Безопасность информационных экономических систем

Отчёт по лабораторной работе №2

Вариант 13

Работу выполнил: Захаров П. Ю.

Группа: 47/1

Задание: разработать консольное приложение для шифрования/дешифрования произвольных файлов с помощью алгоритма RSA.

Ход работы

В рамках текущей работы были созданы генератор ключей и программа для шифрования/дешифрования произвольных файлов с использованием алгоритма RSA. Обе консольные программы написаны на Python и имеют улучшенный командный интерфейс (CLI) с дополнительным функционалом. На изображениях приведены сокращённые названия скриптов (генератор ключей – keygen.py, программа шифрования/дешифрования – rsacrypt.py или ursacrypt.py). Также, в скрипте rsacrypt.py используется измененный алгоритм из-за сложностей и ограничений, которые возникли в процессе разработки. На базе rsacrypt.py реализована программа ursacrypt.py с несколькими алгоритмическими улучшениями.

Для использования алгоритма RSA потребуются два ключа ­– публичный, для шифрования и приватный, для дешифрования. Ниже приведена справочная информация консольного скрипта на Python для генерации ключей. Так как написание генератора ключей не было основной задачей, не буду вдаваться в подробности. Код генератора можно посмотреть в файле lab2\_keygen\_zaharov\_47\_1.py

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 1 – Описание keygen.py

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 2 – Работа keygen.py с параметром случайного зерна 123

В результате работы генератора были созданы публичный и приватный ключи и записаны в файлы public.key и private.key соответственно.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 3 – Cодержимое файлов public.key и private.key

Теперь, когда у нас есть ключи перейдём к работе скрипта rsacrypt.py. В процессе разработки возникла проблема сохранения достоверности информации. Дело в том, что расчёта *k* (длины блока *mi* в битах) используется целая часть от *log2n,* (*k =* [*log2n*]). Это позволяет выполняться условию, что число *mi* <= *n*. Однако, при вычислении *ci* = ((*mi*)*e*) mod *n*, из-за операции взятия модуля может возникнуть ситуация, что *ci* > *k* и для хранения *ci* потребуется дополнительный бит.

На рисунке 4 изображена такая ситуация. Подсчитанное *k* = 8, далее идёт таблица с выводом преобразований блоков: столбец № хранит порядковые номера блоков, bin – бинарное представление блока, int – его числовое представление, cr\_bin – бинарное представление преобразованного блока (зашифрованного или дешифрованного), а cr\_int – его числовое представление.

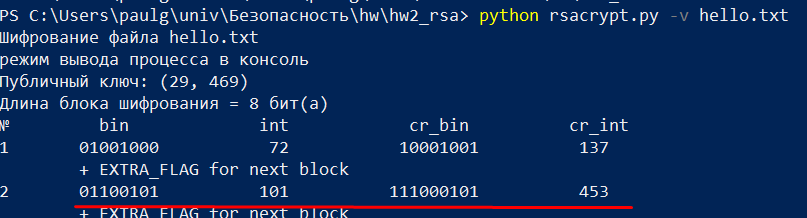


Рисунок 4 – Выход за длину блока

Как видно на рисунке, длина бинарного представления преобразованного второго блока равна 9, что больше 8 (*k)*. Его числовое значение всё ещё меньше 469 (*n*), но больше 255 (2k), поэтому потребовался дополнительный бит. Как же разрешить эту ситуацию?

Наивный алгоритм

Рассмотрим ситуацию, когда мы оставили всё как есть, т. е. воспользовались наивным алгоритмом. Пусть некий файл изначально состоит только из двух байт: 01001000 и 01100101, как на рисунке 4. Так как длина блока *k* = 8, эти же байты соответствуют двум блокам, подлежащим шифрованию. Тогда:

m1 = 72, m2 = 101

с1 = 7229 mod 469 = 137, c2 = 10129 mod 469 = 453

Бинарное представление c1 – 10001001, с2 – 111000101. Для записи зашифрованных данных в файл, необходимо склеить зашифрованные блоки (конкатенировать) и разбить по байтам. Склеим в 10001001111000101, затем разобьем по 8 (10001001 | 11100010 | 1). Получим три байта: 10001001, 11100010 и 00000001. Бинарное представление зашифрованного файла: 100010011110001000000001. При дешифровании также разобьём на блоки по 8 бит и преобразуем. Получим:

m’1 = 13941 mod 469, m’2 = 22641 mod 469, m’3 = 141 mod 469

m’1 = 72, m’2 = 158, m’3 = 1

Как видно из примера, m2 ≠ m’2. К тому же, добавился лишний блок, т.е. дешифрованные данные не соответствуют изначальным. Из этого следует, что наивный алгоритм неприменим для практического использования.

