

Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого

Институт Кибербезопасности и Защиты Информации

ЛАБОРАТОРНАЯ РАБОТА № 2

«Атаки на криптосистему RSA, использующие данные о показателях»

по дисциплине «Криптографические методы защиты информации»

Выполнил
студент гр. 4851003/80802

<подпись>

Сошнев М.Д.

Проверил
преподаватель

<подпись>

Ярмак А.В.

Санкт-Петербург

2022

Цель работы

Изучение уязвимостей криптосистемы RSA по отношению к атакам, использующим показатели, обладающие определенными свойствами, генерация параметров криптосистемы RSA.

Задача

- Получить у преподавателя вариант задания.
- Разработать программу **П-1**, которая реализует разложение составного числа на множители по известным показателям RSA и вычисляет закрытый ключ другого пользователя в случае общего модуля n . Программа **П-1** должна принимать на вход открытый и закрытый ключ RSA, а также открытый ключ RSA другого пользователя; на выходе – возвращать множители модуля RSA, закрытый ключ другого пользователя.
- Разработать программу **П-2**, которая реализует атаку Винера на криптосистему RSA. Программа **П-2** должна принимать на вход открытый ключ RSA: число n и показатель e ; на выходе – возвращать закрытый показатель d .
- Разработать программу **П-3**, которая, в соответствии с вариантом задания, реализует одну из следующих атак:
- бесключевое дешифрование широковещательного сообщения в случае малого общего показателя e . Для восстановления числа m^e по китайской теореме об остатках и нахождения корня степени e допускается использовать готовые процедуры. Тело сообщения – файл, зашифрованный алгоритмом AES-256 в режиме CBC. При длине исходного файла, не кратной 16 байтам, открытый текст дополняется символами с кодом 0x03 так, чтобы длина сообщения (файла) перед зашифрованием была кратна 16 байтам. Изначальная длина доступна в элементе данных «Длина сообщения» заголовка. Удаление выравнивания не обязательно. Программа **П-3** должна принимать на вход открытые ключи нескольких пользователей, а также шифртексты широковещательного сообщения; на выходе – возвращать восстановленный открытый текст;

- бесключевое дешифрование сообщения в случае малого порядка элемента e в группе $(\mathbb{Z}/\varphi(n)\mathbb{Z})^\times$.. Программа **П-3** должна принимать на вход открытый ключ RSA, шифртекст; на выходе – возвращать восстановленный открытый текст.

Разработать программу **П-4**, которая осуществляет генерацию параметров криптосистемы RSA. Разработка процедур генерации случайных чисел и проверки числа на простоту не требуется. Генерация ключей должна обеспечивать выработку безопасных параметров криптосистемы с учетом произведенных атак. Программа **П-4** должна принимать на вход битовую длину модуля RSA; на выходе – возвращать параметры криптосистемы: n, e, d . Выполнить анализ сгенерированных параметров с помощью инструмента CrypTool. Описание утилиты представлено в *Приложении Б*.

Ход работы

Была разработана программа, реализующая 3 атаки на систему RSA и генерация оптимальных параметров для RSA. Использовался язык python и библиотека rsa.

