# How to implement RSA from scratch: A simple Ruby implementation with lots of comments

By Chris Hulbert - [chris.hulbert@gmail.com](mailto:chris.hulbert@gmail.com) - <http://github.com/chrishulbert/crypto/blob/master/ruby/ruby_rsa.rb>

As a learning exercise recently, I set about implementing a bunch of crypto algorithms from scratch, in such a way as to learn how they work, rather than simply cut and pasting some open source code. This is my attempt to explain RSA, from the perspective of implementation, as simply as possible.

Keep in mind that this implementation is more aimed at learning how the algorithm works, rather than implementing it in an optimised and secure way, because I often find that these optimisations make it harder to understand what’s really going on. So, if you keep in mind that this code isn’t production-ready, we should get along just fine.

# RSA quick summary

RSA is a really simple public/private key algorithm, and messages encoded with one key can be decoded with the other. The public key is made up of two numbers: ‘e’ and ‘n’; the private key is made up of two numbers ‘n’ and ‘d’. Now you’ve probably heard of something like ‘1024-bit keys’, but they’re really just plain old numbers, albiet ~300 digits long. Once you’ve got your head around the fact that numbers with 300 decimal digits can be multiplied, added, divided, etc just like a small number like you’re used to, it all begins to make sense. An added bonus is that ruby has most of the implementation already built in for these big numbers.

The numbers are as follows:

e – Public exponent (most of the time this is 65537, or sometimes 3 or something else)

n – Modulus

d – Secret exponent

Encoding is quite a simple equation:

c = m^e mod n.

In other words: Encrypted data = Original Message ^ Exponent mod Modulus. Decoding is very similar:

m = c^d mod n

In other words: Decrypted data = Ciphertext ^ Secret exponent mod Modulus.

The only catch with these formulas is that you need some tricky algorithms to get them to work on normal computers. But don’t worry, we’ll get to that next.

# Modular Exponentiation

The generic formula: b^e mod m is not really going to work with 300-digit numbers without some help, so we’ll use the modular exponentiation technique, which is probably explained better on wiki than I can, so read about it here: <http://en.wikipedia.org/wiki/Modular_exponentiation>. Basically, we’re breaking the ‘power’ function down into a series of multiplications, and at each step we’re applying the mod (%) so that it doesn’t grow too large, instead of doing the modulus at the end. So we’ll implement the right-to left binary method here, which is a port of the python code on the wiki page:

# Calculate a modular exponentiation eg: b^p mod m

def mod\_pow(base, power, mod)

result = 1

while power > 0

result = (result \* base) % mod if power & 1 == 1

base = (base \* base) % mod

power >>= 1;

end

result

end

# Encrypting

Now we’ve got the mod\_pow function, we can start applying the encryption formula: c = m^e mod n, where c = ciphertext, m=cleartext message, e=exponent, and n=modulus aka public key.

Lets begin by converting a string into a big number. What we’ll do is get the ascii value of each character, and use that as two (hex) digits. So the string “ABC” becomes 0x41 (A), 0x42 (B), 0x43 (C) = 0x414243 = 4276803(decimal). Let’s implement that and it’s reverse (number to string) now:

# Convert a string into a big number

def str\_to\_bignum(s)

n = 0

s.each\_byte{|b|n=n\*256+b}

n

end

# Convert a bignum to a string

def bignum\_to\_str(n)

s=""

while n>0

s = (n&0xff).chr + s

n >>= 8

end

s

end

We can now go ahead and encrypt – and we’re only up to page 2 of this document! Careful with the line breaks on the long numbers, they aren’t there in the code:

# The original message, represented as a big number

m = str\_to\_bignum("This is going to be embarrasing if it doesn't work!")

