The UNIVERSITY of VERMONT

Benjamin Fenster
CS-266 (Ling)
19 December 2005

Notes – Public Key Cryptography

I. Mathematical Background
   a. Modular Addition
   b. Additive inverse: Number you add to x to get 0
   c. Modular Multiplication
      i. Multiply in the normal way, then apply mod
      ii. Multiplicative Inverse: Multiply to get 1
      iii. Not all x has a multiplicative inverse
      iv. Only 1, 3, 7, 9 when using mod 10 (they're relatively prime to 10, or coprime)
      v. Euclid's algorithm finds the inverse for a number
      vi. Public and Private keys are multiplicative inverses (but don't know what they're relatively prime TO)
   d. Totient Function
      i. $\phi(n)$ = number of numbers less than n that are relatively prime to n
      ii. $\phi(10) = 4$ (they are {1, 3, 7, 9})
      iii. $\phi(7) = 6$ (they are {1, 2, 3, 4, 5, 6})
      iv. $\phi(0) = 0$ (not relatively prime to any number)
      v. If n is prime, $\phi(n) = n - 1$ since all numbers < n are relatively prime to it (as in the case of 7 above).
      vi. If p and q are both prime and n = pq
         1. If you know how many AREN'T relatively prime, just subtract
         2. $n = 2 * 5 = 10$
         3. How many multiples of 2? 5 (0, 2, 4, 6, 8)
         4. How many multiples of 5? 2 (0, 5)
         5. How many multiples of 10? 1 (0)
         6. So $\phi(10) = 10 - 5 - 2 + 1$ (add one because zero has been double-counted)
         7. In general $\phi(pq) = pq - (p + q - 1) = (p - 1)(q - 1)$
         8. $pq - q - p + 1$
      vii. Used in encrypting / decrypting
      viii. $x^{\phi(n)+1} = x \bmod n$ (more on this later
   e. Modular Exponentiation: If n = 10, $x^1 = x^5 = x^9 = x^{13}$
   f. Euclidian Algorithm
      i. Multiplicative Inverse
         1. We want to find x such that 3 * x is = 1 mod 10
         2. How to find it?
         3. The obvious solution: Try 1, 2, 3, 4, 5, 6, 7, 8, and 9
         4. Complexity is $\theta(N)$ to find the inverse of a number mod N.
         5. Since N is really about 2^{512} that would take "forever"
      ii. Say n = 101, a = 38; find the multiplicative inverse
      iii. The inverse exists when GCD(38, 101) = 1
      iv. 101 = 2(38) + 25 (find quotient and remainder)
      So GCD(38, 101) = GCD(38, 25)
      38 = 1(25) + 13
      25 = 1(13) + 12
      13 = 1(12) + 1
      12 = 1(12) + 0 and GCD(1, 0) = 1 by default, so the GCD is indeed 1
      v. If you have a common divisor between 25 and 13 and you subtract them, the same number is a common divisor of the result
      vi. Now we know these numbers are relatively prime
      vii. Now start plugging in those numbers to compute the multiplicative inverse:
      viii. 1 = 13 – 1 * 12
      = 13 – 1 * (25 – 1 * 13) = 2 * 23 – 1 = 25

= 2 * (38 – 1 * 25) – 1 * 25 = 2 * 38 – 3 * 25
= 2 * 38 – 3(101 – 2 * 38) = 8 * 38 – 3 * 101

ix. So 8 * 38 – 3 * 101 = 1 mod 101, so (-3)(101) = 0 mod 101

II. RSA
- a. Rivest, Shamir, Adleman
- b. Key length is variable
  - i. Any length will *work* to encrypt and decrypt
  - ii. 512 bit is common for security purposes
- c. Much slower than secret key cryptography
- d. Usually you'll use RSA to establish a secret key and then fall back on that
- e. Basics
  - i. Choose two large primes about 256 bits named p and q
  - ii. Set n = pq (don't ever reveal p and q)
  - iii. It takes much more effort to factor a number than to multiply two numbers
  - iv. Public Key
    1. Pick e relatively prime to $\phi(n)$
    2. The public key is <e, n>
  - v. Private Key
    1. $d = (e \bmod \phi(n))^{-1}$
    2. Private key is <d, n>
  - vi. Encryption: $m < n$, $c = m^e \bmod n$
  - vii. Decryption: $m = c^d \bmod n$
  - viii. Example
    1. Pick 7, 11, so n = 77
    2. $\phi(77) = 6 * 10 = 60$
    3. Pick e relatively prime to 60, e = 13
    4. d = e mod 60
       13*d = 1 mod 60
       60 = 4(13) + 8
       13 = 1(8) + 5
       8 = 1(5) + 3
       5 = 1(3) + 2
       3 = 1(2) + 1

