CPSC 467b: Cryptography and Computer Security

Michael J. Fischer

Lecture 9
February 6, 2012

- Euler's Theorem
- Generating RSA Modulus
    - Finding primes by guess and check
    - Density of primes
- Primality Tests
    - Strong primality tests
    - Weak tests of compositeness
    - Reformulation of weak tests of compositeness
    - Examples of weak tests
- RSA Security
    - Factoring n
    - Computing $\phi(n)$
    - Finding d directly
    - Finding plaintext

Euler's Theorem

Repeated multiplication in $\mathbf{Z}_n^*$

If any element $x \in \mathbf{Z}_n^*$ is repeatedly multiplied by itself, the result is eventually 1. [1]

Example, for $x = 5 \in \mathbf{Z}_{26}^*$: 5, 25, 21, 1, 5, 25, 21, 1, ...

Let x^k denote the result of multiplying x by itself k times.
The *order of x*, written $\mathrm{ord}(x)$, is the smallest integer $k \geq 1$ for which $x^k = 1$.

Theorem

$\mathrm{ord}(x) \mid \phi(n)$. *(Recall, $\phi(n)$ is the size of $\mathbf{Z}_n^*$).*

[1] The first repeated element must be x. If not, then some $y \neq x$ is the first to repeat. The element immediately preceding each occurrence of y is yx^{-1}. But then yx^{-1} is the first to repeat, a contradiction. Hence, $x = x^{k+1}$ for some $k \geq 1$, so $x^k = x^{k+1}x^{-1} = xx^{-1} = 1$.

Euler's and Fermat's theorem

Theorem (Euler's theorem)

$x^{\phi(n)} \equiv 1 \pmod{n}$ *for all* $x \in \mathbf{Z}_n^*$.

Proof.

Since $\mathrm{ord}(x) \mid \phi(n)$, we have

$$x^{\phi(n)} \equiv (x^{\mathrm{ord}(x)})^{\phi(n)/\mathrm{ord}(x)} \equiv 1^{\phi(n)/\mathrm{ord}(x)} \equiv 1 \pmod{n}.$$

□

As a special case, we have

Theorem (Fermat's theorem)

$x^{(p-1)} \equiv 1 \pmod{p}$ *for all* x, $1 \le x \le p-1$, *where* p *is prime.*

An important corollary

Corollary

Let $r \equiv s \pmod{\phi(n)}$*. Then* $a^r \equiv a^s \pmod{n}$ *for all* $a \in \mathbf{Z}_n^*$*.*

Proof.

If $r \equiv s \pmod{\phi(n)}$, then $r = s + u\phi(n)$ for some integer u. Then using Euler's theorem, we have

$$a^r \equiv a^{s+u\phi(n)} \equiv a^s \cdot (a^u)^{\phi(n)} \equiv a^s \cdot 1 \equiv a^s \pmod{n},$$

as desired. □

Application to RSA

Recall the RSA encryption and decryption functions

$$\begin{aligned} E_e(m) &= m^e \bmod n \\ D_d(c) &= c^d \bmod n \end{aligned}$$

where $n = pq$ is the product of two distinct large primes p and q.

This corollary gives a sufficient condition on e and d to ensure that the resulting cryptosystem works. That is, we require that

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

Then $D_d(E_e(m)) \equiv m^{ed} \equiv m^1 \equiv m \pmod{n}$ for all messages $m \in \mathbf{Z}_n^*$.

Messages not in $\mathbf{Z}_n^*$

What about the case of messages $m \in \mathbf{Z}_n - \mathbf{Z}_n^*$?

There are several answers to this question.

1. 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 very small — only about $2/\sqrt{n}$.
3. RSA does in fact work for all $m \in \mathbf{Z}_n$, even though Euler's theorem fails for $m \notin \mathbf{Z}_n^*$.

Why Alice might want to avoid sending messages not in $\mathbf{Z}_n^*$

If $m \in \mathbf{Z}_n - \mathbf{Z}_n^*$, either $p \mid m$ or $q \mid m$ (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), then Eve will succeed in breaking the cryptosystem.

Why?

Why a random message is likely to be in $\mathbf{Z}_n^*$

The number of messages in $\mathbf{Z}_n - \mathbf{Z}_n^*$ 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 about $2 \cdot 2^{512}/2^{1024} = 1/2^{511}$.

Such a low-probability event will likely never occur during the lifetime of the universe.

RSA works anyway

For $m \in \mathbf{Z}_n - \mathbf{Z}_n^*$, RSA works anyway, but for different reasons.

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.

Generating RSA Modulus

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.

