Mathematical Introduction

Overview

This short write-up is an exploration of public-key cryptography.

To use the method, one first generates a pair of keys called the **public** key and a corresponding **private** key. These have a symmetry property, so that if either one is used to encrypt a message, then the other one can be used to decrypt the message.

Suppose that Alice generates a public/private key pair and then distributes the public key in such a way that Bob is certain that he really knows Alice's public key.

Bob can encrypt a message with Alice's public key and send it to Alice with the knowledge that only Alice can decrypt it using her private key.

Alternatively, Alice can encrypt a message with her private key and send it to Bob. Successful decryption with Alice's public key proves that the message is from Alice.

The process of encryption and decryption is computationally expensive, and only short messages should be exchanged by this method. Such a message would typically consist of yet another key which can be used to encode the main message.

Key generation

Start with two prime numbers, *p* and *q*. I obtained mine from a list of primes (near the 50 millionth prime):

The prime numbers for real-world use are much larger than this.

```
p = 961748941
q = 982451653
```

Compute `n = p*q = 944871836856449473`

```
>>> p = 961748941
>>> q = 982451653
>>> n = p*q
>>> n
944871836856449473
>>> len(bin(n))
62
>>>
```

This means we can securely encode not more than about 60 bits. Compute Euler's totient function.

```
ϕ(n) = (p − 1)(q − 1)
```

```
>>> phi = (p-1)*(q-1)
>>> phi
944871834912248880
>>>
```

Now choose a number *e* such that:

```
1 < e < ϕ(n)
```

e is called the public key exponent, with the requirement that *e* should be **coprime** to ϕ(n) (i.e., they should have no common factor other than 1).

The easiest way to do that is to pick a prime number like e = 2^{16} + 1 = 65537

Note that 65537 is `10000001` in binary.

I also confirmed that *e* is on a list of primes:

To test that *e* and ϕ(n) are coprime, just show that there is a remainder when dividing by ϕ(n)

```
>>> phi
944871834912248880
>>> e = 2**16 + 1
>>> e
65537
>>> phi % e
42186
>>>
```

The public key is then (*n*, *e*).

Actually, in practice, there is little variation in *e*. It is typically the same value as used here, 65537.

According to Laurens Van Houtven's *Crpto 101* here:

"*e* is either 65537 or 3, and this is because there are very few binary `1` 's in these numbers (only two single digits `1`), and as a result the exponentiation which we will compute *a lot*"

`m**e` is much more efficient with `e = 65537` .

The private key consists of *n* plus another number *d* which is computed from *ϕ(n)* and thus requires knowledge of *p* and *q* (below).

That is why the process of breaking this method of encryption is described as being the same as the problem of finding two primes that can factor a large number: *n*, the product of the primes *p* and *q*. *n* is known from the public key.

Encryption

We also need to have a message *m*, and generate a number to represent it.

ASCII-encoding is certainly a possible way to do this, though it is quite wasteful since many bytes never appear in any message (especially if we just use lowercase).

In this code sample, we convert the message `hello world` into a binary number using the `ord` and `bin` functions.

Because bin doesn't show leading zeroes, we add them back with `zfill` .

```
s = "hello world"
L = [bin(ord(c))[2:].zfill(8) for c in s]
```

```
>>> b = "".join(L)
>>> int(b,2)
126207244316550804821666916L
>>>
```

A more compact encoding might be achieved like this:

```
lc = 'abcdefghijklmnopqrstuvwxyz'
D = dict(zip(lc,range(1,len(lc)+1)))
D[" "] = 0  # space
```

The dictionary `D` assigns an integer for each lowercase letter. Now do:

```
s = "hello world"
m=0
N = 27
for i,c in enumerate(s):
    m += D[c] * N**i
```

Suppose the message was simply `hel`. Since `h` is the eighth letter of the alphabet, `e` is the fifth, and `l` is letter number 12, the value would be

```
8 + (5 × 27) + (12 × 272) = 8891
```

The number *m* is being formed from the integer values for each character of `hel`.

