МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Факультет безопасности информационных технологий

Дисциплина:

«Криптографические методы обеспечения информационной безопасности»

Лабораторная работа 4

“ Основные структурные элементы блочного симметричного алгоритма RSA”

Выполнили:
студент гр. N34511

Бехит М. М. _  ___

Проверил:
Дата: _____
Оценка: _____

Таранов Сергей Владимирович_____

Санкт-Петербург
2024 г.

Цель: изучить основные принципы работы асимметричных криптосистем на примере алгоритма RSA.

Задачи практической работы:

1. Проанализировать эмуляцию алгоритма RSA и примитивных атак на шифр, используя Cryptool 2. Выделить основные необходимые настройки шифра и требуемые ограничения на параметры.
2. Программно реализовать и модифицировать любую асимметричную криптосистему, реализация алгоритма на псевдокоде или в виде блок схем, включающих основные этапы алгоритма с отображением формул и основных математических действий. *Если атака или модификация не применима для реализуемого алгоритма разрешается* найти любую альтернативу (атаки, применимой к алгоритму; модификации для ускорения алгоритма и дополнительной защиты).
3. Для созданной реализации криптосистемы предлагается провести примитивный криптоанализ на устойчивость к следующим атакам, а также сделать минимальные модификации по оптимизации (ускорению процессов шифрования, дешифрования, процесса генерации ключей).

Ход Работа

1. визуализации алгоритма RSA:
    a. Ключ
        - основные необходимые настройки шифра и требуемые ограничения на параметры:
            - p – простой число
            - q – простой число
            - e < φ(N), где N – p * q

