YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 467b: Cryptography and Computer Security

Week 5 (rev. 4)
Professor M. J. Fischer *February 8 & 10, 2005*

Lecture Notes, Week 5

1 Computation with Big Integers

Recall that RSA is based on a number n chosen to be the product of two distinct large primes p and q. The public key e and private key d are numbers in $\mathbf{Z}_n^*$ and are related by $ed \equiv 1 \pmod{\phi(n)}$. The RSA encryption and decryption functions are then

$$E_e(m) = m^e \bmod n$$

$$D_d(c) = c^d \bmod n$$

Before we complete our discussion of RSA, we need to show how to generate suitable numbers n, e, and d, and we need to show that D_d really decrypts messages encrypted by E_e. We also need to discuss reasons for believing that RSA is secure, given that n, p, q are sufficiently large.

What is sufficiently large? That's hard to say, but p and q are typically chosen to be roughly 512 bits long when written in binary, in which case n is about 1024 bits long. Already this presents a major computational problem since the arithmetic built into typical computers can handle only 32 bit integers (or 64 bit integers for the most advanced technology). This means that all arithmetic on large integers must be performed by software routines.

The straightforward algorithms for addition and multiplication that you learned in grade school have time complexities $O(N)$ and $O(N^2)$, respectively, where N is the length of the integers involved. Asymptotically faster multiplication algorithms are known, but they involve large constant factor overheads, so it's not clear whether they are practical for numbers of the size we are talking about. What is clear is that a lot of cleverness is possible in the careful implementation of even the $O(N^2)$ multiplication algorithms, and a good implementation can be many times faster in practice than a poor one. They are also not particularly easy to implement correctly since there are many special cases that must be handled.

Most people choose to use big number libraries written by others rather than write their own code. Two such libraries that you can use in this course are ln3 (the third in a succession of Large Number packages by René Peralta) and gmp (Gnu Multiprecision Package). Ln3 provides a nice C++ user interface but has some limitations on the size numbers that it can handle. I have made it available on the Zoo in `/c/cs467/ln3`. Documentation is in `/c/cs467/ln3/doc`. Gmp provides a large number of highly-optimized function calls for use with C and C++. It is preinstalled on all of the Zoo nodes and supported by the open source community. Type `info gmp` at a shell for documentation.

1.1 Exponentiation

We now turn to the basic operation of RSA, modular exponentiation of big numbers. This is the problem of computing $m^e \bmod n$ for big numbers m, e, and n.

The obvious way to compute this would be to compute $t = m^e$ and then compute $t \bmod n$. The problem with this approach is that m^e is too big! m and e are both numbers about 1024 bits long, so their values are each about 2^{1024}. The value of t is then $(2^{1024})^{2^{1024}}$. This number, when written in binary, is about $1024 * 2^{1024}$ bits long, a number far larger than the number of atoms in the universe (which is estimated to be only around $10^{80} \approx 2^{266}$). The trick to get around this problem is to do all arithmetic in $\mathbf{Z}_n$ using equations (1) of lecture notes week 4, that is, reduce the result modulo n after each arithmetic operation. The product of two length ℓ numbers is only length 2ℓ before reduction mod n, so one never has to deal with numbers longer than about 2048 bits.

Nevertheless, there is still a problem with the naive algorithm, for it will execute its main loop $e - 1$ times. Since the value of e is roughly 2^{1024}, this would run longer than the current age of the universe (which is estimated to be 15 billion years). Assuming one loop iteration could be done in one microsecond (very optimistic seeing as each iteration requires computing a product and remainder of big numbers), only about 30×10^{12} iterations could be performed per year, and only about 450×10^{21} iterations in the lifetime of the universe. But $450 \times 10^{21} \approx 2^{79}$, far less than $e - 1$.