       1 = 3 – 1(2) = 3 – (5 – 1*3) = 2 * 3 – 5
       = 2(8 – 5) – 1 * 5 = 2 * 8 – 3 * 5 = 2 * 8 – 3(13 – 1 * 8)
       = 5 * 8 – 3 * 13 = 5(1 * 60 – 4 * 13) – 3 * 13
       = 5 * 60 **– 23** * 13
    5. So the multiplicative inverse of 13 mod 60 is -23 + 60 = 37
    6. d = 37
    7. So publish <13, 77> as the public key and keep <37, 77> private
- f. Why does RSA Work?
  - i. n = pq, so $\phi(n) = (p – 1)(q – 1)$. This would be really hard to figure out without knowing p and q
  - ii. $d * e = 1 \bmod \phi(n)$. For this to even be possible, e must be relatively prime to $\phi(n)$, so how can a bad guy figure out what d is without knowing $\phi(n)$?
- g. Why is it Secure? It's hard to factor a 512-bit number.
- h. Encryption: Just do x^e
- i. Decryption: Do $(x^e)^d = x^{ed} = x^1 = x$ (because it's all mod n)

III. Exponentiating
- a. Hard to do exponentiation directly with large numbers
- b. Want to compute $123^{54} \bmod 678$
- c. Can perform mod after every multiply

d. $123^2 = 15129 \bmod 678 = 213$
   $123^3 = 123 * 213 = 26199 \bmod 678 = 435$
   $123^4 = 123 * 435 = 53505 \bmod 678 = 621$
e. So the largest number we ever have is n^2 – that's still large, but much more manageable.
f. Would still have to do 53 steps to get the answer (each step of divide/multiply)
g. Binary Expansion of 54 = 110110
h. Compute $123^1 \bmod 678 = 123$
i. Compute $123^{11} \bmod 678 = (123^1)^2 * 123^1 = 435$
j. Compute $123^{110} \bmod 678 = (123^{11})^2 = \ldots$
k. It now only takes two multiplications to get each step.
l. Need to square the previous result. Then, if the last digit is a 1, also multiply by another copy of 123.
m. In base 10, this would mean:
   i. Multiply by 10 to get 87 -> 870
   ii. Multiply by 123^9 if it's 87 -> 87**9**
   iii. We just do it in binary, not base 10.
n. Called "Square and Multiply"
o. For 512-bit numbers, just have 512 steps (instead of 2^{512})

IV. Picking Large Primes
a. Chance of picking a prime number at random is about $1 / \ln 2^{256}$ (to get a number of size 2^{256}.
b. Picking 10-digit number = 1/23. 100-digits: 1/230.
c. Need a primality testing algorithm
d. Could just try dividing all primes less than the square root of n, but that's still too much work.
e. We'll use a probabilistic algorithm that will "guarantee" that a number is prime with a certain probability
f. Euler's Theorem
   i. a is relatively prime to n, $a^{\phi(n)} = 1 \bmod n$
   ii. In RSA, $x^{de} = x \bmod n$, $de = 1 \bmod \phi(n)$, $de = 1 + k\phi(n)$ for some k, $x^{1+k\phi(n)} = x (x^{\phi(n)})^k$
   iii. This theorem is why RSA works
g. Fermat's Theorem
   i. If p is prime and $0 < a < p$, $a^{p-1} = 1 \bmod p$
   ii. If p is prime then "something" is true. If "something" is false, then p is not prime.
   iii. That's the logic behind identifying primes!
   iv. $a^{p-1} \neq 1 \bmod p$ means p is not prime
   v. So pick a p, then see if a^{p-1} is equal to 1 mod p
   vi. May be equal to 1 by luck though, in which case you can conclude nothing.
   vii. If it's *not* equal to 1 you *know* it's composite.
   viii. There are some false negatives too: composites that pass the test. Called Carmichael numbers
h. Miller and Rabin Algorithm
   i. Goal: Compute $a^{n-1} \bmod n$
   ii. Set $2^b c = n - 1$ (c must be odd)
   iii. Compute $a^c \bmod n$
   $a^{2\wedge 1c} \bmod n$
   $a^{2\wedge 2c} \bmod n$
   $a^{2\wedge 3c} \bmod n$
   …
   $a^{2\wedge bc} \bmod n$
   iv. These steps are done in the square and multiply algorithm (each step is squaring)
   v. Why does this help?
      1. If p is prime then $x^2 = 1 \bmod p$ has only two solutions for x: 1 or -1

