SageMath 3: RSA Public Key Cryptosystem

Cheng-Hsin Hsu

National Tsing Hua University

Department of Computer Science

How to Secretly Send Messages

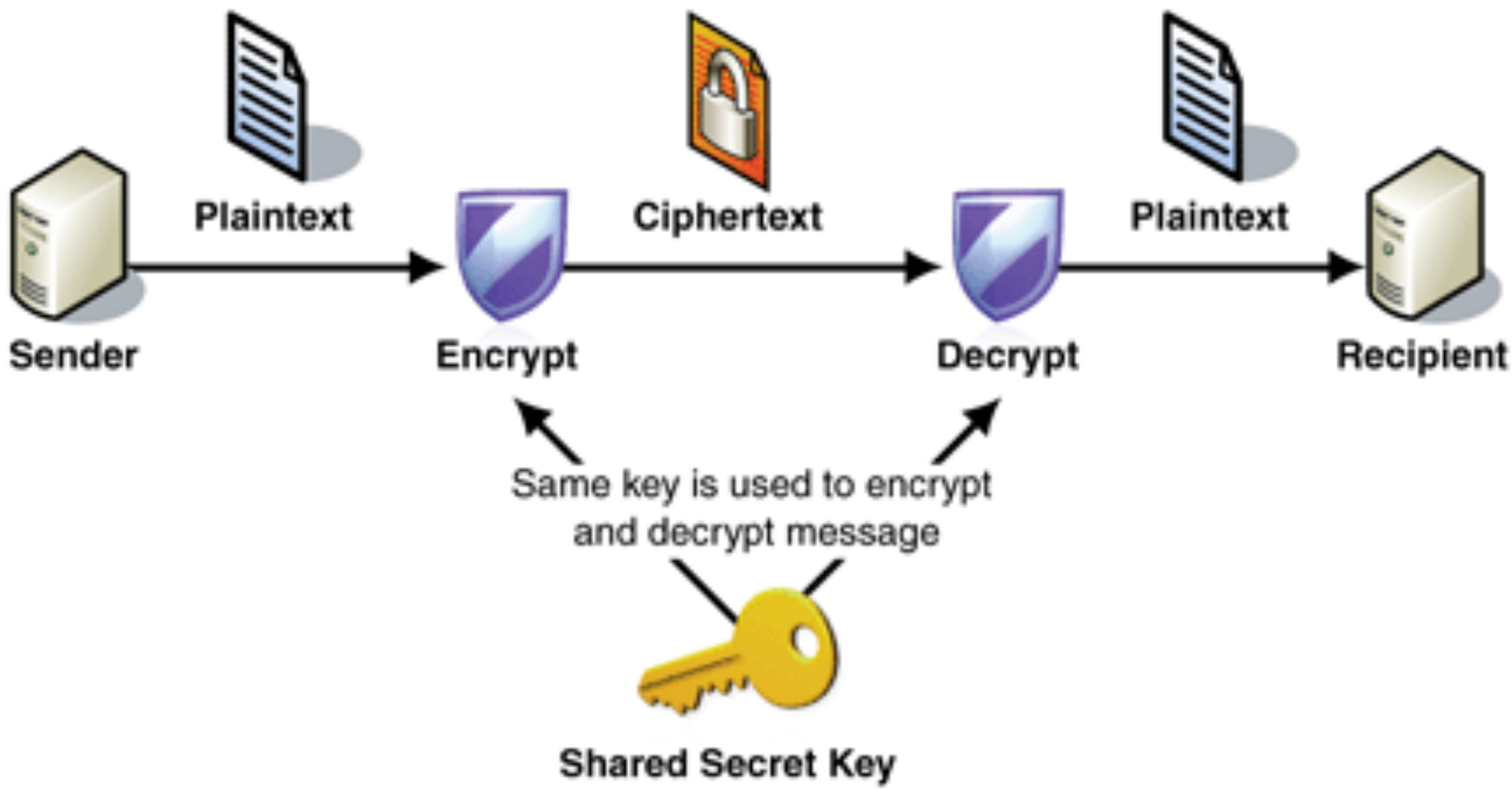
- Plaintext: human-readable messages
- Ciphertext: scrambled message
- Encryption: plaintext → ciphertext
- Decryption: ciphertext → plaintext