The trick here is to use a more efficient exponentiation algorithm based on repeated squaring. To compute $m^e \bmod n$ where $e = 2^k$ is a power of two requires only k squarings, i.e., one computes

$$\begin{array}{rcl} m_0 & = & m \\ m_1 & = & (m_0 * m_0) \bmod n \\ m_2 & = & (m_1 * m_1) \bmod n \\ & \vdots & \\ m_k & = & (m_{k-1} * m_{k-1}) \bmod n. \end{array}$$

Clearly, each $m_i = m^{2^i} \bmod n$. m^e for values of e that are not powers of 2 can be obtained as the product modulo n of certain m_i's. In particular, express e in binary as $e = (b_s b_{s-1} \ldots b_2 b_1 b_0)_2$. Then m_i is included in the final product if and only if $b_i = 1$.

It is not necessary to perform this computation in two phases as described above. Rather, the two phases can be combined together, resulting in a slicker and simpler algorithm that does not require the explicit storage of the m_i's. I will give two versions of the resulting algorithm, a recursive version and an iterative version. I'll write both in C notation, but it should be understood that the C programs only work for numbers smaller than 2^{16}. To handle larger numbers requires the use of big number functions.

```
/* computes m^e mod n recursively */
int modexp( int m, int e, int n)
{
  int r;
  if ( e == 0 ) return 1;           /* m^0 = 1 */
  r = modexp(m*m % n, e/2, n);      /* r = (m^2)^(e/2) mod n */
  if ( (e&1) == 1 ) r = r*m % n;    /* handle case of odd e */
  return r;
}
```

This same idea can be expressed iteratively to achieve even greater efficiency.

```
/* computes m^e mod n iteratively */
int modexp( int m, int e, int n)
{
  int r = 1;
  while ( e > 0 ) {
    if ( (e&1) == 1 ) r = r*m % n;
```

```
        e /= 2;
        m = m*m % n;
      }
      return r;
    }
```

The loop invariant is $e > 0 \land (m_0^{e_0} \bmod n = rm^e \bmod n)$, where m_0 and e_0 are the initial values of m and e, respectively. It is easily checked that this holds at the start of each iteration. If the loop exits, then $e = 0$, so r is the desired result. Termination is ensured since e gets reduced during each iteration. Note that the last iteration of the loop computes a new value of m that is never used. A slight efficiency would result by restructuring the code to eliminate this unnecessary computation.

2 Some More Number Theory Review

Recall that in lecture notes week 4, section 5.1, it was claimed that $\mathbf{Z}_n^*$ is an Abelian group under multiplication $(\cdot)$ mod n. This means that it satisfies the following properties:

Associativity $(\cdot)$ is an associative binary operation on $\mathbf{Z}_n^*$. In particular, this means if $x, y \in \mathbf{Z}_n^*$, then $x \cdot y \in \mathbf{Z}_n^*$.

Identity 1 is an identity element for $(\cdot)$ in $\mathbf{Z}_n^*$, that is $1 \cdot x = x \cdot 1 = x$ for all $x \in \mathbf{Z}_n^*$.

Inverses For all $x \in \mathbf{Z}_n^*$, there exists another element $x^{-1} \in \mathbf{Z}_n^*$ such that $x \cdot x^{-1} = x^{-1} \cdot x = 1$.

Commutativity $(\cdot)$ is commutative. (This is only true for *Abelian* groups.)

Example: Let $n = 26 = 2 \cdot 13$. Then

$$\mathbf{Z}_{26}^* = \{1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25\}$$

and

$$\phi(26) = |\mathbf{Z}_{26}^*| = 12.$$

The inverses of the elements in $\mathbf{Z}_{26}^*$ are given in Table 1. The bottom row of the table gives equiva-

Table 1: Table of inverses in $\mathbf{Z}_{26}^*$.

x	1	3	5	7	9	11	15	17	19	21	23	25
x^{-1}	1	9	21	15	3	19	7	23	11	5	17	25
$=$	1	9	-5	-11	3	-7	7	-3	11	5	-9	-1

lent integers in the range $[-12, \ldots, 13]$. This makes it apparent that $(26 - x)^{-1} = -x^{-1}$. In other words, the last row reads back to front the same as it does from front to back except that all of the signs flip from $+$ to $-$ or $-$ to $+$, so once the inverses for the first six numbers are known, the rest of the table is easily filled in.

