Contents

1 Recommended Reading

For numerics: *The Feeling of Power* by Isaac Asimov `http://downlode.org/Etext/power.html`. After reading this was the first time I ever looked up an algorithm for computing square roots. Read it and you'll know why I did.

For cryptography: `http://xkcd.com/177/`

A Mathematician's Apology, "Two Proofs Everyone Should Know", Chapter 12-13, by G.H. Hardy `web.njit.edu/~akansu/PAPERS/GHHardy-AMathematiciansApology.pdf`

2 Public Key/Private Key Cryptography

2.1 Overview

Public Key: Alice wants anyone in the world to be able to send her a message. So she publishes a *public key* that allows anyone to encrypt a message.

Private Key: However, Alice wants the messages sent to her to be secure so that only she can read them. Therefore, she keeps secret a *private key*. It is only possible to decode encrypted messages if the private key is known.

2.2 RSA Algorithm

RSA key generation:

1. Choose primes p and q.
2. $n = pq$
3. $\phi(n) = (p-1)(q-1)$
4. Choose e, $1 < e < \phi(n)$ s.t. $\gcd(e, \phi(n)) = 1$
5. Find $d = e^{-1} \bmod \phi(n)$. So $d < \phi(n)$ is the number such that $ed \equiv 1 \bmod \phi(n)$. It can be shown that d is unique.
6. Release e and n (public key), keep p, q, and d private (private key)

Modular Inverses: One point in all of this should give you pause: how do we find d? Are modular inverses easy to find? In fact, they are, *if they exist.* A number a only has an inverse modulo n if $\gcd(a, n) = 1$ (we say "a is relatively prime to n"). If a is relatively prime to n, $x = a^{-1} \bmod n$ is the number such that $ax \equiv 1 \bmod n$. This number is unique and can be found in polynomial time using the Extended Euclidean Algorithm (look it up if you're curious). Therefore, since by construction e is relatively prime to $\phi(n)$ and we know e and $\phi(n)$ when constructing the keys, we can find d easily.

Euler Totient Function and Fermat-Euler Theorem: The number $\phi(n)$ is actually an important number in general algebra and number theory, known as the *Euler Totient function* or *Euler Phi function.* It is defined as the number of numbers less than n that are relatively prime to n. You can prove to yourself that if p and q are distinct primes and $n = pq$ then $\phi(n) = (p-1)(q-1)$ as claimed. More important, for our purposes, the Fermat-Euler Theorem proves that if a is relatively prime to n (so $\gcd(a, n) = 1$) then $a^{\phi(n)} \equiv 1 \bmod n$.

RSA Encryption Of message $m < n$: $c = m^e \bmod n$

RSA Decryption RSA decryption works for any $m < n$. Here, we will just show that if m and n are relatively prime (and for most m, they will be) then the decryption works:

$$\begin{aligned} c^d \bmod n &= m^{ed} \bmod n \\ &= m^{N\phi(n)+1} \bmod n \end{aligned}$$

where N is an integer. If $\gcd(m, n) = 1$ then the Fermat-Euler Theorem says that $(m^{\phi(n)})^N \equiv 1 \bmod n$ so we get

$$m^{N\phi(n)+1} \bmod n = 1(m) \bmod n = m \tag{1}$$

where the last step follows because we know $m < n$.

The proof that the decryption works for any $m < n$ is based on the Chinese Remainder Theorem if you wish to look it up.

Attacking RSA: Firstly, note that if you encrypt a small message m with a small exponent e, you may have $c = m^e < n$. This can easily be decrypted by taking the eth root c.

A more serious flaw is that the product of two encryptions is equal to the encryption of the product of two plaintext messages: $m_1^e m_2^e \bmod n = (m_1 m_2)^e \bmod n$. Therefore, if Eve sees an encryption c that she would like to decode, she can generate some other message r and ask the holder of the private key to decrypt $(cr^e) \bmod n$, which gives her the decryption of c. There *are* situations in which the holder of the private key will decrypt any message that he thinks looks relatively "harmless". This attack is a *chosen-ciphertext attack* and can be solved by padding.

Most RSA attacks have been done by simply trying to factor big numbers using various sieve algorithms, bringing us to...

RSA and Factoring: We do not know if RSA is as hard as factoring. Clearly, if we could factor, we could break RSA since, given $n = pq$, we could find $\phi(n) = (p-1)(q-1)$ and $d \equiv e^{-1} \bmod \phi(n)$. Therefore, factoring is at least as hard as RSA (but might be harder).

