Indian Institute of Technology Kanpur

SURGE-2022

PROJECT REPORT

"Cryptanalysis of RSA"

Submitted by:

Shreyansh Sinha

SURGE Roll No: 2230134

Department of Mathematics and Scientific Computing
Indian Institute of Technology Kanpur

Under the guidance of:

Dr. Urbi Chatterjee

Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur
Kanpur, Uttar Pradesh

Certificate

Acknowledgements

I would like to express my gratitude to IIT Kanpur, and my Project Supervisor and Mentor Dr. Urbi Chatterjee for selecting me for the prestigious internship program SURGE 2022 and allowing me to work under her guidance. I am deeply indebted to my professor for emphasizing me and encouraging me to work on Rivest–Shamir–Adleman encryption scheme, which opened new doors of opportunity for me and helped me explore a vast area I had left untouched. I am thankful to her for guiding and mentoring me in my project, whensoever I was stuck, and for giving me a chance to explore my calibre.

I am extremely thankful to the Department of Computer Science Engineering, IIT Kanpur for giving me the opportunity, that helped me in developing an interest in research and further studies.

This internship would not have been possible without the constant support and inspiration of my parents, who had faith in me, much more than I could have. They guided and motivated me throughout, helping me go through difficult days and inspiring me to work harder.

Shreyansh Sinha

Abstract

Rivest Shamir Adleman (known as RSA) algorithm is a well-known asymmetric public key cryptography used in several security critical application for more than 25 years. In this project, we have explored the mathematical formation of the same and how we can cryptanalyze the algorithm to get the plaintext. Most of the work in the project in based on Glenn Durfee's PhD Thesis[6]. The target of the project is three-fold:

1. First we develop the Wiener's attack[1], where using continued fractions, we can perform an attack on RSA with short secret exponent to generate the private key. Through this we came up with a lower bound on the size of the secret exponent in comparison to the modulus.

2. Secondly, using the Chinese Reminder Theorem, we develop the Håstad Broadcast Attack[2] on RSA with a small public exponent, demonstrating that if enough communications are intercepted, the cypher-text can be decrypted. Through this, we are able to identify the necessary size of the public exponent.

3. Lastly, we develop the Coppersmith Short-Padding Attack[3], where we employ the Coppersmith theory and implement it in polynomial time using the Lenstra-Lenstra-Lovász lattice basis reduction[4] algorithm to retrieve the private key of RSA system having short random padding.

Contents

Chapter 1

Introduction

1. Cryptography:

Cryptography is the study and practice of techniques for secure communication in the presence of third parties called adversaries. It deals with developing and analysing protocols which prevents malicious third parties from retrieving information being shared between two entities thereby following the various aspects of information security.

Secure Communication refers to the scenario where the message or data shared between two parties can't be accessed by an adversary. In Cryptography, an Adversary is a malicious entity, which aims to retrieve precious information or data thereby undermining the principles of information security.

1.1 Principles:

Confidentiality refers to certain rules and guidelines usually executed under confidentiality agreements which ensure that the information is restricted to certain people or places.

Data integrity refers to maintaining and making sure that the data stays accurate and consistent over its entire life cycle.

Authentication is the process of making sure that the piece of data being claimed by the user belongs to it.

Non-repudiation refers to ability to make sure that a person or a party associated with a contract or a communication cannot deny the authenticity of their signature over their document or the sending of a message.

1.2 Types of Cryptography:

I] **Secret Key Cryptography**, or symmetric cryptography, uses a single key to encrypt data. Both encryption and decryption in symmetric cryptography use

the same key, making this the easiest form of cryptography. The cryptographic algorithm utilizes the key in a cipher to encrypt the data, and when the data must be accessed again, a person entrusted with the secret key can decrypt the data.

