

## CTF Training Camp for Hackers



# Crypto 103: Fantastic RSA and How to Attack Them

Cousin Wu

November 4, 2021



Open Innovation Lab  
CUHK



## 1 Introduction

Public Key Cryptography

## 2 Mathematical Backgrounds

Modular Arithmetic

Elementary Number Theory

RSA

## 3 RSA

RSA

Choices of  $e$

Intermission: Encoding

Implementing RSA for Yourself

RSA Certificate with Openssl and PyCrypto

## 4 Attacking RSA

Introduction

Meet in the Middle Attack

Factoring



- Common Modulus Attack
- Broadcast Attack
- Fermat Factorization
- Weiner's Attack
- Decryption Oracles
- Small Messages and Public Exponents
- Fixed Padding
  - Coppersmith's Method
  - Håstad's broadcast attack
- Padding Oracle
- Boneh-Durfee Attack

## 5 Variants of RSA

- RSA-CRT
- Carmichael Function



## Special Moduli

### 6 General Formulation of RSA

RSA on Polynomial Ring  
Ring of Integers

### 7 Reference



# Introduction



# About the Speaker



No one cares



RSA is one of the most widely used (public key) encryption scheme out there! But the math behind is relatively simple: just making use of the theorems of elementary number theory. Some attacks require some deeper mathematics.




Before discussing RSA, it is necessary to talk about what it is, and the mathematical backgrounds for understanding it. Note that this training requires somewhat intensive elementary number theory.

For further reference, interested readers can reference chapter 5 of [1], chapter 2 and 3 of [2] and/or chapter 24 of [3](in increasing difficulty). Note that the first two are for undergraduates taking first course on cryptography and **mathematical cryptography** respectively, and the third book is targeted at graduate students in Mathematics studying mathematical cryptography. [4] gives a survey on attacks on RSA, and is one of the major references for the slides.





# Software Installation

A horizontal bar with a teal segment on the left and an orange segment on the right.

Be sure to have the python library gmpy2 and pycrypto installed. Sage is also necessary. If you don't want to install it, [sagecell](#) or [cocalc](#) would do the trick for now.

This may help you. [crypto-commons module for python](#)



# Public Key Cryptography

With public key cryptography, we still have the encryption and decryption function  $E_k$  and  $D_{k'}$  such that  $D_{k'}(E_k(m)) = m$  for any message  $m$  supported by the encryption scheme.

Note that now the encryption and decryption key are different. The key  $k$  to do encryption is called the **public key**, and is meant to be published, to be known by everyone. The  $k'$  to do decryption is called the **private key**, and as the name suggests, the key must be kept private to yourself.

A **Public Key Infrastructure** (PKI) is a system of creating, storing and distributing **digital certificates**. A **Certificate Authority** (CA) is responsible for storing, issuing and signing digital certificates. (Won't discuss)



# Examples

A horizontal bar with a teal segment on the left and an orange segment on the right.

Popular public key cryptography **in use**: RSA, Elliptic Curve Cryptography (ECC)

Other public key cryptography: ElGamal, Rabin cryptosystem, NTRU (Lattice-based)

**BAD** public key cryptography: GGH, Merkle–Hellman knapsack cryptosystem (both are lattice-based)




# Symmetric vs Public Key Cryptography

	Symmetric Key	Public Key
Keys	One Key	Public and Private key
Speed	Usually Fast	Usually Slow
Key Exchange	Yes	Public Key announced publicly



# Why Public Key?

A horizontal bar with a teal segment on the left and an orange segment on the right.

Another problem of symmetric key cryptography is the exchange of keys. We need a side channel to exchange keys. But then we need to encrypt the side channel to make sure it is safe. But to encrypt that we need another secure channel to exchange keys!



# Diffie-Hellman Key Exchange

There is a way to exchange keys without the aforementioned problem. Choose a prime  $p$  and a primitive root mod  $p$ ,  $g$  (such that  $g^n$  runs through all 1 to  $p$  as  $n$  runs through 0 to  $p-1$ ). Then

	Diffie	Hellman
Generate	$a$	$b$
Calculate and send	$n_a = g^a \pmod{p}$	$n_b = g^b \pmod{p}$
Calculate key	$n_a^b = (g^a)^b \pmod{p}$ $= g^{ab} \pmod{p}$	$n_a^b = (g^a)^b \pmod{p}$ $= g^{ab} \pmod{p}$



Symmetry key cryptography can only be used for communication between two parties! So if we have  $n$  parties that want to communicate with each other, we need  $\frac{n(n-1)}{2}$  pairs of keys! For public key cryptography, to encrypt a message to send to a particular person, everyone uses the same public key. So we only need  $n$  keys!



# Mathematical Backgrounds





# Crash Course in Modular Arithmetic

Here we need to go back to modular arithmetic, finally addressing some of the important concepts.

## Definition (GCD)

Say the **greatest common divisor** ( or GCD, highest common factor, HCF) of two integers  $m$  and  $n$  (not both 0) is  $d$  if

1.  $m$  and  $n$  are both multiples of  $d$  (Common Factor)
2.  $d$  is the largest of all the common factors.

We have the notation  $\gcd(m, n) = d$ .

The second condition can be put in a more strict way. If  $m$  and  $n$  is also multiples of another number  $k$ , then  $k$  is a multiple of  $d$ . This implies that  $d$  is the largest one.



### Example

Factors of 12: 1, 2, 3, **4**, 6, 12

Factors of 16: 1, 2, **4**, 8, 16

So  $\gcd(12, 16) = 4$ .