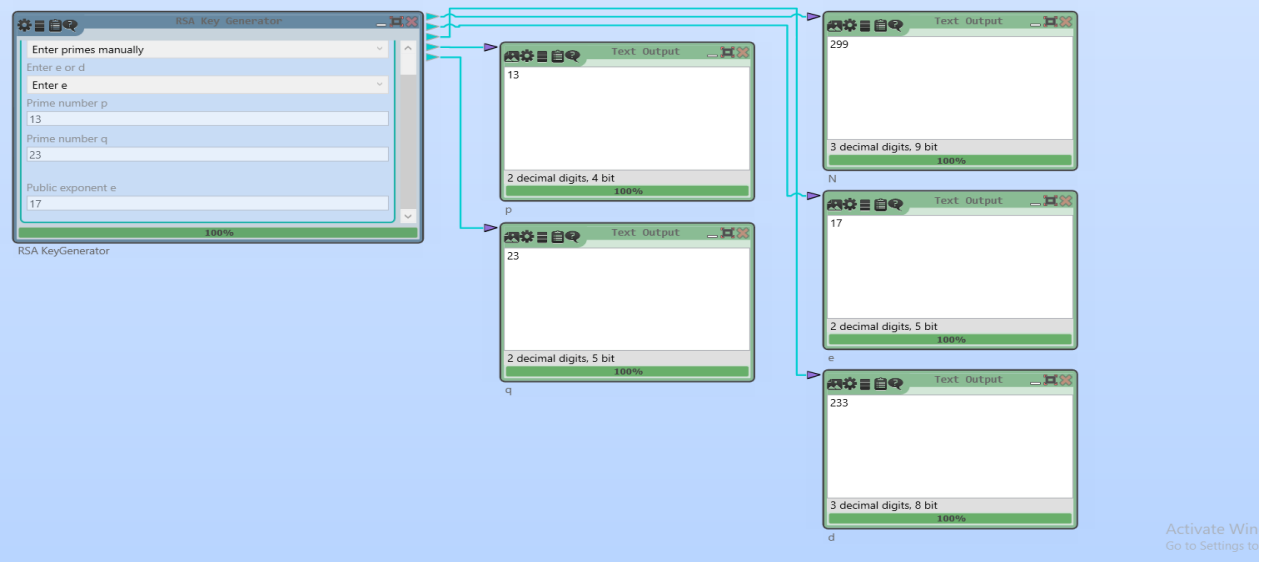

Рис1 - ганеревно ключ

    b. Проанализировано эмуляцию алгоритма RSA

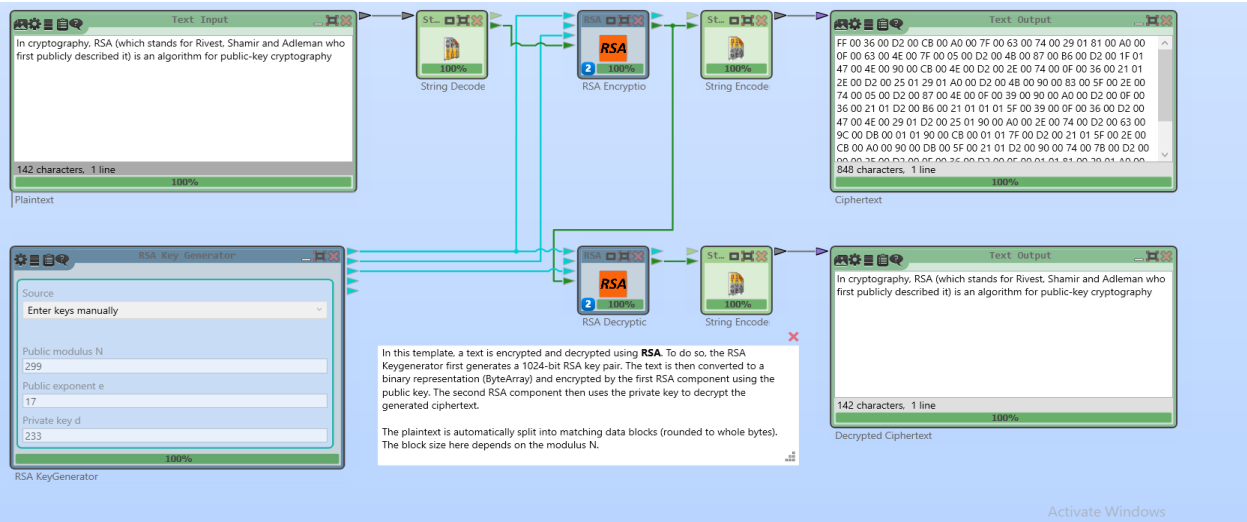

Рис2 - визуализации алгоритма

Оригинальный текст:

In cryptography, RSA (which stands for Rivest, Shamir and Adleman who first publicly described it) is an algorithm for public-key cryptography

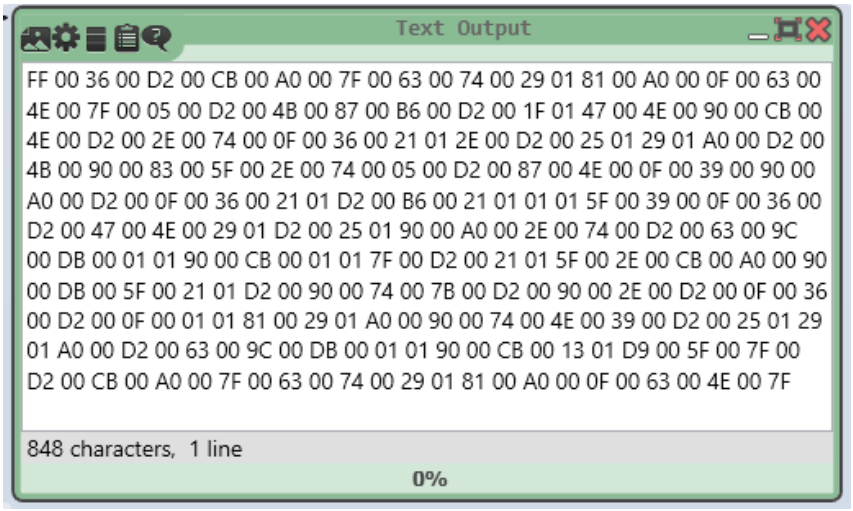

Рис3 – зашифровен текст

2. примитивных атак на шифр алгоритм RSA използуя Cryptool2:
    - Използуя:
      templet - **Factorization with Quadratic Sieve (QS)**
      Чтобы найти факторы **N** – **p** , **q**