Examples:

- AES
- DES
- Caesar Cipher

II] **Public Key Cryptography,** or asymmetric cryptography, uses two keys to encrypt data. One is used for encryption, while the other key can decrypt the message. Unlike symmetric cryptography, if one key is used to encrypt, that same key cannot decrypt the message, rather the other key shall be used. One key is kept private, and is called the "private key", while the other is shared publicly and can be used by anyone, hence it is known as the "public key"

Examples:

- ECC
- Diffie-Hellman
- RSA

III] **Hash functions** are irreversible, one-way functions which protect the data, at the cost of not being able to recover the original message. Hashing is a way to transform a given string into a fixed length string. A good hashing algorithm will produce unique outputs for each input given. The only way to crack a hash is by trying every input possible, until you get the exact same hash. A hash can be used for hashing data (such as passwords) and in certificates.

2. RSA:

RSA algorithm is asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e., Public Key and Private Key. As

the name describes that the Public Key is given to everyone and Private key is kept private.

2.1 Principle:

RSA is based on the fact that it is difficult to factorize a large integer (currently best can be done in $O(\sqrt{n})$). The public key consists of two numbers where one number is multiplication of two large prime numbers. And private key is also derived from the same two prime numbers. So, if somebody can factorize the large number, the private key is compromised. Therefore, encryption strength totally lies on the key size and if we double or triple the key size, the strength of encryption increases exponentially

2.2 Algorithm:

The public key in this cryptosystem consists of the value n, which is called the **modulus**, and the value e, which is called the **public exponent**. The private key consists of the modulus n and the value d, which is called the **private exponent**. RSA public-private key-pair can be generated through following way:

- Generate a pair of large random primes p and q
- Compute the modulus n=p*q
- Select an odd public exponent e between 3 and n-1 that is relatively prime to p-1 and q-1
- Compute the private exponent d from e, p and q
- Output (n, e) as the public key and (n, d) as private key

The encryption operation in the RSA is exponentiation to the eth power modulo n:

$$C=Encrypt(m)=M^e \bmod n$$

The decryption operation is exponentiation to the d^{th} power modulo n:

$$m=Decrypt(C)=C^d \bmod n$$

Computing the private exponent:

Let n be the product of two distinct prime numbers p and q, and let e be the public exponent. Let L=LCM (p-1, q-1). The private exponent d for the RSA encryption cryptosystem is any integer solution to congruence

$$de \equiv 1 \bmod L$$

As e is relatively prime to p-1 and q-1, which ensures that inverse exists.

The RSA cryptosystem works because exponentiation to the d^{th} power modulo n is the inverse of exponentiation to the e^{th} power when the exponents d and e are inverses modulo L. That is, for all m between 0 and n-1,

$$m = (m^e)^d \bmod n$$

2.3 Proof of Correctness

As $de \equiv 1 \bmod L$, we can write de as (p-1)k+1
By Fermat's Little theorem: $a^{p-1} \equiv 1 \pmod p$ for any integer a and prime p, not dividing a.

To show $m^{ed} \equiv m \pmod p$, we consider two cases:

- If $m \equiv 0 \pmod p$, m is a multiple of p. Thus, m^{ed} is a multiple of p. So, $m^{ed} \equiv 0 \equiv m \pmod p$
- If m is not equal to 0(mod)p
  $m^{ed} = m^{(ed-1)}\, m = m^{(p-1)k} m = (m^{p-1})^k\, m \equiv 1^h m \equiv m \pmod p$

where we used Fermat's little theorem to replace $m^{p-1} \bmod p$ with 1.

Similarly for $m^{ed} \equiv m \pmod q$
By Chinese remainder theorem, we can conclude that

$$m^{de} = m \pmod{pq}$$

Chapter 2

Attacks on RSA

Cryptanalysis is the study of ciphertext, cyphers, and cryptosystems with the goal of comprehending how they operate and developing ways for defeating or compromising them. cryptanalysts target secure hashing, digital signatures, and

other cryptographic algorithms. Below are some possible attacks on the RSA encryption scheme: -

1. Factoring Method

The most straightforward attack on RSA is factorization of the modulus N = pq. Once a factor p is discovered, the factor q = N/p may be computed, so $\varphi(N) = N-p-q+1$ is revealed. This is enough to compute $d \equiv e^{-1} \bmod \varphi(N)$.