Properties of the GCD:

1.  $\gcd(a, a) = a$
2.  $\gcd(a, b) = \gcd(b, a)$
3.  $\gcd(1, n) = 1 \forall n \neq 0$
4.  $\gcd(0, n) = n \forall n \neq 0$
5.  $\gcd(a, b) = \gcd(a \bmod b, b)$



# Division

## Definition (Division (Euclidean domain))

Given two integers  $a$ ,  $b \neq 0$ . Then there exists **unique** integers  $q$  and  $0 \leq r \leq b - 1$  such that

$$a = bq + r$$

$q$  is called the quotient, and  $r$  is called the remainder. Then we say

$$r = a \mod b$$

We also denote the set of integers from 0 to  $n - 1$  as  $\mathbb{Z}/n\mathbb{Z}$ . We can freely add, subtract and multiply integers in the set, take mod  $n$ , and the result is still in  $\mathbb{Z}/n\mathbb{Z}$ .



# Euclidean Algorithm

The GCD can be found in a more elegant (and fast) way when we have such a division algorithm. Note that  $\gcd(a, b) = \gcd(a \bmod b, b)$ . So if we have  $a > b$ ,  $\gcd(a, b) = \gcd(b, a \bmod b)$ , we have that  $b > a \bmod b$ . So we can do the same trick again to obtain the answer. This is known as the **Euclidean Algorithm**, and is known 2300 years ago by the great mathematician Euclid.

Of course, as we are doing cryptography (not maths), we will make use of computers to compute the gcd. In python, there is a gcd function in the **math** library.



# Bézout's Identity

## Theorem

*Given integers  $a$  and  $b$ , there exists integers  $r$  and  $s$  such that*

$$ar + bs = \gcd(a, b)$$

Recall before, we have mentioned briefly about the existence of multiplicative inverses. Recall that we want to solve the equation

$$ax \equiv 1 \pmod{n}.$$

If  $\gcd(a, n) = 1$ , then we get integers  $r, s$  such that  $ar + ns = 1$ . Taking mod  $n$ . We get exactly  $ar \equiv 1 \pmod{n}$ !



## Theorem

*Multiplicative inverse of integer  $a$  mod  $n$  exists if and only if  $\gcd(a, n) = 1$ .*

Note that by Euclidean Algorithm, we can obtain the coefficients in the Bézout's Identity. The exact calculations will not be introduced.

To get the multiplicative inverse of an element, in gmpy2, we can do `gmpy.invert(x,n)` to find the multiplicative inverse of  $x$  mod  $n$ . In sage, we do `inverse_mod(x,n)`. Starting from Python 3.8, you can do `pow(x, -1, p)`.

To get the coefficients in Bézout's identity, sage has the `xgcd` function.

```
d,u,v = xgcd(12,15)
assert d == u*12 + v*15
```



# Elementary Number Theory

The following is a function that is crucial to RSA.

## Definition (Euler's totient function)

Define the Euler's totient function  $\phi : \mathbb{N} \rightarrow \mathbb{N}$  as  $\phi(n)$  = number of integers  $x$  between 1 and  $n$  inclusive such that  $\gcd(x, n) = 1$  (or multiplicative inverse exists). Some may also define the function as  $\phi(n) = |(\mathbb{Z}/n\mathbb{Z})^*|$ .


## Theorem (Euler's Theorem)

*Suppose  $n$  is positive integer and  $a$  is an integer with  $\gcd(a, n) = 1$ . Then*

$$a^{\phi(n)} \equiv 1 \pmod{n}$$



# RSA



Let's design an encryption such that  $D(E(m)) = m^{k\phi(n)+1} \pmod{n}$ , then Euler's Theorem will imply  $m^{1+k\phi(n)} = m \cdot (m^{\phi(n)})^k = m \pmod{n}$ ! This is exactly what we do in RSA.

Now that we can compute  $d$  given  $\phi(n)$  and  $e$ , **how do we compute  $\phi(n)$ ? Is computing it hard?**





# Evaluating Phi Function

It seems that in order to compute  $\phi(n)$ , one needs to check all the numbers between 1 and  $n$  and see if the gcd is 1. However, there is a way to compute the phi function fast.

## Theorem

1. If  $p$  is a prime, then  $\phi(p) = p - 1$
2. If  $n = ab$  with  $\gcd(a, b) = 1$ , then  $\phi(n) = \phi(a)\phi(b)$

The first property is obvious. The second property follows directly from the Chinese Remainder Theorem.



## Corollary

*If  $p$  and  $q$  are prime, and  $n = pq$ , then  $\phi(n) = (p - 1)(q - 1)$ .*

Now if we know the factorization of  $n = pq$ , then computing  $\phi(n)$  is easy! **Computing  $\phi(n)$  requires factorization.**

## Exercise

Proof that

$$\phi(n) = n \cdot \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

Where  $p$  is all the prime that divides  $n$ . In other words, if  $n = p_1^{k_1} \cdots p_j^{k_j}$  where  $p_i$  are primes and  $k_i > 0$ , then

$$\phi(n) = n \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_j}\right)$$



# Integer Factorization Problem

How fast is factorization?

Given integer  $n$ , can we tell if  $n$  is prime? Can we factorize  $n$  into prime factors in polynomial time?

In fact, the first question can be solved in **polynomial time** deterministically (AKS primality test). Although a more practical algorithm to use would be the probabilistic polynomial time algorithm **Miller-Rabin Test**.



We don't know the answer to the second problem. Although algorithms exist to factor integers faster than brute force (e.g. General Number Field Sieve), for now there is no known algorithm for integer factorization in polynomial time for usual computation models.

Note that there is a **quantum computer algorithm** for integer factorization, the **Shor's Algorithm**, that can factorize integers in polynomial time! Note that currently quantum computers do not have such computational capability yet.



# RSA Problem

Note that importantly, however, that the security of RSA is certainly not as hard as factoring, since to break RSA we just need to solve the following **RSA Problem**:

Can we find plaintext  $m$  such that  $m^e \equiv c \pmod{N}$  given  $c$ ,  $e$  and  $N$ ?

