2016-05-25

William A. Stein

5/25/2016

Contents

1 Math 480: Open Source Mathematical Software

1.0.1 2016-05-25

1.0.2 William Stein

1.1 Lectures 26: Public key cryptography (part 2 of 3): RSA

1.2 Recall the Diffie-Hellman key exchange:

- A and B agree on $g \in /p$.
- A and B choose random a, b and send g^a and g^b.
- The shared secret is $s = (g^a)^b = g^{ab} = (g^b)^a$, which both A and B can easily compute, but an eavesdropper (presumably) can't.

DH is simple, beautifully symmetric, and is useful for setting up an active secure channel for temporary communication:

- login to a website, ssh to a remote computer, etc.

DH does **not** solve a lot of interesting problems though. For example:

- B publishes some information a "public key" on their website.

- Anybody at any time, and without B having to do anything, can encrypt a message that only B can decrypt.

There is a solution to this problem, which also uses modular arithmetic. It's completely different than Diffie-Hellman!

width=500 class=

1.3 The RSA Cryptosystem

RSA = Rivest-Shamir-Adleman

(Errata: In the lecture yesterday I said GCHQ secretly also discovered DH, but actually they also discovered RSA.)

How RSA works. There's one basic idea from number theory that you have to know about to understand RSA.

Let p be a prime. Fermat's Little Theorem says that for every integer a coprime to p, we have

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

Try it out:

```
p = 23
for a in [1..p]:
    print a, Mod(a, p)^(p-1)
```

```
1 1
2 1
3 1
4 1
5 1
6 1
7 1
8 1
9 1
10 1
11 1
12 1
13 1
14 1
15 1
16 1
17 1
18 1
19 1
20 1
21 1
22 1
23 0
```

Proof: The cardinality of the finite group $(/p)^*$ is $p-1$ and it's a basic result in group theory that the order of any element of a group divides the cardinality of the group.

Similarly for component n, we have that if $\gcd(a,n)=1$, then

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

since $(/n)^*$ is a group of order $\varphi(n)$.

```
n = 20
r = euler_phi(n)
for a in range(n):
    if gcd(a,n) == 1:
        print a, Mod(a, n)^r
```

```
1 1
3 1
7 1
9 1
11 1
13 1
17 1
19 1
```

So A wants to send a secret message to B. here's how.

- Step 1. **Setup:**
  - B secretly chooses two large prime numbers p and q at random and an integer e, and publishes $n=pq$ and e.
  - B knows p and q, so B can compute $\varphi(n)=(p-1)(q-1)$, and also an integer d such that $ed \equiv 1 \pmod{\varphi(n)}$. Thus $ed = 1 + k\varphi(n)$ for some k.
- Step 2. **Encryption:**
  - A *encodes* the message as an integer m modulo n.
  - Then A *encrypts* the message as $t = m^e \pmod{n}$.
- Step 3. **Decryption:**
  - B decrypts the message by using that $t^d = (m^e)^d = m^{ed} \equiv m^{1+k\varphi(n)} \equiv m \pmod{n}$.

This last step uses that $m^{\varphi(n)} \equiv 1 \pmod{n}$.

Let's try it out!

```
# Step 1: Setup

p = next_prime(ZZ.random_element(2^512))
q = next_prime(ZZ.random_element(2^512))
n = p*q
```

```
phi_n = (p-1)*(q-1)
e = 3
while gcd(e,phi_n) != 1:
    e += 1
d = lift(Mod(e, phi_n)^(-1))

print "The public key is"
print "n = ", n
print "e = ", e
```

```
The public key is
n =  4193576142777251301471599104738520602392854975960019601550218390270026417272300606098
090097196163103938420340736828032525936334750757206483709727364066566202122942452387368507
273372745472523937410798982231833094252812192865894903884237776459556540024429351006827869
6461981911236835601612664086526107573847997
e =  5
```

```
# Step 2: Encryption

## Encode our message as a number in base 26:

mesg = "tectonicxxandxxsharkxxtankxxarexxundercover"
m = 0
for i, a in enumerate(mesg):
    m += 26^i * (ord(a) - ord('a'))
print "m = ", m

## Then encrypt it!

t = Mod(m, n)^e
print "encrypted message = ", t
```

```
m =  4613687604820056968581129277215495307270152505781622265499707
encrypted message =  209045548710188638563028852749394486816125582581249826260854304526112
204303871376059461427490581957698337463043849162002626202832920356299568865268548753933095
083337430898122323707774804796790756309311192086908387367757323797641988477429836234638155
8097543809500667583082778777187834544360943888902615307
```

```
# Step 3: Decryption

r = lift(t^d)
print "decrypted message = ", r

## And decode
z = ''
while r:
    z += chr((r % 26) + ord('a'))
    r = r//26
```

```
print "message = ", z
decrypted message =  4613687604820056968581129277215495307270152505781622265499707
message =  tectonicxxandxxsharkxxtankxxarexxundercover
```

1.4 Attacking RSA

To attack RSA the observer has to compute $m \pmod{n}$ given $m^e \pmod{n}$.

That is **NOT** the discrete log problem again! (Why?)

In theory, the attacker has all the information they need. They know n, so they can "just" factor n to find p, q, then compute $\varphi(n) = (p-1)(q-1)$, and d as above.

However factoring large numbers seems to be really frickin' hard.

```
n
4193576142777251301471599104738520602392854975960019601550218390270026417272300606098090097196163103938420340736828032525936334750757206483709727364066566202122942452387368507273372745472523937410798982231833094252812192865894903884237776459556540024429351006827869646198191123683560161266408652610757384799 7
```

```
is_prime(n)
False
False
```

```
factor(n) # will just waste cpu forever
Error in lines 1-1
Traceback (most recent call last):
  File ''/projects/sage/sage-6.10/local/lib/python2.7/site-
packages/smc_sagews/sage_server.py'', line 905, in execute
    exec compile(block+'\n', '', 'single') in namespace, locals
  File '''', line 1, in <module>
  File ''/projects/sage/sage-6.10/local/lib/python2.7/site-packages/sage/rings/arith.py'',
line 2459, in factor
    int_ = int_, verbose=verbose)
  File ''sage/rings/integer.pyx'', line 3511, in sage.rings.integer.Integer.factor
(/projects/sage/sage-6.10/src/build/cythonized/sage/rings/integer.c:22694)
    F = factor_using_pari(n, int_=int_, debug_level=verbose, proof=proof)
  File ''sage/rings/factorint.pyx'', line 352, in sage.rings.factorint.factor_using_pari
(/projects/sage/sage-6.10/src/build/cythonized/sage/rings/factorint.c:6295)
    p, e = n._pari_().factor(proof=proof)
  File ''sage/libs/pari/gen.pyx'', line 8977, in sage.libs.pari.gen.gen.factor
(/projects/sage/sage-6.10/src/build/cythonized/sage/libs/pari/gen.c:140617)
    pari_catch_sig_on()
  File ''sage/ext/interrupt/interrupt.pyx'', line 88, in
sage.ext.interrupt.interrupt.sig_raise_exception
(/projects/sage/sage-6.10/src/build/cythonized/sage/ext/interrupt/interrupt.c:924)
    raise KeyboardInterrupt
KeyboardInterrupt
```

OK, now work on your homework