The current fastest method for factoring is the General Number Field Sieve [4]. It has a running time of $\exp(c + o(1)) \cdot (\log N)^{1/3} (\log \log N)^{2/3}$ for some $1 < c < 2$.

The size of N is chosen to foil this attack. Even though the speed of computer hardware continues to accelerate, it seems unlikely that the best factoring algorithms will be able to factor say 1024-bit RSA moduli in the next twenty years

2. Wiener Attacks:

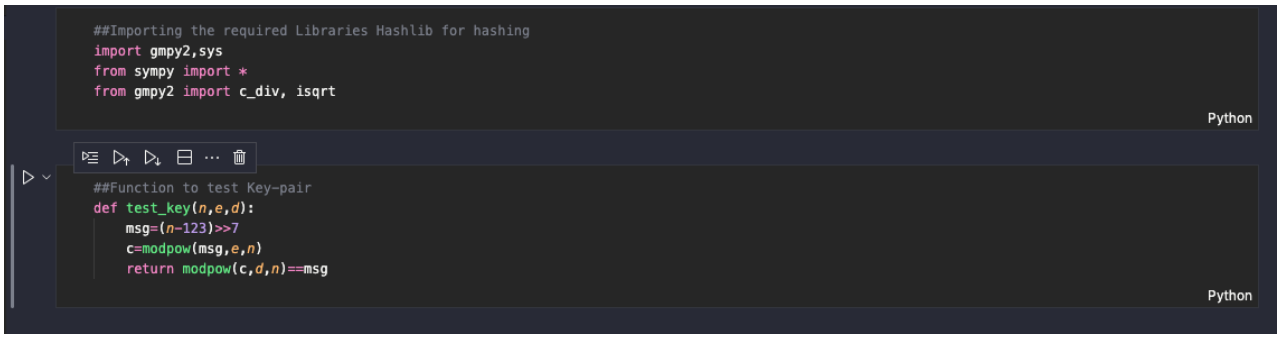
To speed up RSA decryption and signing, it is tempting to use a small secret exponent d rather a random $d \leq \varphi(N)$. Since modular exponentiation takes time linear in $\log_2 d$, using a d that is substantially shorter than N can improve performance by a factor of 10 or more.

Unfortunately, a classic attack by Wiener [1] shows that a sufficiently short d leads to an efficient attack on the system. His method uses approximations of continued fractions

Wiener's Theorem : Suppose N = pq and $\frac{\sqrt{N}}{2} < q < p < \sqrt{N}$. Furthermore suppose $d < \frac{1}{3} N^{1/4}$. There is an algorithm which, given N and e, generates a

list of length logN of candidates for d, one of which will equal d. This algorithm runs in time linear in logN.