Атака по известным показателям

В начале программа генерирует две пары ключей (закрытый-открытый) при общем модуле n . Далее программа пытается взломать один из закрытых ключей по одному известному закрытому ключу. Поиск данного ключа сводится к нахождению корня из 1 в кольце $\mathbb{Z}/n\mathbb{Z}$. Продемонстрируем:

```
(maxim@maxim)-[~/.../КМЗМ/Work/RSA/attacker]
$ python main.py --attack 1
p = 265010569022868596855981254879662151673488681563516243704069
9858375245906267067552920154929733434037523063660789444967114149
q = 971793658123705322457297840044298555260902212881756668240540
321764484751215584605093056992150647693626600617066625959
n = 257535590312178177287567268799762886222436768829864772511258
329793308664887313070561808109328961882669804972734043336996382
973710179318118684399358279609448262736399023330912115541603879

Первый ключ:
e = 65537
d = 986845793904165672249672218955467195889963619882569240593133
0695941774114340133269784529425482124234381818332041488369232349
4392116555589291555958613277649703310562205025615679936823795994

Второй ключ:
e = 16777217
d = 469861213674194624042482719004652448151414861785627103234526
0710071842935238926089888393299218962619518033156220327771068273
4365651755337470029221717480922906235627562621645233382736642019

[Attack] m = 646749127950973056622267682136844536170405457562439
2919911359270099360501315093141018687049578219759486812280270030
9402536649843561427036554007028596383773286058643152307637748160
[Attack] s = 808436409938716320777834602671055670213006821953049
6149889199087624200626643866426273358811972774699358515350337537
1753170812304451783795692508785745479716607573303940384547185209
[Attack] a = 336243929947082637240778668397368822357534879074659
0279123491119164825685261712732654762659880379593398172303225114
6853983648830949151018658953198781843765456953900010937695367699
[Attack] wait ...
[Attack] a = 200781413994591764628316499713753435721413230902044
5564123431043258798886464722044224551869430006264462943862037453
7843849755216440255484459373799887517402930224586651355302016119
[Attack] wait ...
[Attack] p = 265010569022868596855981254879662151673488681563516
2800321929858375245906267067552920154929733434037523063660789444
[Attack] q = 971793658123705322457297840044298555260902212881756
6485688183217644847512155846050930569921506476936266006170666259
[Attack] f(n) = 257535590312178177287567268799762886222436768829
3941353048763297933086648873130705618081093289618826698049727340
7040338683809120201261114384584489545806464096706344583406059709
[Attack] d = 469861213674194624042482719004652448151414861785627
7627045490710071842935238926089888393299218962619518033156220327
0370502284365651755337470029221717480922906235627562621645233382
Атака прошла успешно

(maxim@maxim)-[~/.../КМЗМ/Work/RSA/attacker]
$
```

Рисунок 1 — атака по известным показателям

Стоит отметить что атака прошла успешно и ключ был скомпрометирован даже при большом размере ключа (2048 бит)

Атака Винера

Данная атака пытается скомпрометировать закрытый ключ по известному открытому. При этом она имеет место быть только при малых показателях d . Продемонстрируем атаку для следующего ключа:


$p = 863$

$q = 919$

$n = 793097$

$e = 678271$

$d = 7$



```
(maxim@maxim)-[~/.../КМЗИ/Work/RSA/attacker]
$ python main.py --attack 2
Ключ:
p = 863
q = 919
n = 793097
e = 678271
d = 7
[Attack] m = 100
[Attack] m^e (mod n) = 710412
[Attack] e/n = [0, 1, 5, 1, 9, 1, 2, 1, 16, 1, 15, 1, ...]
[Attack] a = 0
[Attack] a = 1
[Attack] a = 5
[Attack] a = 1
[Attack] d = 7
Атака прошла успешно

(maxim@maxim)-[~/.../КМЗИ/Work/RSA/attacker]
$ |
```

Рисунок 2 — атака Винера

Атака «бесключевое дешифрование»

Данная атака предполагает, что у параметра e малый порядок в кольце $\mathbb{Z}/n\mathbb{Z}$. Нарушитель пытается скомпрометировать исходное сообщение по шифртексту и открытому ключу. Программа написана таким образом, что нарушитель делает определенное число итераций и если за это время не удалось посчитать исходное сообщение, то считается, что атака закончилась неудачей. Продемонстрируем атаку для следующего ключа:

$$p = 863$$

$$q = 919$$

$$n = 793097$$

$$e = 678271$$

$$d = 7$$

и исходного сообщения $m = 739764$

```
(maxim@maxim)-[~/.../КМЭИ/Work/RSA/attacker]
$ python main.py --attack 3
Ключ:
p = 863
q = 919
n = 793097
e = 678271
d = 7

Исходное сообщение: 739764

Шифртекст: 421015
[Attack] i=0 c=421015
[Attack] i=1 c=738999
[Attack] i=2 c=478158
[Attack] i=3 c=769336
[Attack] i=4 c=617178
[Attack] i=5 c=519421
[Attack] i=6 c=745294
[Attack] i=7 c=517142
[Attack] i=8 c=518273
[Attack] i=9 c=343249
[Attack] i=10 c=307948
[Attack] i=11 c=493394
[Attack] i=12 c=412337
[Attack] i=13 c=538906
[Attack] i=14 c=349400
[Attack] i=15 c=509712
[Attack] i=30942 c=652851
[Attack] i=30943 c=204028
[Attack] i=30944 c=566109
[Attack] i=30945 c=106527
[Attack] i=30946 c=316491
[Attack] i=30947 c=131675
[Attack] i=30948 c=514660
[Attack] i=30949 c=620265
[Attack] i=30950 c=155351
[Attack] i=30951 c=37480
[Attack] i=30952 c=233262
[Attack] i=30953 c=577738
[Attack] i=30954 c=221569
[Attack] i=30955 c=263551
[Attack] i=30956 c=261780
[Attack] i=30957 c=499210
[Attack] i=30958 c=780084
[Attack] i=30959 c=739764
Атака прошла успешно
```

Рисунок 3, 4 — атака «бесключевое дешифрование»

Генерация параметров для RSA

Программа работает следующим образом:

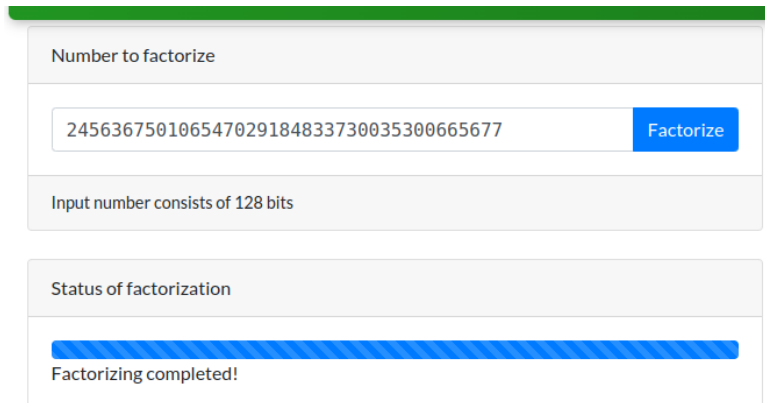
1. В зависимости от размерности, необходимой пользователю определяет промежуток для p и q
2. Рандомно генерирует p , пока оно не будет простым
3. Рандомно генерирует q , пока оно не будет простым
4. Считает $n=pq$ и $f(n)=(p-1)(q-1)$
5. Рандомно генерирует e , пока оно не будет взаимно-простым с $f(n)$
6. Считает d , как $e^{-1}(\text{mod } f(n))$

Пример работы программы для размерности 64 бит:

```
(maxim@maxim) - [~/.../КМЗИ/Work/RSA/attacker]
$ python main.py --generate 64
p = 13699888233400882417
q = 17929836063017994781
n = 245636750106547029184833730035300665677
e = 151076153224452811607777079388964457427
d = 45610700877368240233102131293488113883
```

Рисунок 5 — генерация ключа размерность 64 бит

Проанализируем данный ключ с помощью CryptTools:



The screenshot shows the CryptTools web interface. At the top, there is a green header. Below it, a section titled "Number to factorize" contains a text input field with the value "245636750106547029184833730035300665677" and a blue "Factorize" button. Below the input field, a message states "Input number consists of 128 bits". Further down, a section titled "Status of factorization" features a blue progress bar that is nearly full and the text "Factorizing completed!" below it.