Prime number theorem

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 $512 \times 0.693\ldots \approx 355$.

Primality Tests

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.
- In 2002, Manindra Agrawal, Neeraj Kayal and Nitin Saxena found a deterministic primality test which runs in time $\tilde{O}(N^{12})$. This was later improved to $\tilde{O}(N^6)$. Here, N is the length of the number to be tested when written in binary, and $\tilde{O}$ hides a polylogarithmic factor in N. (See Wikipedia.)
- 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 correctly answers '**composite**' or '**prime**' for each input $n \geq 2$.

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 a 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 is "high" that a random helper string is a witness, so the algorithm P_2 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* $T(n, a)$ is only required to have many witnesses to the correct answer when n is 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.

Use of a weak test of compositeness

Let $T(n, a)$ be a weak test of compositeness. Algorithm P_2 is a "best effort" attempt to prove that n is composite.

Since T is a weak test, we can slightly simplify P_2.

```
Algorithm $P_3(n, t)$:
    **repeat** $t$ times {
        Generate a random helper string $a$;
        **if** $(T(n, a) =$ 'composite') **return** 'composite';
    }
    **return** '?';
```

P_3 returns '**composite**' just in case it finds a witness a to the compositeness of n.

Algorithm P_3 using a weak test

When algorithm P_3 answers '**composite**', n is definitely composite.

Turning this around, we have:

- If n is composite and t is sufficiently large, then with high probability, $P_3(n, t)$ outputs '**composite**'.
- If n is prime, then $P_3(n, t)$ always outputs '**?**'.

Meaning of output **?**

It is tempting to interpret '**?**' as meaning "n is probably prime".

However, it makes no sense to say that n is *probably prime*; n either is or is not prime.

It also is not true that if I guess "*prime*" whenever I see output **?** that I will be correct with high probability.
Why not?

Imagine the test is run repeatedly on $n = 15$. Every now and then the output will be **?**, but "*prime*" is *never* correct in this case.