Formula for Euler's ϕ function

Theorem 1 *Write n in factored form, so*

$$n = p_1^{e_1} \cdots p_k^{e_k}$$

where $p_1, \dots, p_k$ are distinct primes and $e_1, \dots, e_k$ are positive integers.[1] *Then*

$$\phi(n) = (p_1 - 1) \cdot p_1^{e_1 - 1} \cdots (p_k - 1) \cdot p_k^{e_k - 1}.$$

When p is prime, we have simply $\phi(p) = (p-1)$, and for the product of two distinct primes, $\phi(pq) = (p-1)(q-1)$. Thus, $\phi(26) = (13-1)(2-1) = 12$, as we have seen.

Theorem 2 (Euler's theorem) $x^{\phi(n)} \equiv 1 \pmod{n}$ *for all $x \in \mathbf{Z}_n^*$.*

As a special case, we have

Theorem 3 (Fermat's theorem) $x^{(p-1)} \equiv 1 \pmod{p}$ *for all x, $1 \le x \le p-1$, where p is prime.*

3 RSA Decryption

We now show why RSA decryption works. Recall that $n = pq$ for large distinct primes p and q, and $\phi(n) = |\mathbf{Z}_n^*| = (p-1)(q-1)$. Alice chooses the public key e and private key d to satisfy $e, d \in \mathbf{Z}_{\phi(n)}^*$ and

$$ed \equiv 1 \pmod{\phi(n)}. \tag{1}$$

The encryption function is $E_e(m) = m^e \bmod n$ and decryption function is $D_d(c) = c^d \bmod n$.

To be a proper cryptosystem, $E_e()$ must be a one-to-one function, and $D_d()$ must invert $E_e()$ on its range. That is, one must show

$$D_d(E_e(m)) = m$$

for all messages m. Plugging in the definitions of D_d and E_e, it is sufficient to verify that

$$(m^e)^d \equiv m \pmod{n} \tag{2}$$

By (1), we have $ed = 1 + u\phi(n)$ for some u. If $m \in \mathbf{Z}_n^*$, then (2) follows using Euler's theorem:

$$(m^e)^d \equiv m^{ed} \equiv m^{1+u\phi(n)} \equiv m \cdot (m^{\phi(n)})^u \equiv m \cdot 1^u \equiv m \pmod{n}.$$

What about the case of messages $m \in \mathbf{Z}_n - \mathbf{Z}_n^*$? There are several answers to this question.

1. For such m, either $p|m$ or $q|m$ (but not both because $m < pq$). If Alice ever sends such a message and Eve is astute enough to compute $\gcd(m, n)$ (which she can easily do, see below), then Eve will succeed in breaking the cryptosystem. So Alice doesn't really want to send such messages if she can avoid it.

2. If Alice sends random messages, her probability of choosing a message not in $\mathbf{Z}_n^*$ is only about $2/\sqrt{n}$. This is because the number of "bad" messages is only $n - \phi(n) = pq - (p-1)(q-1) = p + q - 1$ out of a total of $n = pq$ messages altogether. If p and q are both 512 bits long, then the probability of choosing a bad message is only $2 \cdot 2^{512}/2^{1024} = 1/2^{511}$. Such a small probability event will likely never occur during the known life of the universe.

3. For the purists out there, RSA does in fact work for all $m \in \mathbf{Z}_n$, even though Euler's theorem fails for $m \notin \mathbf{Z}_n^*$. For example, if $m = 0$, it is clear that $(0^e)^d \equiv 0 \pmod{n}$, yet Euler's theorem fails since $0^{\phi(n)} \not\equiv 1 \pmod{n}$. We omit the proof of this curiosity.

[1] By the fundamental theorem of arithmetic, every integer can be written uniquely in this way up to the ordering of the factors.

4 Generating RSA Encryption and Decryption Exponents

We next turn to the question of how Alice chooses e and d to satisfy (1). One way she can do this is to choose a random integer $d \in \mathbf{Z}^*_{\phi(n)}$ and then solve (1) for e. We will show how to do this in Sections 6 and 7 below.