Алгоритм Extra Bit Crypt (EBC)

Первой идеей было увеличение k на 1, но это приводило к нарушению условия mi < n. Значит, нужно его динамически изменять.

Алгоритм EBC базируется на наивном, но со следующими изменениями:

1. При шифровании перед записью зашифрованного блока (cr\_bin), длина которого больше k, вставлять блок длины k, состоящий только из единиц (Extra Flag).
2. При дешифровании, во время чтения зашифрованных данных, разово увеличивать k длины следующего блока на единицу при считывании блока полностью из единиц (Extra Flag). Сам блок Extra Flag никак не обрабатывается и в дешифрованный файл не попадает.

Этот алгоритм реализован в консольной программе rsacrypt.py, описание приведено на рисунке 5. На рисунке 6 изображена работа программы. Неправильное кодирование последнего символа (последнего байта) вызвано смещением дополнительных битов или не кратности 8 длины бинарного представления после шифрования. Была попытка исправить это сложением двух последних байт, но на корню это проблему не решило. Алгоритм можно применять, если сохранностью последнего байта можно пренебречь (простой текст, неструктурированные данные).

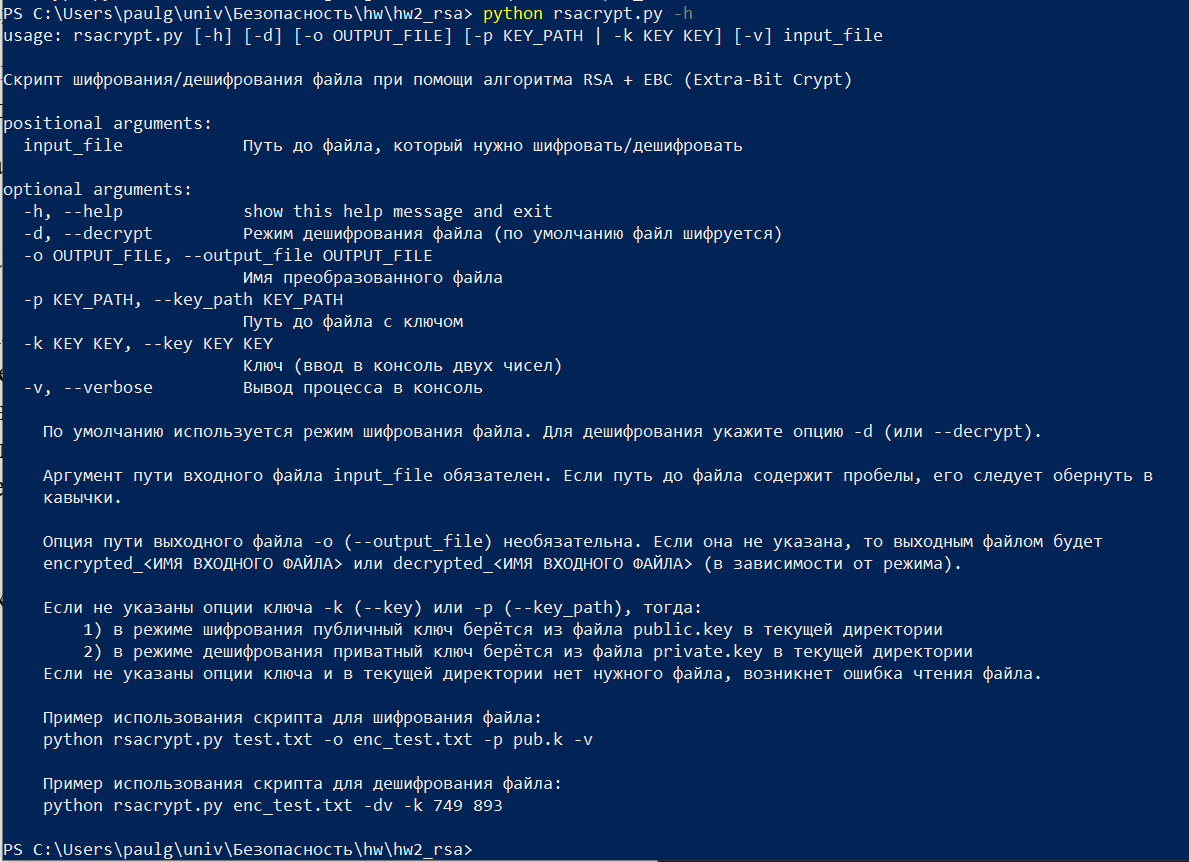


Рисунок 5 – Описание rsacrypt.py

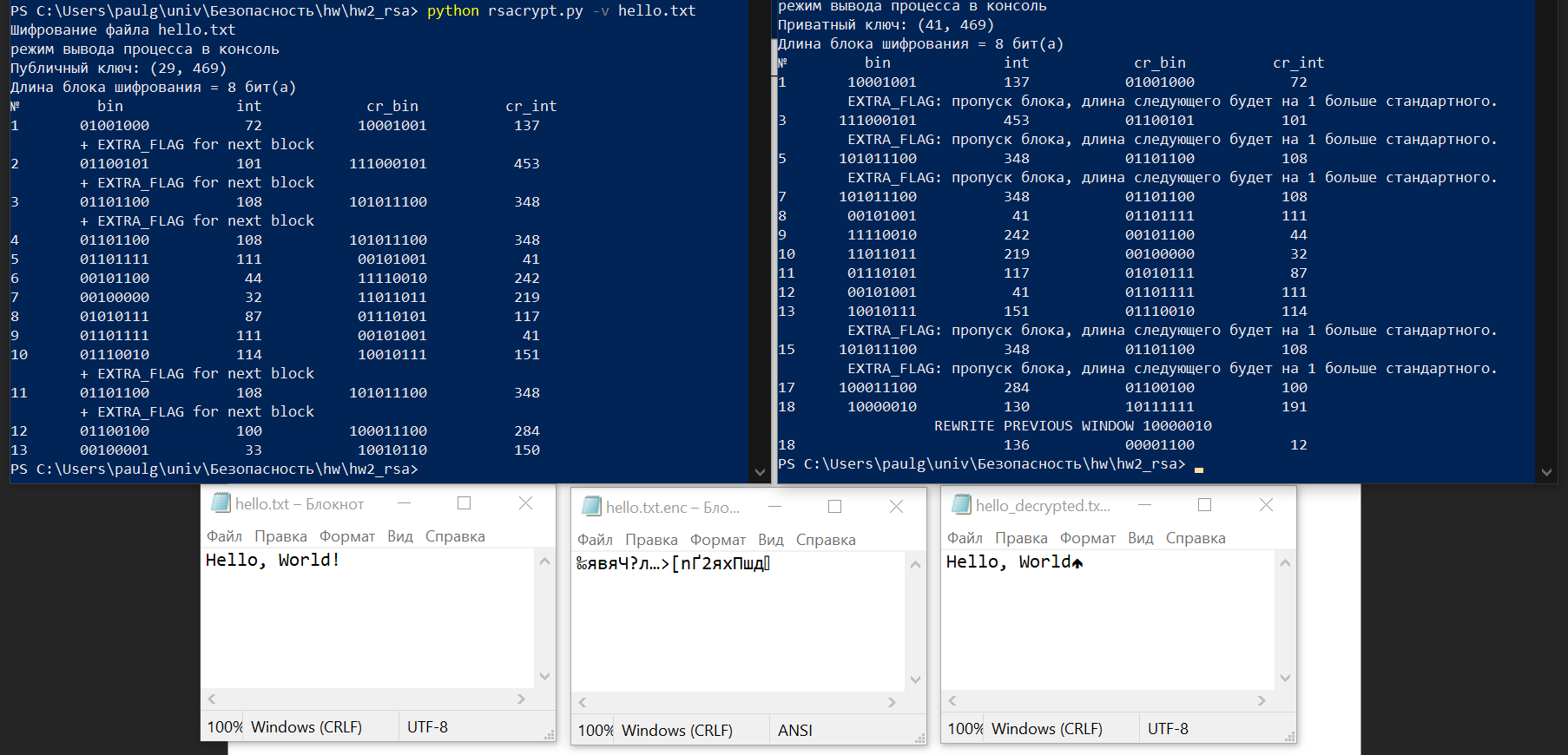


Рисунок 6 – Пример работы rsacrypt.py

Алгоритм Unified RSA (uRSA)

