|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**  **«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ**  **(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)»** | | | | | |
| **Журнал практики** | | | | | |
| Институт №4 | «Радиоэлектроника, инфокоммуникации и информационная безопасность» | | | | |
|  |  | | | | |
| Кафедра | 402 | | Учебная группа | | М4О-309Б-22 |
|  |  | |  | |  |
| ФИО обучающегося | | Кривов Василий Сергеевич | | | |
|  | |  | | | |
| Направление подготовки/ | | 11.03.02 | | | |
| специальность | | Инфокоммуникационные технологии и системы связи | | | |
|  | | *шифр, наименование направления подготовки/специальности* | | | |
|  | |  | | | |
| Вид практики | | Проектная | | | |
|  | | *учебная, производственная, преддипломная или другой вид практики* | | | |
| Оценка за практику | |  | | Куприянов Александр Ильич | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1. **Место и сроки проведения практики:** | | | | |
| Наименование организации: | | Кафедра 402, Московский авиационный институт | | |
| Сроки проведения практики | |  | | |
| дата начала практики: | | 30.06 | | |
| дата окончания практики: | | 27.07 | | |
| 1. **Инструктаж по технике безопасности:** | | | | |
|  |  | Куприянов А. И. |  | \_\_30\_ \_\_\_июня\_\_\_ 2025 г. |
| *подпись проводившего* | *расшифровка подписи* | *дата проведения* |
| 1. **Индивидуальное задание обучающегося:** | | | | |
| Изучить алгоритм блочного шифрования «Кузнечик» (ГОСТ 34.12–2018), разработать | | | | |
| программу для шифрования и дешифрования данных. | | | | |
|  | | | | |
| 1. **План выполнения индивидуального задания обучающегося:** | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **№ п/п** | **Место проведения** | **Тема** | **Период выполнения** |
| 1 | Кафедра 402, МАИ | Инструктаж по технике безопасности, цели и задачи практики | 30.06 |
| 2 | Кафедра 402, МАИ | Изучение основ криптографии, симметричных шифров и блочной структуры | 01.07 – 05.07 |
| 3 | Кафедра 402, МАИ | Изучение алгоритма «Кузнечик» (ГОСТ 34.12–2018) | 08.07 – 12.07 |
| 4 | Кафедра 402, МАИ | Разработка и отладка программы на C++ для реализации шифра | 15.07 – 27.07 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Утверждаю** | | | | | | |  | |
|  |  | Куприянов А. И. |  | \_\_27\_ \_\_\_июля\_\_\_ 2025г. | | | | |
| *подпись руководителя от МАИ* |  | *расшифровка подписи* |  | *дата утверждения* | | | | |
|  |  | Куприянов А. И. |  | \_\_27\_ \_\_\_июля\_\_\_ 2025г. | | | | |
| *подпись руководителя от организации/предприятия* |  | *расшифровка подписи* |  | *дата утверждения* | | | | |
| **Ознакомлен** |  |  | | |  |  |  |  |
|  |  | Кривов В. С. |  | \_\_27\_ \_\_\_июля\_\_\_ 2025г. | | | | |
| *подпись обучающегося* |  | *расшифровка подписи* |  | *дата ознакомления* | | | | |

|  |
| --- |
| 1. **Отзыв руководителя практики от организации/предприятия:** |

Студент Кривов Василий Сергеевич учебной группы М4О-309Б-22 в период с 30.06.2024 по 27.07.2024 проходил учебную практику на кафедре 402 «Радиосистемы и комплексы управления, передачи информации и информационная безопасность». В рамках практики студент изучил теоретические основы криптографии, включая цели и задачи криптографической защиты информации, назначение и принципы работы блочных шифров, а также особенности полей Галуа. Основное внимание было уделено алгоритму симметричного блочного шифрования «Кузнечик» (ГОСТ 34.12–2018), его структуре, раундовым преобразованиям и ключевому расписанию. Студентом была **разработана программа, реализующая алгоритм «Кузнечик» на языке программирования C++**, демонстрирующая применение теоретических знаний на практике. Материалы, изложенные в отчёте обучающегося, полностью соответствуют индивидуальному заданию.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Куприянов А. И. |  | \_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ 2025г. |
| *подпись руководителя от организации/предприятия* |  | *расшифровка подписи* |  | *дата утверждения* |

|  |
| --- |
| 1. **Отчёт обучающего по практике:**  **1. Основные цели учебной практики** Целью учебной практики является приобретение практических навыков и теоретических знаний в области симметричной криптографии путём изучения и реализации алгоритма блочного симметричного шифрования «Кузнечик» (ГОСТ 34.12–2018). Практика направлена на развитие умений анализа действующих стандартов в области информационной безопасности и применение их в программной реализации. **2. Основные задачи учебной практики** В процессе прохождения учебной практики необходимо решить следующие задачи:   1. Изучить теоретические основы симметричного блочного шифрования и общую структуру алгоритма «Кузнечик». 2. Проанализировать государственный стандарт ГОСТ 34.12–2018, регламентирующий алгоритм «Кузнечик». 3. Реализовать алгоритм шифрования и дешифрования «Кузнечик» на языке программирования C++. 4. Провести тестирование реализованного программного модуля, оценить корректность и эффективность работы. 5. Подготовить отчёт о выполненной работе с выводами по результатам практики. |

**1. Теоретические основы симметричного блочного шифрования и структура алгоритма «Кузнечик»**

**1.1 Основы криптографии**

Криптографические алгоритмы занимают особое место среди методов защиты данных от несанкционированного доступа, так как основываются исключительно на свойствах самих сообщений, не завися от материальных носителей или особенностей обработки и передачи информации. С ростом объёмов информационных потоков и развитием компьютерных технологий интерес к криптографии постоянно увеличивается.

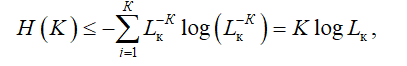
Криптографические методы обеспечивают преобразование информации таким образом, что её содержание становится недоступным без ключа для расшифровки. Это самый надёжный способ защиты, поскольку охраняется сама информация, а не доступ к ней — например, зашифрованный файл останется непонятным даже при краже носителя. Такие методы реализуются в виде программных средств. [1]

Особое значение криптографическая защита имеет в информационных системах аэрокосмических комплексов, где надёжность и безопасность информации критичны из-за высокой стоимости оборудования и важности результатов.

Современная криптография использует различные типы алгоритмов, в частности традиционные симметричные (одноключевые) криптосистемы, где для шифрования и расшифровки сообщения обе стороны используют один и тот же ключ.