The upshot is that if integer factorization is easy, then RSA is broken, but **not vice versa**.



RSA



# RSA

RSA is developed by **R**ivest (The R in CLRS!), **S**hamir and **A**dleman in 1977. The algorithm for generate n-bit RSA keys are as follows:

1. Choose two large primes  $p$  and  $q$  with roughly  $\frac{n}{2}$  bits and compute  $n = pq$
2. Calculate  $\phi(n) = (p - 1)(q - 1)$
3. Choose  $e$  such that  $\gcd(e, \phi(n)) = 1$  and compute  $d \equiv e^{-1} \pmod{\phi(n)}$
4. Discard  $p$ ,  $q$  and  $\phi(n)$

Then we get the public key of RSA as  $(n, e)$ , and the private key of RSA as  $(n, d)$ .



Now the encryption is as follows:

$$E_{(n,e)}(m) = m^e \pmod{n}$$

And the decryption is

$$E_{(n,d)}(c) = c^d \pmod{n}$$

$e$  is sometimes called the public exponent,  $d$  the private exponent.

As we have discussed, in general it is very hard to compute  $\phi(n)$  nor  $m$  given only the ciphertext and the public key  $(n, e)$ .

One important note is that we need  $m < n$ , otherwise we may not be able to get back the same message after decryption.





# Properties

## Theorem (Properties of RSA)

*With the implied meaning of the variables,*

1. (Multiplicative)  $E_k(m)E_k(n) = E_k(mn)$
2. (Bijection)  $D_{k'}(E_k(m)) = E_k(D_{k'}(m)) = m$

## Exercise

Prove the above properties.

The second property allows us to use RSA as digital signature, but we will not cover it here.

The first property is much more interesting. We will see it being exploited soon.



## Exercise (Correctness of RSA)

In previous discussions, we mentioned that RSA made use of the Euler's Theorem. But Euler's Theorem works only if  $m$ , the message, is coprime to  $n$ . (i.e.  $\gcd(m, n) = 1$ ). If the assumption is false, do we still get  $D_{(n,d)}(E_{(n,e)}(m)) = m \pmod{n}$ ?

## Exercise (Modified RSA)

Suppose we have an encryption scheme as follows. Take a prime  $p$ , and pick a pair  $(e, d)$  integers such that  $de \equiv 1 \pmod{p-1}$ . To encrypt a message, we do  $E_e(x) = x^e \pmod{p}$ . To decrypt, we do  $D_d(x) = x^d \pmod{p}$ . Show that  $E_e$  and  $D_d$  are indeed inverses of each other. Also show that this scheme is **NOT** secure if the attacker knows the **public key**  $(p, e)$ .



# Speed of RSA

For encryption, we do a modular exponentiation  $m^e \pmod n$ . If we just loop  $e$  times and each time multiplying  $m$ , the running time of encryption would be very slow. This approach requires  $O(e)$  multiplication which is exponential time since time is counted in terms of length of input, not the actual input.

```
1: function NaiveModExp( $m, e, n$ ):  
2:    $x \leftarrow 1$   
3:   while  $e > 0$  do  
4:      $x \leftarrow x \times m \pmod n$   
5:      $e \leftarrow e - 1$   
6:   return  $x$ 
```



# Repeated Squaring

A smarter approach is to note that we can compute  $m^{2^k} \pmod n$ , for each  $k < \log e$ , and depending on the digits on the binary expansion of  $e$ , we may choose to multiply  $m^{2^k}$  into the result. For example, if  $e = 1101_2$ , then  $m^e \equiv m^{1101_2} \equiv m^{8+4+1} \equiv m \cdot m^4 \cdot m^8 \pmod n$ .

An important observation is that  $m^{2^k}$  can be computed from  $m^{2^{k-1}}$  by squaring  $\left(m^{2^{k-1}}\right)^2$ . This leads to the repeated squaring algorithm of modular exponentiation.




# Algorithm

```
1: function REPEATED-SQUARING( $m, e, n$ ):  
2:    $e[] \leftarrow$  representation of  $e$  in base-2  
3:    $res \leftarrow m$   
4:   for  $i$  from 0 to  $\log e$  do  
5:      $res \leftarrow res^2 \pmod n$   
6:     if  $e[i] == 1$  then  
7:        $res \leftarrow res \cdot m \pmod n$   
8:   return  $x$ 
```

Now the algorithm only requires  $O(\log(e))$  multiplications, which is polynomial time.




# Choice of e



We note that from the previous algorithm, if we have a lot of zeroes in the base-2 representation, then the number of multiplications can be greatly reduced! For example, if we have exponents in the form of  $2^k + 1$  for some integer  $k$ , then we only need to carry out about  $k$  multiplications! So for RSA encryption, we choose the public exponent to be primes in the form of  $2^k + 1$ , so for example 3, 17, 257 and 65537.



# Side-Channel Attack

A horizontal bar with a teal segment on the left and an orange segment on the right.

Let's say we have an embedded system that performs RSA decryption. If the system is using repeated squaring for modular exponentiation, then since multiplication is slow and takes more power, by measuring the voltage the system draws (with high precision) we would be able to tell if multiplication is performed or not. Since we know that the multiplication occurs whenever the bit of  $d$  in base-2 is 1, then we would be able to recover  $d$  by observing the voltage.

This method that does not target the cryptosystem, but observing the physical properties or other feedbacks not from the cryptosystem itself, is called **Side-Channel Attacks**.



# Encrypting messages

Note that RSA is a mathematical function that takes numbers. Our messages are words (say ASCII for simplicity). How to use RSA to encrypt words?

Encoding!

We can first convert the message to ASCII, then treat the ASCII string as a long number.