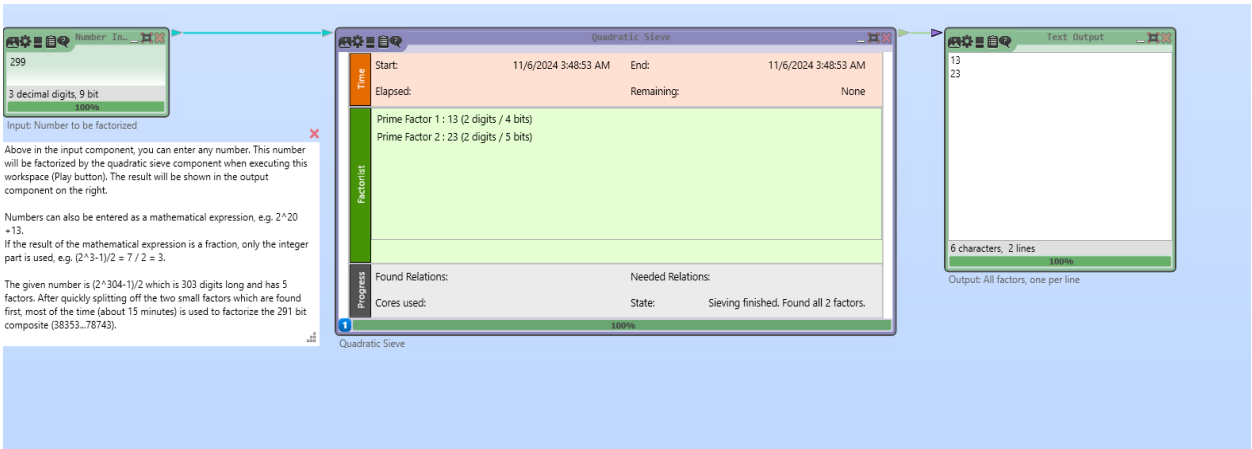

Рис4 – Factorization

Получаем факторов **p**, **q**

И **e** – откреte ключ (известно)