RSA is very close to being as hard as factoring: We know that given n and e, finding $\phi(n)$ is as hard as factoring. $\phi(n)$ can be used to factor n as follows:

$$\begin{aligned} & \phi(n) = (p-1)(q-1) = n + 1 - (p+q) \\ \Rightarrow \quad & p + q = n + 1 - \phi(n) \\ & p - q = \sqrt{p^2 - 2pn + q^2} = \sqrt{(p+q)^2 - 4n} \end{aligned}$$

This is two equations and two unknowns so we can solve for p and q! Therefore, finding $\phi(n)$ must be as hard as factoring since if we could find $\phi(n)$, we could factor n.

It has also been shown that finding d itself (without necessarily finding $\phi(n)$ first) is as hard as factoring. However, neither of these is exactly the same as breaking RSA. To break RSA, we need a polynomial time algorithm that can find m given c, e, and n. While the most obvious way to go about this would be to find d, that is **not** exactly the same thing. It is, in fact, unknown if RSA and factoring are equivalently hard problems.

Note that factoring has also not been proved to be NP-Complete.

3 A Number Theory Digression

3.1 Requirements

In order to make RSA (and a number of other cryptographic schemes) work, we would like to have the following:

1. Fast method for generating large primes p and q
2. Fast method for computing powers modulo numbers
3. Difficult to factor n

3.2 Running Time

Factoring sounds "easy". After all, if you want to factor n, you can simply try every number from 2 to n and see which ones divide n... This is $O(n)$ time. The problem is $O(n)$ time is *not* polynomial in the size of the input to the problem. Technically, size of input is defined as the number of bits. Usually, though, we shortcut and consider the number of items (i.e. number of vertices, number of edges, etc) because each item takes a constant number of bits to represent and we are interested in how the algorithm scales as we add more items. With cryptographic algorithms, though, we input a constant number of numbers and want to know how the running time depends on the *value* of the numbers. The number of bits it takes to represent a number n is $\log(n)$. Therefore, when we talk about algorithms that run "efficiently" or "quickly" on input number n, we mean the algorithm must be polynomial in $\log(n)$. The scheme for factoring outlined above takes $O(n)$, which is exponential in the size of the input. When we talk about things being "easy", we mean it is possible to compute that thing in time polynomial in $\log(n)$.

3.3 Primality Testing

Fermat's Little Theorem: If p is prime then $a^{p-1} \equiv 1 \bmod p$ for $1 \leq a \leq p-1$.

Fermat Primality Test: So, we randomly choose a bunch of a's and check them. If $a^{p-1} \equiv 1 \bmod p$ for all of them, we decide p is prime. This is a *probabilistic test.* In fact, even if we checked almost every a less than p, we could still be wrong because of Carmichael numbers, which are composite numbers n such that $a^{n-1} \equiv 1 \bmod n$ for all a relatively prime to n. Of course, if you find $a < n$ that is *not* relatively prime to n (and computing GCD is a polynomial time algorithm using the Euclidean Algorithm) that's an immediate indicator that n is not prime, but these are few and far between. Therefore, almost any a we pick will be relatively prime to n and, if n is a Carmichael number (rather than a true prime), we are highly likely to mis-identify it. The lowest known Carmichael number is 561.

Probability in Cryptography: This is OK! *All* of cryptography relies on probabilistic proofs. Namely, assume we have an awesome cryptographic scheme that encrypts a message m to c. We want it to be "hard" for anyone to guess m only given c (and maybe some public information). However, "hard" does not mean impossible. Consider an algorithm that just outputs a random message. This algorithm has a (tiny) chance of guessing m. So we only require that there be an exponentially small (again, remember, in the log of the values of the input numbers) chance that something bad can happen. Like a program guessing our message or the primality test failing. So it is fine to use a probabilistic primality test, assuming that the chance of it outputting a composite number is exponentially small.

Note, however, that because of Carmichael numbers, the Fermat primality test is not usually used in practice. Instead people use the Miller-Rabin or Solovay-Strassen tests.

3.4 An Infinity of Primes

Primes are clearly very important to cryptography. What happens should we run out? Luckily, thanks to Euclid, we know we cannot. There are an infinite number of prime numbers. This is one of my favorite proofs (along with, actually, the proof that the square root of two is irrational) so I'm including it here.