Below is the code implementation of Wiener attack on Jupyter notebook

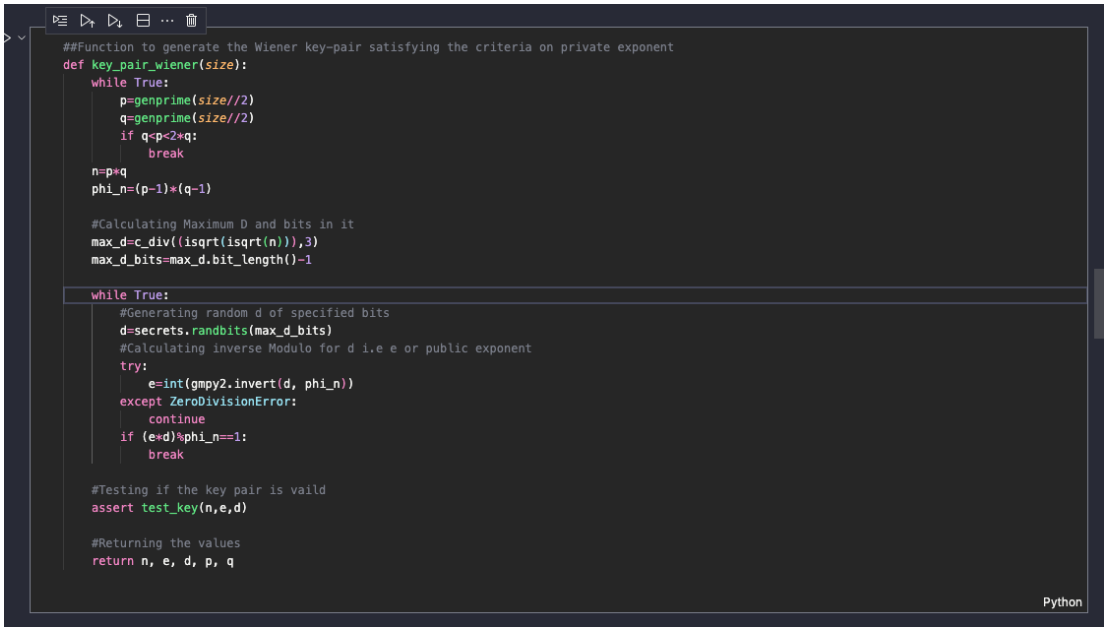


```python
##Importing the required Libraries Hashlib for hashing
import gmpy2,sys
from sympy import *
from gmpy2 import c_div, isqrt
```

Python

```python
##Function to test Key-pair
def test_key(n,e,d):
    msg=(n-123)>>7
    c=modpow(msg,e,n)
    return modpow(c,d,n)==msg
```

Python



```python
##Function to generate the Wiener key-pair satisfying the criteria on private exponent
def key_pair_wiener(size):
    while True:
        p=genprime(size//2)
        q=genprime(size//2)
        if q<p<2*q:
            break
    n=p*q
    phi_n=(p-1)*(q-1)

    #Calculating Maximum D and bits in it
    max_d=c_div((isqrt(isqrt(n))),3)
    max_d_bits=max_d.bit_length()-1

    while True:
        #Generating random d of specified bits
        d=secrets.randbits(max_d_bits)
        #Calculating inverse Modulo for d i.e e or public exponent
        try:
            e=int(gmpy2.invert(d, phi_n))
        except ZeroDivisionError:
            continue
        if (e*d)%phi_n==1:
            break

    #Testing if the key pair is vaild
    assert test_key(n,e,d)

    #Returning the values
    return n, e, d, p, q
```

Python

```python
if __name__ == '__main__':
    #Generating a pair of private and public key
    #Hashing it to make it more secure
    N, e, d, p, q = key_pair_wiener(1024)
    print('[+] Generated an RSA keypair with a short private exponent.')
    print('[+] ++ e:    ', (e))
    print('[+] -- d:    ', (d))
    print('[+] ++ N:    ', (N))
    print('[+] -- p:    ', (p))
    print('[+] -- q:    ', (q))
    print('[+] -- phiN: ', ((p - 1)*(q - 1)))
    print('[+] -----------------')

    #Generating the cf_expansion of e and N
    cf_expansion = cf_exp(e, N)

    #Calculating the convergence of the expansion
    convergents = convergent(cf_expansion)
    print('[+] Found the continued fractions expansion convergents of e/N.')

    print('[+] Iterating over convergents; '
          'Testing correctness through factorization.')
    print('[+] ...')
    for pk, pd in convergents: # pk - possible k, pd - possible d
        if pk == 0:
            continue

        possible_phi = (e*pd - 1)//pk

        p = Symbol('p', integer=True)
        roots = solve(p**2 + (possible_phi - N - 1)*p + N, p)

        if len(roots) == 2:
            pp, pq = roots # pp - possible p, pq - possible q
            if pp*pq == N:
                print('[+] Factored N! :) derived keypair components:')
                print('[+] ++ e:    ', (e))
                print('[+] ++ d:    ', (pd))
                print('[+] ++ N:    ', (N))
                print('[+] ++ p:    ', (pp))
                print('[+] ++ q:    ', (pq))
                print('[+] ++ phiN: ', (possible_phi))
                sys.exit(0)
```

```python
    print('[-] Wiener\'s Attack failed; Could not factor N')
    sys.exit(1)
```