However, there is another issue, namely, how does Alice find $d \in \mathbf{Z}^*_{\phi(n)}$? If $\mathbf{Z}^*_{\phi(n)}$ is large enough, then she can just choose random elements from $\mathbf{Z}_{\phi(n)}$ until she encounters one that lies in $\mathbf{Z}^*_{\phi(n)}$. But how large is large enough? If $\mathbf{Z}^*_{\phi(n)}$ is much smaller than $\phi(n)$ (the size of $\mathbf{Z}_{\phi(n)}$), she might have to search for a long time before finding a suitable d.

In general, $\mathbf{Z}^*_m$ can be considerably smaller than m. For example, if $m = |\mathbf{Z}_m| = 210$, then $|\mathbf{Z}^*_m| = 48$. In this case, the probability that a randomly-chosen element of $\mathbf{Z}_m$ falls in $\mathbf{Z}^*_m$ is only $48/210 = 8/35 = 0.228\ldots$.

The following theorem provides a crude lower bound on how small $\mathbf{Z}^*_m$ can be relative to the size of $\mathbf{Z}_m$ that is nevertheless sufficient for our purposes.

Theorem 4 *For all $m \geq 2$,*

$$\frac{|\mathbf{Z}^*_m|}{|\mathbf{Z}_m|} \geq \frac{1}{1+\log_2 m}.$$

Proof: Write m in factored form as $m = \prod_{i=1}^t p_i^{e_i}$, where p_i is the i^{th} prime that divides m and $e_i \geq 1$. Then $\phi(m) = \prod_{i=1}^t (p_i - 1)p_i^{e_i-1}$, so

$$\frac{|\mathbf{Z}^*_m|}{|\mathbf{Z}_m|} = \frac{\phi(m)}{m} = \frac{\prod_{i=1}^t (p_i-1)p_i^{e_i-1}}{\prod_{i=1}^t p_i^{e_i}} = \prod_{i=1}^t \left(\frac{p_i-1}{p_i}\right). \tag{3}$$

To estimate the size of $\prod_{i=1}^t (p_i - 1)/p_i$, note that $(p_i - 1)/p_i \geq i/(i+1)$. This follows since $(x-1)/x$ is monotonic increasing in x, and $p_i \geq i+1$. Then

$$\prod_{i=1}^t \left(\frac{p_i-1}{p_i}\right) \geq \prod_{i=1}^t \left(\frac{i}{i+1}\right) = \frac{t!}{(t+1)!} = \frac{1}{t+1}. \tag{4}$$

Clearly $t \leq \log_2 m$ since $2^t \leq \prod_{i=1}^t p_i \leq m$. Combining this fact with equations (3) and (4) gives the desired result. ■

For n a 1024-bit integer, $\phi(n)$ is also at most 1024 bits long, so the fraction of elements in $\mathbf{Z}_{\phi(n)}$ that also lie in $\mathbf{Z}^*_{\phi(n)}$ is at least 1/1025. Therefore, the expected number of random trials before Alice finds a number in $\mathbf{Z}^*_{\phi(n)}$ is provably at most 1025 and is most likely much smaller.

5 Euclidean algorithm

To test if $d \in \mathbf{Z}^*_{\phi(n)}$, Alice can test if $\gcd(d, \phi(n)) = 1$. How does she do this?

The greatest common divisor of two numbers a and b is easily found if a and b are already given in factored form. Namely, let p_i be the i^{th} prime and write $a = \prod p_i^{e_i}$ and $b = \prod p_i^{f_i}$. Then $\gcd(a,b) = \prod p_i^{\min(e_i,f_i)}$. However, factoring is believed to be a hard problem, and no polynomial-time factorization algorithm is currently known. Indeed, if it were, then Eve could use it to break RSA, and RSA would be of no interest as a cryptosystem.

