YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 467a: Cryptography and Computer Security — *Notes 9 (rev. 1)*
Professor M. J. Fischer — *October 5, 2006*

Lecture Notes 9

43 Generating RSA Encryption and Decryption Exponents

We showed in section 42 (lecture notes 8) that RSA decryption works for $m \in \mathbf{Z}_n^*$ if e and d are chosen so that

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

that is, d is e^{-1} (the inverse of e) in $\mathbf{Z}_{\phi(n)}^*$.

We now 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 $e \in \mathbf{Z}_{\phi(n)}^*$ and then solve (1) for d. We will show how to do this in Sections 45 and 46 below.

However, there is another issue, namely, how does Alice find random $e \in \mathbf{Z}_{\phi(n)}^*$? 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 $\phi(\phi(n))$ (the size of $\mathbf{Z}_{\phi(n)}^*$) 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 candidate for e.

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 1 *For all $m \geq 2$,*

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

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

$$\frac{|\mathbf{Z}_m^*|}{|\mathbf{Z}_m|} = \frac{\phi(m)}{m} = \frac{\prod_{i=1}^t (p_i - 1)p_i^{e_i-1}}{\prod_{i=1}^t p_i^{e_i}} = \prod_{i=1}^t \left(\frac{p_i - 1}{p_i}\right). \tag{2}$$

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

$$\prod_{i=1}^t \left(\frac{p_i - 1}{p_i}\right) \geq \prod_{i=1}^t \left(\frac{i}{i+1}\right) = \frac{1}{2} \cdot \frac{2}{3} \cdot \frac{3}{4} \cdots \frac{t}{t+1} = \frac{1}{t+1}. \tag{3}$$

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

For n a 1024-bit integer, $\phi(n) < n < 2^{1024}$. Hence, $\log_2(\phi(n)) < 1024$, so $\lfloor \log_2(\phi(n)) \rfloor \leq 1023$. By Theorem 1, the fraction of elements in $\mathbf{Z}_{\phi(n)}$ that also lie in $\mathbf{Z}_{\phi(n)}^*$ is at least 1/1024. Therefore, the expected number of random trials before Alice finds a number in $\mathbf{Z}_{\phi(n)}^*$ is provably at most 1024 and is most likely much smaller.

44 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 basic ideas underlying the Euclidean algorithm were sketched in section 40.2 (lecture notes 8). 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