Our message is a much larger number.

```
>>> m
920321254041092
>>>
```

To read the message, just reverse the process:

```
N = 27
m = 920321254041092
rD = dict(zip(D.values(),D.keys()))
pL = list()
while m:
    pL.append(rD[m % N])
    m /= N

print ''.join(pL)
```

```
>>> m = 920321254041092
>>> pL = list()
>>> while m:
...     pL.append(rD[m % N])
...     m /= N
...
>>> ''.join(pL)
'hello world'
>>>
```

While the above encoding could be viewed as a form of encryption, it is very weak. The encryption function we will use is

```
c = m**e % n
```

```
>>> m = 920321254041092
>>> e = 65537
>>> n = 944871836856449473
>>> x = m**e
>>> len(str(x))
980692
>>> c = x % n
>>> c
448344912451359241L
>>>
```

The number *c* **is** our ciphertext. (The L on the end signifies a Python long, a type of number).

`x = m**e` is a very large number! Its decimal representation has nearly one million digits.

It is much more efficient to do the mod operation at the same time as the exponentiation. The Python built-in

function `pow` allows that as an option:

```
>>> pow(m,e,n)
448344912451359241L
>>>
```

Public key encryption is `m**e % n`.

Decryption

There is one more number we need to decode the encrypted text.

This number is called *d*, and it is referred to as the exponent of the private key. The private key is (*d*,*n*), although only the *d* part is actually secret.

Finding *d* is the tricky part of the whole operation, but luckily, it only needs to be computed once for a given key pair.

d is called the modular multiplicative inverse of e, mod ϕ(n). For example `2` is the modular multiplicative inverse of `7` mod `13`, since `2 * 7 = 14, and 14 % 13 = 1`.

What this means is that we want *d* such that

```
d * e = 1 (mod ϕ(n))
```

Substituting the known values for *e* and ϕ(n)

```
d × 65537 = 1 (mod 944871834912248880)
```

I found a simple implementation for computing *d* here:

```
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b \% a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception("modular inverse does not exist")
    else:
        return x % m
```

I have not yet figured out how it works exactly. But it is easy to show that it does. I saved the code in a file `mod.py` .

Let's try it out:

```
from mod import modinv
e = 65537
phi = 944871834912248880
d = modinv(e,phi)
```

Output:

```
>>> d
8578341116816273
>>> d*e % phi
1L
>>>
```

Remember that the secret pair of primes was used to compute ϕ.

So now, we are finally ready to decrypt:

```
>>> c = 448344912451359241
>>> n = 944871836856449473
>>> d = 8578341116816273
>>>
>>> p = pow(c,d,n)
>>> p
920321254041092L
>>>
>>>
```

Recall that `m` is equal to `920321254041092`.

We have successfully generated a key pair, and used it to encrypt and decrypt a simple message.

Public key decryption is `c**e % n`.

The last thing we need to do is demonstrate the symmetry property. We can encrypt with the private key, and decrypt with public one.

Private key encryption is `m**d % n`.

```
>>> m = 920321254041092
>>> d = 8578341116816273
>>> n = 944871836856449473
>>> c = pow(m,d,n)
>>>
```

This *c* is different than what we had before.

```
>>> c
461000660869754451L
>>>
```

Private key decryption is `c**e % n`.

```
e = 65537
p = pow(c,e,n)
>>> p
920321254041092L
```

We have again recovered our plaintext message: *p* is equal to *m*.

Summary

Once again, we find two large primes `p` and `q`. We compute `n = p*q` and `phi = (p-1)*(q-1)`. We choose `e = 65537` so that *e* is prime (and co-prime to ϕ). We find

```
d * e = 1 (mod ϕ(n))
```

Then

- `m**e % n = c`, public key encryption
- `c**d % n = p`, decryption
- `m**d % n = c`, private key encryption
- `c**e % n = p`, decryption

decryption of a message encrypted with the public key, and *encryption* of a message to be decrypted by the public key, both use the secret, private key.