Рисунок 6 — данный ключ легко раскладывается на множители

Тогда покажем, что ключ размерностью 2048 бит уже будет стойким:

```
main@main: ~/./KMS/keys/rsa/attacker:
$ python main.py --generate 1024
p = 1786730188046434097449670722280230992847409727917257672586222291519801648445401715959212705878522048290636787857323493682552198080828886482398256143782054547753998815744620738230336294486628475404728691459822569120491799318864
1144899237876951727808977162303538038652191463334856288159215787528248429488883
q = 1101586934514791323290181290839858658678428875381964672732258386448544694897880462899076177865174675945182011348719637299726768283141612986826392019849835565862626280676846106829873940571362187425982293712558725181
42378118284579628682914497181234798192183329114519333322704769897426447131971
n = 19682380124545127420778354446087287818784697987427635881589837878613111469350699377112795968082674641744822113596162765198188616711365682221975443165322898639858826159946631625998783880199238808917333946176772158821278842191
14164298885282242818460568051389805545408167365818916311925841882763258718483151250576620116618268661408487388514803189425334245988746445554241512563274177456256816643399149180958946464176585587577614112186866953242355389536141482
74284920879258329768338959576839217252498163268105736868941314459452019406576427182228732532896880839874208598782651986835863568757488721451137480515121593
e = 1049458846879372996874297227765665672394178456524471553961296111558477362857848745482992138245124445538945685998466426195668897125264923454918309998881315848639966778830780109738293074248825318726748638198720994665298936167
5673779256651920455966473158626977685848528559976878738056179784880718598536659388266367181249228784394778418154921981583619613166518982633355476585722088373711595613924964881814437162815599667888429939818794122818356955862299388
7354127782589906749705152150973344974135154546408139426451084082959767621828838209774187868271038139748599533964533294284828885433991613529360512748297
d = 7867958617670827774248459048328237282862787254711399961914409171503854188889383751348394344257231893861639132788632185767438871858332611626459911674916775829831328721460615711663235963689540488887158337034683972186667696487
28799167633191964685658969636719925988338288813866299072233964538510676478956248422019573031880167314839804148333912664115717609680719315358854837549838721182621356458581783779302554207932548584538827755824099472636107948611016
82141515730286128399853877854366884879841349646743726928801615940122488170141972461889527963264128323384528762555228393832815861481508964681633217815989923
```

Рисунок 7 - генерация ключа размерность 2048 бит

Number to factorize

5882572956753836573302923712425577770757434209120:

Factorize

Input number consists of 1023 bits

Status of factorization

An error occured: "Program terminated with exit(-1)"

Рисунок 8 — CryptTools заканчивает работу с ошибкой

Контрольные вопросы

1. При каком условии возможна атака Винера?

$q < p < 2 * q, d < \frac{1}{3}(n)^{(1/4)}$ (достаточно малый показатель d)

2. Перечислите методы противодействия атаке Винера.

Использовать $e' = e + t * \varphi(n)$

Использовать $d_p = d \pmod{p-1}, d_q = d \pmod{q-1}$

3. Почему числа p и q необходимо выбирать так, чтобы $p-1, q-1$ имели большие простые делители?

В противном случае у параметра e будет маленький порядок в Z/nZ и возможна атака «Безключевое дешифрование»

4. Предложите способы защиты от атаки бесключевого дешифрования широковещательных сообщений без отказа от использования общих малых закрытых показателей.

Перед шифрованием применять некое преобразование над сообщением, например $m' = m \oplus n_i$, тогда после преобразования сообщения m будут уникальны для каждого пользователя и применения К.Т.О. будет невозможным.

