CPSC 467b: Cryptography and Computer Security

Michael J. Fischer

Lecture 10
February 19, 2013

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

One-way and Trapdoor Permutations

Primality Tests

Algorithms for testing primality

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

A *deterministic primality test* is a *deterministic* procedure that correctly answers '**composite**' or '**prime**' for each input $n \geq 2$.

- 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 primality test is feasible in practice.

Extended primality tests

There do exist fast *probabilistic* algorithms for testing primality.

To arrive at a probabilistic algorithm, we extend the notion of a deterministic 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".

Thus, an *extended primality test* is a deterministic algorithm $T(n, a)$ with three possible answers: '**composite**', '**prime**', or '**?**'.

If the algorithm gives a non-'**?**' answer, that answer must be correct for n, and we say that the helper string a is a *witness* to that answer for n.

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 no witness to the correct answer for n is ever found), but if it does terminate, the returned 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 answer.

Strong primality tests

A primality test $T(n, a)$ is *strong* if, for every $n \geq 2$, there are "many" witnesses to the correct answer for n.

For a strong test, the probability is "high" that a random helper string is a witness, so the algorithm P_2 will usually succeed.

We do not know of any feasibly computable strong primality test.

Fortunately, a weaker test can still be useful.

Weak tests of compositeness

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

$$\zeta_a(n) = (2 \leq a \leq n-1 \text{ and } a^{n-1} \not\equiv 1 \pmod{n}).$$

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*[1] 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 $561 = 3 \cdot 11 \cdot 17$. [2]

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.

[1] Carmichael numbers are sometimes called Fermat pseudoprimes.
[2] 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 $b_i = a^{2^i t} \bmod n$, $0 \leq i \leq s$.
- 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.[3] Hence, the expected number of iterations of the do loop is at most 2.

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

Example

Suppose $n = 55$, $e = 3$, and $d = 27$.[4]
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.

$$b_0 = a^t \bmod n = 2^5 \bmod 55 = 32$$
$$b_1 = (b_0)^2 \bmod n = (32)^2 \bmod 55 = 1024 \bmod 55 = 34$$
$$b_2 = (b_1)^2 \bmod n = (34)^2 \bmod 55 = 1156 \bmod 55 = 1$$
$$b_3 = (b_2)^2 \bmod n = (1)^2 \bmod 55 = 1$$
$$b_4 = (b_3)^2 \bmod n = (1)^2 \bmod 55 = 1$$

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.

[4]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.

One-way and Trapdoor Permutations

One-way permutations

A *one-way permutation* is a permutation f that is easy to compute but hard to invert.

To *invert* f means to find, for each element y, the unique x such that $f(x) = y$.

Here, "easy" means computable in probabilistic polynomial time.

"Hard" to invert means that no probabilistic polynomial time algorithm can find x given y with non-negligible success probability.

Negligible functions

A function is said to be *negligible* if it goes to zero faster than any inverse polynomial.

Definition (2.1 in Goldwasser-Bellare)

ν is negligible if for every constant $c \geq 0$ there exists an integer k_c such that $\nu(k) < k^{-c}$ for all $k \geq k_c$.

In other words, no matter what c you choose, $\nu(k) < k^{-c}$ for all sufficiently large k.

One-way functions

More generally, any function f is *one-way* if it is easy to compute but hard to invert, where now "invert" means to find, given y in the range of f, any element x such that $f(x) = y$.

Definition (2.2 in Goldwasser-Bellare)

A function $f\colon \{0,1\}^* \to \{0,1\}^*$ is *one-way* if:

1. There exists a PPT[5] that on input x outputs $f(x)$;
2. For every PPT algorithm A there is a negligible function ν_A such that for all sufficiently large k,

$$\mathrm{Pr}[\, \mathrm{f(z) = y : x \overset{\$}{\leftarrow} \{0,1\}^k;\ y \leftarrow f(x);\ z \leftarrow A(1^k, y)}\,] \le \nu_{\mathrm{A}}(\mathrm{k})$$

[5]Probabilistic polynomial time Turing machine.

Fine points

Here, k is a *security parameter* that measures the length of the argument x to f. It is also provided to A as input, in unary, so that A's running time is measured correctly as a function of k.

Again we have several sources of randomness which jointly determine the probability that A successfully inverts f.

In particular, the string y on which we wish to invert f is not chosen uniformly from strings of a given length but instead is chosen according the probability of y being the image under f of a uniformly chosen x from $\{0,1\}^k$.

Trapdoor functions

A *trapdoor function* f is a one-way function that becomes easy to invert on a subset of its domain given a secret *trapdoor* string.

Definition (2.15 of Goldwasser-Bellare, corrected)

A *trapdoor* function is a one-way function $f\colon \{0,1\}^* \to \{0,1\}^*$ such that there exists a polynomial p and a probabilistic polynomial time algorithm I such that for every k there exists a $t_k \in \{0,1\}^*$ such that $|t_k| \leq p(k)$ and for all $x \in \{0,1\}^k$, then $I(f(x), t_k) = z$, where $f(z) = f(x)$.

Taking this apart, it says that f, when restricted to strings of length k, can be inverted by the algorithm I when I is given a suitable "trapdoor" string t_k.

A more intuitive formulation of trapdoor function

In the usual formulation of trapdoor function, we imagine f being chosen at random from a family of one-way functions. The trapdoor string is the secret that allows f to be inverted.

The idea is that f is regarded as a function of two arguments, k and x, where $|k|$ and $|x|$ are both bounded by a polynomial in the security parameter s. $f(k, x)$ should be a one-way function but with the additional *trapdoor property*:

> *There is a PPT algorithm I and a polynomial p such that for every s and every $k \in \{0,1\}^*$ with $|k| \leq p(s)$, there exists a $t_k \in \{0,1\}^*$ with $|t_k| \leq p(s)$ such that for all $x \in \{0,1\}^s$, then $I(f(k,x), t_k) = z$, where $f(k,z) = f(k,x)$.*

The RSA function

The RSA function is the map $f((n, e), x) = x^e \pmod{n}$. It is believed to be a trapdoor function with security parameter s when n is restricted to length s strings, n is is the product of two distinct primes of approximately the same length, $e \in \mathbf{Z}^*_{\phi(n)}$, and $x \in \mathbf{Z}_n$.

Here, the index k is the pair (n, e), and the trapdoor $t_k = (n, d)$, where d is the RSA decryption exponent.

The mathematically inclined will notice that this function does not quite fit the earlier definitions because of the restrictions on the domains of n, e, and x. We leave it to a more advanced course to properly sort out such details.