# Exponent (part of the public key)

e = 0x10001

# The modulus (aka the public key, although this is also used in the private key as well)

n=0xcd0b9724a2a09b16d7739bc9daa29274563765110ba93a2f8f880ce9191909050599ee361eadec462e7c9b167aa020d61c93d6787921242c76a8cfd7c29ec5fe626e36ed06134ffc37fe3638dff132be1f15ed0fa43c3d957436b67ef42a1bb58f6154f30e5b30e7048bbff83375ae666900808558269c30d297190883d83bf1

# The secret exponent (aka the private key)

d=0x5771a2f287bc70878c388c6ec823decb685d5567b08e69f7108217e76cc1a57c13c872b377dffa4c6fd4ca1b0b0eb1123ebbb992452e220c284a93e0d2e9fd4b5ce90a889a612a00912fed6daf910140b95b5ae80af6f5e3df2eda981a7a6f937a4563ee8f1cc2770da989c98abc9e65df9869c42c063048b363f74e0a12d941

# Encrypt – c = m^e mod n

c = mod\_pow(m,e,n)

puts "Encrypted: %x" % c

You should get the following result:

Encrypted: 7d7a2a676485af8f6868a74d86dbb8fba033dab7e98beddc2ab72eac1170153c9ee24851712886f443e40299be74666f6ebc8325855c98d917b5db9a87025c892c28ae1a586636403d4c897088040fccae64973c1cada19ad49dbc9292b23a8fba55365a72b704471483d89658c2320e26762522182b48316e3cfaf0cb71136a

# Decryption

Decryption is pretty simple. We just grab the ‘c’ value from before, and do the following formula: m = c^d mod n, where m = decrypted message, c=ciphertext, d =secret exponent, n=modulus. Lets try it out:

# Decrypt

a = mod\_pow(c,d,n)

s = bignum\_to\_str(a)

puts "Decrypted: " + s

You should get this result:

Decrypted: This is going to be embarrasing if it doesn't work!

# Creating keys

Now this is where it starts getting complicated. I bet you wondered before where the big ‘n’ and ‘d’ values came from. Well let’s work that out:

First up, we need two very large prime numbers, called ‘p’ and ‘q’. For a 1024-bit key, these are to be ~512 bits each. We generate the ‘n’ modulus (aka public key) easily by multiplying them: n = p \* q. We then generate another number, ‘phi’: phi = (p-1) \* (q-1). To calculate the secret exponent ‘d’, we then get the ‘modular multiplicative inverse’ of e modulo phi (this will be explained more later). Once you have your e, n, and d values, you should discard the p, q, and phi values.

# Generating primes

To generate p and q, our two big primes, we need a big-number generator, and a prime-number tester. We’ll use the Miller-Rabin primality test here because it’s simple and fast, however it’ll be wrong one in a million times, so it’s probably not good for production use (the AKS primality test algorithm is probably more suited for that). You can read up on the primality test here: <http://en.wikipedia.org/wiki/Miller-Rabin_primality_test>, and here’s an implementation I found, but was unable to find the original writer:

# Perform a primality test

class Integer # From http://snippets.dzone.com/posts/show/4636

def prime?

n = self.abs()

return true if n == 2

return false if n == 1 || n & 1 == 0

# cf. http://betterexplained.com/articles/another-look-at-prime-numbers/ and

# http://everything2.com/index.pl?node\_id=1176369

return false if n > 3 && n % 6 != 1 && n % 6 != 5 # added

d = n-1

d >>= 1 while d & 1 == 0

20.times do # 20 = k from above

a = rand(n-2) + 1

t = d

y = mod\_pow(a,t,n) # implemented below

while t != n-1 && y != 1 && y != n-1

y = (y \* y) % n

t <<= 1

end

return false if y != n-1 && t & 1 == 0

end

return true

end

end

Now we need a function to generate big numbers of a certain bit length, and a function to continually generate them until it finds a prime. When generating these numbers, the highest two bits will be set (so that it fills the necessary bit length), as well as the lowest one (so that it is always an odd number). The function will create a big string of 0’s and 1’s, then convert that to a big number base 2:

# Make a random bignum of size bits, with the highest two and low bit set

def create\_random\_bignum(bits)

middle = (1..bits-3).map{rand()>0.5 ? '1':'0'}.join

str = "11" + middle + "1"

str.to\_i(2)

end

# Create random numbers until it finds a prime

def create\_random\_prime(bits)

while true

val = create\_random\_bignum(bits)

return val if val.prime?