Получно праметрах: **p**, **q**, **e**

Используя: templet – **RSA decryption**

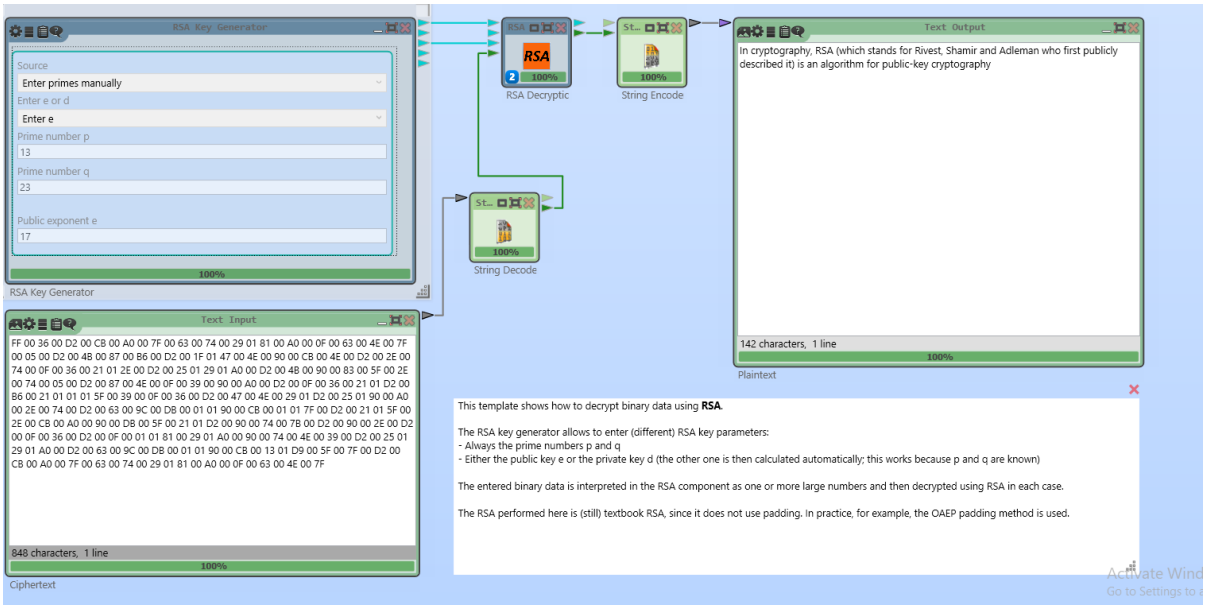

Рис 5 – RSA decryption

Получно текст: ***In cryptography, RSA (which stands for Rivest, Shamir and Adleman who first publicly described it) is an algorithm for public-key cryptography***

3. Программно реализовать и модифицировать RSA криптосистему

```
import random
import math
from sympy import isprime, gcd
from sympy.ntheory.generate import randprime
import hashlib

class RSA:
# Set to 1023 bits for primes to make N approximately 2046 bits

    def __init__(self, bits=1023):

        self.bits = bits
        self._generate_keys()  # Private method to generate keys

    def _generate_prime(self):
        """Generate a random prime number with specified bit length."""
        return randprime(2**(self.bits - 1), 2**self.bits)
```

```
    def _generate_keys(self):
        #Generate the public key components (e, N)
        # Step 1: Generate two distinct prime numbers, p and q
        p = self._generate_prime()
        q = self._generate_prime()
        while p == q:  # Ensure p and q are different
            q = self._generate_prime()

        # Step 2: Calculate N and Euler's totient (f_elar)
        self.N = p * q  # Public modulus with approximately 2046 bits
        self.f_elar = (p - 1) * (q - 1)

        # Step 3: Find a public exponent e
        self.e = random.randrange(2, self.f_elar)
        while gcd(self.e, self.f_elar) != 1 or not isprime(self.e):
            self.e = random.randrange(2, self.f_elar)

    def _oaep_pad(self, message, label=b"", hash_alg=hashlib.sha256):
        """Apply OAEP padding to the message."""
        k = (self.N.bit_length() + 7) // 8  # Length of the RSA modulus in
bytes
        h_len = hash_alg().digest_size  # Hash output length

        if len(message) > k - 2 * h_len - 2:
            raise ValueError("Message too long.")

        l_hash = hash_alg(label).digest()
        ps = b"\x00" * (k - len(message) - 2 * h_len - 2)
        db = l_hash + ps + b"\x01" + message
        seed = random.randbytes(h_len)
        db_mask = self._mgf1(seed, k - h_len - 1, hash_alg)
        masked_db = bytes(a ^ b for a, b in zip(db, db_mask))
        seed_mask = self._mgf1(masked_db, h_len, hash_alg)
        masked_seed = bytes(a ^ b for a, b in zip(seed, seed_mask))
        return b"\x00" + masked_seed + masked_db

    def _oaep_unpad(self, padded_message, label=b"",
hash_alg=hashlib.sha256):
        """Remove OAEP padding from the message."""
        k = (self.N.bit_length() + 7) // 8
        h_len = hash_alg().digest_size

        if len(padded_message) != k or padded_message[0] != 0:
            raise ValueError("Decryption error.")
```

```
        masked_seed = padded_message[1:h_len + 1]
        masked_db = padded_message[h_len + 1:]

        seed_mask = self._mgf1(masked_db, h_len, hash_alg)
        seed = bytes(a ^ b for a, b in zip(masked_seed, seed_mask))
        db_mask = self._mgf1(seed, k - h_len - 1, hash_alg)
        db = bytes(a ^ b for a, b in zip(masked_db, db_mask))

        l_hash = hash_alg(label).digest()
        if not db.startswith(l_hash):
            raise ValueError("Decryption error.")

        db = db[len(l_hash):]
        sep_index = db.find(b"\x01")
        if sep_index == -1:
            raise ValueError("Decryption error.")

        return db[sep_index + 1:]

    def _mgf1(self, seed, mask_len, hash_alg):
        """Mask Generation Function (MGF1) based on a hash function."""
        h_len = hash_alg().digest_size
        mask = b""
        for counter in range((mask_len + h_len - 1) // h_len):
            c = counter.to_bytes(4, byteorder="big")
            mask += hash_alg(seed + c).digest()
        return mask[:mask_len]

    def encrypt(self, message):
        """Encrypt the message with the public key (e, N)."""
        padded_message = self._oaep_pad(message.encode())
        message_int = int.from_bytes(padded_message, 'big')  # Convert padded
message to integer
        if message_int >= self.N:
            raise ValueError("Message is too long for encryption with the
current key size.")
        ciphertext = pow(message_int, self.e, self.N)
        return ciphertext

    def decrypt(self, ciphertext):
        """Decrypt the ciphertext using a private decryption key (d)
calculated locally."""
        d = pow(self.e, -1, self.f_elar)  # Calculate d as the modular
inverse of e mod f_elar
        decrypted_int = pow(ciphertext, d, self.N)
```

```
        padded_message = decrypted_int.to_bytes((decrypted_int.bit_length() +
7) // 8, 'big')
        return self._oaep_unpad(padded_message).decode('utf-8', 'ignore')

# Example usage
rsa = RSA(bits=1023)  # Initialize RSA with 1023-bit primes to create a 2046-
bit modulus N

# Display the public keys
print(f"Public key (e, N): ({rsa.e})")
print(f"N: {rsa.N}")
print(f"N bit length: {rsa.N.bit_length()} bits")  # Should be close to 2046
bits

# Get the message from the user
message = input("Enter the message to encrypt: ")
print(f"Original message: {message}")

# Encrypt the message
ciphertext = rsa.encrypt(message)
print(f"Encrypted message (as integer): {ciphertext}")

# Decrypt the message
decrypted_message = rsa.decrypt(ciphertext)
print(f"Decrypted message: {decrypted_message}")
```