Шифрованное сообщение Ш(С, К) сформировано из исходного открытого текста С и ключа К по известному алгоритму и воспроизведена техническим средством разведки (перехвата сообщений) совершенно точно. Вся неизвестность заключена в исходном открытом тексте С и в конкретном выбранном при шифрации ключе К. Исходное сообщение, информационную стойкость которого нужно обеспечить, называется открытым текстом. В результате шифрации образуется криптограмма (шифрограмма, шифровка). Для шифрации и для расшифровки используется ключ. Этот ключ должен быть известен источнику сообщений (передатчику) и получателю (приемнику), причем известен только им одним. Теоретически достижимую предельную способность шифра обеспечивать защиту информации можно характеризовать условной вероятностью Ринф=Р(С|Ш), т.е. вероятностью получения открытого текста (сообщения) С при том условии, что была принята криптограмма Ш.

Ключ – это некоторая последовательность символов. Если К знаков ключа выбираются из алфавита объемом Lк символов, то всего можно сформировать разных ключевых последовательностей:



Цели криптографической защиты информации

1. Конфиденциальность - обеспечение доступа к информации только авторизованным пользователям. Данные шифруются так, чтобы злоумышленник не мог их прочитать без ключа.

2. Целостность - гарантия того, что данные не были изменены или повреждены в процессе хранения или передачи. Это предотвращает подделку и искажение информации.

3. Аутентификация - подтверждение подлинности пользователя или устройства, участвующего в обмене информацией, чтобы исключить возможность подделки личности.

4. Невозможность отказа от ответственности (неподтверждаемость) — механизм, позволяющий доказать авторство отправленного сообщения, чтобы отправитель не мог впоследствии отказаться от его отправки.

5. Контроль доступа - определение и ограничение прав пользователей на доступ к информации и ресурсам.

6. Доступность - обеспечение своевременного и непрерывного доступа авторизованных пользователей к информации и сервисам.

**1.2. Назначение блочных шифров и области их применения**

Блочные шифры разбивают исходный текст на блоки фиксированной длины и шифруют каждый блок. Шифрование может основываться на перестановках символов или замене символов другими из того же или другого алфавита. Современные криптоалгоритмы используют комбинацию этих методов, так как чистые шифры замены или перестановок устарели. [2]

Главное свойство блочного шифрования - каждый символ зашифрованного блока зависит от большинства символов исходного блока, и одинаковые блоки открытого текста не дают одинаковые шифротексты. Это обеспечивает высокую чувствительность к изменениям текста и ключа. Однако при использовании одного ключа для всех блоков возможен криптоанализ на основе повторяющихся шаблонов, особенно из-за избыточности исходного текста.

Кроме того, ошибка в одном символе зашифрованного блока приводит к неправильному расшифрованию всего блока, что ограничивает применение простых блочных шифров для длинных сообщений.

Для повышения стойкости криптосистемы устраняют избыточность текста, применяя сжатие.

Основные методы сжатия:

1. Удаление часто повторяющихся символов (например, пробелов и служебных пометок), которые малоинформативны и снижают безопасность.

2. Повышение энтропии текста - выравнивание вероятностей появления символов. Это достигается рандомизацией (сложением текста с псевдослучайной последовательностью) или использованием многоалфавитных подстановок, при которых текст шифруется несколько раз разными алфавитами.

При равномерном распределении символов в шифротексте частотный анализ становится неэффективным, что значительно повышает криптостойкость.

Сеть Фейстеля - это метод блочного шифрования, разработанный Хорстом Фейстелем в лаборатории IBM в 1971 году. Сегодня многие криптографические протоколы основаны именно на этом подходе.

В основе сети Фейстеля лежит работа с блоками открытого текста, которые делятся на две равные части - левую (L) и правую (R). Далее происходит следующее:

- Левая часть L подвергается преобразованию с помощью функции f и ключа K: вычисляется значение X = f(L, K). Функция f может быть любой, отвечающей требованиям безопасности.

- Результат X складывается по модулю 2 (операция XOR) с правой частью R, которая при этом остаётся без изменений: X = X ⊕ R.

- Затем полученные части меняются местами и объединяются.

Такой набор операций образует одну итерацию - ячейку Фейстеля (Рис.1.):

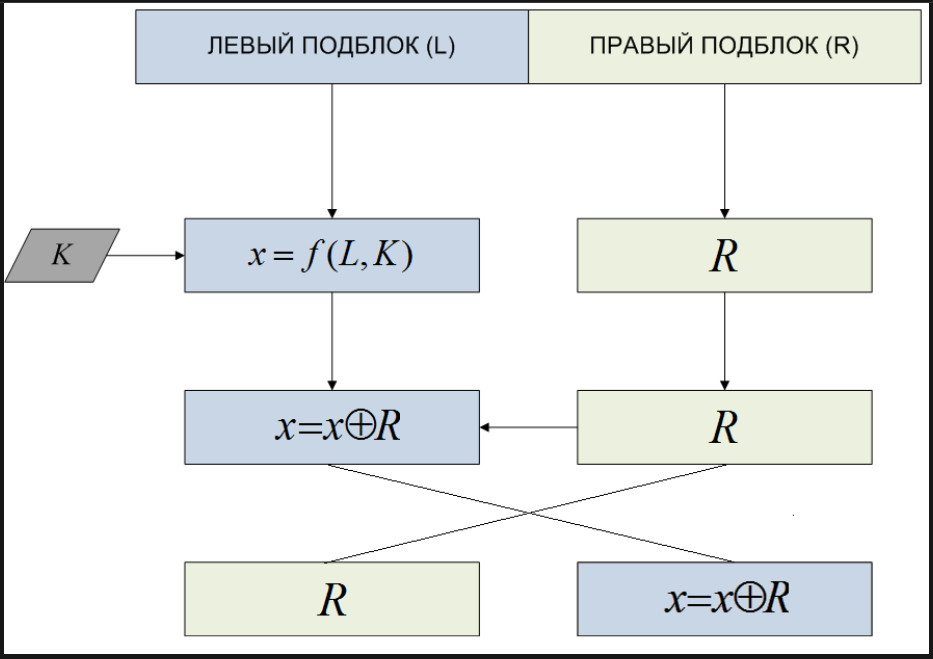


Рисунок. 1. Блочное шифрование

Полученные на выходе первой ячейки подблоки поступают на вход второй ячейки, результирующие подблоки из второй ячейки попадают на вход третьей ячейки и так далее.

**1.3. Математические аспекты для анализа ГОСТ 34.12**

Арифметика полей Галуа – это вид полиномиальной арифметики, где каждый элемент поля представляется в виде полинома. Результатом любых операций (сложения, умножения и т.д.) над элементами поля также является элемент того же поля. [3]

Порядок поля, т.е. количество его элементов, является некоторой натуральной степенью характеристики , где m ϵ N.

При m=1 поле называется простым. В случаях, когда m>1, для образования поля необходим еще порождающий полином степени m, такое поле называется расширенным.

- обозначение поля Галуа. Порождающий полином является неприводимым, то есть простым (по аналогии с простыми числами делится без остатка на 1 и на самого себя).