What does make sense is to say that the probability is small that P_3 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 ( $P_3(x, t)$ == ‘composite’ );
    return $x$;
```

The number x that GenPrime() returns has the property that P_3 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 number returned by GenPrime(k) is prime.

This probability depends on *both* the failure probability of P_3 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_3 to fail.

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

Boolean test of compositeness

We now reformulate weak tests of compositeness as Boolean functions.

A Boolean function $\tau(n, a)$ can be interpreted as a weak test of compositeness by taking true to mean '**composite**' and false to mean '**?**'.

There's nothing deep here. We're just changing notation.

Meaning of a Boolean test of compositeness

Let $\tau(n, a)$ be a Boolean test of compositeness.
Write $\tau_a(n)$ to mean $\tau(n, a)$.

- If $\tau_a(n) = \text{true}$, we say that τ_a *succeeds* on n, and a is a *witness* to the compositeness of n.
- If $\tau_a(n) = \text{false}$, then τ_a *fails* on n and gives no information about the compositeness of n.

Clearly, if n is prime, then τ_a fails on n for all a, but if n is composite, then τ_a may succeed for some values of a and fail for others.

Useful tests

A test of compositeness τ is *useful* if

- there is a feasible algorithm that computes $\tau(n, a)$;
- for every composite number n, $\tau_a(n)$ succeeds for a fraction $c > 0$ of the helper strings a.

Sample use of a useful test

Suppose for simplicity that $c = 1/2$ and one computes $\tau_a(n)$ for 100 randomly-chosen values for a.

- If any of the τ_a succeeds, we have a proof a that n is composite.
- If all fail, we don't know whether or not n is prime or composite. But we do know that if n is composite, the probability that all 100 tests fail is only $1/2^{100}$.

Application to RSA

In practice, we use GenPrime(k) to choose RSA primes p and q, where the constant t is set according to the number of witnesses and the confidence levels we would like to achieve.

For $c = 1/2$, using $t = 20$ trials gives us a failure probability of about one in a million when testing a composite number a.

We previously argued that we expect to test 355 numbers, 354 of which are composite, in order to find one suitable RSA prime. For an RSA modulus $n = pq$, we then expect to test 708 composite numbers on average, giving P_3 that many opportunities to fail. Hence, the probability that the resulting RSA modulus is bad is roughly $708/10^6 \approx 1/1412$. t can be increased if this risk of failure is deemed to be too large.

Finding weak tests of compositeness

We still need to find useful weak tests of compositeness.

We begin with two simple examples. While neither is useful, they illustrate some of the ideas behind the useful tests that we will present later.

The division test $\delta_a(n)$

Let

$$\delta_a(n) = (2 \leq a \leq n-1 \text{ and } a|n).$$

Test δ_a succeeds on n iff 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 useful since the fraction of witnesses to n's compositeness is exponentially small.

For example, if $n = pq$ for p, q prime, then the *only* witnesses are p and q, and the only tests that succeed are δ_p and δ_q.

The Fermat test $\zeta_a(n)$

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 n is prime and $\gcd(a, n) = 1$, then $a^{n-1} \equiv 1 \pmod{n}$.

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 useful, we would need to know that every composite number n has a constant fraction of witnesses.

Carmichael numbers (Fermat pseudoprimes)

Unfortunately, there are certain composite numbers n called *Carmichael numbers*[2] for which there are no witnesses, and all of the tests ζ_a fail. Such n are fairly rare, but they do exist. The smallest such n is $561 = 3 \cdot 11 \cdot 17$. [3]

Hence, Fermat tests are not useful tests of compositeness according to our definition, and they are unable to distinguish Carmichael numbers from primes.

Further information on primality tests may be found in section C.9 of Goldwasser and Bellare.

[2] Carmichael numbers are sometimes called Fermat pseudoprimes.

[3] See `http://en.wikipedia.org/wiki/Carmichael_number` for further information.

RSA Security

Attacks on RSA

The security of RSA depends on the computational difficulty of several different problems, corresponding to different ways that Eve might attempt to break the system.

- Factoring n
- Computing $\phi(n)$
- Finding d directly
- Finding plaintext

We examine each in turn and look at their relative computational difficulty.

RSA factoring problem

Definition (RSA factoring problem)

Given a number n that is known to be the product of two primes p and q, find p and q.

Clearly, if Eve can find p and q, then she can compute the decryption key d from the public encryption key (e, n) (in the same way that Alice did when generating the key).

This completely breaks the system, for now Eve has the same power as Bob to decrypt all ciphertexts.

This problem is a special case of the general factoring problem. It is believed to be intractable, although it is not known to be NP-complete.

$\phi(n)$ problem

Definition ($\phi(n)$ problem)

Given a number n that is known to be the product of two primes p and q, compute $\phi(n)$.

Eve doesn't really need to know the factors of n in order to break RSA. It is enough for her to know $\phi(n)$, since that allows her to compute $d = e^{-1} \pmod{\phi(n)}$.

Computing $\phi(n)$ is no easier than factoring n. Given n and $\phi(n)$, Eve can factor n by solving the system of quadratic equations

$$\begin{aligned} n &= pq \\ \phi(n) &= (p-1)(q-1) \end{aligned}$$

for p and q using standard methods of algebra.

Decryption exponent problem

Definition (Decryption exponent problem)

Given an RSA public key (e, n), find the decryption exponent d.

Eve might somehow be able to find d directly from e and n even without the ability to factor n or to compute $\phi(n)$.

That would represent yet another attack that couldn't be ruled out by the assumption that the RSA factoring problem is hard. However, that too is not possible, as we now show.

Factoring n knowing e and d

We begin by finding unique integers s and t such that

$$2^s t = ed - 1$$

and t is odd.

This is always possible since $ed - 1 \neq 0$.

Express $ed - 1$ in binary. Then s is the number of trailing zeros and t is the value of the binary number that remains after the trailing zeros are removed.

Since $ed - 1 \equiv 0 \pmod{\phi(n)}$ and $4 \mid \phi(n)$ (since both $p - 1$ and $q - 1$ are even), it follows that $s \geq 2$.

Square roots of $1 \pmod{n}$

Over the reals, each positive number has two square roots, one positive and one negative, and no negative numbers have real square roots.

Over $\mathbf{Z}_n^*$ for $n = pq$, $1/4$ of the numbers have square roots, and each number that has a square root actually has four.

Since 1 does have a square root modulo n (itself), there are four possibilities for b:

$$\pm 1 \bmod n \quad \text{and} \quad \pm r \bmod n$$

for some $r \in \mathbf{Z}_n^*$, $r \not\equiv \pm 1 \pmod{n}$.

Finding a square root of $1 \pmod{n}$

Using randomization to find a square root of $1 \pmod{n}$.

- Choose random $a \in \mathbf{Z}_n^*$.
- Define a sequence $b_0, b_1, \ldots, b_s$, where $b_i = a^{2^i t} \bmod n$, $0 \le i \le s$.
- Each number in the sequence is the square of the number preceding it $\pmod{n}$.
- The last number in the sequence is $b_s = a^{ed-1} \bmod n$.
- Since $ed \equiv 1 \pmod{\phi(n)}$, it follows using Euler's theorem that $b_s \equiv 1 \pmod{n}$.
- Since $1^2 \bmod n = 1$, every element of the sequence following the first 1 is also 1.

Hence, the sequence consists of a (possibly empty) block of non-1 elements, following by a block of one or more 1's.

Using a non-trivial square root of unity to factor n

Suppose $b^2 \equiv 1 \pmod{n}$. Then $n \mid (b^2 - 1) = (b+1)(b-1)$.

Suppose further that $b \not\equiv \pm 1 \pmod{n}$. Then $n \nmid (b+1)$ and $n \nmid (b-1)$.

Therefore, one of the factors of n divides $b+1$ and the other divides $b-1$.

Hence, $p = \gcd(b-1, n)$ is a non-trivial factor of n.
The other factor is $q = n/p$.

Randomized factoring algorithm knowing *d*