Naïve Way: ASCII Encoding

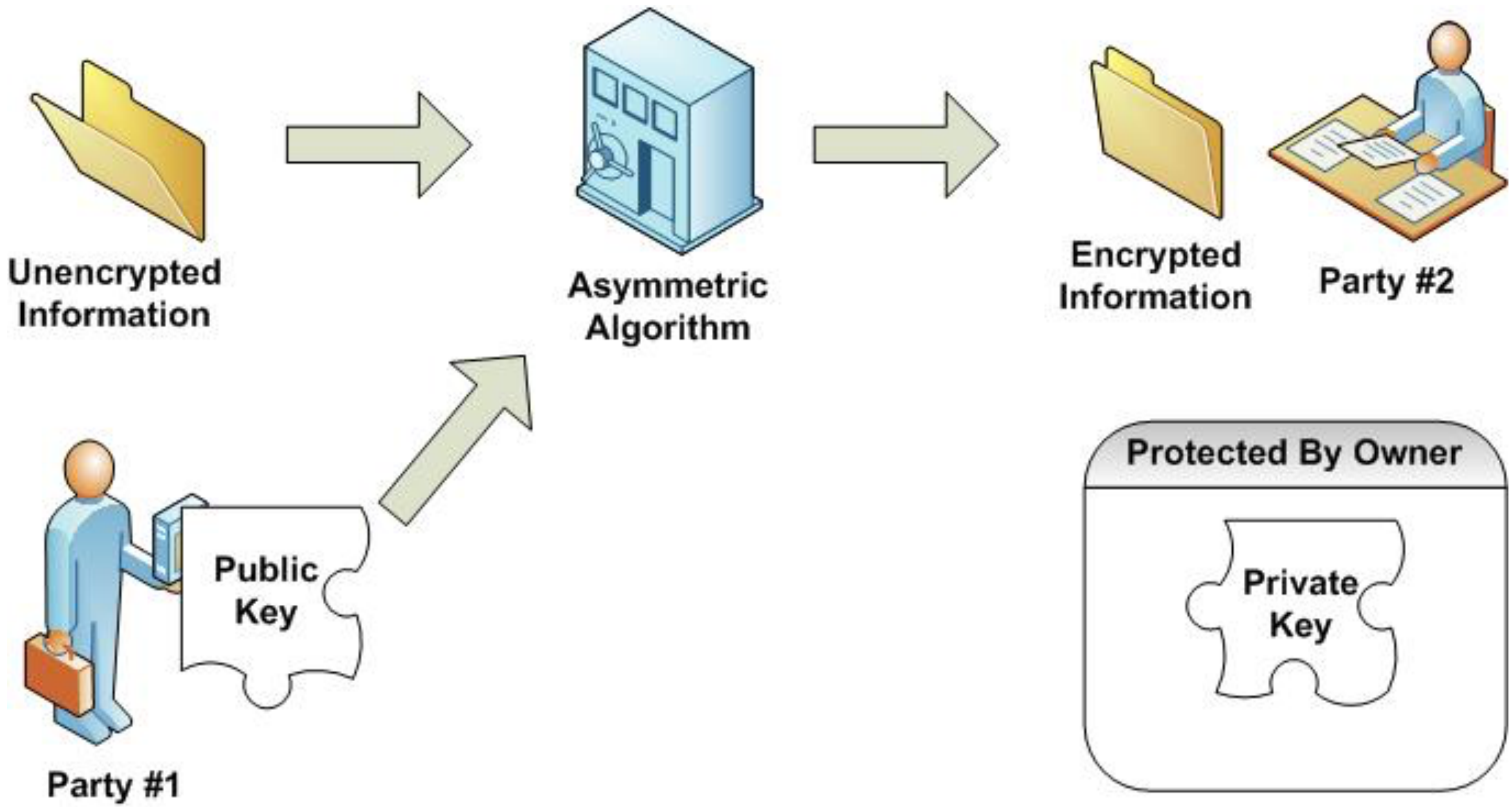
- Let $\Sigma = \{A, B, \ldots, Z\}$ be the English (uppercase) alphabet ← plaintext
- Let $\Phi = \{65, 66, \ldots, 90\}$ be the ASCII encodings, where $f : \Sigma \rightarrow \Phi$