Message	H	a	h	a
ASCII	0x48	0x61	0x68	0x61
Hex	0x48	61	68	61
Integer	1214343265			





# Implement RSA with PyCrypto

```
from Crypto.PublicKey import RSA

def encrypt(m, n, e):
    return pow(m, e, n)

def decrypt(m, n, d):
    return pow(m, d, n)

m = 123456
e = 17
rsa = RSA.generate(1024, e=e)
n, d = rsa.n, rsa.d
assert decrypt(encrypt(m, n, e), n, d) == m
```



# Certificates



In CTF, we are often given the RSA keys in certificates. (Public key) Certificates are a way to store a key.



# Generate keys with Openssl

The following command generates a RSA-2048 key pair with private key named `private.pem` and public key named `public.pem` (2048 means that the  $n$  will be 2048 bits long)

```
openssl genrsa -out private.pem 2048  
openssl rsa -in private.pem -out public.pem -pubout
```



# Reading keys with PyCrypto

```
from Crypto.PublicKey import RSA

# Import Public Key
pub = None
with open("public.pem", "r") as f:
    pub = RSA.import_key(f.read())
print(pub.n, pub.e)

# Import Private Key
private = None
with open("private.pem", "r") as f:
    private = RSA.import_key(f.read())
print(private.d, private.p, private.q)
assert private.p * private.q == pub.n
```



### Exercise

Suppose we know the public key of RSA  $(n, e)$  where  $n = pq$  with  $p, q$  distinct primes.

1. How to get  $\phi(n)$  if we have  $p$ ?
2. How to get  $d$  if we have  $\phi(n)$ ?
3. How to get  $d$  if we have  $p$ ?

### Exercise


Can we factorize  $n$  if we have  $d$ ?



## Attacking RSA



# Attacks on RSA

A horizontal bar with a teal segment on the left and an orange segment on the right.

Breaking RSA is conjectured to be as hard as integer factorization. On the other hand, if we can factorize large integers, then RSA is broken. Factorizing large integers is hard (NP, BQP). That said, bad implementations of RSA leads to interesting attacks.

There is not enough time to cover every attacks of RSA (even every attacks in the slides), some attacks are included just for the sake of completeness (of RSA and cryptography).



# Brute Force




Note that brute-forcing  $n$  and  $d$  are not practical, nor does brute-force encrypting  $m$  to see if it encrypts to our given ciphertext. However, meet-in-the-middle still outperforms brute force by a little bit.





# Meet in the Middle Attack

A horizontal bar with a teal segment on the left and an orange segment on the right.

Given that the plaintext is at most  $k$  bit (e.g. if we know the ciphertext is an AES key), if  $m = m_1m_2$  where they are roughly the same size (this happens with probability 18% if  $k = 64$  for example), then we can try a meet-in-the-middle attack on  $m_1$  and  $m_2$  ([5]).



The methodology is as follows.

1. Generate two list of integers between 1 to  $2^{\frac{k}{2}}$ , call them  $\{a_i\}$  and  $\{b_i\}$ .
2. Compute  $A_i = a_i^e \pmod{n}$  and  $B_i = b_i^{-e} \pmod{n}$ .
3. calculate  $cB_i \pmod{n} = (mb_i^{-1})^e \pmod{n}$  and check if any  $A_j = cB_i$ .
4. If yes, then  $m = a_i b_j$ .

This allows us to search through the whole message space from requiring  $O(2^k)$  modular exponentiations with brute-force to about  $O(\sqrt{2^k})$  with space complexity  $O(2^k)$ .



# Factoring

Even though we concluded that factoring large integers is hard, some times if one of the factors is small, then we may be able to do it. In sage, there is a factor function that can be used.

There are better algorithms that are better than trial and error.


The general number field sieve is currently the fastest algorithm to factor integers, with running time  $O(e^{(\sqrt[3]{\frac{64}{9}} + o(1))}(\ln n)^{\frac{1}{3}}(\ln \ln n)^{\frac{2}{3}})$ , but it is still very slow.

Some people have already factored some numbers and put it online. [factordb](#), [yafu](#), [Integer factorization calculator](#)

Lesson: Don't use easily factorizable numbers as  $n$ .



# Pollard's $p-1$ algorithm

A horizontal bar with a teal segment on the left and an orange segment on the right.

This algorithm attempts to factorize  $n$  by the assumption that one of the factors  $p$  or  $q$ , say  $p$ , has the property that  $p - 1$  is smooth (even the largest prime factor is small).

There is also another algorithm, Williams's  $p+1$  algorithm, that can factor  $n$  when  $p + 1$  is smooth.



# Common Factors

Consider a large list of public keys generated by the same system. What if there is a bad random number generation, such that some of the keys share the same prime? In other words,

What if  $n_1$  and  $n_2$  distinct public keys of RSA has a common factor  $q$ ?

How do we tell? We do not know  $q$ !



Luckily, if  $n_1$  and  $n_2$  has one common factor  $q$ , then  $\gcd(n_1, n_2) = q$  (in the case that they are a product of two primes). So checking the gcd for all possible pairs of public keys will work. In fact, the Euclidean algorithm works in polynomial time, which is fast.



# Common Modulus Attack

What if there are two pairs of public keys with the same  $n$  but different  $e$ ? Suppose there are keys  $(n, e_1)$  and  $(n, e_2)$  where  $\gcd(e_1, e_2) = 1$ , and we get the ciphertexts of the same unknown message  $m$  using the two encryption. Then we can recover  $m$ . We can find integers  $a$  and  $b$  such that  $ae_1 + be_2 = 1$ . Then let the two ciphertexts be  $c_1 = m^{e_1} \pmod{n}$  and  $c_2 = m^{e_2} \pmod{n}$ . Now  $m^{ae_1+be_2} = m^1 = m$ , so

$$\begin{aligned} c_1^a \cdot c_2^b &= (m^{e_1} \pmod{n})^a (m^{e_2} \pmod{n})^b \\ &= m^{ae_1} m^{be_2} \pmod{n} \\ &= m^{ae_1+be_2} \pmod{n} \\ &= m \end{aligned}$$

Lesson: don't reuse  $n$  with different  $e$ .