Вывод

В результате выполнения лабораторной работы были исследованы некоторые атаки на криптосистему RSA. Можно сделать следующие выводы: для каждого абонента необходимо генерировать не только уникальные параметры e , d но и уникальные параметры n , причем данный параметр должен быть произведением уникальных p и q , таких, что $p-1$ и $q-1$ не является сильно составным числом. В идеале: $p-1=2p_1$, $q-1=2q_1$. Также необходимо, чтобы параметр d был достаточно большим (больше, чем $\frac{1}{3}(n)^{(1/4)}$) и чтобы у параметра e был большой порядок в кольце Z/nZ . Рекомендуется брать e такой, что его двоичное представление имеет вид $10...01$. Также, размер ключа должен составлять как минимум 1024 бит.

Приложение

```
import random
import math
import rsa
from rsa.common import inverse

class AttackCommonModule:
    def __init__(self, n, e, d, logs=True):
        self.__e_b = e
        self.__d_b = d
        self.__n = n
        self.__logs = logs

    def __call__(self, e):
        """
        :return: private_key
        """

        m = self.__e_b * self.__d_b - 1
        if self.__logs:
            print('[Attack] m = {}'.format(m))

        s = m
        while (s & 1) == 0:
            s >>= 1

        print('[Attack] s = {}'.format(s))

        last_b = self.__n - 1
        while last_b == self.__n - 1:
            a = random.randint(0, self.__n)
            if self.__logs:
                print('[Attack] a = {}'.format(a))

            b = pow(a, s, self.__n)

            if self.__logs:
                print('[Attack] wait...')

            while b != 1:
                # print('[Attack] b = {}'.format(b))
                last_b = b
                b = pow(b, 2, self.__n)

        t = last_b
        p, q = math.gcd(t + 1, self.__n), math.gcd(t - 1, self.__n)
        f_n = (p - 1) * (q - 1)
        d = inverse(e, f_n)

        if self.__logs:
            print('[Attack] p = {}'.format(p))
            print('[Attack] q = {}'.format(q))
```

```

        print('[Attack] f(n) = {}'.format(f_n))
        print('[Attack] d = {}'.format(d))

    return p, q, d

class AttackViner:
    def __init__(self, logs=True):
        self.__logs = logs

    def __call__(self, public_key):
        """
        :return: private_key.d
        """

        m = 100
        if self.__logs:
            print('[Attack] m = {}'.format(m))

        c = pow(m, public_key.e, public_key.n)
        if self.__logs:
            print('[Attack] m^e (mod n) = {}'.format(c))

        frac = self.__to_continued_fraction(public_key.e, public_key.n)
        if self.__logs:
            print('[Attack] e/n = {}'.format(frac))

        Q_, Q__, P_, P__ = 0, 1, 1, 0
        for a in frac:
            if self.__logs:
                print('[Attack] a = {}'.format(a))

            P = a * P_ + P__
            Q = a * Q_ + Q__
            P__, Q__ = P_, Q_
            P_, Q_ = P, Q

            if pow(c, Q, public_key.n) == m:
                return Q

        return None

    @staticmethod
    def __to_continued_fraction(a, b):
        a, q = divmod(a, b)
        t, res = b, [a]

        while q != 0:
            next_t = q
            a, q = divmod(t, q)
            t = next_t
            res.append(a)

        return res

```

```

class AttackKeyLessDecrypt:
    def __init__(self, iter=1_000_000, logs=True):
        self.__iter = iter
        self.__logs = logs

    def __call__(self, public_key, cipher_text):
        """
        :return: open_text
        """

        c = cipher_text
        for i in range(self.__iter):
            if self.__logs:
                print('[Attack] i={} c={}'.format(i, c))

            c_pred = c
            c = pow(c, public_key.e, public_key.n)

            if (c - cipher_text) % public_key.n == 0:
                return c_pred

        return None

import random
import rsa.common
import rsa.prime
import argparse
from attacks import *

def generate_keys_with_common_n(bits):
    # Первый ключ
    _, private_key_1 = rsa.newkeys(bits)

    # Второй ключ на тех же p, q, n но новые e, d
    e = 0b10000000000000000000000000000001
    d = rsa.common.inverse(e, (private_key_1.p - 1) * (private_key_1.q - 1))
    private_key_2 = rsa.PrivateKey(private_key_1.n, e, d, private_key_1.p,
private_key_1.q)

    print('p = {}'.format(private_key_1.p))
    print('q = {}'.format(private_key_1.q))
    print('n = {}'.format(private_key_1.n))

    print('\nПервый ключ:')
    print('e = {}'.format(private_key_1.e))
    print('d = {}'.format(private_key_1.d))

    print('\nВторой ключ:')
    print('e = {}'.format(private_key_2.e))
    print('d = {}'.format(private_key_2.d))

    return private_key_1, private_key_2