Proof there are an infinite number of primes: We proceed by contradiction. Assume there are a finite number of primes P. Now we take the product of all primes $n = p_1 p_2 ... p_P$ and consider $n+1$. Clearly $n+1$ is not divisible by any of $p_1, p_2, ..., p_P$ because it leaves a remainder of 1 after any division by them and just as clearly $n+1$ is not equal to any primes in $p_1, p_2, ..., p_P$. Therefore, $n+1$ itself is prime *or* it is divisible by some prime not in our list. However, our list supposedly included all primes.

3.5 Modular Exponentiation

Another thing that crops up all over cryptography is modular exponentiation. We would like to be able to do that quickly.

```
ExponMod($c, d = \langle d_{k-1}, d_{k-2}, ..., d_0 \rangle, n$)
    Input: Base $c$, exponent $d$ of $k$ bits, modulus $n$
    Output: $c^d$ mod $n$.
1   $i \leftarrow 0, m \leftarrow 1$
2   for $j = k - 1$ downto $0$
3       $i \leftarrow 2i$
4       $m \leftarrow m^2$ mod $n$
5       if $d_j = 1$
6           $i \leftarrow i + 1$
7           $m \leftarrow mc$ mod $n$
8   return $m$
```

Running Time: If n, c, and d are k-bit numbers, total bit operations is $O(k^3)$. (Assuming multiplication of k-bit numbers takes $O(k^2)$.)

Correctness: Firstly note that we output $m = c^i$. Therefore, we we must just show that $i = d$. To do this, note that multiplying by 2 in binary is equivalent to just adding a zero to the end of the number. Then proceed by induction by decreasing bit significance to show that the bits of i and d are the same. Do it yourself!

4 Collision-Resistant Hash Functions

Cryptography has other applications than just encoding messages. One other application is collision-resistant hash functions. These are hash functions h such that, given the hash function, it is difficult to find any x and y such that $h(y) = h(x)$. More formally:

Definition: A family of hash functions $H = \{h_0, h_1, ...\}$ is *collision resistant* if for all algorithms $p(h_i)$ running in time logarithmic in the size of the input such that for all i, the probability that $p(h_i) = \{x, y\}$ where $x \neq y$ and $h_i(x) = h_i(y)$ is exponentially small in $\log(m)$.

So even if you want to break the hash function for just *one* input, you cannot do it!

Universal Hash Functions: Note that this is not the same as a universal hash function. For those functions, *randomly* chosen keys should have had a small probability of colliding, but if you are allowed to choose any two keys, it may be easy to find a collision.

For example, $h_{a,b}(x) = ((ax+b) \bmod p) \bmod m$ with p larger than a, b, m, or x is a universal hash function, but given a, b, p, m, we can easily find x and y such that $((ax+b) \bmod p) \bmod m = ((ay+b) \bmod p) \bmod m$. For example $x = 1$ and $y = (m+1)$ collide.

Example: Choose p and q as odd primes and compute $n = pq$. Let g be any number relatively prime to $\phi(n) = (p-1)(q-1)$. Then $h_{g,n}(x) = g^x \bmod n$ is a collision resistant hash function provided p and q are unknown and factoring is hard.

Sketchy Proof: Assume we could find x and y with $y \neq x$ such that $g^x \bmod n = g^y \bmod n$. Therefore, we have that $g^{x-y} = 1 \bmod n$. For n, we define the Carmichael function, $\lambda(n)$ as the smallest integer m such that $a^m \equiv 1 \bmod n$. Therefore $x - y = k\lambda(n)$. Moreover, for any g relatively prime to $\phi(n)$, the Fermat-Euler theorem states that $g^{\phi(n)} \equiv 1 \bmod n$. Therefore, $\phi(n) = k'\lambda(n)$ is also a multiple of $\lambda(n)$. We showed that $\phi(n)$ can be used to factor n. Actually, Miller and Bach showed that any multiple of $\lambda(n)$ is enough to factor n.

5 Box and Locks Problem

Problem: Alice and Bob have never met in person. Alice wants to send a box containing valuables to Bob. However, the only way to do this is through the evil postal service. The evil postal service opens any box without a lock on it and takes whatever is inside. How can Alice send her box to Bob without it being emptied?

Solution: Alice puts a lock on the box and sends it to Bob. Bob then puts his own lock on the box and sends it back to Alice. Alice removes her lock and sends it to Bob, who can now remove his lock and get at the contents of the box.

Analogy: This is actually a method for key-exchage. If Alice has a semi-public key that she is willing to share with Bob, but not the whole world, she can use this method to securely send Bob her key.

It is also similar to public-key encryption because both are *assymetric key systems*. In both, Alice and Bob are able to exchange information in full view of the public without anyone being able to read the information. Moreover, they are able to do this *without* any previous communication.