- Example: "SCIENCE" → 83677369786769
- But it's too weak

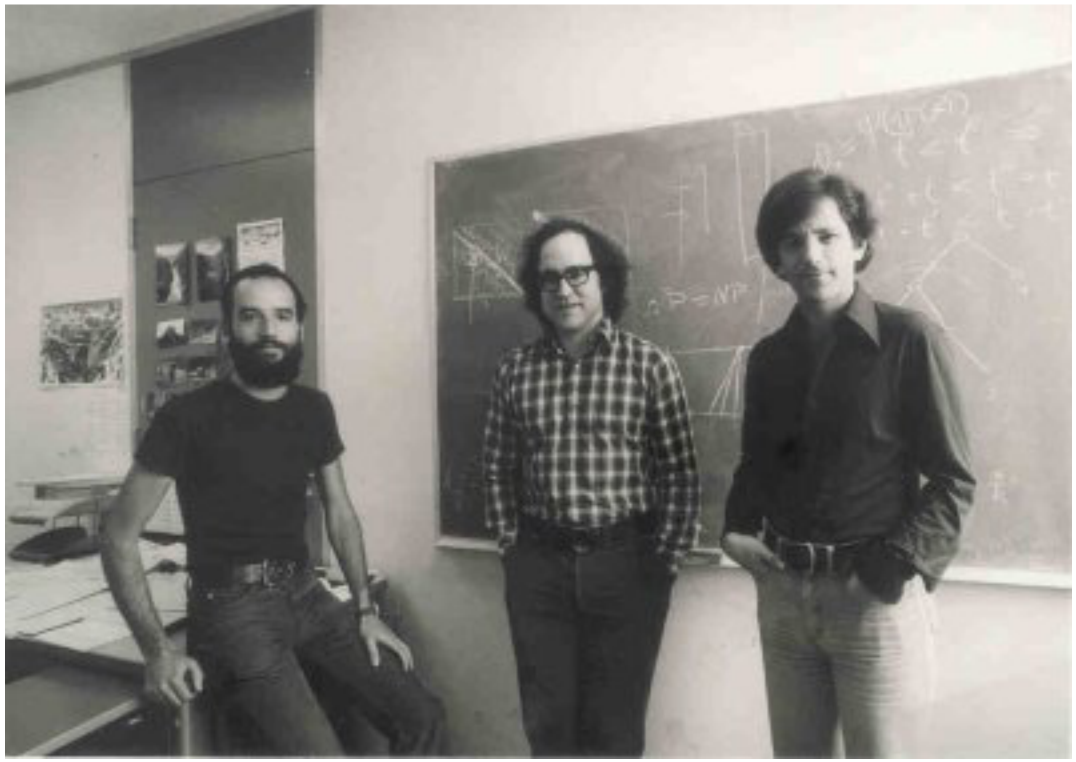
Symmetric Cryptography



Asymmetric Cryptography



A Popular Asymmetric Algorithm: RSA



R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. ACM Communications, 21, 2 (February 1978), 120-126.

RSA Pseudocode

1. Choose two huge primes *p* and *q*, and let *n*=*pq*
2. Let $e \in \mathbb{Z}$ be positive s.t. $\gcd(e, \phi(n)) = 1$
3. Find a $d \in \mathbb{Z}$ so that $de \equiv 1 \pmod{\phi(n)}$
4. Public key (*n*, *e*), private key (*p*, *q*, *d*)
5. For any integer *m* < *n*, encrypt *m* by $c \equiv m^e \pmod{n}$
6. Decrypt *c* using $m \equiv c^d \pmod{n}$

Let's try to walk through this in Sage!

Mersenne Primes

- Studied by Marin Mersenne in 17th century
- Power of two minus 1: $M_m = 2^m - 1$
- If M_m is a prime, the it's called Mersenne primes
  - Sounds like a good way to create huge primes
  - is_prime(.) tells us if a number is prime
- Alternatively, we may use random_prime(...)

Generate the Primes for Keys

```
sage: p = 2^31 - 1
sage: is_prime(p)
True
sage: q = 2^61 -1
sage: is_prime(q)
True
sage: n = p*q
sage: n
4951760154835678088235319297
```

BTW, far-apart p and q is very bad choices in the sense of security

RSA Pseudocode, Step 2

1. Choose two huge primes *p* and *q*, and let *n*=*pq*
2. Let $e \in \mathbb{Z}$ be positive s.t. $\gcd(e, \phi(n)) = 1$
3. Find a $d \in \mathbb{Z}$ so that $de \equiv 1 \pmod{\phi(n)}$
4. Public key (*n*, *e*), private key (*p*, *q*, *d*)
5. For any integer *m* < *n*, encrypt *m* by $c \equiv m^e \pmod{n}$
6. Decrypt *c* using $m \equiv c^d \pmod{n}$

Find a Coprime of Euler Phi

- We learned how to calculate euler_phi(.)
- Let's randomly pick a number < phi, and wish they are coprime
- We stop only when we find a coprime *e*
  - Usage of while loop....