Так как работа с любой информацией — это работа с байтами, а байт представляет из себя 8 бит, в качестве поля берут и порождающий полином

Однако для начала разберем основные операции в более простом поле с порождающим полиномом

XOR – побитовое сложение по модулю 2, показано на (Рис.2.)

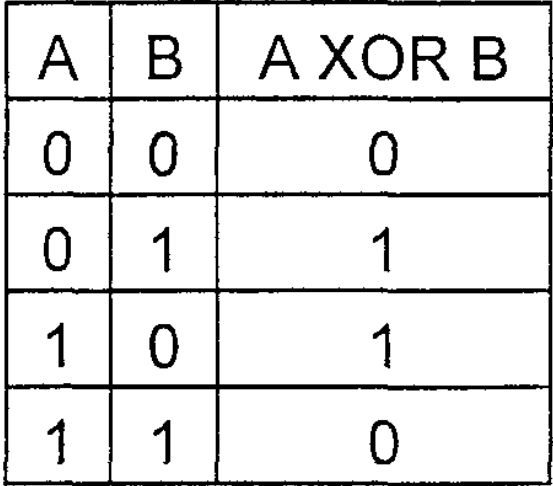


Рисунок 2. XOR

Пример:

В полиномиальном виде данная операция будет выглядеть как:

**2. Структура и принципы работы алгоритмов блочного шифрования по ГОСТ 34.12–2018**

**2.1. Введение**

ГОСТ 34.12–2018 определяет два алгоритма блочного шифрования, реализующих симметричную криптографическую защиту информации: **«Кузнечик»** (с длиной блока 128 бит) и **«Магма»** (с длиной блока 64 бита). Оба алгоритма удовлетворяют современным требованиям по криптографической стойкости, адаптированы как для программной, так и для аппаратной реализации и являются стандартом Российской Федерации с 1 июня 2019 года. Несмотря на общую цель – преобразование данных с использованием общего ключа – их внутренняя структура принципиально различается.

**2.2. Общие принципы**

Блочные шифры работают по схожему принципу: входные данные разбиваются на блоки фиксированной длины, каждый из которых обрабатывается алгоритмом шифрования с использованием определённого секретного ключа. В результате формируется шифртекст той же длины. Обратное преобразование – расшифрование – возвращает исходный открытый текст. [4]

**2.3. Алгоритм «Кузнечик» (блок 128 бит)**

Алгоритм «Кузнечик» реализует блочное шифрование с размером блока 128 бит и длиной ключа 256 бит. Он основан на **SP-сети**, в которой каждая раундовая итерация состоит из следующих компонентов:

1. **XOR с раундовым ключом (операция X)**
2. **S-преобразование (S-блоки)**
3. **Линейное преобразование (операция L)**

В отличие от алгоритмов, построенных на сети Фейстеля (например, Магма или DES), SP-сети обеспечивают высокую диффузию и нелинейность за счёт последовательного применения подстановок и линейных преобразований ко всему блоку сразу. В «Кузнечике» применяется 9 таких полных раундов, а в финале добавляется десятый раунд, включающий только операцию XOR с ключом.

Каждый из первых девяти раундов включает:

1. **Операцию-X[k]**  
   Выполняется поэлементное побитовое сложение по модулю 2 (XOR) между входным блоком и текущим раундовым ключом. Это необходимый шаг, позволяющий связать входные данные с ключом.
2. **Преобразование-S**  
   После XOR входной блок разбивается на 16 байт. Для каждого байта применяется нелинейная подстановка – так называемый S-блок. В ГОСТе она задана в виде таблицы размером 256 значений (от 0 до 255), где каждому значению соответствует новое байтовое значение. Это биективное отображение, то есть оно однозначно обратимо. S-блок в «Кузнечике» предназначен для внедрения криптографической **нелинейности**.
3. **Линейное-преобразование-L**  
   После S-преобразования полученные байты проходят через линейное преобразование. Оно выполняется в поле GF(2^8) и представляет собой цепочку из 16 итераций так называемого R-преобразования. В каждой итерации байты перемешиваются по определённому правилу с использованием фиксированных коэффициентов. Конкретно, каждый байт в новом состоянии определяется как XOR линейной комбинации текущих байтов с коэффициентами в поле GF(2^8).  
   В результате, один изменённый байт на входе преобразования приводит к изменению всех 16 байт на выходе – это свойство называется **диффузией** и крайне важно для защиты от линейного и дифференциального криптоанализа.

Таким образом, каждый раунд реализует комбинацию:

​

Финальный раунд (десятый) состоит только из XOR с десятым ключом:

​

**Ключевое расписание (Key Schedule)**

Генерация раундовых ключей – важнейший компонент алгоритма, поскольку от криптографической стойкости ключевого расписания зависит стойкость всего шифра. В «Кузнечике» мастер-ключ длиной 256 бит разделяется на два 128-битных подблока: K₁ и K₂. Далее при помощи раундовых констант и специальной функции **F** происходит 32 раундовых преобразования, в которых подблоки циклически обновляются и перемешиваются.

Каждая итерация выглядит так:

Где:

* Ci​ – раундовая константа (определяется как результат преобразования L от шестнадцатеричного представления номера раунда),

В результате формируются **10** 128-битных раундовых ключей ​, которые применяются по одному на каждый раунд.

**Обратное преобразование (расшифрование)**

Для расшифрования необходимо применить **обратные операции в обратном порядке**:

* Обратная линейная трансформация,
* Обратная подстановка , то есть обратная таблица S-блока,
* XOR с соответствующим ключом.

Выглядит расшифрование как последовательность:

**Преимущества конструкции**

Архитектура «Кузнечика» была выбрана не случайно. SP-сеть обеспечивает:

* Хорошую диффузию благодаря линейному слою L,
* Сильную нелинейность, устойчивость к атакам на основе различий,
* Возможность параллельной обработки байтов (в отличие от Фейстеля),
* Высокую скорость при аппаратной реализации (S и L можно реализовать таблицами и схемами с конвейером).

**2.4. Алгоритм «Магма» (блок 64 бита)**

Алгоритм «Магма» основан на классической **сети Фейстеля**, аналогичной структуре шифров DES и GOST 28147-89. Он работает с блоками по 64 бита и ключами длиной 256 бит. В отличие от «Кузнечика», здесь структура итеративна и предполагает деление входного блока на две половины по 32 бита. На каждом раунде одна половина подвергается преобразованию, а вторая – просто копируется и XOR’ится с результатом. После чего половины меняются местами.

На каждом из 32 раундов выполняется следующая последовательность:

1. Левая половина блока Li​ передаётся в функцию **F**, которая использует текущий раундовый ключ Ki​.
2. Выполняется сложение по модулю 232 :

Результат разбивается на 8 четырёхбитных участков, каждый из которых заменяется по **одному из восьми S-блоков** (разных для каждого участка).