```
Factor $(n, e, d)$ { //finds $s$, $t$ such that $ed - 1 = 2^s t$ and $t$ is odd
    $s = 0$; $t = ed - 1$;
    while ($t$ is even ) {$s$++; $t$/=2; }

    // Search for non-trivial square root of 1 $\pmod{n}$
    do {
        // Find a random square root $b$ of 1 $\pmod{n}$
        choose $a \in \mathbf{Z}_n^*$ at random;
        $b = a^t \bmod n$;
        while ($b^2 \not\equiv 1 \pmod{n}$) $b = b^2 \bmod n$;
    } while ($b \equiv \pm 1 \pmod{n}$);

    // Factor $n$
    $p = \gcd(b - 1, n)$;
    $q = n/p$;
    return $(p, q)$;
}
```

Notes on the algorithm

Notes:

- b_0 is the value of b when the innermost while loop is first entered, and b_k is the value of b after the k^{th} iteration.
- The inner loop executes at most $s - 1$ times since it terminates just before the first 1 is encountered, that is, when $b^2 \equiv 1 \pmod{n}$.
- At that time, $b = b_k$ is a square root of $1 \pmod{n}$.
- The outer do loop terminates if and only if $b \not\equiv \pm 1 \pmod{n}$. At that point we can factor n.

The probability that $b \not\equiv \pm 1 \pmod{n}$ for a randomly chosen $a \in \mathbf{Z}_n^*$ is at least 0.5.[4] Hence, the expected number of iterations of the do loop is at most 2.

[4] (See Evangelos Kranakis, *Primality and Cryptography*, Theorem 5.1.)

Example

Suppose $n = 55$, $e = 3$, and $d = 27$.[5]
Then $ed - 1 = 80 = (1010000)_2$, so $s = 4$ and $t = 5$.

Now, suppose we choose $a = 2$. We compute the sequence of b's.

$$\begin{aligned}
b_0 &= a^t \bmod n = 2^5 \bmod 55 = 32\\
b_1 &= (b_0)^2 \bmod n = (32)^2 \bmod 55 = 1024 \bmod 55 = 34\\
b_2 &= (b_1)^2 \bmod n = (34)^2 \bmod 55 = 1156 \bmod 55 = 1\\
b_3 &= (b_2)^2 \bmod n = (1)^2 \bmod 55 = 1\\
b_4 &= (b_3)^2 \bmod n = (1)^2 \bmod 55 = 1
\end{aligned}$$

The last $b_i \neq 1$ in this sequence is $b_1 = 34 \not\equiv -1 \pmod{55}$, so 34 is a non-trivial square root of 1 modulo 55.

It follows that $\gcd(34 - 1, 55) = 11$ is a prime divisor of n.

[5] These are possible RSA values since $n = 5 \times 11$, $\phi(n) = 4 \times 10 = 40$, and $ed = 81 \equiv 1 \pmod{40}$.

A ciphertext-only attack against RSA

Eve isn't really interested in factoring n, computing $\phi(n)$, or finding d, except as a means to read Alice's secret messages.

A problem we would like to be hard is

Definition (ciphertext-only problem)

Given an RSA public key (n, e) and a ciphertext c, find the plaintext message m.

Hardness of ciphertext-only attack

A ciphertext-only attack on RSA is no harder than factoring n, computing $\phi(n)$, or finding d, but it does not rule out the possibility of some clever way of decrypting messages without actually finding the decryption key.

Perhaps there is some feasible probabilistic algorithm that finds m with non-negligible probability, maybe not even for all ciphertexts c but for some non-negligible fraction of them.

Such a method would “break” RSA and render it useless in practice.

No such algorithm has been found, but neither has the possibility been ruled out, even under the assumption that the factoring problem itself is hard.