Python

```
[+] Generated an RSA keypair with a short private exponent.
[+] ++ e:
6030864414228476681916285809774971580010522599140301752360809525929899745729670334243088529003715617800143508783931528612221396861977008766748140181
[+] -- d:     5331867266387788009502181936198395258153087173479822263864248828278569914835
[+] ++ N:
1179646261619451738348893930777327718313184870100143679117024526860731412736439190858062180920301999597267105553633115570873325758296955062253051806
[+] -- p:     1303930593073882464146581059038431014464666702477770384001250131014542652684925763175095254314782515499450772413673701399551380114420 4
[+] -- q:     9046848566061762705085433070412372671865519170767397385420337859172870968500174858918620849497553790200801463285161734840465549293252 9
[+] -- phiN:
1179646261619451738348893930777327718313184870100143679117024526860731412736439190858062180920301999597267105553633115570873325758296955062253051806
[+] -----------------
[+] Found the continued fractions expansion convergents of e/N.
[+] Iterating over convergents; Testing correctness through factorization.
[+] ...
[+] Factored N! :) derived keypair components:
[+] ++ e:
6030864414228476681916285809774971580010522599140301752360809525929899745729670334243088529003715617800143508783931528612221396861977008766748140181
[+] ++ d:     5331867266387788009502181936198395258153087173479822263864248828278569914835
[+] ++ N:
1179646261619451738348893930777327718313184870100143679117024526860731412736439190858062180920301999597267105553633115570873325758296955062253051806
[+] ++ p:     9046848566061762705085433070412372671865519170767397385420337859172870968500174858918620849497553790200801463285161734840465549293252 9
[+] ++ q:     1303930593073882464146581059038431014464666702477770384001250131014542652684925763175095254314782515499450772413673701399551380114420 4
[+] ++ phiN:
1179646261619451738348893930777327718313184870100143679117024526860731412736439190858062180920301999597267105553633115570873325758296955062253051806
```

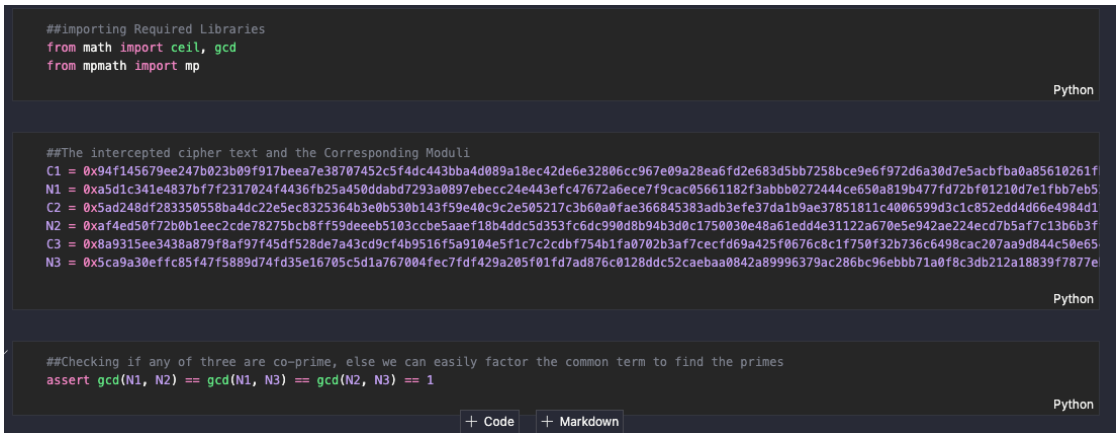
3. Håstad Broadcast Attack

To accelerate RSA encryption (and signature verification), it is useful to reduce the public exponent e to a small value, such as 3. This however exposes RSA to the following attack, which was developed by Håstad [2].