Euclid's algorithm is remarkable, not only because it was discovered a very long time ago, but also because it works without knowing the factorization of a and b. It relies on the equation

$$\gcd(a,b) = \gcd(a-b.\, b) \tag{5}$$

which holds when $a \geq b$. This allows the problem of computing $\gcd(a, b)$ to be reduced to the problem of computing $\gcd(a - b, b)$, which is "smaller" if $b > 0$. Here we measure the size of the problem (a, b) by the sum $a + b$ of the two arguments. (5) leads in turn leads to an easy recursive algorithm:

```
int gcd(int a, int b)
{
  if ( a < b ) return gcd(b, a);
  else if ( b == 0 ) return a;
  else return gcd(a-b, b);
}
```

Nevertheless, this algorithm is not very efficient, as you will quickly discover if you attempt to use it, say, to compute $\gcd(1000000, 2)$.

Repeatedly applying (5) to the pair (a, b) until it can't be applied any more produces the sequence of pairs $(a, b), (a - b, b), (a - 2b, b), \ldots, (a - qb, b)$. The sequence stops when $a - qb < b$. But the number of times you can subtract b from a is just the quotient $\lfloor a/b \rfloor$, and the amount $a - qb$ that is left is just the remainder $a \bmod b$. Hence, one can go directly from the pair (a, b) to the pair $(a \bmod b, b)$. Since $a \bmod b < b$, it is also convenient to swap the elements of the pair. This results in the Euclidean algorithm (in C notation):

```
int gcd(int a, int b)
{
  if ( b == 0 ) return a;
  else return gcd(b, a % b);
}
```

6 Diophantine equations and modular inverses

Now that Alice knows how to choose $d \in \mathbf{Z}^*_{\phi(n)}$, how does she find e? That is, how does she solve (1)? Note that e, if it exists, is a multiplicative inverse of $d \pmod{n}$, that is, a number that, when multiplied by d, gives 1.

Equation (1) is an instance of the general Diophantine equation

$$ax + by = c \tag{6}$$

Here, a, b, c are given integers. A solution consists of integer values for the unknowns x and y. To put (1) into this form, we note that $ed \equiv 1 \pmod{\phi(n)}$ iff $ed = 1 + u\phi(n)$ for some integer u. This is seen to be an equation in the form of (6) where the unknowns x and y are e and u, respectively, and $a = d$, $b = -\phi(n)$, and $c = 1$.

7 Extended Euclidean algorithm

It turns out that (6) has a solution iff $\gcd(a, b) \mid c$. It can also be solved by a process akin to the Euclidean algorithm, which we call the *Extended Euclidean algorithm*. Here's how it works. The algorithm generates a sequence of triples of numbers (r_i, u_i, v_i), each satisfying the invariant

$$r_i = au_i + bv_i \tag{7}$$

The first two triples are $(a, 1, 0)$ and $(b, 0, 1)$. The algorithm generates triple $i + 2$ from triples i and $i + 1$ much the same as the Euclidean algorithm generates $(a \bmod b)$ from a and b. More precisely, let $q_{i+1} = \lfloor r_i / r_{i+1} \rfloor$. Then

$$\begin{aligned} r_{i+2} &= r_i - q_{i+1} r_{i+1} \\ u_{i+2} &= u_i - q_{i+1} u_{i+1} \\ v_{i+2} &= v_i - q_{i+1} v_{i+1} \end{aligned}$$

Note that $r_{i+2} = (r_i \bmod r_{i+1})$, so one sees that the sequence of generated pairs (r_1, r_2), (r_2, r_3), $(r_3, r_4), \ldots$, is exactly the same as the sequence of pairs generated by the Euclidean algorithm. Like the Euclidean algorithm, we stop when $r_t = 0$. Then $r_{t-1} = \gcd(a, b)$, and from (7) it follows that

$$\gcd(a, b) = a u_{t-1} + b v_{t-1} \tag{8}$$

If $c = \gcd(a, b)$, then $x = u_{t-1}$ and $y = v_{t-1}$ solves (6). If $c = k \gcd(a, b)$, then $x = k u_{t-1}$ and $y = k v_{t-1}$ solves (6). Otherwise, $\gcd(a, b)$ does not divide c, and one can show that (6) has no solution. See Handout 4 for further details, as well as for a discussion of how many solutions (6) has and how to find all solutions.

8 Probabilistic Primality Testing

The remaining problem for generating an RSA key is how to find two large primes p and q that, when multiplied together, form the RSA modulus n.