На основе алгоритма EBC возникла следующая идея:

1. На этапе шифрования считывать изначальные данные блоками по k, а зашифрованные данные записывать блоками длины h, кратной 8. Причём h не меньше k + 1.
2. На этапе шифрования последним байтом записывается длина бинарного представления числа последнего блока j.
3. На этапе дешифрования считывать зашифрованные блоки длины h, дешифрованные блоки будут иметь длину k, кроме последнего.
4. На этапе дешифрования считывается последний байт j, обозначающий длину последнего блока. Бинарное представление последнего дешифрованного блока будет иметь длину j.

Скрипт ursacrypt.py реализует этот алгоритм. На рисунке 7 приведено описание программы. Программа стабильно работает для всех типов файлов. На рисунках 8 и 9 изображены примеры шифрования и дешифрования текстового файла и изображения.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 7 – Описание ursacrypt.py

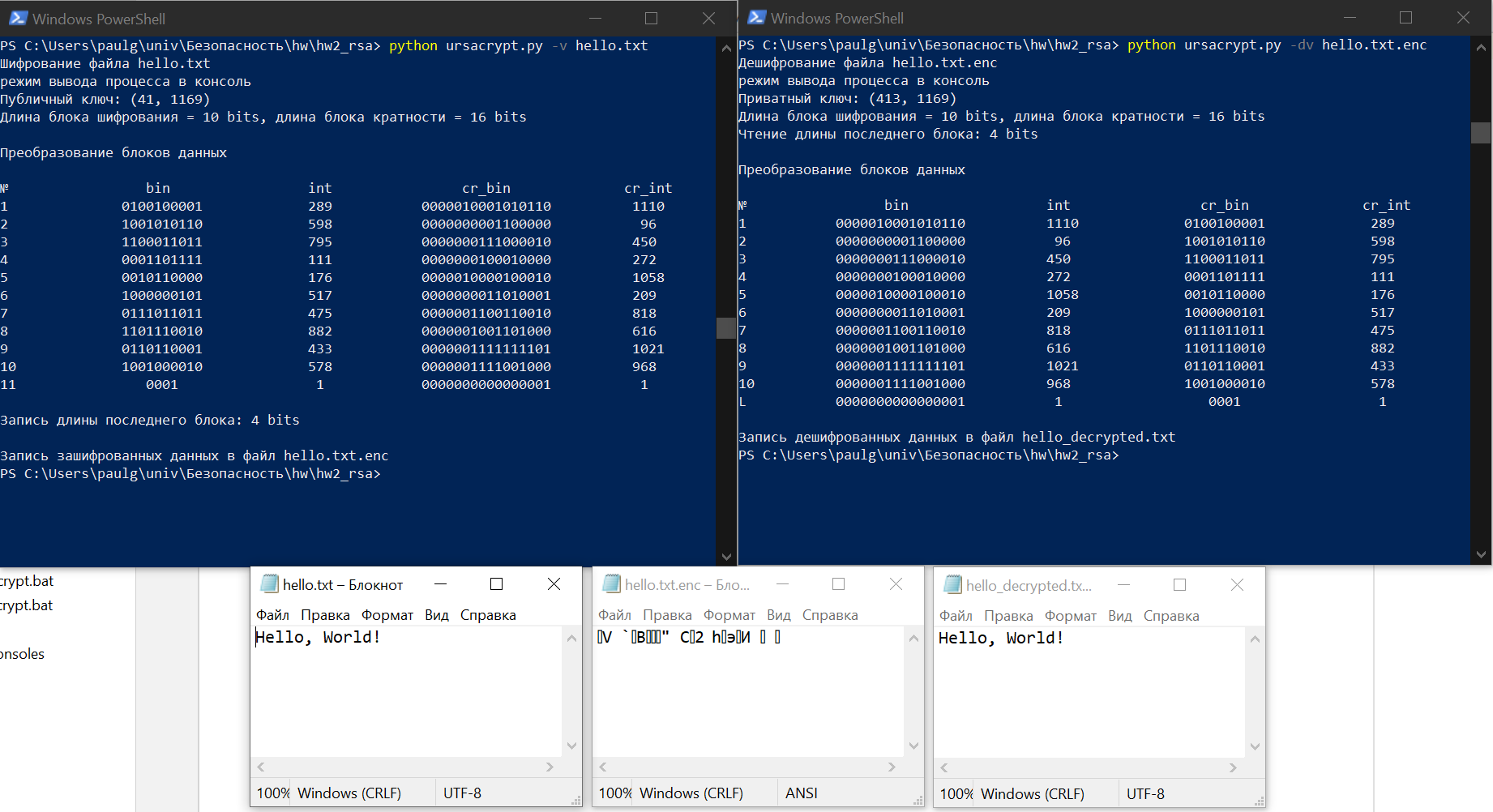


Рисунок 8 – Шифрование и дешифрование текстового файла

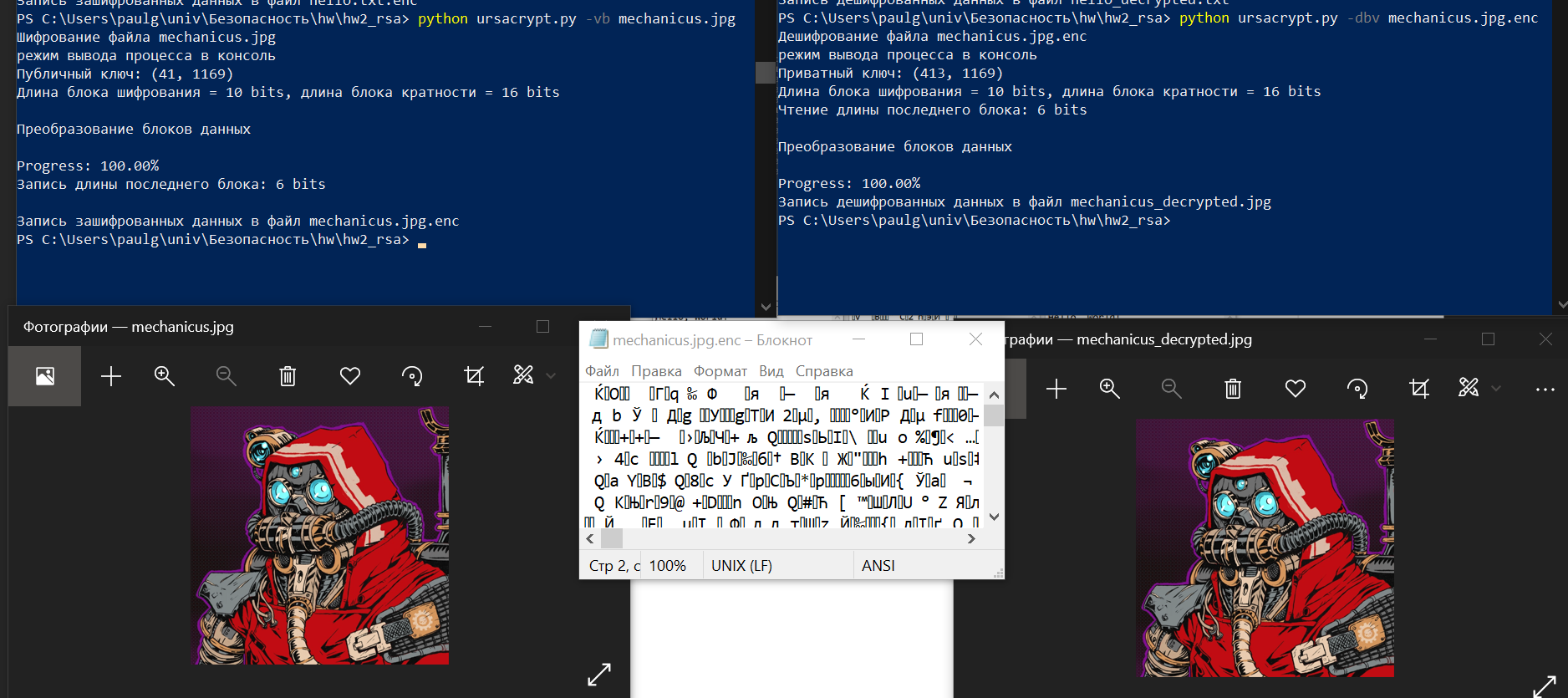


Рисунок 9 – Шифрование и дешифрование изображения