Let us start with a simpler version. Suppose Bob wishes to send the same message M to k recipients, all of whom are using public exponent equal to 3. He obtains the public keys $\langle N_i, e_i \rangle$ for i = 1,...,k, where $e_i = 3$ for all i. Naively, Bob computes the ciphertext $C_i = M^3 \bmod N_i$ for all i and sends C_i to the ith recipient.

A simple argument shows that as soon as $k \geq 3$, the message M is no longer secure. Suppose Eve intercepts C_1, C_2, and C_3, where $C_i = M^3 \bmod N_i$. We may assume $\gcd(N_i, N_j) = 1$ for all $i \neq j$ (otherwise, it is possible to compute a factor of one of the N_i's.) By the Chinese Remainder Theorem, she may compute $C \in Z^*_{N_1N_2N_3}$ such that $C \equiv C_i \bmod N_i$. Then $C \equiv M^3 \bmod N_1N_2N_3$; however, since $M < N_i$ for all i, we have $M^3 < N_1N_2N_3$. Thus $C = M^3$ holds over the integers, and Eve can compute the cube root of C to obtain M.

The Code Implementation is of above example is as follows:-



```
##importing Required Libraries
from math import ceil, gcd
from mpmath import mp
```

Python

```
##The intercepted cipher text and the Corresponding Moduli
C1 = 0x94f145679ee247b023b09f917beea7e38707452c5f4dc443bba4d089a18ec42de6e32806cc967e09a28ea6fd2e683d5bb7258bce9e6f972d6a30d7e5acbfba0a85610261f
N1 = 0xa5d1c341e4837bf7f2317024f4436fb25a450ddabd7293a0897ebecc24e443efc47672a6ece7f9cac05661182f3abbb0272444ce650a819b477fd72bf01210d7e1fbb7eb5
C2 = 0x5ad248df283350558ba4dc22e5ec8325364b3e0b530b143f59e40c9c2e505217c3b60a0fae366845383adb3efe37da1b9ae37851811c4006599d3c1c852edd4d66e4984d1
N2 = 0xaf4ed50f72b0b1eec2cde78275bcb8ff59deeeb5103ccbe5aaef18b4ddc5d353fc6dc990d8b94b3d0c1750030e48a61edd4e31122a670e5e942ae224ecd7b5af7c13b6b3f
C3 = 0x8a9315ee3438a879f8af97f45df528de7a43cd9cf4b9516f5a9104e5f1c7c2cdbf754b1fa0702b3af7cecfd69a425f0676c8c1f750f32b736c6498cac207aa9d844c50e65
N3 = 0x5ca9a30effc85f47f5889d74fd35e16705c5d1a767004fec7fdf429a205f01fd7ad876c0128ddc52caebaa0842a89996379ac286bc96ebbb71a0f8c3db212a18839f7877e
```

Python

```
##Checking if any of three are co-prime, else we can easily factor the common term to find the primes
assert gcd(N1, N2) == gcd(N1, N3) == gcd(N2, N3) == 1
```