While-Loop to Find *e*

```
sage: phi=euler_phi(n); phi
4951760152529835076874141700
sage: e=int(random() * (phi-1)) + 1
sage: while gcd(e, phi) !=1 :
....:     e=int(random() * (phi-1)) + 1
....:
sage: e
3093458420861290024932474881
```

What does this do?

RSA Pseudocode, Step 3

1. Choose two huge primes *p* and *q*, and let *n*=*pq*
2. Let $e \in \mathbb{Z}$ be positive s.t. $\gcd(e, \phi(n)) = 1$
3. Find a $d \in \mathbb{Z}$ so that $de \equiv 1 \pmod{\phi(n)}$ ←
4. Public key (*n*, *e*), private key (*p*, *q*, *d*)
5. For any integer *m* < *n*, encrypt *m* by $c \equiv m^e \pmod{n}$
6. Decrypt *c* using $m \equiv c^d \pmod{n}$

How to Find *d*?

- Sounds tricky: $de \equiv 1 \pmod{\phi(n)}$
  - $\phi(n) | de - 1$
  - or $de - 1 = k \times \phi(n)$ for some integer *k*
  - or $de - k\phi(n) = 1$
- Think again
  - What are given? ← *e* and *phi*
  - What do we want to determine? ← *d* and *k*
- How can we find two integers *d* and *k?*
  - *Recall that e and phi are coprime*

Extended Euclidean Algorithm

- We know $\gcd(a,b) = xa + yb$ for some *x* and *y*
- Sage command xgcd(a, b) returns *(gcd(a,b), x, y)* as a 3-tuple

sage: tuple=xgcd(e, phi); tuple

(1, -16522784699765489228624745 79,
103220967678441436335607125 3)

sage: d = Integer(mod(tuple[1], phi)); d

3299481682553286154011667121 ← Found our d

sage: mod(d*e, phi)

1 ← Validate d

RSA Pseudocode, Step 4

1. Choose two huge primes *p* and *q*, and let *n*=*pq*
2. Let $e \in \mathbb{Z}$ be positive s.t. $\gcd(e, \phi(n)) = 1$
3. Find a $d \in \mathbb{Z}$ so that $de \equiv 1 \pmod{\phi(n)}$
4. Public key (*n*, *e*), private key (*p*, *q*, *d*)
5. For any integer *m* < *n*, encrypt *m* by $c \equiv m^e \pmod{n}$
6. Decrypt *c* using $m \equiv c^d \pmod{n}$

Public and Private Keys

sage: (n,e) ← Public Key

(49517601548356780882353l9297,
30934584208612900249324748 81)

sage: (p,q,d) ← Private Key

(2147483647, 2305843009213693951,
3299481682553286154011667121)

RSA Pseudocode, Step 5

1. Choose two huge primes *p* and *q*, and let *n*=*pq*
2. Let $e \in \mathbb{Z}$ be positive s.t. $\gcd(e, \phi(n)) = 1$
3. Find a $d \in \mathbb{Z}$ so that $de \equiv 1 \pmod{\phi(n)}$
4. Public key (*n*, *e*), private key (*p*, *q*, *d*)
5. For any integer *m* < *n*, encrypt *m* by $c \equiv m^e \pmod{n}$
6. Decrypt *c* using $m \equiv c^d \pmod{n}$

Encrypt the Message (and Fail)

- "SCIENCE" → m=83677369786769
- $c \equiv m^e \pmod{n}$

e=3093458420861290024932474881

```
sage: m=83677369786769
sage: c=mod(m^e, n)
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-19-c5605db94841> in <module>()
----> 1 c=mod(m**e, n)
/usr/lib/sagemath/local/lib/python2.7/site-packages/sage/rings/
integer.so in sage.rings.integer.Integer.__pow__ (sage/rings/integer.c:
14001)()
RuntimeError: exponent must be at most 9223372036854775807
```

Repeated Squaring

- Start from $d = 1$
- Convert b into binary $(b_1, b_2, \ldots, b_k)$
- Iterate i from 1 to k
  - $d = d * d \bmod n$ ← Move 1 digit toward left
  - If $b_i = 1$, let $d = d * a \bmod n$