## Exercise (CODE BLUE CTF 2017)

1. Common Modulus 1 (98 points)
2. Common Modulus 2 (133 points)
3. Common Modulus 3 (210 points)





# Chinese Remainder Theorem

## Theorem

*Suppose  $n_1, \dots, n_k$  is pairwise coprime ( $\gcd(n_i, n_j) = 1 \ \forall i \neq j$ ), then the system of congruence equations*

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \\ \vdots \\ x \equiv a_k \pmod{n_k} \end{cases}$$

*has a unique solution  $x^* \pmod{n_1 n_2 \dots n_k}$ .*



# Broadcast Attack

Suppose a plaintext is encrypted  $k$  times, where all the public exponent  $e$  are the same, and  $k \geq e$ . i.e. the following holds:

$$\begin{cases} m^e \equiv a_1 \pmod{n_1} \\ m^e \equiv a_2 \pmod{n_2} \\ \vdots \\ m^e \equiv a_k \pmod{n_k} \end{cases}$$

Then by Chinese Remainder Theorem, we get  $x^* \bmod n_1 n_2 \cdots n_k$  that satisfies all the equations above. Then  $x^* \equiv a_i \pmod{n_i}$  for any  $i$ , at the same time  $m^e \equiv a_i \pmod{n_i}$  obviously.



But  $m^e = \underbrace{m \cdot m \cdots m}_e \leq \underbrace{m \cdot m \cdots m}_k < n_1 \cdot n_2 \cdots n_k$ , so  $x^* = m^e$   
(without mod), since the solution is unique mod  $n_1 \cdots n_k$ . So we  
get  $m = \sqrt[e]{x^*}$ .

### Exercise

There is one implicit assumption on  $n_i$  we made on the derivation of the broadcast attack. What is it? Why can it be assumed?

Also note that we only have  $m < n_i$  as an assumption ( $m = O(n_i)$ ). With this weak assumption we need the full  $e$  equations.



# Lesson



1. Do not reuse  $e$ ...?
2. Instead, use large  $e$ ! (e.g. 65537)
3. Use random padding to randomize the ciphertext!



# Chinese Remainder Theorem in Sage

For example, if we want to solve the following system of congruent equations

$$\begin{cases} x \equiv 2 \pmod{3} \\ x \equiv 3 \pmod{5} \\ x \equiv 2 \pmod{7} \end{cases}$$

, then in sage, we can do

```
CRT_list([2,3,2],[3,5,7])
```

to get the solution mod  $3 \times 5 \times 7 = 105$ .

## Exercise

What are **all** the possible solutions to the above equation?



# Fermat Factorization

What if the primes  $p$  and  $q$  are very close? i.e.  $|p - q|$  is small?

Recall from secondary school that

$$(p + q)^2 = p^2 + 2pq + q^2$$

$$(p - q)^2 = p^2 - 2pq + q^2$$

$$\text{so } (p + q)^2 - (p - q)^2 = 4pq$$

$$\left(\frac{p + q}{2}\right)^2 - \left(\frac{p - q}{2}\right)^2 = n$$

$$\left(\frac{p + q}{2}\right)^2 - n = \left(\frac{p - q}{2}\right)^2$$



Let  $x = \left(\frac{p+q}{2}\right)^2$  and  $y = \left(\frac{p-q}{2}\right)^2$ , we obtain the following Diophantine equation

$$x^2 - n = y^2$$

If we can solve the equation for  $x$  and  $y$ , we can obtain

$$(p, q) = (x + y, x - y).$$

If we brute force to check if  $x^2 - n$  is a perfect square, we can get  $y$ . By AM-GM inequality, we know that  $x^2 > n$ , so brute-forcing  $x$  from  $\lceil \sqrt{n} \rceil$  works.



The pseudocode for Fermat factorization is as follows:

```
1: function Fermat( $n$ ):  
2:    $x \leftarrow \lceil \sqrt{n} \rceil$   
3:    $y^2 \leftarrow x^2 - n$   
4:   while  $y^2$  is not a perfect square do  
5:      $x \leftarrow x + 1$   
6:      $y^2 \leftarrow x^2 - n$   
7:   return  $p = x + \sqrt{y^2}, q = x - \sqrt{y^2}$ 
```





### Exercise

Implement Fermat Factorization in python, sage, or any language of choice.

### Exercise

“Zygote RSA” in Firebird intro CTF

Lesson: Don't use close primes.



# Exercise



SECCON 2017: Ps and Qs (200 points)

Description: Decrypt it.

Ps and Qs (password: seccon2017)



# Weiner's Attack

Weiner attacks can find  $d$  if the private exponent  $d$  is small.

(when  $d < \frac{1}{3}N^{\frac{1}{4}}$ ). It works by the fact that if the above fact holds,

then  $\frac{k}{d}$  is a convergent of the continued fraction expansion of  $\frac{e}{N}$ , where  $k$  is an integer such that  $de = k\phi(n) + 1$ . So by looking at each convergent of  $\frac{e}{N}$  (which we know), we can try to recover  $\frac{k}{d}$  and thus  $d$ . (Will not talk about here)

Note: How do we know that  $d$  is small?

An improvement was made to make the bound  $d < N^{0.292}$  by Boneh and Durfee, but that made use of Coppersmith method (will mention later).

Lesson: Don't use large  $e$ .



# Decryption Oracles

Imagine a situation where a service can decrypt any message besides one particular ciphertext. Can we recover the plaintext?


Suppose  $c = E_k(m)$ , and we can obtain any  $x = D_k(y)$  except when  $y = c$ . Then we can craft a ciphertext  $\tilde{c} = c \cdot E_k(2) = E_k(m)E_k(2) = E_k(2m)$ , and get the plaintext  $2m \pmod{n}$ . Now we can obtain  $m$  by dividing by 2 directly if  $2m < n$ .

