YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 467a: Cryptography and Computer Security

Notes 11 (rev. 2)
October 6, 2005

Professor M. J. Fischer

Lecture Notes 11

1 Generating RSA Modulus: Part 2

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

1.1 Probabilistic Primality Testing

A deterministic test for primality is a procedure that, given as input a number n, correctly returns the answer 'composite' or 'prime'.[1] 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** '?';
```

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

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.

1.2 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 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.

1.3 Formal definition

Formally, 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.

A test of compositeness T is *useful* if there is a feasible algorithm $T(n,a)$ that computes $\tau_a(n)$, and 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 to choose RSA primes p and q is to choose numbers at random and apply some fixed number of randomly-chosen tests to each candidate,[2] 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).

1.4 Example tests of compositeness

Here are two examples of tests for compositeness.

[2]This is what Algorithm P_2 does.

1. Let $\delta_a(n) = (2 \le a \le n-1 \text{ and } a|n)$. 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 $\zeta_a(n) = (2 \le a \le n-1 \text{ and } a^{n-1} \not\equiv 1 \pmod n)$. 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 tests of compositeness that do have the properties needed to make them useful in probabilistic primality algorithms.

2 Chinese Remainder Theorem

We now return to a basic result of number theory that will be used later.

Let $n_1, n_2, \ldots, n_k$ be positive *pairwise relatively prime* positive integers[3], 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:

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned} \tag{1}$$

The *Chinese Remainder Theorem* says that (1) 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 lecture 10, section 5. Now let

$$x = (\sum_{i=1}^{k} a_i M_i N_i) \bmod n \tag{2}$$

If $j \ne i$, 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,

$$x \equiv \sum_{i=1}^{k} a_i M_i N_i \equiv \underbrace{0a_1 + \ldots + 0a_{i-1}} + 1a_i + \underbrace{0a_{i+1} \ldots 0a_k} \equiv a_i \pmod{n_i} \tag{3}$$

for all $1 \le i \le k$, establishing that (2) is a solution of (1).

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

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[4] 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 (1) has only one solution in $\mathbf{Z}_n$.

2.1 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

$$\chi((x \odot y) \bmod n) = ((a_1 \odot b_1) \bmod n_1,\ \ldots,\ (a_k \odot b_k) \bmod n_k). \tag{4}$$

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 $n_i \,|\, n$.

2.2 RSA Decryption Works for All of Z_n

In lecture 9, section 2, we showed that RSA decryption works when $m, c \in \mathbf{Z}_n^*$. We now show using the Chinese Remainder Theorem 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

$$\begin{aligned} m &\equiv a \pmod{p} \\ m &\equiv b \pmod{q} \end{aligned} \tag{5}$$

Raising both sides to the power ed gives

$$\begin{aligned} m^{ed} &\equiv a^{ed} \pmod{p} \\ m^{ed} &\equiv b^{ed} \pmod{q} \end{aligned} \tag{6}$$

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,

$$a^{ed} \equiv a^{1+u\phi(p)\phi(q)} \equiv a \cdot \left(a^{\phi(p)}\right)^{u\phi(q)} \equiv a \cdot 1^{u\phi(q)} \equiv a \pmod{p} \tag{7}$$

Similarly,

$$b^{ed} \equiv b \pmod{q} \tag{8}$$

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

Combining the pair (6) with (7) and (8) 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

$$\begin{aligned} x &\equiv a \pmod p \\ x &\equiv b \pmod q \end{aligned} \tag{9}$$

From (5), m is also a solution of (9). By the Chinese Remainder Theorem, the solution to (9) is unique modulo n, so $m^{ed} \equiv m \pmod n$ as desired.