2. $n = 15$, $x^2 = 1 \bmod 15$, $x = 1, 14, 4, 11$
3. Remember it's modulo arithmetic
4. Since $4 * 4 = 1$, 15 is not prime

vi. If the last step is 1, but the previous step is not 1, then it's not prime! (you squared a number and got 1 – that violates the theorem)

vii. As you're computing, see if each answer is equal to 1 or -1. If not, you're done: n is not prime

viii. Proof of that Theorem
1. $x^2 = 1 \bmod p$, $x^2 - 1 = 0 \bmod p$
2. $(x - 1)(x + 1) = p$, so either $(x - 1)$ or $(x + 1)$ is a multiple of p

ix. Implementation
1. Pick add n, check n / {3, 5, 7, 11, …}
2. Pick a random a and compute a^c with $n - 1 = 2^b c$
3. Run test as described, and repeat
4. Each step gives ½ probability
5. There's some research presently generating truly independent values of a (to compensate for pseudorandom qualities)
6. Usually do about 30 values of a

V. How to pick e

a. Just needs to be relatively prime to $(p - 1)(q - 1)$
b. Can test whether it's relatively prime using Euclid's algorithm
c. Could generate p and q, then generate random e values until you find one that's relatively prime
d. Could also generate e3 first, then create $(p - 1)(q - 1)$ to be relatively prime to e
e. It's desirable to control the value of e to make encryption/decryption easier
f. Could use a small constant for e like 3 or 65537 (both are primes, so it's easy to satisfy the condition that it's relatively prime to $(p - 1)(q - 1)$
   i. Also , binary expansions are 11 and 100000000000000001
   ii. This makes square and multiply particularly easy
g. Small d
   i. If $d < n^{1/4}$ and $q < p < 2q$ (they are close together), d can be found easily
   ii. Technique involves continuous fractions
   iii. Don't even need to search one-by-one
h. Small e
   i. Since e is published, there's nothing *necessarily* wrong with having a small e
   ii. Message must be bigger than the cube root of n
   iii. An attack
      1. Broadcast a message to three people (B, C, D) each of whom has her own key
      2. Each message encrypted with $<3, n_i>$
      3. Messages encrypt to $m^3 \bmod n_1$, $m^3 \bmod n_2$, $m^3 \bmod n_3$. All different due to different values of n for each person
      4. A bad guy intercepts these messages
      5. $m^3 \bmod n_1 n_2 n_3$ (if n_1, n_2, n_3 are all relatively prime – unlikely that three people chose any of the same primes, so they probably are)
      6. Since $m < n_1$, $m < n_2$, and $m < n_3$, $m^3 < n_1 n_2 n_3$.
      7. So $m^3 \bmod n_1 n_2 n_3 = m^3$ so just take the ordinary cube root to obtain m
      8. The same attack would work for e = 65537 but would need to send the message to 65,537 people, and at that point it's not really a secret anyway.
      9. One could easily prevent this attack by padding each message with something unique.

VI. Chinese Remainder Theorem

a. Have 1 mod 3, 2 mod 5, 4 mod 7
b. It's possible to find an x such that $x = 1 \bmod 3$, $x = 2 \bmod 5$, and $x = 4 \bmod 7$
c. This is since 3, 5, and 7 are relatively prime

d. Pick two
    i. Try to solve 1 mod 3 = x = 4 mod 7
    ii. x = 4 + 7y (4 + some integer multiple of y)
    iii. Substitute 4 + 7y = 1 mod 3
    iv. Then 1 + y = 1 mod 3
    v. y = 0 mod 3, y = 3z for some integer z
    vi. x = 4 + 7(3z) = 4 + 21z
    vii. x = 4 mod 21
e. Bring in the last equation
    i. x = 4 mod 21
    ii. x = 2 mod 5
    iii. x = 2 + 5m for m = z
    iv. 2 + 5m = 4 mod 21
    v. 5m = 2 mod 21
    vi. Need the multiplicative inverse of 5 mod 21 (which we conveniently know how to calculate): it's 17
    vii. so 17 * 5m = 2 * 17 mod 21
    viii. 85m = 34 mod 21, m = 13 mod 21
    ix. m = 13 + 21n for some integer n
    x. x = 2 + 5(13 + 21n) = 2 + 65 + 105n
    xi. x = 67 mod 105 (since 105 is 7 * 5 * 3)

