шифр.", "Ошибка",

MessageBoxButton.OK, MessageBoxImage.Error);

}

}

private void rbTEA\_Checked(object sender, RoutedEventArgs e)

{

tbDecryptMessage.Text = "";

tbEncryptMessage.Text = "";

}

private void rbDFC\_Checked(object sender, RoutedEventArgs e)

{

tbDecryptMessage.Text = "";

tbEncryptMessage.Text = "";

}

public string RepeatOrTruncate(string message)

{

int targetLength = 16;

int messageLength = message.Length;

if (messageLength == targetLength)

{

// Сообщение уже имеет желаемую длину

return message;

}

else if (messageLength < targetLength)

{

// Добавляем повторения

int repetitionsNeeded = (int)Math.Ceiling((double)targetLength / messageLength);

string repeatedMessage = string.Concat(Enumerable.Repeat(message, repetitionsNeeded));

return repeatedMessage.Substring(0, targetLength);

}

else

{

// Урезаем сообщение

return message.Substring(0, targetLength);

}

}

# **Приложение Г**

<Window x:Class="course\_project\_TEA\_DFC.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:course\_project\_TEA\_DFC"

mc:Ignorable="d"

Title="MainWindow" Height="620" Width="800" ResizeMode="NoResize">

<Grid Background="#FFC3E2E7">

<Grid.RowDefinitions>

<RowDefinition Height="0.1\*"></RowDefinition>

<RowDefinition Height="0.1\*"></RowDefinition>

<RowDefinition Height="0.1\*"></RowDefinition>

<RowDefinition Height="0.05\*"></RowDefinition>

<RowDefinition Height="0.2\*"></RowDefinition>

<RowDefinition Height="0.05\*"></RowDefinition>

<RowDefinition Height="0.2\*"></RowDefinition>

<RowDefinition Height="0.05\*"></RowDefinition>

<RowDefinition Height="0.2\*"></RowDefinition>

<RowDefinition Height="0.05\*"></RowDefinition>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="0.1\*"></ColumnDefinition>

<ColumnDefinition Width="0.2\*"></ColumnDefinition>

<ColumnDefinition Width="0.1\*"></ColumnDefinition>

</Grid.ColumnDefinitions>

<RadioButton x:Name="rbTEA" Grid.Row="0" Grid.Column="0" Content="Алгоритм TEA"

HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="16"

FontFamily="Times New Roman" Checked="rbTEA\_Checked"/>

<Button x:Name="btOpenFile" Grid.Row="0" Content="Открыть файл с исходным сообщением"

FontSize="16" FontFamily="Times New Roman" Height="30" HorizontalAlignment="Center"

VerticalAlignment="Center" Grid.Column="1" Background="#FFA0D4D6" Width="350"

Click="LoadTextFromFile\_Click" BorderBrush="#00707070"/>

<RadioButton x:Name="rbDFC" Grid.Row="0" Grid.Column="2" Content="Алгоритм DFC"

HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="16"

FontFamily="Times New Roman" Checked="rbDFC\_Checked"/>

<Label Content="Ключ:" Grid.Row="1" Grid.Column="0" HorizontalAlignment="Right"

VerticalAlignment="Center" FontSize="16" FontFamily="Times New Roman"

Height="27" Margin="0,0,20,0"/>

<TextBox x:Name="tbKey" Grid.ColumnSpan="3" Grid.Row="1" Text="" FontSize="18"

FontFamily="Times New Roman" HorizontalAlignment="Center" VerticalAlignment="Center"

Width="400" TextAlignment="Justify"/>

<Button x:Name="btEncrypt" Content="Зашифровать" Click="btEncrypt\_Click" FontSize="16"

FontFamily="Times New Roman" Height="30" Grid.Row="2" HorizontalAlignment="Left"

VerticalAlignment="Center" Width="295" Background="#FFA0D4D6" BorderBrush="#00707070"

Grid.ColumnSpan="2" Margin="30,0,0,0" />

<Button x:Name="btDecrypt" Content="Расшифровать" Click="btDecrypt\_Click" FontSize="16"

FontFamily="Times New Roman" Height="30" Grid.Row="2" HorizontalAlignment="Right"

VerticalAlignment="Center" Width="295" Background="#FFA0D4D6" BorderBrush="#00707070"

Grid.ColumnSpan="2" Grid.Column="1" Margin="0,0,30,0" />

<Label Grid.Row="3" Grid.Column="1" Content="Исходное сообщение" HorizontalAlignment="Center"

FontSize="16" FontFamily="Times New Roman" VerticalAlignment="Center"/>

<TextBox x:Name="tbOriginalMessage" Width="700" Grid.Column="0" Grid.ColumnSpan="3"

HorizontalAlignment="Center" Grid.Row="4" Text="" FontSize="16" FontFamily="Times New Roman"

TextAlignment="Justify"/>

<Label Grid.Row="5" Grid.Column="1" Content="Зашифрованное сообщение" HorizontalAlignment="Center"

FontSize="16" FontFamily="Times New Roman" VerticalAlignment="Center"/>

<TextBox x:Name="tbEncryptMessage" Width="700" Grid.Column="0" Grid.ColumnSpan="3"

HorizontalAlignment="Center" Grid.Row="6" Text="" FontSize="16" FontFamily="Times New Roman"

TextAlignment="Justify"/>

<Label Grid.Row="7" Grid.Column="1" Content="Расшифрованное сообщение" HorizontalAlignment="Center"

VerticalAlignment="Center" FontSize="16" FontFamily="Times New Roman"/>

<TextBox x:Name="tbDecryptMessage" Width="700" Grid.Column="0" Grid.ColumnSpan="3"

HorizontalAlignment="Center" Grid.Row="8" Text="" FontSize="16" FontFamily="Times New Roman"

TextAlignment="Justify"/>

</Grid>

</Window>