$$\gcd(a, b) = \gcd(a - b,\, b) \tag{4}$$

which holds when $a \geq b \geq 0$. 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. (4) 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 (4) 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);
}
```

45 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

$$ax + by = c \tag{5}$$

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 $ed + u\phi(n) = 1$ for some integer u. This is seen to be an equation in the form of (5) where the unknowns x and y are e and u, respectively, and the coefficients a, b, c are d, $\phi(n)$, and 1, respectively.

46 Extended Euclidean algorithm

It turns out that (5) is closely related to the greatest common divisor, for it has a solution iff $\gcd(a, b) \mid c$. It can 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 $T_i = (r_i, u_i, v_i)$, each satisfying the invariant

$$r_i = au_i + bv_i \geq 0. \tag{6}$$

The first triple T_1 is $(a, 1, 0)$ if $a \geq 0$ and $(-a, -1, 0)$ if $a < 0$. The second trip T_2 is $(b, 0, 1)$ if $b \geq 0$ and $(-b, 0, -1)$ if $b < 0$.

The algorithm generates T_{i+2} from T_i and T_{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 $T_{i+2} = T_i - q_{i+1} T_{i+1}$, that is,

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

Note that $r_{i+2} = (r_i \bmod r_{i+1})$, [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 (6) it follows that

$$\gcd(a, b) = au_{t-1} + bv_{t-1} \tag{7}$$

Returning to equation (5), if $c = \gcd(a, b)$, then $x = u_{t-1}$ and $y = v_{t-1}$ is a solution. If c is a multiple of $\gcd(a, b)$, then $c = k \gcd(a, b)$ for some k and $x = ku_{t-1}$ and $y = kv_{t-1}$ is a solution. Otherwise, $\gcd(a, b)$ does not divide c, and one can show that (5) has no solution. See Handout 5 for further details, as well as for a discussion of how many solutions (5) has and how to find all solutions.

Here's an example. Suppose one wants to solve the equation

$$31x - 45y = 3 \tag{8}$$

In this example, $a = 31$ and $b = -45$. We begin with the triples

$$\begin{aligned} T_1 &= (31, 1, 0) \\ T_2 &= (45, 0, -1) \end{aligned}$$

The computation is shown in the following table:

i	r_i	u_i	v_i	q_i
1	31	1	0	
2	45	0	-1	0
3	31	1	0	1
4	14	-1	-1	2
5	3	3	2	4
6	2	-13	-9	1
7	1	16	11	2
8	0	-45	-31	

[1] This follows from the division theorem, which can be written in the form $a = b \cdot \lfloor a/b \rfloor + (a \bmod b)$.

From $T_7 = (1, 16, 11)$ and (6), we obtain

$$1 = a \times 16 + b \times 11$$

Plugging in values $a = 31$ and $b = -45$, we compute

$$31 \times 16 + (-45) \times 11 = 496 - 495 = 1$$

as desired. The solution to (8) is then $x = 3 \times 16 = 48$ and $y = 3 \times 11 = 33$.

47 Generating RSA Modulus

We finally turn to the question of generating the RSA modulus, $n = pq$. Recall that the numbers p and q should be random distinct primes of about the same length. The method for finding p and q is similar to the "guess-and-check" method used in Section 43 to find random numbers in $\mathbf{Z}_n^*$. Namely, keep generating random numbers p of the right length until a prime is found. Then keep generating random numbers q of the right length until one is found that is prime and different from p.

To generate a random prime of a given length, say k bits long, generate $k-1$ random bits, put a "1" at the front, regard the result as binary number, and test if it is prime. We defer the question of how to test if the number is prime and look now at the expected number of trials before this procedure will terminate.

The above procedure samples uniformly from the set $B_k = \mathbf{Z}_{2^k} - \mathbf{Z}_{2^{k-1}}$ of binary numbers of length exactly k. Let p_k be the fraction of elements in B_k that are prime. Then the expected number of trials to find a prime will be $1/p_k$. While p_k is difficult to determine exactly, the celebrated *Prime Number Theorem* allows us to get a good estimate on that number.

Let $\pi(n)$ be the number of numbers $\leq n$ that are prime. For example, $\pi(10) = 4$ since there are four primes $\leq$ 10, namely, 2, 3, 5, 7. The prime number theorem asserts that $\pi(n)$ is "approximately"[2] $n/(\ln n)$, where $\ln n$ is the natural logarithm $(\log_e)$ of n. The chance that a randomly picked number in $\mathbf{Z}_n$ is prime is then $\pi(n-1)/n \approx ((n-1)/\ln(n-1))/n \approx 1/(\ln n)$.

Since $B_k = \mathbf{Z}_{2^k} - \mathbf{Z}_{2^{k-1}}$, we have

$$\begin{aligned} p_k &= \frac{\pi(2^k - 1) - \pi(2^{k-1} - 1)}{2^{k-1}} \\ &= \frac{2\pi(2^k - 1)}{2^k} - \frac{\pi(2^{k-1} - 1)}{2^{k-1}} \\ &\approx \frac{2}{\ln 2^k} - \frac{1}{\ln 2^{k-1}} \approx \frac{1}{\ln 2^k} \\ &= \frac{1}{k \ln 2}. \end{aligned}$$

Hence, the expected number of trials before success is approximately $k \ln 2$. For $k = 512$, this works out to $512 \times 0.693\ldots \approx 355$.

The remaining problem for generating an RSA key is how to test if a large number is prime. 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, which we now discuss

[2]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.

48 Probabilistic Primality Tests

A deterministic test for primality is a procedure that, given as input a number n, correctly returns the answer 'composite' or 'prime'.[3] To arrive at a probabilistic algorithm, we extend the notion of a deterministic primality test in two ways: We give it an extra "helper" string a, and we allow it to answer '?', meaning "I don't know". Given input n and helper string a, such an output may correctly answer either 'composite' or '?' when n is composite, and it may correctly answer either 'prime' or '?' when n is prime. If the algorithm gives a non-"?" answer, we say that the helper string a is a *witness* to that answer.

Given an extended primality test $T(n, a)$, we can use it to build a strong probabilistic primality testing algorithm. On input n, do the following:

```
Algorithm $P_1(n)$:
    **repeat forever** {
        Generate a random helper string $a$;
        Let $r = T(n, a)$;
        **if** $(r \neq$ '?') **return** $r$;
    };
```

This algorithm has the property that it might not terminate (in case there are no witnesses to the correct answer for n), but when it does terminate, the answer is correct.

Unfortunately, we do not know of any test that results in an efficient strong probabilistic primality testing algorithm. However, the above algorithm can be weakened slightly and still be useful. What we do is to add a parameter t which is the maximum number of trials that we are willing to perform. The algorithm then becomes:

```
Algorithm $P_2(n, t)$:
    **repeat** $t$ times {
        Generate a random helper string $a$;
        Let $r = T(n, a)$;
        **if** $(r \neq$ '?') **return** $r$;
    }
    **return** '?';
```

Now the algorithm is allowed to give up and return '?', but only after trying t times to find the correct answer. If there are lots of witnesses to the correct answer, then the probability will be high of finding one, so most of the time the algorithm will succeed. But even this assumption is stronger than we know how to achieve.

49 Tests of compositeness

The tests that we will present are asymmetric. When n is composite, there are many witnesses to that effect, but when n is prime, there are none. Hence, the test either outputs 'composite' or '?' but never 'prime'. We call these *tests of compositeness* since an answer of 'composite' means that n is definitely composite, but these tests can never say for sure that n is prime.

When algorithm P_2 uses a test of compositeness, an answer of 'composite' likewise means that n is definitely composite. Moreover, if there are many witnesses to n's being composite and t is

[3] We assume that $n \geq 2$, so that n is either composite or prime.

sufficiently large, then the probability that $P_2(n, t)$ outputs 'composite' will be high. However, if n is prime, then both the test and P_2 will always output '?'. It is tempting to interpret P_2's output of '?' to mean "n is probably prime", but of course, it makes no sense to say that n is probably prime; n either is or is not prime. But what does make sense is to say that the probability is very small that P_2 answers '?' when n is composite.

In practice, we will indeed interpret the output '?' to mean 'prime', but we understand that the algorithm has the possibility of giving the wrong answer when n is composite. Whereas before our algorithm would only report an answer when it was sure and would answer '?' otherwise, now we are considering algorithms that are allowed to make mistakes with (hopefully) small probability.