VII. A Threat: Smooth Numbers
a. A smooth number is the product of "small" primes (where small is defined in terms of modern computational ability)
b. We want to sign message m
    i. Compute h = hash(m)Include $s = h^d \bmod n$ (where <d, n> is the private key) in the message.
    ii. Recipient computes $s^e \bmod n$, compares to hash(m) (where <e, n> is the sender's public key)
c. First, imagine we're JUST sending h (and it's known by everyone)
d. I observe the signature of m_1, m_2 (signed by Sean), $m_1^d \bmod n$, and $m_2^d \bmod n$
e. I could sign m_1m_2 as Sean $(m_1m_2)^d \bmod n$
f. If m_1/m_2 is prime, I can fake signatures on that prime and its multiples
g. Could pad with 0 on the right, but that just multiplies by 10 which is 2^n5^m (smooth!)
h. Zeros on the left don't change the value
i. As a protocol designer you want a way to counter threats like this to maintain security
j. In this case, consider padding.

VIII. Diffie-Hellman Key Exchange
a. DHKE can be used just to exchange a secret key
b. Pick p = 512 bit (or so) prime; make it public
c. Pick a random g, g < p and make that public
d. Working on the "multiplicative subgroup generated by g"
e. Two parties pick two secrets s_a, s_b (one each)
f. Send $g^{Sa} \bmod p$, and $g^{Sb} \bmod p$
g. Now person A knows S_a and $g^{Sb} \bmod p$, so can calculate $g^{SaSb} \bmod p$. Person B can do the same, given S_b and $g^{Sa} \bmod p$. A bad guy doesn't have either secret, however, so can't do that calculation.

IX. Zero Knowledge Proofs
a. With RSA
    i. Publish <e, n>
    ii. Challenger generates a challenge (m) randomly, encrypts with the public key, and sends that to the prover.
    iii. The prover sends back the decrypted m, which the challenger can verify.
    iv. Nobody else could decrypt it, so if the challenger gets back the original m the prover must really be who s/he claims
    v. Zero-knowledge since the challenger gains no information about d

vi. Why have any other scheme?
    1. RSA is not terribly efficient. Have (on average) 768 multiplies and divides plus modulus over a 512 bit number. Uhg.
    2. Want a scheme that takes less work, but it may *only* do zero-knowledge proof if desired

b. Graphs
    i. Generate graphs A and B, where A is isomorphic to B
    ii. Prover generates graphs G_1, G_2, …, G_{30} (some number of these) and sends them to the challenger.
    iii. Challenger sends a random string of length 30 of As and Bs (AAAABBBABB)
    iv. Prover returns 30 permutations that map A or B to G_i
    v. If all challenges answered correctly, identity is proven
    vi. The prover knows how the graphs are constructed, so it's easy to map. If the challenger supplies graphs it's "impossible" for anyone else to find even the original mapping (A to G_i)

c. Fiat Shamir
    i. Publish (n, v) where n = pq for primes p and q, and
        1. Generate two random primes (like for RSA) and multiply
        2. Finding the square root mod n is hard
        3. Take a random s, set $v = s^2 \bmod n$
    ii. Prover generates $r_i^2 \bmod n$, …, $r_{30}^2 \bmod n$ (some number of random numbers)
    iii. Challenger again sends ABAABBB...
    iv. If the ith question is A, send back $s*r_i$
    v. If the ith question is B, send back r_i
    vi. Prover takes $s*r_i$, squares it = $s^2r_i^2 = vr_i^2 \bmod n$ (so the challenger can compare)
    vii. Why not have only A questions:
        1. Need to construct y such that the square root of yv is known
        2. First, generate a, then a^2v^{-1}, use $y = a^2v^{-1}$
        3. Send that for different values of a
        4. Then for answers to A, send a_i
        5. Challenger computes a_i^2, compares to $(a^2v^{-1})v$
        6. But if you generate r_i^2 that way, you don't know r_i. Could ONLY answer A-type questions.
    viii. Do 30 multiplications to square numbers, then on average 15 more to reply to A challenges.
    ix. To verify the answers, would square 15 times (15 multiplies), then multiply vr_i^2 (so 15 more) for A answers is 30. For B would only need r_i^2 so 15 multiplications.
    x. Total (for both parties) is about 90 multiplications and 90 divisions