Alice uses a similar procedure to find p and q as she did to find a suitable d. Namely, she repeatedly chooses p and q at random from numbers of the right bit length until she finds numbers with the right properties. All we need for RSA are that p and q be distinct primes. This raises the same two questions we had in choosing d: How dense are the "good" values of p and q (that is, how likely is a chosen number to be a prime), and how does Alice test if a number is prime? We defer the first question and look at the question of primality testing.

Until very recently, no deterministic polynomial time algorithm was known for testing primality, and even now it is not known whether any deterministic algorithm is feasible in practice. However, there do exist fast *probabilistic* algorithms for testing primality. These algorithms consist of a set of *tests*. Each test operates on a number n and either *succeeds* in proving that n is composite (not prime), or it *fails* to show that n is composite. There are two reasons that a test might fail. One possibility is that n really is composite, but the test is not able to show that fact. The other possibility is that n is prime (and hence no test could prove otherwise).

The key to a useful probabilistic primality algorithm is to find a set of tests with the property that for every composite number n, a fraction $c > 0$ of the tests succeed on n. Suppose for simplicity that $c = 1/2$ and one applies 100 randomly-chosen tests to n. If any of them succeeds, we have a proof that n is composite. If all fail, we don't know whether or not n is prime or composite. But what we do know is that if n is composite, the probability that all 100 tests fail is only $1/2^{100}$.

In practice, what we do is to choose candidates for p and q at random and apply some fixed number of randomly-chosen tests to each candidate, rejecting the candidate if it proves to be composite. We keep the candidate (and assume it to be prime) if all of the tests for compositeness fail. We never know whether or not our resulting numbers p and q really are prime, but we can adjust the parameters to reduce the probability to an acceptable level that we will end up a number p or q that is not prime (and hence that we have unknowingly generated a bad RSA key).

9 Chinese Remainder Theorem

Let $n_1, n_2, \ldots, n_k$ be positive *pairwise relatively prime* positive integers[2], let $n = \prod_{i=1}^{k} n_i$, and let $a_i \in \mathbf{Z}_i$ for $i = 1, \ldots, k$. Consider the system of congruence equations with unknown x:

$$\begin{array}{rcl} x & \equiv & a_1 \pmod{n_1} \\ x & \equiv & a_2 \pmod{n_2} \\ & \vdots & \\ x & \equiv & a_k \pmod{n_k} \end{array} \tag{9}$$

The *Chinese Remainder Theorem* says that (9) has a unique solution in $\mathbf{Z}_n$.

To solve for x, let

$$N_i = n/n_i = \underbrace{n_1 n_2 \ldots n_{i-1}} \cdot \underbrace{n_{i+1} \ldots n_k},$$

and compute $M_i = N_i^{-1} \bmod n_i$, for $1 \le i \le k$. Note that $N_i^{-1} \pmod{n_i}$ exists since $\gcd(N_i, n_i) = 1$ by the pairwise relatively prime condition. We can compute N_i^{-1} using the methods of section 7. Now let

$$x = (\sum_{i=1}^{k} a_i M_i N_i) \bmod n \tag{10}$$

If $j \neq i$, then $M_j N_j \equiv 0 \pmod{n_i}$ since $n_i | N_j$. On the other hand, $M_i N_i \equiv 1 \pmod{n_i}$ by definition of M_i. Hence,

$$x \equiv \sum_{i=1}^{k} a_i M_i N_i \equiv \underbrace{0a_1 + \ldots + 0a_{i-1}} + 1a_i + \underbrace{0a_{i+1} \ldots 0a_k} \equiv a_i \pmod{n_i} \tag{11}$$

for all $1 \le i \le k$, establishing that (10) is a solution of (9).

To see that the solution is unique in $\mathbf{Z}_n$, let χ be the mapping $x \mapsto (x \bmod n_1, \ldots, x \bmod n_k)$. χ is a surjection[3] from $\mathbf{Z}_n$ to $\mathbf{Z}_{n_1} \times \ldots \times \mathbf{Z}_{n_k}$ since we have just shown for all $(a_1, \ldots, a_k) \in \mathbf{Z}_{n_1} \times \ldots \times \mathbf{Z}_{n_k}$ that there exists $x \in Z_n$ such that $\chi(x) = (a_1, \ldots, a_k)$. Since also $|\mathbf{Z}_n| = |\mathbf{Z}_{n_1} \times \ldots \times \mathbf{Z}_{n_k}|$, χ is a bijection, and (9) has only one solution in $\mathbf{Z}_n$.