↑

If there is a 1, multiply by a

Example of Repeated Squaring

- Derive 3^6 ← $6 = (110)_2$
- Step 1: d=1
- Step 2: i=1, d=1*1 = 1, d = 1*3 = 3
- Step 3: i=2, d=3*3 = 9, d=9*3 = 27
- Step 4: I=3, d=27*27=729

- Note that I ignore modulus here for brevity

Repeated Square Function

- Save the following code as rsmod.sage ← Uah, pay attentions to indents, like all python sources
- Load it using %runfile rsmod.sage
- Test it

```
def rsmod(a, b, n):
   d=1
   for i in list(Integer.binary(b)):
      d=mod(d*d, n)
      if Integer(i) == 1:
         d = mod(d*a, n)
   return Integer(d)
```

```
sage: %runfile rsmod.sage
sage: rsmod(3,6,100000)
729
```

Now We are Back on Track

- Use *e* and *n* (=*pq*) to encrypt *m* into *c*

sage: c=rsmod(m, e, n)

sage: c

1406082576299748012744893983

- Last step, decode c using *d* and *n*

sage: m2=rsmod(c, d, n); m2==m

True

Recap: RSA Pseudocode

1. Choose two huge primes *p* and *q*, and let *n*=*pq*
2. Let $e \in \mathbb{Z}$ be positive s.t. $\gcd(e, \phi(n)) = 1$
3. Find a $d \in \mathbb{Z}$ so that $de \equiv 1 \pmod{\phi(n)}$
4. Public key (*n*, *e*), private key (*p*, *q*, *d*)
5. For any integer *m* < *n*, encrypt *m* by $c \equiv m^e \pmod{n}$
6. Decrypt *c* using $m \equiv c^d \pmod{n}$

We have done this!