## Exercise

What if  $2m \geq n$ ?



# Discussion

A horizontal bar with a teal segment on the left and an orange segment on the right.

Again this attack works because of the multiplicative property of RSA. This property is also known as malleability, where we can modify the ciphertext to produce another ciphertext that are related to the original plaintext. Recall the bit-flipping and padding oracle of CBC mode!



This attack is a type of **chosen-ciphertext attack**.

### Definition

A **chosen-ciphertext attack** (CCA) is where we can choose some specially crafted ciphertext, have it decrypted and obtain the plaintext.



# LSB Oracle

In fact, if an oracle only gives the parity (odd/even, or equivalently the LSB), then we can still obtain the original plaintext!

Note that  $n$  is odd (assume  $n$  is a product of odd primes) while  $2m$  is always even and so are  $2^k m$  for all natural number  $k$ . If

$D_{k'}(E_k(2m))$  is even, then we can conclude that  $0 < 2m < n$ . If  $D_{k'}(E_k(2m))$  is odd, we get  $2m > n$ . Since  $m < n$  we will obtain  $2n > 2m > n$ . We can then try to ask for

$D_{k'}(E_k(4m)), \dots, D_{k'}(E_k(2^k m))$  to recover the plaintext.

## Exercise

Generalize the result above to  $2^k m$  (perhaps by induction).



# Lesson



1. Don't let users decrypt anything.
2. Pad your messages.





# Small Messages

We recall that the encryption of RSA is

$$E_k(m) = m^e \pmod{n}$$

If  $e$  is small (e.g. 3) and  $m$  is small, then we may have that  $m^e < n$ , so  $m^e \pmod{n} = m^e$ . This means that we can just take the  $e$ -th root to recover the message.

In practice,  $e$  is larger than 3. Popular ones include 257 and 65537 so  $m^e$  is much larger than  $n$ .



## Existing RSA Implementation

We have seen that missing several details (like small  $e$ , bad primes, small messages) can break RSA. So in real life, RSA is not used directly.

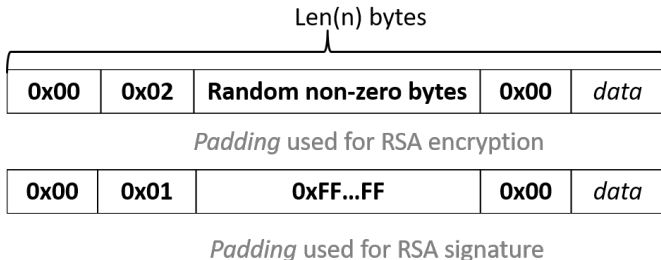


Figure: Diagram for PKCS1 v1.5



Why random bytes are used for padding?



# Coppersmith Method

## Theorem (Coppersmith-Howgrave-Graham<sup>1</sup>)

*Let  $f(x)$  be a (monic) polynomial of degree  $d$  with integer coefficients. Then if there exists integer(s)  $x$  such that  $f(x) \equiv 0 \pmod{N}$  and  $|x| < N^{\frac{1}{d}}$ , then such integers can be found in polynomial time of  $\log N$  and  $d$ .*

The member function `small_roots` of the class `sage.rings.polynomial.polynomial_modn_dense_ntl`. `Polynomial_dense_mod_n` in sage can be used to compute such integers.

---

<sup>1</sup>Howgrave-Graham is one person, not two people named Howgrave and Graham.



# Stereotyped Message

Let  $e = 3$ . Say we have a small message  $m$  to encrypt, and  $m < N^{\frac{1}{3}}$ .

If we use textbook RSA, the plaintext can be directly recovered by taking cube root. Suppose we pad the message with '1' bits on the left until the padded message has about the same size as  $N$ . Then we have the equation

$$(P + m)^3 \equiv c \pmod{N}$$

where we use  $P$  to denote the padding, so  $P = \underbrace{111 \dots 1}_{\lfloor \log_2 N \rfloor - k} \underbrace{00 \dots 0}_k$ .



Rearrange to get

$$(P + m)^3 - c \equiv 0 \pmod{N}$$

which is a cubic polynomial! Check to see the condition of Coppersmith's theorem is satisfied, so we can apply that to recover the plaintext.



# Håstad's broadcast attack

How about when the message is large? If the padding is linear (perhaps just append or prepend **known** bits), given at least  $e$  ciphertexts from the same plaintext, we can still recover the message.

## Theorem

*Given  $N_1, \dots, N_k$  are pairwise coprime integers, and  $g_i(x)$  be polynomials of degree at most  $q$ . If  $M < \min\{N_1, \dots, N_k\}$  and  $g_i(M) \equiv 0 \pmod{N_i}$  for each  $i$ , and  $k > q$ , then there exists an efficient algorithm to compute  $M$ .*



## Proof

Use Chinese remainder theorem to construct  $g(x)$  such that  $g(M) \equiv 0 \pmod{N_1 N_2 \cdots N_k}$ , then  $M$  and  $g$  satisfies the requirement of Coppersmith's theorem. Use Coppersmith's method to recover  $M$ . □





# Lesson



1.  $e = 3$  is really really  $\dots$  really bad (without random padding)!

$10^{10^{10^{10^{\dots}}}}$

2. Use random padding for encryption!



# Bleichenbacher's Attack ([7])


Recall the PKCS#1 v1.5 padding. Suppose an oracle returns if a decrypted message has valid padding or not, then we are able to recover the message.

We use our decryption oracle idea, except now there is an approximately  $2^{-17}$  to  $2^{-15}$  probability that a decryption message conforms with the padding scheme (assume random message from a uniform distribution). So we need a lot of trials to decrypt, still it is possible.