9.1 Homomorphic property of χ

The bijection χ is interesting in its own right, for it establishes a one-to-one correspondence between members of $\mathbf{Z}_n$ and k-tuples $(a_1, \ldots, a_k)$ in $\mathbf{Z}_{n_1} \times \ldots \times \mathbf{Z}_{n_k}$. This lets us reason about and compute with k-tuples and then translate the results back to $\mathbf{Z}_n$.

The *homomorphic property* of χ means that performing an arithmetic operation on $x \in \mathbf{Z}_n$ corresponds to performing the similar operation on each of the components of $\chi(x)$. More precisely, let $\odot$ be one of the arithmetic operations $+$, $-$, or $\times$. If $\chi(x) = (a_1, \ldots, a_k)$ and $\chi(y) = (b_1, \ldots, b_k)$, then

$$\chi((x \odot y) \bmod n) = ((a_1 \odot b_1) \bmod n_1, \ \ldots, \ (a_k \odot b_k) \bmod n_k). \tag{12}$$

In other words, if one first performs $z = (x \odot y) \bmod n$ and then computes $z \bmod n_i$, the result is the same as if one instead first computed $a_i = (x \bmod n_i)$ and $b_i = (y \bmod n_i)$ and then performed $(a_i \odot b_i) \bmod n_i$. This relies on the fact that $(z \bmod n) \bmod n_i = z \bmod n_i$, which holds because $n_i \,|\, n$.

[2]This means that $\gcd(n_i, n_j) = 1$ for all $1 \le i < j \le k$.

[3]A *surjection* is an onto function.

9.2 RSA Decryption, part 2

In section 3, we showed that RSA decryption works when $m, c \in \mathbf{Z}_n^*$. We are now in a position to show that it works for all $m, c \in \mathbf{Z}_n$.

Let $n = pq$ be an RSA modulus, p, q distinct primes, and let e and d be the RSA encryption and decryption exponents, respectively. We show $m^{ed} \equiv m \pmod{n}$ for all $m \in \mathbf{Z}_n$.

Define $a = (m \bmod p)$ and $b = (m \bmod q)$, so

$$\begin{aligned} m &\equiv a \pmod{p} \\ m &\equiv b \pmod{q} \end{aligned} \tag{13}$$

Raising both sides to the power ed gives

$$\begin{aligned} m^{ed} &\equiv a^{ed} \pmod{p} \\ m^{ed} &\equiv b^{ed} \pmod{q} \end{aligned} \tag{14}$$

We now argue that $a^{ed} \equiv a \pmod{p}$. If $a \equiv 0 \pmod{p}$, then obviously $a^{ed} \equiv 0 \equiv a \pmod{p}$. If $a \not\equiv 0 \pmod{p}$, then $\gcd(a, p) = 1$ since p is prime, so $a \in \mathbf{Z}_p^*$. By Euler's theorem,

$$a^{\phi(p)} \equiv 1 \pmod{p}$$

Since $ed \equiv 1 \pmod{\phi(n)}$, we have $ed = 1 + u\phi(n) = 1 + u\phi(p)\phi(q)$ for some integer u. Hence,

$$a^{ed} \equiv a^{1+u\phi(p)\phi(q)} \equiv a \cdot \left(a^{\phi(p)}\right)^{u\phi(q)} \equiv a \cdot 1^{u\phi(q)} \equiv a \pmod{p} \tag{15}$$

Similarly,

$$b^{ed} \equiv b \pmod{q} \tag{16}$$

Combining the pair (14) with (15) and (16) yields

$$\begin{aligned} m^{ed} &\equiv a \pmod{p} \\ m^{ed} &\equiv b \pmod{q} \end{aligned}$$

Thus, m^{ed} is a solution to the system of equations

$$\begin{aligned} x &\equiv a \pmod{p} \\ x &\equiv b \pmod{q} \end{aligned} \tag{17}$$

From (13), m is also a solution of (17). By the Chinese Remainder Theorem, the solution to (17) is unique modulo n, so $m^{ed} \equiv m \pmod{n}$ as desired.