1. Результат сдвигается циклически влево на 11 бит.
2. Полученное значение XOR’ится с правой половиной Ri, давая новую левую часть.
3. Половины меняются местами:

Так как это сеть Фейстеля, расшифрование выполняется точно так же, но в обратном порядке применения ключей.

**Ключевое расписание**

Ключ длиной 256 бит разбивается на **восемь 32-битных слов**:

Раундовые ключи формируются следующим образом:

* Раунды 1–24: k1 … k8​ повторяются 3 раза подряд (8×3=24),
* Раунды 25–32: используются те же ключи, но в **обратном порядке**: k8 … k1​.

Таким образом получается 32 раундовых ключа.

**Преобразование F и S-блоки**

Функция F включает:

* Сложение по модулю 232,
* Применение 8 различных S-блоков (один на каждые 4 бита),
* Циклический сдвиг на 11 бит.

S-блоки заданы явно в стандарте как таблицы из 16 элементов (для входов от 0 до 15). Каждый байт 32-битного результата пропускается через свою таблицу подстановки.

**Преимущества и особенности**

* Простая и проверенная временем архитектура Фейстеля.
* Высокая скорость на программном уровне благодаря простоте операций (сложение, сдвиг, XOR).
* Возможность использовать аппаратные реализации ГОСТ 28147-89.

Однако в сравнении с «Кузнечиком» он демонстрирует меньшую диффузию, и сегодня рассматривается как совместимый алгоритм для обеспечения преемственности.

3. Программная реализация шифрования на основе алгоритма Кузнечик

**Работа основного алгоритма шифрования в программной интерпретации:**

Для того, чтобы зашифровать нужный нам файл, который весит 10 Мб, нам надо для начала установить и открыть программу (Рис.3):

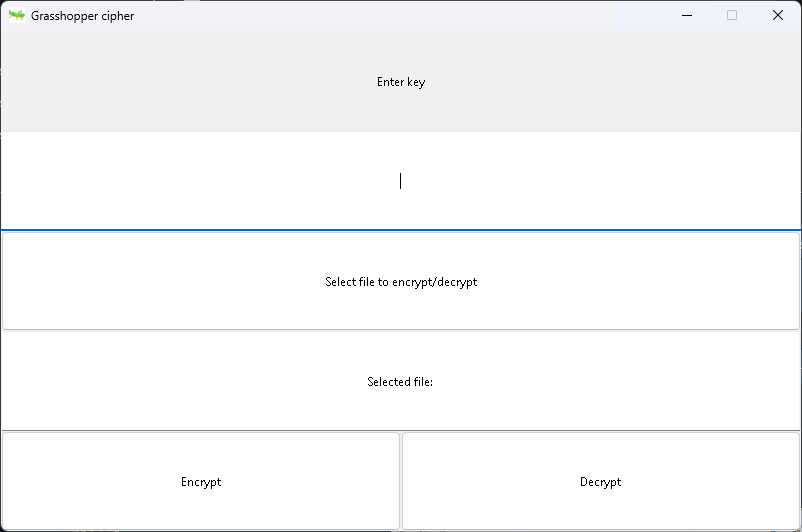


Рисунок 3. Программа шифрования “Кузнечик”

Далее нужно придумать ключ для нашего шифра, который будет длиной 64 шестнадцатеричных символа, с нижним регистром, для примера возьмём такой ключ:

111111aaaaaa222222bbbbbb333333dddddd444444dddddd555555eeeeee66ff

Данный ключ состоит только из символов от 0 до 9 и от a до f, все буквы в нижнем регистре. Вбиваем наш ключ в программу (Рис.4):

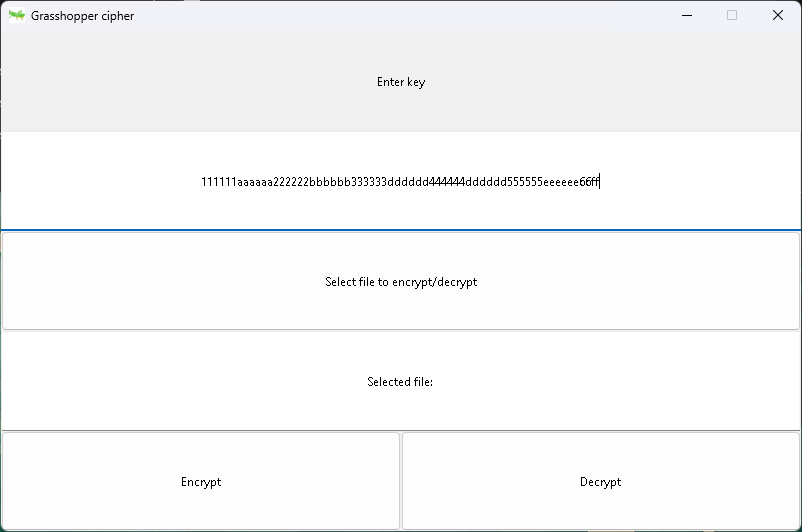


Рисунок 4. Программа шифрования “Кузнечик” с введённым ключом шифрования

После того, как мы ввели наш ключ шифрования, нужно выбрать файл, который мы хотим зашифровать, важно, чтобы файл весил 10 Мб, так как программа будет работать очень долго, если не удовлетворить это условие.

Для примера возьмём картинку из интернета, весом в 8 Кб и добавим её в нашу программу (Рис.5):

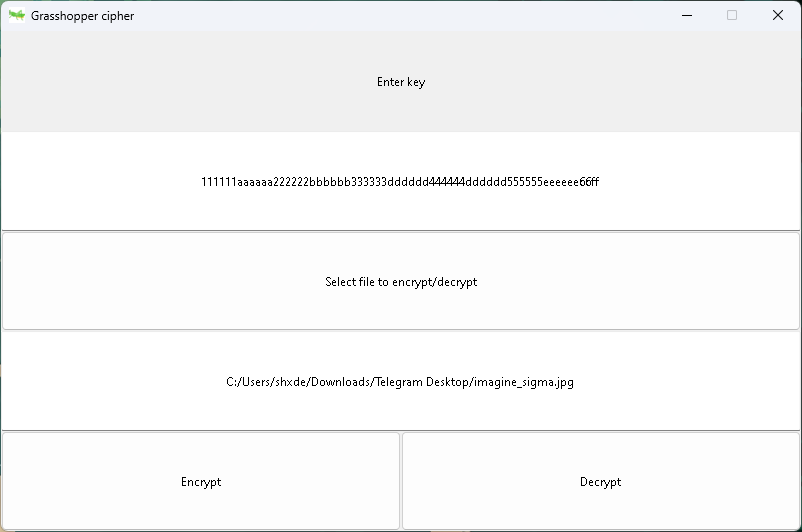


Рисунок 5. Программа шифрования “Кузнечик” с добавленным файлом для шифрования

Перед тем, как зашифровать файл, надо проверить, что он исправно открывается (Рис.6):