end

end

So let’s test our prime number generator (and create the modulus ‘n’ while we’re at it):

puts "Generating primes, please wait:"

p = create\_random\_prime(512)

q = create\_random\_prime(512)

puts "p: %x" % p

puts "q: %x" % q

n = p\*q

puts "n: %x" % n

This should take a couple seconds to run, and give you a result like this (different/longer numbers of course):

Generating primes, please wait:

p: c1ee397e7aa766561c7f0eb493379aa278b28bcc4de2264f09eceea87fe4fefa22a8914e9501742a969041c9db0d72…

q: e4b139ac7cd81ba66b2d68a55ef6f82d16917794799ad0b99180b543da81c31fc87f6fba23bbee508308731e5ace54…

n: e98f4ca3afb2067720b5decf5499068519ce8b593182efddff70494153a8b7303fb97959e2f1f92c5132a974d2e62…

# Modular Multiplicative Inverse

In order to create the secret exponent ‘d’, we need to find the modular multiplicative inverse of e mod phi. You can read about the maths of this here: <http://en.wikipedia.org/wiki/Modular_multiplicative_inverse>. We’ll be using the Extended Euclidean algorithm, described here: <http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm>. I cannot explain the maths any better than wiki, so you can read about it there.

Basically, we’re using the Extended Euclidean algorithm to solve the formula: ax + by = gcd(a,b). We input the exponent e as ‘a’ and phi as ‘b’ to that formula, and get back ‘x’ and ‘y’. The returned ‘y’ value is ignored, but the ‘x’ is used as our secret exponent ‘d’, with one catch: if ‘x’ is negative, d = x + phi.

Here’s the ruby implementation of the extended euclidean algorithm, ported from the python version on wiki: <http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Recursive_method_2>

# Solve the extended euclidean algorithm: ax + by = gcd(a,b)

def extended\_gcd(a, b)

return [0,1] if a % b == 0

x,y = extended\_gcd(b, a % b)

[y, x-y\*(a / b)]

end

And our function to use the above algorithm to find our secret exponent, given p, q, and e:

# Get the modular multiplicative inverse of a modulo b: a^-1 equiv x (mod m)

def get\_d(p,q,e)

phi = (p-1)\*(q-1)

x,y = extended\_gcd(e,phi)

x += phi if x<0

x

end

So now we’ve got everything we need to generate a public+private key-pair, so lets test it out:

puts "Generating key-pair, please wait:"

e = 65537

p = create\_random\_prime(512)

q = create\_random\_prime(512)

n = p\*q

d = get\_d(p,q,e)

puts "e (exponent): %x" % e

puts "n (public): %x" % n

puts "d (private): %x" % d

You should get a result like this:

Generating key-pair, please wait:

e (exponent): 10001

n (public): c6df190f06faa4c053ac77fb1f57e480caab7accf39d247b0186bba54264accea74d5567869493e9c95988c6ad0171acbbc0e4a66c

f758a84096edb817178d7e36ba753a73380de0970e4bf2afdc74f54ec896ea6b3d6fd351bb8c143fa03089b94a0c5c8a5b2bf1721b78f2bf2ccfb3b52d514e208756303f4ca1d316592667

d (private): 7e9957627196ed7a61c9d13753e4a7da352aa4aa040b6d45c0dafc695fb2a72f86e17c14c35fa22999bc1d8e1c6466f10734ec59e5

d42fe42bf9e8aae04866116b44595af5ed79b6046429423b1b5ba8b5cdd33c600beaaaf3320b55d5a0afd22264633480bf1bcaf517af8517f89c772b9211afd751e41aa9fba2e1cec49001

Thanks for reading - that’s it for RSA encryption/decryption/key generation! Please check out my code at github at the address below. What’s remaining is the padding and signature schemes that make this a truly secure and viable solution, but we’ve got the crux of RSA sorted so far. You may want to read further about RSA at the links here:

<http://github.com/chrishulbert/crypto/blob/master/ruby/ruby_rsa.rb>

<http://www.di-mgt.com.au/rsa_alg.html>

<http://islab.oregonstate.edu/koc/ece575/02Project/Mor/>

<http://people.csail.mit.edu/rivest/Rsapaper.pdf>