CPSC 467b: Cryptography and Computer Security

Lecture 10

Michael J. Fischer

Department of Computer Science
Yale University

February 10, 2010

1. Generating RSA Encryption and Decryption Exponents
2. Diophantine equations and modular inverses
   - General form
   - Extended Euclidean algorithm
3. Generating RSA Modulus
   - Finding primes by guess and check
   - Density of primes
4. Primality Tests
   - Strong primality tests
   - Weak tests of compositeness

Recall RSA exponent requirement

We showed in the last lecture 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 $\mathbf{Z}_{\phi(n)}^*$.

How does Alice choose e and d to satisfy (1)?

- Choose a random integer $e \in \mathbf{Z}_{\phi(n)}^*$.
- Solve (1) for d.

Sampling from $\mathbf{Z}_n^*$

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 also lies in $\mathbf{Z}_{\phi(n)}^*$.

A candidate element e lies in $\mathbf{Z}_{\phi(n)}^*$ iff $\gcd(e, \phi(n)) = 1$, which can be computed efficiently using the Euclidean algorithm.[1]

[1] $\phi(n)$ itself is easily computed for an RSA modulus $n = pq$ since $\phi(n) = (p-1)(q-1)$ and Alice knows p and q.

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 $\mathbf{Z}_{\phi(n)}$), Alice 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.
Example:

$$m = |\mathbf{Z}_m| = 2 \cdot 3 \cdot 5 \cdot 7 = 210$$
$$\phi(m) = |\mathbf{Z}^*_m| = 1 \cdot 2 \cdot 4 \cdot 6 = 48.$$

In this case, the probability that a randomly-chosen element of $\mathbf{Z}_m$ falls in $\mathbf{Z}^*_m$ is only $48/210 = 8/35 = 0.228\ldots$.

A lower bound on $\phi(m)/m$

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

Theorem

For all $m \geq 2$,

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

Proof.

Write $m = \prod_{i=1}^{t} p_i^{e_i}$, 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

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

Proof.

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

To estimate the size of $\prod_{i=1}^{t}(p_i-1)/p_i$, note that

$$\left(\frac{p_i-1}{p_i}\right) \geq \left(\frac{i}{i+1}\right).$$

This follows since $(x-1)/x$ is monotonic increasing in x, and $p_i \geq i+1$. Then

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

Proof.

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

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

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 with equations (2) and (3) gives the desired result.

$$\frac{|\mathbf{Z}_m^*|}{|\mathbf{Z}_m|} \geq \frac{1}{t+1} \geq \frac{1}{1 + \lfloor \log_2 m \rfloor}. \quad (4)$$

Expected difficulty of choosing RSA exponent e

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 the theorem, the fraction of elements in $\mathbf{Z}_{\phi(n)}$ that also lie in $\mathbf{Z}^*_{\phi(n)}$ is at least

$$\frac{1}{1 + \lfloor \log_2 \phi(n) \rfloor} \geq \frac{1}{1024}.$$

Therefore, the expected number of random trials before Alice finds a number in $\mathbf{Z}^*_{\phi(n)}$ is provably at most 1024 and is likely much smaller.

RSA decryption exponent d

After Alice chooses $e \in \mathbf{Z}^*_{\phi(n)}$, how does she find d?

That is, how does she solve

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

Note that d, if it exists, is a multiplicative inverse of $e \pmod{\phi(n)}$, that is, a number that, when multiplied by e, gives $1 \pmod{\phi(n)}$.

General Diophantine equations

A *Diophantine equation* is a linear equation in two unknowns over the integers.

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

Here, a, b, c are given integers. A solution consists of integer values for the unknowns x and y that make (5) true.

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 d and u, respectively, and the coefficients a, b, c are $e, \phi(n), 1$, respectively.

Existence of solution

Theorem

The Diophantine equation

$$ax + by = c$$

has a solution over $\mathbf{Z}$ *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*.