Python

+ Code + Markdown

```python
# Using Chinese Remainder Theorem
# Z_p x Z_q isomorph with Z_N where N=pq (where Z_p is a Ring of integers redcued by p)
# (x_p, x_q) <-> [(x_p * 1_p + x_q * 1_q) mod N](where 1_p is the identity element in ring p and 1_q is the identity element in q)
# 1_p <-> (1, 0)
# 1_q <-> (0, 1)
N12 = N1 * N2
#Using the extended Eculiedan
k= eucalg(N1, N2)
a,b=k[0],k[1]
p1 = b * N2 % N12
q1 = a * N1 % N12
C12 = (C1 * p1 + C2 * q1) % N12

N = N12 * N3

k = eucalg(N12, N3)
a,b=k[0],k[1]
p1 = b * N3 % N
q1 = a * N12 % N
C123 = (C12 * p1 + C3 * q1) % N

mp.dps = len(str(C123))     # set floating-point precision of mpmath

m = int(mp.cbrt(C123))
bits = bin(m)[2:]
message = int.to_bytes(m, ceil(len(bits) / 8), byteorder="big").decode("ASCII")
print(message)
```

Python

```
This message is secret and unfortunately e is not high. The message should also be sufficiently long so it is a residue.
```

Håstad demonstrates a considerably stronger result. Consider the following naive defense against the aforementioned attack to comprehend it. Let's say Bob pads the message M before encrypting it, resulting in slightly different messages for the recipients. Bob might encrypt I 2m + M and send it to the ith recipient, for example, if M is m bits long. This linear padding scheme is insecure, as Håstad demonstrated. In reality, he demonstrated that the message will produce an insecure scheme when any fixed polynomial is applied to it.

Håstad Theorem : Suppose $N_1, \ldots, N_k$ are relatively prime integers and set $N_{min} = \min_i(N_i)$. Let $g_i(x) \in Z_{N_i}[x]$ be k polynomials of maximum degree d. Suppose there exists a unique $M < N_{min}$ satisfying

$$g_i(M) = 0 \pmod{N_i} \text{ for all } i \in \{0,\ldots,k\}$$

Furthermore suppose $k > d$. There is an efficient algorithm which, given $\langle N_i, g_i(x)\rangle$ for all i, computes M.

This can be applied to the problem of broadcast RSA as follows. Suppose the ith plaintext is padded with a polynomial $f_i(x)$, so that $C_i \equiv (f_i(M))^{e_i} \mod N_i$. Then the polynomials $g_i(x) := (f_i(x))^{e_i} - C_i$ satisfy the above relation. The attack succeeds once $k > \max_i(e_i \cdot \deg f_i)$.

4. Coppersmith Short Padding Attack

Coppersmith showed that if randomized padding is used improperly then RSA encryption is not secure [3].

Let's say Bob encrypts a message M and sends it to Alice using a short random pad. When Eve gets hold of this and interferes with the transmission, Bob is forced to send the message again using a fresh random pad. The attack that comes after illustrates that if the random pads are too short, Eve still can decipher the message M even if she is unaware of the ones being deployed.

For simplicity, we will assume the padding is placed in the least significant bits, so that $C_i = (2^m M + r_i)^e \bmod N$ for some small m and random $r < 2^m$. Eve now knows

$$C_1 = (2^m M + r_1)^e \pmod N \text{ and } C_2 = (2^m M + r_2)^e \pmod N$$

for some unknown M, r_1, and r_2. Define $f(x,y) := x^e - C_1$ and $g(x,y) := (x+y)^e - C_2$. We see that when $x = 2^m M + r_1$, both of these polynomials have $y = r_2 - r_1$ as a root modulo N. We may compute the resultant $h(y) := \mathrm{Res}_x(f,g)$ which will be of degree at most e^2 Then $y = r_2 - r_1$ is a root of h(y) modulo N. If $|r_i| < (1/2)N^{1/e^2}$ for i = 1,2 then we have that $|r_2 - r_1| < N^{1/e^2}$.