Рисунок. 6 Файл, который мы хотим зашифровать

В том, что файл открывается исправно, мы убедились, теперь приступим к шифрованию, в программе необходимо нажать кнопку “Encrypt”, это запустит процесс шифрования файла и без обратной дешифровки при помощи ключа мы больше не сможем открыть данный файл. Проверим работу шифрования, попытаемся открыть зашифрованный файл (Рис.7):

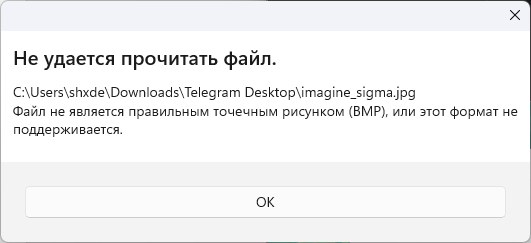


Рисунок. 7 Результат работы программы шифрования

Как и ожидалось, файл мы не смогли открыть и компьютер выдал ошибку, это означает, что наш файл был успешно зашифрован и доступ к нему имеют только люди, у которых есть ключ шифрования.

Теперь попробуем дешифровать зашифрованный файл, повторяем все те же действия, как на (Рис.5), вводим ключ шифрования, выбираем зашифрованный файл. Далее, чтобы дешифровать файл, в программе необходимо нажать кнопку “Decrypt”, это запустит процесс дешифрования файла. Когда программа выполнила все действия, мы можем убедиться, что всё работает исправно, откроем дешифрованный файл (Рис.8):



Рисунок. 8 Результат работы программы дешифрования

Как можно заметить, программа без ошибки шифрует и дешифрует файлы, это говорит о исправности её работы. [5], [6]

**Принцип работа основного алгоритма шифрования в кодовой интерпретации:**

#include "grasshopper\_cipher.h"

std::vector<unsigned char> Pi = {

0xFC, 0xEE, 0xDD, 0x11, 0xCF, 0x6E, 0x31, 0x16, 0xFB, 0xC4, 0xFA, 0xDA,

0x23, 0xC5, 0x04, 0x4D, 0xE9, 0x77, 0xF0, 0xDB, 0x93, 0x2E, 0x99, 0xBA,

0x17, 0x36, 0xF1, 0xBB, 0x14, 0xCD, 0x5F, 0xC1, 0xF9, 0x18, 0x65, 0x5A,

0xE2, 0x5C, 0xEF, 0x21, 0x81, 0x1C, 0x3C, 0x42, 0x8B, 0x01, 0x8E, 0x4F,

0x05, 0x84, 0x02, 0xAE, 0xE3, 0x6A, 0x8F, 0xA0, 0x06, 0x0B, 0xED, 0x98,

0x7F, 0xD4, 0xD3, 0x1F, 0xEB, 0x34, 0x2C, 0x51, 0xEA, 0xC8, 0x48, 0xAB,

0xF2, 0x2A, 0x68, 0xA2, 0xFD, 0x3A, 0xCE, 0xCC, 0xB5, 0x70, 0x0E, 0x56,

0x08, 0x0C, 0x76, 0x12, 0xBF, 0x72, 0x13, 0x47, 0x9C, 0xB7, 0x5D, 0x87,

0x15, 0xA1, 0x96, 0x29, 0x10, 0x7B, 0x9A, 0xC7, 0xF3, 0x91, 0x78, 0x6F,

0x9D, 0x9E, 0xB2, 0xB1, 0x32, 0x75, 0x19, 0x3D, 0xFF, 0x35, 0x8A, 0x7E,

0x6D, 0x54, 0xC6, 0x80, 0xC3, 0xBD, 0x0D, 0x57, 0xDF, 0xF5, 0x24, 0xA9,

0x3E, 0xA8, 0x43, 0xC9, 0xD7, 0x79, 0xD6, 0xF6, 0x7C, 0x22, 0xB9, 0x03,

0xE0, 0x0F, 0xEC, 0xDE, 0x7A, 0x94, 0xB0, 0xBC, 0xDC, 0xE8, 0x28, 0x50,

0x4E, 0x33, 0x0A, 0x4A, 0xA7, 0x97, 0x60, 0x73, 0x1E, 0x00, 0x62, 0x44,

0x1A, 0xB8, 0x38, 0x82, 0x64, 0x9F, 0x26, 0x41, 0xAD, 0x45, 0x46, 0x92,

0x27, 0x5E, 0x55, 0x2F, 0x8C, 0xA3, 0xA5, 0x7D, 0x69, 0xD5, 0x95, 0x3B,

0x07, 0x58, 0xB3, 0x40, 0x86, 0xAC, 0x1D, 0xF7, 0x30, 0x37, 0x6B, 0xE4,

0x88, 0xD9, 0xE7, 0x89, 0xE1, 0x1B, 0x83, 0x49, 0x4C, 0x3F, 0xF8, 0xFE,

0x8D, 0x53, 0xAA, 0x90, 0xCA, 0xD8, 0x85, 0x61, 0x20, 0x71, 0x67, 0xA4,

0x2D, 0x2B, 0x09, 0x5B, 0xCB, 0x9B, 0x25, 0xD0, 0xBE, 0xE5, 0x6C, 0x52,

0x59, 0xA6, 0x74, 0xD2, 0xE6, 0xF4, 0xB4, 0xC0, 0xD1, 0x66, 0xAF, 0xC2,

0x39, 0x4B, 0x63, 0xB6};