```

```

def try_attack_1():
    private_key_1, private_key_2 = generate_keys_with_common_n(1024)

    attack = AttackCommonModule(private_key_1.n, private_key_1.e,
private_key_1.d)
    p_attacked, q_attacked, d_attacked = attack(private_key_2.e)

    if private_key_2.p == p_attacked and private_key_2.q == q_attacked and \
        private_key_2.d == d_attacked:
        print('Атака прошла успешно')
    else:
        print('Атака прошла безуспешно')

def try_attack_2():
    # public_key, private_key = rsa.newkeys(128)
    public_key = rsa.PublicKey(793097, 678271)
    private_key = rsa.PrivateKey(793097, 678271, 7, 863, 919)

    print('Ключ:')
    print('p = {}'.format(private_key.p))
    print('q = {}'.format(private_key.q))
    print('n = {}'.format(private_key.n))
    print('e = {}'.format(private_key.e))
    print('d = {}'.format(private_key.d))

    attack = AttackViner()
    private_key_attacked = attack(public_key)

    if private_key.d == private_key_attacked:
        print('Атака прошла успешно')
    else:
        print('Атака прошла безуспешно')

def try_attack_3():
    # public_key, private_key = rsa.newkeys(64, exponent=50)
    public_key = rsa.PublicKey(793097, 678271)
    private_key = rsa.PrivateKey(793097, 678271, 7, 863, 919)

    print('Ключ:')
    print('p = {}'.format(private_key.p))
    print('q = {}'.format(private_key.q))
    print('n = {}'.format(private_key.n))
    print('e = {}'.format(private_key.e))
    print('d = {}'.format(private_key.d))

    m = random.randint(0, public_key.n)
    print('Исходное сообщение: {}'.format(m))

    c = pow(m, public_key.e, public_key.n)
    print('Шифртекст: {}'.format(c))

    attack = AttackKeyLessDecrypt(logs=True)

```

```

m_attacked = attack(public_key, c)

if m_attacked == m:
    print('Атака прошла успешно')
else:
    print('Атака прошла безуспешно {} {}'.format(m_attacked, m))

def generate(bits):
    left = 1 << (bits - 1)
    right = (1 << bits) - 1

    # generate p
    while True:
        p = random.randint(left, right)
        if rsa.prime.is_prime(p):
            break

    # generate q
    while True:
        q = random.randint(left, right)
        if rsa.prime.is_prime(q) and p != q:
            break

    n = p * q
    f_n = (p - 1) * (q - 1)

    # generate e
    while True:
        e = random.randint(1, f_n - 1)
        if math.gcd(e, f_n) == 1:
            break

    d = rsa.common.inverse(e, f_n)

    print('p = {}'.format(p))
    print('q = {}'.format(q))
    print('n = {}'.format(n))
    print('e = {}'.format(e))
    print('d = {}'.format(d))

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--attack', required=False)
    parser.add_argument('--generate', required=False)
    args = parser.parse_args()

    if args.attack == '1':
        try_attack_1()
    elif args.attack == '2':
        try_attack_2()
    elif args.attack == '3':
        try_attack_3()

```



```
if args.generate:  
    generate(int(args.generate))
```