By Coppersmith's Theorem[5] we may compute all of

the roots h(y), which will include $r_2 - r_1$. Once $r_2 - r_1$ is discovered, we may use a result of Franklin and Reiter [6] to extract M

Below is the code for the above attack on Sage Notebook

In [2]:
```
import random
import binascii
```

In [3]:
```
#Function to implement Coppersmith Short padding attack
def coppersmith_short_pad(C1, C2, N, e = 3, eps = 1/25):
    P.<x, y> = PolynomialRing(Zmod(N))#Polynomial ring of two varibles
    P2.<y> = PolynomialRing(Zmod(N))#Polynomial ring of one variable

    g1 = (x^e - C1).change_ring(P2)#f(x,y)=x^e-C_1(modN)
    g2 = ((x + y)^e - C2).change_ring(P2)#(f(x,y)=(x+y)^-C_2(modN))

    # Changes the base ring to Z_N[y] and finds resultant of g1 and g2 in x
    res = g1.resultant(g2, variable=x)

    # coppersmith's small_roots only works over univariate polynomial rings, so we
    # convert the resulting polynomial to its univariate form and take the coefficients modulo N
    # Then we can call the sage's small_roots function and obtain the delta between m_1 and m_2.
    # Play around with these parameters: (epsilon, beta, X)
    roots = res.univariate_polynomial().change_ring(Zmod(N))\
        .small_roots(epsilon=eps)

    return roots[0]
```

In [4]:
```
#Using Franklin-Reiter attack to determine the final message
def franklin_reiter(C1, C2, N, r, e=3):
    P.<x> = PolynomialRing(Zmod(N))
    equations = [x ^ e - C1, (x + r) ^ e - C2]
    g1, g2 = equations
    return -composite_gcd(g1,g2).coefficients()[0]
```

In [22]:
```
#Recovering message intercepted Ciphertext
def recover_message(C1, C2, N, e = 3):
    delta = coppersmith_short_pad(C1, C2, N)
    recovered = franklin_reiter(C1, C2, N, delta)
    #message = int.to_bytes(m, ceil(len(recovered) / 8), byteorder="big").decode("ASCII")
    return recovered
```

In [6]:
```
def composite_gcd(g1,g2):
    return g1.monic() if g2 == 0 else composite_gcd(g2, g1 % g2)
```

In [24]:
```
# Takes a long time for larger values and smaller epsilon
def test():
    N = random_prime(2^10, proof=False) * random_prime(2^10, proof=False)
    e = 3

    m = Integer(math.log(N, 2) // e^2)

    ##Random pad for the message
    r1 = random.randint(1, pow(2, m))
    r2 = random.randint(1, pow(2, m))

    #The message transmitted and the corresponding ciphertext
    M = int(binascii.hexlify(b"hello"), 16)
    print(M)
    C1 = pow(pow(2, m) * M + r1, e, N)
    C2 = pow(pow(2, m) * M + r2, e, N)

    # Using eps = 1/125 is slooooowww
    print(coppersmith_short_pad(C1, C2, N, eps=1/200))
    print(recover_message(C1, C2, N))
```

In [25]:
```
if __name__ == "__main__":
    test()
```

```
448378203247
0
9916
```

References:

[1] M. J. Wiener, "Cryptanalysis of short RSA secret exponents," in *IEEE Transactions on Information Theory*, vol. 36, no. 3, pp. 553-558, May 1990, doi: 10.1109/18.54902.

[2] J. Hastad. Solving simultaneous modular equations of low degree. SIAM J. of Computing, 17:336{341, 1988.

[3] D. Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. Journal of Cryptology, vol. 10, pp. 233–260, 1997

[4] NAPIAS,Huguette."AGeneralizationoftheLLL-AlgorithmoverEuclideanRingsorOrders."*JournaldeThéorie Des Nombres de Bordeaux*, vol. 8, no. 2, 1996, pp. 387–96. *JSTOR*,

[5] D. Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. Journal of Cryptology, vol. 10, pp. 233–260, 1997.

[6] G. Durfee. Public Key Cryptanalysis Using Algebraic and Lattice Methods. Ph.D. Thesis, Stanford University, January 2002.

GitHub Link for above code:-
https://github.com/shreyanshsinha13/Cryptanalysis-of-RSA