std::vector<unsigned char> reverse\_Pi = {

0xA5, 0x2D, 0x32, 0x8F, 0x0E, 0x30, 0x38, 0xC0, 0x54, 0xE6, 0x9E, 0x39,

0x55, 0x7E, 0x52, 0x91, 0x64, 0x03, 0x57, 0x5A, 0x1C, 0x60, 0x07, 0x18,

0x21, 0x72, 0xA8, 0xD1, 0x29, 0xC6, 0xA4, 0x3F, 0xE0, 0x27, 0x8D, 0x0C,

0x82, 0xEA, 0xAE, 0xB4, 0x9A, 0x63, 0x49, 0xE5, 0x42, 0xE4, 0x15, 0xB7,

0xC8, 0x06, 0x70, 0x9D, 0x41, 0x75, 0x19, 0xC9, 0xAA, 0xFC, 0x4D, 0xBF,

0x2A, 0x73, 0x84, 0xD5, 0xC3, 0xAF, 0x2B, 0x86, 0xA7, 0xB1, 0xB2, 0x5B,

0x46, 0xD3, 0x9F, 0xFD, 0xD4, 0x0F, 0x9C, 0x2F, 0x9B, 0x43, 0xEF, 0xD9,

0x79, 0xB6, 0x53, 0x7F, 0xC1, 0xF0, 0x23, 0xE7, 0x25, 0x5E, 0xB5, 0x1E,

0xA2, 0xDF, 0xA6, 0xFE, 0xAC, 0x22, 0xF9, 0xE2, 0x4A, 0xBC, 0x35, 0xCA,

0xEE, 0x78, 0x05, 0x6B, 0x51, 0xE1, 0x59, 0xA3, 0xF2, 0x71, 0x56, 0x11,

0x6A, 0x89, 0x94, 0x65, 0x8C, 0xBB, 0x77, 0x3C, 0x7B, 0x28, 0xAB, 0xD2,

0x31, 0xDE, 0xC4, 0x5F, 0xCC, 0xCF, 0x76, 0x2C, 0xB8, 0xD8, 0x2E, 0x36,

0xDB, 0x69, 0xB3, 0x14, 0x95, 0xBE, 0x62, 0xA1, 0x3B, 0x16, 0x66, 0xE9,

0x5C, 0x6C, 0x6D, 0xAD, 0x37, 0x61, 0x4B, 0xB9, 0xE3, 0xBA, 0xF1, 0xA0,

0x85, 0x83, 0xDA, 0x47, 0xC5, 0xB0, 0x33, 0xFA, 0x96, 0x6F, 0x6E, 0xC2,

0xF6, 0x50, 0xFF, 0x5D, 0xA9, 0x8E, 0x17, 0x1B, 0x97, 0x7D, 0xEC, 0x58,

0xF7, 0x1F, 0xFB, 0x7C, 0x09, 0x0D, 0x7A, 0x67, 0x45, 0x87, 0xDC, 0xE8,

0x4F, 0x1D, 0x4E, 0x04, 0xEB, 0xF8, 0xF3, 0x3E, 0x3D, 0xBD, 0x8A, 0x88,

0xDD, 0xCD, 0x0B, 0x13, 0x98, 0x02, 0x93, 0x80, 0x90, 0xD0, 0x24, 0x34,

0xCB, 0xED, 0xF4, 0xCE, 0x99, 0x10, 0x44, 0x40, 0x92, 0x3A, 0x01, 0x26,

0x12, 0x1A, 0x48, 0x68, 0xF5, 0x81, 0x8B, 0xC7, 0xD6, 0x20, 0x0A, 0x08,

0x00, 0x4C, 0xD7, 0x74};

std::vector<int> lagua\_table = {

1, 2, 4, 8, 16, 32, 64, 128, 195, 69, 138, 215, 109, 218, 119,

238, 31, 62, 124, 248, 51, 102, 204, 91, 182, 175, 157, 249, 49, 98,

196, 75, 150, 239, 29, 58, 116, 232, 19, 38, 76, 152, 243, 37, 74,

148, 235, 21, 42, 84, 168, 147, 229, 9, 18, 36, 72, 144, 227, 5,

10, 20, 40, 80, 160, 131, 197, 73, 146, 231, 13, 26, 52, 104, 208,

99, 198, 79, 158, 255, 61, 122, 244, 43, 86, 172, 155, 245, 41, 82,

164, 139, 213, 105, 210, 103, 206, 95, 190, 191, 189, 185, 177, 161, 129,

193, 65, 130, 199, 77, 154, 247, 45, 90, 180, 171, 149, 233, 17, 34,

68, 136, 211, 101, 202, 87, 174, 159, 253, 57, 114, 228, 11, 22, 44,

88, 176, 163, 133, 201, 81, 162, 135, 205, 89, 178, 167, 141, 217, 113,

226, 7, 14, 28, 56, 112, 224, 3, 6, 12, 24, 48, 96, 192, 67,

134, 207, 93, 186, 183, 173, 153, 241, 33, 66, 132, 203, 85, 170, 151,

237, 25, 50, 100, 200, 83, 166, 143, 221, 121, 242, 39, 78, 156, 251,

53, 106, 212, 107, 214, 111, 222, 127, 254, 63, 126, 252, 59, 118, 236,

27, 54, 108, 216, 115, 230, 15, 30, 60, 120, 240, 35, 70, 140, 219,

117, 234, 23, 46, 92, 184, 179, 165, 137, 209, 97, 194, 71, 142, 223,

125, 250, 55, 110, 220, 123, 246, 47, 94, 188, 187, 181, 169, 145, 225,

1};

std::vector<int> lagua\_table2 = {

0, 1, 157, 2, 59, 158, 151, 3, 53, 60, 132, 159, 70, 152, 216,

4, 118, 54, 38, 61, 47, 133, 227, 160, 181, 71, 210, 153, 34, 217,

16, 5, 173, 119, 221, 55, 43, 39, 191, 62, 88, 48, 83, 134, 112,

228, 247, 161, 28, 182, 20, 72, 195, 211, 242, 154, 129, 35, 207, 218,

80, 17, 204, 6, 106, 174, 164, 120, 9, 222, 237, 56, 67, 44, 31,

40, 109, 192, 77, 63, 140, 89, 185, 49, 177, 84, 125, 135, 144, 113,

23, 229, 167, 248, 97, 162, 235, 29, 75, 183, 123, 21, 95, 73, 93,

196, 198, 212, 12, 243, 200, 155, 149, 130, 214, 36, 225, 208, 14, 219,

189, 81, 245, 18, 240, 205, 202, 7, 104, 107, 65, 175, 138, 165, 142,

121, 233, 10, 91, 223, 147, 238, 187, 57, 253, 68, 51, 45, 116, 32,

179, 41, 171, 110, 86, 193, 26, 78, 127, 64, 103, 141, 137, 90, 232,

186, 146, 50, 252, 178, 115, 85, 170, 126, 25, 136, 102, 145, 231, 114,

251, 24, 169, 230, 101, 168, 250, 249, 100, 98, 99, 163, 105, 236, 8,

30, 66, 76, 108, 184, 139, 124, 176, 22, 143, 96, 166, 74, 234, 94,

122, 197, 92, 199, 11, 213, 148, 13, 224, 244, 188, 201, 239, 156, 254,

150, 58, 131, 52, 215, 69, 37, 117, 226, 46, 209, 180, 15, 33, 220,

172, 190, 42, 82, 87, 246, 111, 19, 27, 241, 194, 206, 128, 203, 79,

0};

std::vector<unsigned char> l\_vec = {148, 32, 133, 16, 194, 192, 1, 251,

1, 192, 194, 16, 133, 32, 148, 1};

std::vector<std::vector<unsigned char>> iter\_C(32,

std::vector<unsigned char>(16,0));

std::vector<std::vector<unsigned char>> iter\_key(10,

std::vector<unsignedchar>(16, 0));

std::vector<unsigned char> GOST\_Kuz\_X(std::vector<unsigned char> a,

std::vector<unsigned char> b) {

std::vector<unsigned char> result(BLOCK\_SIZE);

for (int i = 0; i < BLOCK\_SIZE; i++) {

result[i] = (unsigned char)(a[i] ^ b[i]);