Extended Euclidean algorithm

The algorithm generates a sequence of triples of numbers $T_i = (r_i, u_i, v_i)$, each satisfying the invariant

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

Extended Euclidean algorithm

The algorithm generates a sequence of triples of numbers $T_i = (r_i, u_i, v_i)$, each satisfying the invariant

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

$$T_1 = \begin{cases} (a, 1, 0) & \text{if } a \geq 0 \\ (-a, -1, 0) & \text{if } a < 0 \end{cases}$$

$$T_2 = \begin{cases} (b, 0, 1) & \text{if } b \geq 0 \\ (-b, 0, -1) & \text{if } b < 0 \end{cases}$$

Extended Euclidean algorithm

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

T_{i+2} is obtained by subtracting a multiple of T_{i+1} from from T_i so that $r_{i+2} < r_{i+1}$. This is similar to the way the Euclidean algorithm obtains (a mod b) from a and b.

Extended Euclidean algorithm

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

T_{i+2} is obtained by subtracting a multiple of T_{i+1} from from T_i so that $r_{i+2} < r_{i+1}$. This is similar to the way the Euclidean algorithm obtains (a mod b) from a and b.

In detail, 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,

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

Extended Euclidean algorithm

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

T_{i+2} is obtained by subtracting a multiple of T_{i+1} from from T_i so that $r_{i+2} < r_{i+1}$. This is similar to the way the Euclidean algorithm obtains (a mod b) from a and b.

In detail, 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,

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

The sequence of generated pairs (r_1, r_2), (r_2, r_3), (r_3, r_4), ... is exactly the same as the sequence generated by the Euclidean algorithm. We stop when $r_t = 0$. Then $r_{t-1} = \gcd(a, b)$.

Extended Euclidean algorithm

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

From (6) it follows that

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

Finding all solutions

Returning to the original equation,

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

if $c = \gcd(a, b)$, then $x = u_{t-1}$ and $y = v_{t-1}$ is a solution.

If $c = k \cdot \gcd(a, b)$ is a multiple of $\gcd(a, b)$, then $x = ku_{t-1}$ 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 6 for further details, as well as for a discussion of how many solutions (5) has and how to find all solutions.

Example of extended Euclidean algorithm

Suppose one wants to solve the equation

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

Here, $a = 31$ and $b = -45$. We begin with the triples

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

Computing the triples

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	

Extracting the solution

From $T_7 = (1, 16, 11)$, we obtain the solution $x = 16$ and $y = 11$ to the equation

$$1 = 31x - 45y$$

We can check this by substituting for x and y:

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

The solution to

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

is then $x = 3 \cdot 16 = 48$ and $y = 3 \cdot 11 = 33$.

Recall RSA modulus

Recall the RSA modulus, $n = pq$. 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 to find random numbers in $\mathbf{Z}_m^*$.

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 a prime different from p is found.

Generating random primes of a given length

To generate a k-bit prime:

- 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.

Expected number of trials to find a prime

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 is $1/p_k$.

While p_k is difficult to determine exactly, the celebrated *Prime Number Theorem* allows us to get a good estimate on that number.

Prime number function

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.

Theorem (prime number theorem)

$$\pi(n) \approx n/(\ln n)$$

where $\ln n$ *is the natural logarithm* $\log_e n$.

Notes:

- We ignore the critical issue of how good an approximation this is. The interested reader is referred to a good mathematical text on number theory.
- Here $e = 2.71828\ldots$ is the base of the natural logarithm, not to be confused with the RSA encryption exponent, which, by an unfortunate choice of notation, we also denote by e.

Likelihood of randomly finding a prime

The chance that a randomly picked number in $\mathbf{Z}_n$ is prime is

$$\frac{\pi(n-1)}{n} \approx \frac{n-1}{n \cdot \ln(n-1)} \approx \frac{1}{\ln n}.$$

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 $\approx k \ln 2$.
For $k = 512$, this works out to $512 \times 0.693\ldots \approx 355$.

Algorithms for testing primality

The remaining problem for generating an RSA key is how to test if a large number is prime.

- At first sight, this problem seems as hard as factoring.

Algorithms for testing primality

The remaining problem for generating an RSA key is how to test if a large number is prime.