Объяснение атаки на RSA

Атака на RSA в данном проекте основана на методе факторизации Ферма, который используется для разложения модуля N на простые множители “p” и “q”. Вот как это работает:

Основная идея:

Если N можно представить в виде разности квадратов:

$$N= a2−b2=(a+b)(a−b),$$

то множители “p” и “q” будут равны “a + b” и “a – b” соответственно.

Шаги атаки:

- Находим наименьшее целое “a”, такое что $a^2 > N$.
- Проверяем, является ли “ $b^2 = a^2 - N$” полным квадратом. Если да, то вычисляем “b” как квадратный корень из “b^2”.
- Находим множители:
  - $p=a+b$, $q=a-b$.
- Если $p \cdot q=N$, то атака успешна, и мы можем вычислить приватный ключ “d” как модульное обратное к “e” по модулю $\phi(N) = (p-1)(q-1)$.

Реализация в коде:

В проекте атака реализована в функции rsa_attack(N, e, ciphertext). Код ищет подходящие “a” и “b”, проверяет, является ли “b^2” полным квадратом, и затем вычисляет множители “p” и “q”.

После нахождения множителей вычисляется d и дешифруется сообщение.

Пример использования:

В коде демонстрируется, как атака применяется к зашифрованному сообщению. Если факторизация успешна, сообщение может быть дешифровано без знания оригинального приватного ключа.

Заключение

В ходе выполнения лабораторной работы были детально изучены принципы функционирования асимметричной криптосистемы RSA. Особое внимание было уделено ключевым этапам алгоритма: генерации ключей, процессам шифрования и дешифрования, а также способам визуализации операций с использованием программного инструмента Cryptool 2. На основе практической реализации алгоритма проведён анализ устойчивости RSA к атаке, основанной на методе факторизации Ферма, что позволило наглядно продемонстрировать уязвимости, возникающие при некорректном выборе параметров, в частности простых чисел ppp и qqq.

Также была реализована программная модель RSA с применением padding-схемы OAEP, обеспечивающей дополнительную криптографическую стойкость. Проект подтвердил, что безопасность алгоритма напрямую зависит от размера и случайности выбираемых простых чисел, а также от соблюдения требований к параметрам.