}

return result;

}

std::vector<unsigned char> GOST\_Kuz\_S(std::vector<unsigned char> in\_data) {

std::vector<unsigned char> result(BLOCK\_SIZE);

for (int i = 0; i < BLOCK\_SIZE; i++) {

int data = in\_data[i];

if (data < 0) data = data + 256;

result[i] = Pi[data];

}

return result;

}

unsigned char GOST\_Kuz\_GF\_mul(unsigned char a, unsigned char b) {

if (a == 0 || b == 0) return 0;

int stepen\_a = lagua\_table2[a - 1];

int stepen\_b = lagua\_table2[b - 1];

int stepen\_res = (stepen\_a + stepen\_b) % 255;

int result = lagua\_table[stepen\_res];

return result;

}

std::vector<unsigned char> GOST\_Kuz\_R(std::vector<unsigned char> state) {

unsigned char a\_0 = 0;

std::vector<unsigned char> result(BLOCK\_SIZE);

for (int i = 15; i >= 0; i--) {

if (i != 0) {

result[i] = state[i - 1];

}

a\_0 ^= GOST\_Kuz\_GF\_mul(state[i], l\_vec[i]);

}

result[0] = a\_0;

return result;

}

std::vector<unsigned char> GOST\_Kuz\_L(std::vector<unsigned char> in\_data) {

std::vector<unsigned char> result(BLOCK\_SIZE);

std::vector<unsigned char> internal = in\_data;

for (int i = 0; i < 16; i++) {

internal = GOST\_Kuz\_R(internal);

}

result = internal;

return result;

}

std::vector<unsigned char> GOST\_Kuz\_reverse\_S(

std::vector<unsigned char> in\_data) {

std::vector<unsigned char> result(BLOCK\_SIZE);

for (int i = 0; i < BLOCK\_SIZE; i++) {

int data = in\_data[i];

if (data < 0) {

data = data + 256;

}

result[i] = reverse\_Pi[data];

}

return result;

}

std::vector<unsigned char> GOST\_Kuz\_reverse\_R(

std::vector<unsigned char> state) {

unsigned char a\_15 = 0;

std::vector<unsigned char> result(BLOCK\_SIZE, 0);

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

result[i] = state[i + 1];

a\_15 ^= GOST\_Kuz\_GF\_mul(state[i + 1], l\_vec[i]);

}

a\_15 ^= GOST\_Kuz\_GF\_mul(state[0], l\_vec[15]);

result[15] = a\_15;

return result;

}

std::vector<unsigned char> GOST\_Kuz\_reverse\_L(

std::vector<unsigned char> in\_data) {

std::vector<unsigned char> result(BLOCK\_SIZE, 0);

std::vector<unsigned char> internal;

internal = in\_data;

for (int i = 0; i < 16; i++) {

internal = GOST\_Kuz\_reverse\_R(internal);

}

result = internal;

return result;

}

void GOST\_Kuz\_Get\_C() {

std::vector<unsigned char> iter\_num(16, 0);

for (int i = 0; i < 32; i++) {

iter\_num[15] = i + 1;

iter\_C[i] = GOST\_Kuz\_L(iter\_num);

}

}

std::vector<std::vector<unsigned char>> GOST\_Kuz\_F(

std::vector<unsigned char> in\_key\_1, std::vector<unsigned char> in\_key\_2,

std::vector<unsigned char> iter\_const) {

std::vector<unsigned char> internal;

std::vector<unsigned char> out\_key\_2 = in\_key\_1;

internal = GOST\_Kuz\_X(in\_key\_1, iter\_const);

internal = GOST\_Kuz\_S(internal);

internal = GOST\_Kuz\_L(internal);

std::vector<unsigned char> out\_key\_1 = GOST\_Kuz\_X(internal, in\_key\_2);

std::vector<std::vector<unsignedchar>>key(2, std::vector<unsigned char>());

key[0] = out\_key\_1;

key[1] = out\_key\_2;

return key;

}

void GOST\_Kuz\_Expand\_Key(std::vector<unsigned char> key\_1,

std::vector<unsigned char> key\_2) {

std::vector<std::vector<unsigned char>> iter12(2,

std::vector<unsigned char>());

std::vector<std::vector<unsigned char>> iter34(2,

std::vector<unsigned char>());

iter\_key[0] = key\_1;

iter\_key[1] = key\_2;

iter12[0] = key\_1;

iter12[1] = key\_2;

for (int i = 0; i < 4; i++) {

iter34 = GOST\_Kuz\_F(iter12[0], iter12[1], iter\_C[0 + 8 \* i]);

iter12 = GOST\_Kuz\_F(iter34[0], iter34[1], iter\_C[1 + 8 \* i]);

iter34 = GOST\_Kuz\_F(iter12[0], iter12[1], iter\_C[2 + 8 \* i]);

iter12 = GOST\_Kuz\_F(iter34[0], iter34[1], iter\_C[3 + 8 \* i]);

iter34 = GOST\_Kuz\_F(iter12[0], iter12[1], iter\_C[4 + 8 \* i]);

iter12 = GOST\_Kuz\_F(iter34[0], iter34[1], iter\_C[5 + 8 \* i]);

iter34 = GOST\_Kuz\_F(iter12[0], iter12[1], iter\_C[6 + 8 \* i]);

iter12 = GOST\_Kuz\_F(iter34[0], iter34[1], iter\_C[7 + 8 \* i]);

iter\_key[2 \* i + 2] = iter12[0];

iter\_key[2 \* i + 3] = iter12[1];

}

}

std::vector<unsigned char> GOST\_Kuz\_Encript(std::vector<unsigned char> blk,

std::vector<unsigned char> key\_1,

std::vector<unsigned char> key\_2) {

std::vector<unsigned char> result(BLOCK\_SIZE);

result = blk;

for (int i = 0; i < 9; i++) {

result = GOST\_Kuz\_X(iter\_key[i], result);

result = GOST\_Kuz\_S(result);

result = GOST\_Kuz\_L(result);

}

result = GOST\_Kuz\_X(result, iter\_key[9]);

return result;

}

std::vector<unsigned char> GOST\_Kuz\_Decript(std::vector<unsigned char> blk,

std::vector<unsigned char> key\_1,

std::vector<unsigned char> key\_2) {

std::vector<unsigned char> result(BLOCK\_SIZE, 0);

result = blk;

for (int i = 9; i >= 1; i--) {

result = GOST\_Kuz\_X(iter\_key[i], result);

result = GOST\_Kuz\_reverse\_L(result);

result = GOST\_Kuz\_reverse\_S(result);

}

result = GOST\_Kuz\_X(result, iter\_key[0]);

return result;