- At first sight, this problem seems as hard as factoring.
- Indeed, no deterministic polynomial time algorithm was known for testing primality until 2002 when Manindra Agrawal, Neeraj Kayal and Nitin Saxena found a deterministic primality test which runs in time $\tilde{O}((\log n)^{12})$. This was later improved to $\tilde{O}((\log n)^6)$.

Algorithms for testing primality

The remaining problem for generating an RSA key is how to test if a large number is prime.

- At first sight, this problem seems as hard as factoring.
- Indeed, no deterministic polynomial time algorithm was known for testing primality until 2002 when Manindra Agrawal, Neeraj Kayal and Nitin Saxena found a deterministic primality test which runs in time $\tilde{O}((\log n)^{12})$. This was later improved to $\tilde{O}((\log n)^6)$.
- Even now it is not known whether any deterministic algorithm is feasible in practice.

Algorithms for testing primality

The remaining problem for generating an RSA key is how to test if a large number is prime.

- At first sight, this problem seems as hard as factoring.
- Indeed, no deterministic polynomial time algorithm was known for testing primality until 2002 when Manindra Agrawal, Neeraj Kayal and Nitin Saxena found a deterministic primality test which runs in time $\tilde{O}((\log n)^{12})$. This was later improved to $\tilde{O}((\log n)^6)$.
- Even now it is not known whether any deterministic algorithm is feasible in practice.
- However, there do exist fast *probabilistic* algorithms for testing primality.

Tests for primality

A *primality test* is a deterministic procedure that, given as input an integer $n \geq 2$, correctly returns the answer '**composite**' or '**prime**'.

To arrive at a probabilistic algorithm, we extend the notion of a primality test in two ways:

1. We give it an extra "helper" string a.
2. We allow it to answer '**?**', meaning "I don't know".

Given input n and helper string a, such an algorithm 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.

Probabilistic primality testing algorithm

We can build a probabilistic primality testing algorithm from an extended primality test $T(n, a)$.

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.

Trading off non-termination against possibility of failure

By bounding the number of trials, termination is guaranteed at the cost of possible failure. Let t be 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.

Strong primality tests

A primality test $T(n, a)$ is *strong* if there are "many" witnesses to the correct answer.

For a strong test, the probability will be "high" of finding a witness and the algorithm will usually succeed.

Unfortunately, we do not know of any strong primality test that has lots of witnesses to the correct answer for every $n \geq 2$.

Fortunately, a weaker test can still be useful.

Weak tests

A *weak test of compositeness* is only required to have many witnesses to the correct answer '**composite**' when n is in fact composite.

When n is prime, a weak test always answers '**?**', so there are no witnesses to n being prime.

Hence, the test either outputs '**composite**' or '**?**' but never '**prime**'.

An answer of '**composite**' means that n is definitely **composite**, but these tests can never say for sure that n is prime.

Algorithm P_2 using a weak test

When algorithm P_2 uses a weak test of compositeness, an answer of '**composite**' means that n is definitely composite.

Assuming n is composite, there are many witnesses to n's being composite, and t is sufficiently large, then the probability that $P_2(n, t)$ outputs '**composite**' will be high.

However, if n is prime, then both the underlying weak test T and P_2 itself will always output '**?**'. It is tempting to interpret P_2's output of '**?**' to mean "n is probably prime".

However, 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 small that P_2 answers '**?**' when n is in fact composite.

Finding a random prime

Algorithm GenPrime(k):

```
const int t=20;
do {
    Generate a random k-bit integer x;
} while ( P_2(x, t) == 'composite' );
return x;
```

The number x that GenPrime() returns has the property that P_2 failed to find a witness, but there is still the possibility that x is composite.

Success probability for GenPrime(k)

We are interested in the probability that the result returned by GenPrime(k) is prime.

This probability depends on both the failure probability of P_2 and also on the density of primes in the set being sampled.

The fewer primes there are, the more composite numbers are likely to be tried before a prime is encountered, and the more opportunity there is for P_2 to fail.

What would happen if the set being sampled contained *only* composite numbers?