10 Probabilistic Primality Testing, part 2

10.1 Tests of compositeness

In section 8, we described a probabilistic algorithm based on a set of tests. In a little greater detail, a *test of compositeness* is a set $T = \{\tau_1, \ldots, \tau_s\}$, where $\tau_i : \mathbf{Z} \to \{\mathsf{true}, \mathsf{false}\}$ has the property that

$$\tau_a(n) = \mathsf{true} \Rightarrow n \text{ is composite.}$$

If $\tau_a(n) = \mathsf{true}$, we say that $\tau_a(n)$ *succeeds*, and a is a *witness* to the compositeness of n. If $\tau_a(n) = \mathsf{false}$, then the test *fails* and gives no information about the compositeness of n. Clearly, if n is prime, then all τ_a fail on n, but if n is composite, then $\tau_a(n)$ may either succeed or fail.

Here are two examples of tests for compositeness.

1. Let $\delta_a(n) = (2 \leq a \leq n-1 \text{ and } a|n)$. Test δ_a succeeds on n if a is a proper divisor of n, which indeed implies that n is composite. Thus, $\{\delta_a\}_{a \in \mathbf{Z}}$ is a valid test of compositeness. Unfortunately, it isn't very useful in a probabilistic primality algorithm since the number of tests that succeed when n is composite are too small. For example, if $n = pq$ for p, q prime, then the *only* tests that succeed are δ_p and δ_q.

2. Let $\zeta_a(n) = (2 \leq a \leq n-1 \text{ and } a^{n-1} \not\equiv 1 \pmod{n})$. By Fermat's theorem, if p is prime and $\gcd(a, p) = 1$, then $a^{p-1} \equiv 1 \pmod{p}$. Hence, if $\zeta_a(n)$ succeeds, it must be the case that n is *not* prime. This shows that $\{\zeta_a\}_{a \in \mathbf{Z}}$ is a valid test of compositeness. For this test to be adequate for a probabilistic primality algorithm, we would need to know that for all composite numbers n, a significant fraction of the tests ζ_a succeed on n. Unfortunately, there are certain compositeness numbers n called *pseudoprimes* for which all of the tests ζ_a fail. Such n are fairly rare, but they do exist. The ζ_a tests are unable to distinguish pseudoprimes from true primes, so they are not adequate for testing primality

We will return to this topic later when we have developed sufficient number theory to present a test of compositeness that does have the properties need to make it useful in probabilistic primality algorithms.

10.2 Prime Number Theorem

Even assuming Alice has a feasible algorithm for testing whether or not an arbitrary number n is prime, she still has the problem of finding a prime. If the primes are sufficiently dense, then it works for her to simply choose large numbers at random, testing them in turn until she encounters one that is prime.

The *prime number theorem* allows her to estimate on how many numbers she will have to try before encountering a primer. Let $\pi(n)$ be the number of numbers $\leq n$ that are prime. For example, $\pi(10) = 4$ since there are four primes ≤ 10, namely, 2, 3, 5, 7. The prime number theorem asserts that $\pi(n)$ is "approximately"[4] $n/(\ln n)$, where $\ln n$ is the natural logarithm $(\log_e)$ of n. Alice's chance of a randomly picked number in $\mathbf{Z}_n$ being prime is given by the ratio $\pi(n)/n \approx 1/(\ln n)$. The expected number of trials before Alice encounters a prime is the inverse of that probability, that is, $\ln n$. For example, if $n = 2^{1024}$, then the expected number of random probes to find a prime in $\mathbf{Z}_n$ is $\ln n = 1024 \ln 2 = 1024 \times 0.693 \ldots \approx 710$.

[4]We ignore the critical issue of how good an approximation this is in these notes. The interested reader is referred to a good mathematical text on number theory.