This happens in real life in SSL ([Return of Bleichenbacher's Oracle Threat \(ROBOT\)](#)).



## Remark



The Coppersmith method made use of the idea of **lattices** in mathematics. In fact, lattice is a very important concept in cryptography, as encryptions based on lattice problems are currently resistant against quantum computer attacks.

Lattice-based cryptography is a type of **post-quantum cryptography**. On the other hand, **lattice basis reduction** algorithms like LLL is a useful tool for cryptanalysis.

There are some other attacks of RSA that uses the Coppersmith method, notably we can factorize  $N$  given (half of the bits) of one of the prime factors.



# Boneh-Durfee Attack

Let's discuss the application of Coppersmith's attack to improve the bound of Wiener's attack.

Recall  $de = 1 + k\phi(n)$ . Since  $\phi(n) = n - p - q + 1$  we get  $de = 1 + k(n - p - q + 1)$ . Now reduce modulo  $e$  and rearrange to get  $2k\left(\frac{n+1}{2} - \frac{p+q}{2}\right) + 1 \equiv 0 \pmod{e}$ . This way we get a bivariate polynomial  $f(x, y) = 2x(A + y) + 1 \pmod{e}$  with small roots  $\left(k, -\frac{p+q}{2}\right)$ . We can use Coppersmith's method (provided that  $d < N^{0.292}$ ) to recover the small root and recover  $d$ .



## Variants of RSA



# Variants of RSA

There are many variants of RSA. Before the end (or read at your own leisure depending on if the speaker can finish the material in 2 hours), we shall look into some popular ones.

Recall the public exponent is usually chosen to be in the form  $2^k + 1$  to reduce the time of encryption. This makes encryption easy, but not decryption (since we have no control of  $d$ ).



# RSA-CRT

Instead of doing  $D_k(c) = c^d \pmod{n}$ , upon key generation, we calculate

$$d_p \equiv e^{-1} \pmod{p-1}$$

$$d_q \equiv e^{-1} \pmod{q-1}$$

Then to decrypt ciphertexts, one calculates

$$m_p \equiv c^{d_p} \pmod{p}$$

$$m_q \equiv c^{d_q} \pmod{q}$$

and combine the two together by Chinese Remainder Theorem to obtain the plaintext.



### Exercise

Verify that this decryption scheme works. What do we need to store as private key?

### Exercise (Fault Attack on RSA-CRT)

Suppose in the decryption process, one of the two congruences gave the wrong result (perhaps because of cosmic rays). Explain how to factor  $N$  given the wrong decrypted text and the original undecrypted text.

Note: this situation arises in digital signatures.

### Exercise

This decryption scheme gives a  $4\times$  speed-up compared to usual textbook decryption. Why?





# Carmichael Function

Some will use the Carmichael function instead of the Euler phi function to generate the exponents.

## Definition

The **Carmichael totient function** is a function  $\lambda : \mathbb{N} \rightarrow \mathbb{N}$ , defined to give the smallest integer  $\lambda(n)$  such that  $a^{\lambda(n)} \equiv 1 \pmod{n}$  for all  $a$  coprime to  $n$ .

## Theorem

*If  $p$  and  $q$  are prime numbers, then  $\lambda(pq) = \text{lcm}(p-1, q-1)$ , where  $\text{lcm}(a, b)$  gives the **least common multiple** of the two integers.*

The key generation is the same except we replace all instances of  $\phi(n)$  with  $\lambda(n)$ .



# Multiprime RSA

Why not use  $n = p_1 p_2 \cdots p_r$  for RSA?

For  $N$ -bit RSA, we can choose  $r$  primes that are about  $N/r$  bits long, and the rest of key generation, encryption/decryption are the same with appropriate adjustments. RSA-CRT gives a even larger speed-up for decryption compared to the usual RSA.

## Exercise

What is  $\phi(p_1 p_2 \cdots p_r)$ ?



# Prime powers

Why not use  $n = p^r q$  (Takagi-RSA[8]) or even  $n = p^r q^s$  ([9]) ?

For decryption, we can use Hensel's lifting lemma along with RSA-CRT to speed up the calculation.

## Exercise

Show that these  $n$  can still define a valid encryption/decryption function.



# pqRSA([10])

Quantum computers can factorize integers quickly. How about use insanely long keys?

Post-quantum RSA (pqRSA) proposes using 1TB (“partically” would be 1GB) of multiprime key for RSA.

## Remark

This is regarded as a joke in the community.



## General Formulation of RSA



# General RSA

Most RSA questions we see are just RSA with integers mod  $n$ . However using some general constructions of math we are able to do RSA with other mathematics. More specifically we just need to have some kind of Euler totient function along with a formulation of “Euler’s Theorem”.

Let’s use an example of the polynomials.



# Polynomials over $\mathbb{Z}/p\mathbb{Z}$

Let  $m > 1$  be a positive integer. Then we can talk about a (univariate) polynomial over  $\mathbb{Z}/m\mathbb{Z}$  as the formal expression  $f(x) = a_n x^k + a_{k-1} x^{k-1} + \dots + a_1 x + a_0$ , where  $a_k \neq 0$  and  $a_i \in \mathbb{Z}/m\mathbb{Z}$  for all  $i$ . We say the degree of  $f(x)$  is  $\deg(f) = k$ .

We can add, subtract and multiply polynomials in  $\mathbb{Z}/m\mathbb{Z}$ . Here we are interested in polynomials over  $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$  where  $p$  is a prime.

The set (in fact ring) of polynomials over  $\mathbb{F}_p$  can be denoted as  $\mathbb{F}_p[x]$ .



# Irreducible

## Definition (Irreducible Polynomial)

We say a polynomial  $f(x)$  that is not zero nor degree-zero is irreducible if whenever  $f(x) = g(x)h(x)$ , one of  $g(x)$  and  $h(x)$  must be a degree-zero polynomial.