Naïve Way to Break It

- Figure out the *p* and *q* values. But, how hard is factorization?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
CPU times: user 0.01 s, sys: 0.00 s, total: 0.01 s
sage: time factor(random_prime(2^64)*random_prime(2^64))
CPU times: user 0.05 s, sys: 0.00 s, total: 0.05 s
sage: time factor(random_prime(2^96)*random_prime(2^96))
CPU times: user 3.54 s, sys: 0.04 s, total: 3.58 s
sage: time factor(random_prime(2^128)*random_prime(2^128))
CPU times: user 534.39 s, sys: 0.12 s, total: 534.51 s
```

← Growing into something

- Well there are many primes between 2^{511} and 2^{512} ← Attackers cannot be that lucky

Flawed Random Number Generators

- 1995 Goldberg-Wagner: During any particular second, the Netscape browser generates only about 2^{47} possible keys
- 2008 Bello: Debian and Ubuntu generate $<2^{20}$ possible keys for SSH, OpenVPN, etc
- What we can do is:
  - Generate many private keys on a device
  - Check if any of these private keys divide *n*
  - Finding *p* (and *q*) is no longer impossible

Pollard's *p-1* Attack

- Due to John Pollard in 1974
- Only work on special primes ← Smooth primes
- A number is *k*-smooth if all of its prime factors are smaller than *k*
- Example: 10, 100, and 2^{1024} are all 6-smooth, but 14 is not

Background of Pollard's *p-1* Attack

- RSA's n can be readily factorized if $p-1$ or $q-1$ are smooth ← only have small factors
  - Wait, but we don't know p nor q, right? Indeed …
- Checking if an integer k is B-smooth may be to computationally demanding
  - Compare it against if $k|B!$

Integer k Divides $B!$

Lemma: $k \mid B!$ implies k is B-smooth

Proof:

- Assume k is not B-smooth, then there existing an integer $f|k$, where $f > B$.
- f does not divide any $b' <= B$.
- Since we know $p|ab$ iff $p|a$ or $p|b$, k does not divide $B!$ for sure.

Note: the converse is false, proof is left as exercise

Fermat's Little Theorem

Theorem: Given a prime number *p*, and any $a \not\equiv 0 \pmod{p}$, we know $a^p \equiv a \pmod{p}$

or $a^{p-1} \equiv 1 \pmod{p}$

Proof:

The first *p-1* positive multiples of *a* are: *a*, *2a*, *3a*, ..., (*p-1*)*a*. These multiples are all distinct, because if *xa* = *ya* (mod *p*), we know *x*=*y* (since *p* is a prime).

Fermat's Little Theorem (cont.)

The *p-1* multiples are congruent to *1, 2, …, p-1*, in some order (the precise permutation is not important). Let's multiply all of them together and we have $a \cdot 2a \cdots (p-1)a = 1 \cdot 2 \cdots (p-1) \pmod p$, and then $a^{p-1}(p-1)! = (p-1)! \pmod p$. Getting rid of (*p-1*)*!* at both sides yields the theorem.

How Fermat's Little Theorem Helps?

- Say *p-1* | *B*!, there is a *k* so that *k*(*p-1*) = *B*!
- Then we have

$$2^{B!} = 2^{k(p-1)} = (2^{p-1})^k \equiv 1^k \equiv 1 \pmod{p}$$

- Or $c = (2^{B!} - 1)$ is a multiple of *p*
  - Both in ordinary integers and under mod *p*
  - I skip some technical details
- OK. What I'm talking about? Since *n=pq*, a multiple of *p*; gcd(*c*, *n*) is a multiple of *p*

The Pollard's *p-1* Attack

- We compute $c=2^{B!}-1$ mod n
- We compute gcd(c,n)
- If it is between *1* and *n*, it is *p* (because *q* is a prime) ← we can then compute $q = n/p$
- We have broken the public key!
- But, if gcd(c,n)=n, we fail ← this only happens if both *p-1* and *q-1* divide B*!*
  - Well, we ignore these corner cases for brevity. Just remember *p-1* does not always work

There is a Catch.....

- How to compute $c=2^{B!}-1 \bmod n$? Is it realistic?
- Remember $p-1$ must be B-smooth, and B is not going to be small!
  - Say $2^{(10000!)} \bmod n$
- Can we really do this? NO....
- Need the last twist....

$$c_1 = 2^1 \pmod{n}$$
$$c_2 = (c_1)^2 \pmod{n}$$
$$c_3 = (c_2)^3 \pmod{n}$$
$$c_4 = (c_3)^4 \pmod{n}$$
$$\vdots$$
$$c_B = (c_{B-1})^B \pmod{n}$$

The Last Twist

$$c_1 = 2^1 \pmod{n}$$
$$c_2 = (c_1)^2 \pmod{n}$$
$$c_3 = (c_2)^3 \pmod{n}$$
$$c_4 = (c_3)^4 \pmod{n}$$
$$\vdots$$
$$c_B = (c_{B-1})^B \pmod{n}$$

$$c_4 \equiv (c_3)^4 \equiv ((c_2)^3)^4 \equiv (((c_1)^2)^3)^4 \equiv ((((2^1)^2)^3)^4)$$

Similarly $C_B \equiv 2^{B!} \pmod{n}$, and we can recursively calculate C_B

Put Pollard's *p-1* Together

Sage:n=44426601460658291157725536008128017297890787463719427903128118036605 7

sage: B_fac=factorial(2^25)

Well, I did not apply the last optimization

sage: c=Integer(pow(2,B_fac,n)) - 1

sage: p=gcd(c,n); p

1267650600228229401496703217601

sage: q=n/p; q

35046409044148024855564290035771968285 7

sage: p*q == n

True

Yeah, it works

Summary

- We introduced symmetric and asymmetric cryptography systems
- We walked through RSA algorithm
- We discussed one of the RSA attacks
- There are many other attacks ← out of scope
- References:
  - http://www.gregorybard.com/SAGE.html ← Our textbook
  - http://doc.sagemath.org/html/en/thematic_tutorials/numtheory_rsa.html ← Introduction on RSA
  - https://www.hyperelliptic.org/tanja/vortraege/facthacks-29C3.pdf ← Many more attacks

SageMath #3 Homework (S3)

1. (1%) Let *n* = *44426601460658291157725536008128017297890787463719427903128118036605*7 . Implement "the last twist" of *p-1* attack and compute $c = 2^{10000!} \pmod{n}$. You need to explain and run your code to the TAs to get the point
2. (1%) Disprove that *k* is *B*-smooth implies *k* | *B!*

SageMath #3 Homework (S3) (cont.)

3. (1%) Use Pollard's *p-1* attack to factorize this number:
n=8620215476436315823969982122087229142
8825864423479130795058291644274722203 97
9560941774193227831712 1

You need to explain and run your code in front of the TA, or you get zero point