}

void separate\_key(std::string main\_key, std::vector<unsigned char>& key\_1,

std::vector<unsigned char>& key\_2) {int tmp1 = 0, tmp2 = 0;

for (int i = 0; i < 16; i++) {

if (48 <= main\_key[2 \* i] && main\_key[2 \* i] <= 57)

tmp1 = main\_key[2 \* i] - 48;

if (97 <= main\_key[2 \* i] && main\_key[2 \* i] <= 102)

tmp1 = main\_key[2 \* i] - 87;

if (48 <= main\_key[2 \* i + 1] && main\_key[2 \* i + 1] <= 57)

tmp2 = main\_key[2 \* i + 1] - 48;

if (97 <= main\_key[2 \* i + 1] && main\_key[2 \* i + 1] <= 102)

tmp2 = main\_key[2 \* i + 1] - 87;

key\_1[i] = (unsigned char)(tmp1 << 4 | tmp2);

if (48 <= main\_key[2 \* i + 32] && main\_key[2 \* i + 32] <= 57)

tmp1 = main\_key[2 \* i + 32] - 48;

if (97 <= main\_key[2 \* i + 32] && main\_key[2 \* i + 32] <= 102)

tmp1 = main\_key[2 \* i + 32] - 87;

if (48 <= main\_key[2 \* i + 32 + 1] && main\_key[2 \* i + 32 + 1] <= 57)

tmp2 = main\_key[2 \* i + 32 + 1] - 48;

if (97 <= main\_key[2 \* i + 32 + 1] && main\_key[2 \* i + 32 + 1] <= 102)

tmp2 = main\_key[2 \* i + 32 + 1] - 87;

key\_2[i] = (unsigned char)(tmp1 << 4 | tmp2);

}

}

void Encript\_File(std::string file\_name, std::string main\_key) {

std::vector<unsigned char> key\_1(16, 0);

std::vector<unsigned char> key\_2(16, 0);

separate\_key(main\_key, key\_1, key\_2);

GOST\_Kuz\_Get\_C();

GOST\_Kuz\_Expand\_Key(key\_1, key\_2);

std::vector<unsigned char> buffer(16, 0);

std::ifstream file(file\_name, std::ios::binary);

std::ofstream out(file\_name + ".crypto", std::ios::binary);

while (1) {

file.read(reinterpret\_cast<char\*>(buffer.data()), 16);

int bytesRead = file.gcount();

if (bytesRead == 0) break;

buffer = GOST\_Kuz\_Encript(buffer, key\_1, key\_2);

for (int i = 0; i < 16; i++) out << buffer[i];

buffer = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

}

file.close();

out.close();

QFile file\_remove(QString::fromStdString(file\_name));

file\_remove.remove();

file\_remove.close();

QFile file\_rename(QString::fromStdString(file\_name + ".crypto"));

file\_rename.rename(QString::fromStdString(file\_name));

file\_rename.close();

}

void Decript\_File(std::string file\_name, std::string main\_key) {

std::vector<unsigned char> key\_1(16, 0);

std::vector<unsigned char> key\_2(16, 0);

separate\_key(main\_key, key\_1, key\_2);

GOST\_Kuz\_Get\_C();

GOST\_Kuz\_Expand\_Key(key\_1, key\_2);

std::vector<unsigned char> buffer(16, 0);

std::ifstream file(file\_name, std::ios::binary);

std::ofstream out(file\_name + ".crypto", std::ios::binary);

while (1) {

file.read(reinterpret\_cast<char\*>(buffer.data()), 16);

int bytesRead = file.gcount();

if (bytesRead == 0) break;

buffer = GOST\_Kuz\_Decript(buffer, key\_1, key\_2);

for (int i = 0; i < 16; i++) out << buffer[i];

buffer = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

}

file.close();

out.close();

QFile file\_remove(QString::fromStdString(file\_name));

file\_remove.remove();

file\_remove.close();

QFile file\_rename(QString::fromStdString(file\_name + ".crypto"));

file\_rename.rename(QString::fromStdString(file\_name));

file\_rename.close();

}

int validate(std::string main\_key) {

int result = 0;

if (main\_key.length() != 64) {

result = 1;

} else {

for (int i = 0; i < main\_key.length(); i++) {

if (!(48 <= main\_key[i] && main\_key[i] <= 57 ||

97 <= main\_key[i] && main\_key[i] <= 102)) {

result = 2;

break;

}

}

}

return result;

}

****Заключение****

В ходе прохождения учебной практики на кафедре 402 Московского авиационного института была успешно выполнена поставленная задача по изучению и реализации алгоритма симметричного блочного шифрования "Кузнечик" (ГОСТ 34.12-2018).

В процессе работы:

1. Были глубоко изучены теоретические основы криптографии, включая принципы работы блочных шифров, математические основы полей Галуа и особенности SP-сетей.
2. Проведён детальный анализ стандарта ГОСТ 34.12-2018, рассмотрены структура алгоритма "Кузнечик", его криптографические преобразования и ключевое расписание.
3. Разработана программная реализация алгоритма на языке C++, включающая функции шифрования и дешифрования данных. Программа успешно протестирована на различных файлах, что подтвердило корректность реализации.
4. Создан удобный интерфейс для работы с алгоритмом, позволяющий выполнять шифрование и дешифрование файлов с заданным ключом.

Полученные результаты демонстрируют, что алгоритм "Кузнечик" обладает высокой криптостойкостью и может быть эффективно реализован программными средствами. Практическая работа позволила закрепить теоретические знания в области криптографии и приобрести ценный опыт разработки криптографических систем защиты информации.

Проведённая работа имеет практическую значимость для сферы информационной безопасности и может быть использована в дальнейших исследованиях и разработках в области криптографической защиты данных.

****Список литературы****

1. Зенков А. Информационная безопасность и защита информации. Учебное пособие для вузов. – Litres, 2021.
2. Тишина Н. А. Защита информационных процессов в компьютерных системах: учебное пособие для обучающихся по образовательным программам высшего образования по направлениям подготовки 09.03. 01 Информатика и вычислительная техника и 09.03. 04 Программная инженерия. – 2019.
3. Салий В. Н. Криптографические методы и средства защиты информации //Саратов–2017. – 2017.
4. ГОСТ 34.12-2018. Информационная технология. Криптографическая защита информации. Блочные шифры. — М.: Стандартинформ, 2018.
5. Елисеев В. Л. Нейросетевая обфускация вычислений над зашифрованными данными //Прикладная дискретная математика. Приложение. – 2020. – №. 13. – С. 85-93.
6. Михалевич И. Ф., Жеребятин И. А. ИМИТАЦИОННАЯ МОДЕЛЬ КРИПТОГРАФИЧЕСКОГО ПРОТОКОЛА ШИФРОВАНИЯ //DSPA. – С. 23.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Кривов В. С. |  | \_\_27\_ \_\_\_июля\_\_\_ 2025г. |
| *подпись обучающегося* |  | *расшифровка подписи* |  | *дата утверждения* |