This sounds like the definition of prime numbers, but this is just due to the fact that for integers, irreducible integers are equivalent to prime integers.

## Definition (Prime Polynomial)

A polynomial  $f(x)$  that is not zero nor degree-zero is prime if whenever  $f(x) | g(x)h(x)$ , either we have  $f(x) | g(x)$  or  $f(x) | h(x)$ .





The set of polynomials modulo  $n(x)$  consists of polynomials with degree less than  $\deg(n)$ . So the number of elements in the set is  $m^{\deg(n)}$ . We denote the set of polynomials mod  $n(x)$  as

$\mathbb{F}_p[x]/(n(x))$  (or  $\frac{\mathbb{F}_p[x]}{n(x)\mathbb{F}_p[x]}$ ) (c.f. integers  $\mathbb{Z}$  and integers mod  $n$   $\mathbb{Z}/n\mathbb{Z}$ ).



# Modulo Reduction

We know that we can divide a polynomial to get a quotient and remainder, so we can certainly define the modulo operation for polynomials.

## Definition

For non-zero polynomial  $f(x)$  and  $n(x)$ , we can write

$$f(x) = q(x)n(x) + r(x)$$

where  $\deg(r) < \deg(n)$ . We can also write

$$f(x) \equiv r(x) \pmod{n(x)}$$



# “Modular Inverse”

In the mod  $n(x)$  world, we can talk about the multiplicative inverse of polynomial  $f(x) \bmod n(x)$ . It is just a polynomial  $g(x) \bmod n(x)$  such that  $f(x)g(x) \equiv 1 \pmod{n(x)}$ .

## Theorem

*Every non-zero polynomial mod  $n(x)$  has a multiplicative inverse.*



# “Phi function”

We can talk about a similar construct to  $\phi(n)$  for numbers. We can just define  $\phi(f(x))$  to be the polynomials in  $\mathbb{F}_p[x]/(f(x))$  that has multiplicative inverse. In this case, if  $f(x)$  is irreducible, then  $\phi(f(x)) = m^{\deg(f)} - 1$ .

If  $n(x) = f(x)g(x)$  where  $f$  and  $g$  are irreducible, then  $\phi(n(x)) = (m^{\deg(f)} - 1)(m^{\deg(g)} - 1)$ . See where this is going?



# “Euler’s Theorem”

In the case of polynomials, the “Euler’s theorem” Still holds for the “phi function”.

Theorem (Lagrange’s theorem on Polynomial mod  $n(x)$ )

*If  $\gcd(g(x), n(x)) = 1$ , then*

$$g(x)^{\phi(n(x))} \equiv 1 \pmod{n(x)}.$$



# RSA on Polynomials

Choose a prime number  $r$ , and a polynomial  $n(x) = p(x)q(x)$  where  $p(x), q(x) \in \mathbb{F}_r[x]$  are irreducible. Choose  $e$  be an integer coprime with  $\phi(n(x))$ , and compute the private key  $d$  such that  $de \equiv 1 \pmod{\phi(n(x))}$ .

Now we can encrypt a polynomial  $f(x) \in \mathbb{F}_p[x]/(n(x))$  by computing  $E_k(f(x)) = f(x)^e \pmod{n(x)}$ . The decryption is just replacing  $e$  with  $d$ . “Euler’s theorem” implies that this encryption process is correct.

If we can factor polynomials into irreducible parts, then we can compute  $\phi(n(x))$  and recover the message.




### Exercise

Which attack(s) we mentioned in this training work for polynomial RSA?



# Ring of Integers








RSA can also be defined in rings of integers  $\mathcal{O}_K$  where  $K$  is a number field, let's say  $\mathbb{Q}[\alpha]$  for some root of polynomial  $\alpha$ . We can talk about irreducible elements (using field norm), modulo operation, gcd, phi function and Euler's Theorem. Again the security depends on the (in)ability to factor elements into irreducible parts.

However since this section would require some algebra, so we refrain from discussing this to spare the readers from the pain of learning number theory.








# Reference I

- 
-  D. R. Stinson and M. Paterson, *Cryptography: theory and practice*.  
CRC press, 2018.
  -  J. Hoffstein, J. Pipher, J. H. Silverman, and J. H. Silverman, *An introduction to mathematical cryptography*, vol. 1.  
Springer, 2008.
  -  S. D. Galbraith, *Mathematics of public key cryptography*.  
Cambridge University Press, 2012.
  -  D. Boneh *et al.*, “Twenty years of attacks on the rsa cryptosystem,” *Notices of the AMS*, vol. 46, no. 2, pp. 203–213, 1999.







## Reference II



-  D. Boneh, A. Joux, and P. Q. Nguyen, “Why textbook elgamal and rsa encryption are insecure,” in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 30–43, Springer, 2000.
-  R. L. Rivest and R. D. Silverman, “Are strong primes needed for rsa?,” in *IN THE 1997 RSA LABORATORIES SEMINAR SERIES, SEMINARS PROCEEDINGS*, 1999.
-  D. Bleichenbacher, “Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1,” in *Annual International Cryptology Conference*, pp. 1–12, Springer, 1998.



## Reference III

- 
-  T. Takagi, “Fast rsa-type cryptosystem modulo  $p^k q$ ,” in *Annual International Cryptology Conference*, pp. 318–326, Springer, 1998.
  -  S. Lim, S. Kim, I. Yie, and H. Lee, “A generalized takagi-cryptosystem with a modulus of the form  $p^r q^s$ ,” in *International Conference on Cryptology in India*, pp. 283–294, Springer, 2000.
  -  D. J. Bernstein, N. Heninger, P. Lou, and L. Valenta, “Post-quantum rsa,” in *International Workshop on Post-Quantum Cryptography*, pp. 311–329, Springer, 2017.