Lecture 12: Public-Key Cryptography and RSA

Lecture Notes on "Introduction to Computer Security"

by Avi Kak (kak@purdue.edu)

March 20, 2007

Goals:

- To review public-key cryptography
- To demonstrate that confidentiality and sender-authentication can be achieved simultaneously with public-key cryptography
- To review the Rivest-Shamir-Adleman (RSA) algorithm for public-key cryptography
- To present the proof of the RSA algorithm
- To go over the computational issues related to RSA.
- To discuss the security of RSA

Public-Key Cryptography

- Public-key cryptography is also known as asymmetric-key cryptography.

- Encryption and decryption is carried out using **two different keys**. The two keys in such a key pair are referred to as the **public key** and the **private key**. (As we will see, this solves one of the most vexing problems associated with symmetric-key cryptography — the problem of key distribution).

- With public key cryptography, all parties interested in secure communications can publish their public keys.

- Party A, if wanting to communicate **confidentially** with party B, can encrypt a message using B's publicly available key. Such a communication would only be decipherable by B as only B would have access to the corresponding private key.

- Party A, if wanting to send an **authenticated message** to party B, would encrypt the message with A's own private key. Since this message would only be decipherable with A's public key, that would establish the authenticity of the message — meaning that A was indeed the source of the message.

- The figure on slide 5 shows how public-key encryption can be used to provide **confidentiality and authentication** at the same time. Note again that confidentiality means that we want to protect a message from eavesdroppers and authentication means that the recipient needs a guarantee as to the identity of the sender.

- In the figure, A's public and private keys are designated PU_a and PR_a. B's public and private keys are designated PU_b and PR_b.

- A wants to send a message X to B. The processing steps undertaken by A to convert X into its encrypted form Z that can be placed on the wire are:

$$Z \quad = \quad E\left(PU_b,\ E\left(PR_a,\ X\right)\right)$$

  where $E()$ stands for encryption. The processing steps undertaken by B to recover X from Z are

$$X \quad = \quad D\left(PU_a,\ D\left(PR_b,\ Z\right)\right)$$

  where $D()$ stands for decryption.

- The sender A encrypting his/her message with its own private key PR_a provides authentication. This step constitutes A putting his/her **digital signature** on the message. (Instead of applying the private key to the entire message, a sender may also "sign" a

message by applying his/her private key to just **a small block of data** that is derived from the message to be sent.)

- The sender A **further encrypting** his/her message with the receiver's public key PU_b provides **confidentiality**.

- Of course, the price paid for achieving confidentiality and authentication at the same time is that now the message must be processed **four** times in all for encryption/decryption. The message goes through two encryptions at the sender's place and two decryptions at the receiver's place. Each of these four steps involves separately the **computationally complex** public-key algorithm.

- Note that public-key cryptography does **not** make obsolete the more traditional symmetric-key cryptography. Because of the greater computational overhead associated with public-key cryptosystems, symmetric-key systems will continue to be used for the foreseeable future.

- However, it is generally agreed that public-key encryption is indispensable for key management and digital signature applications.

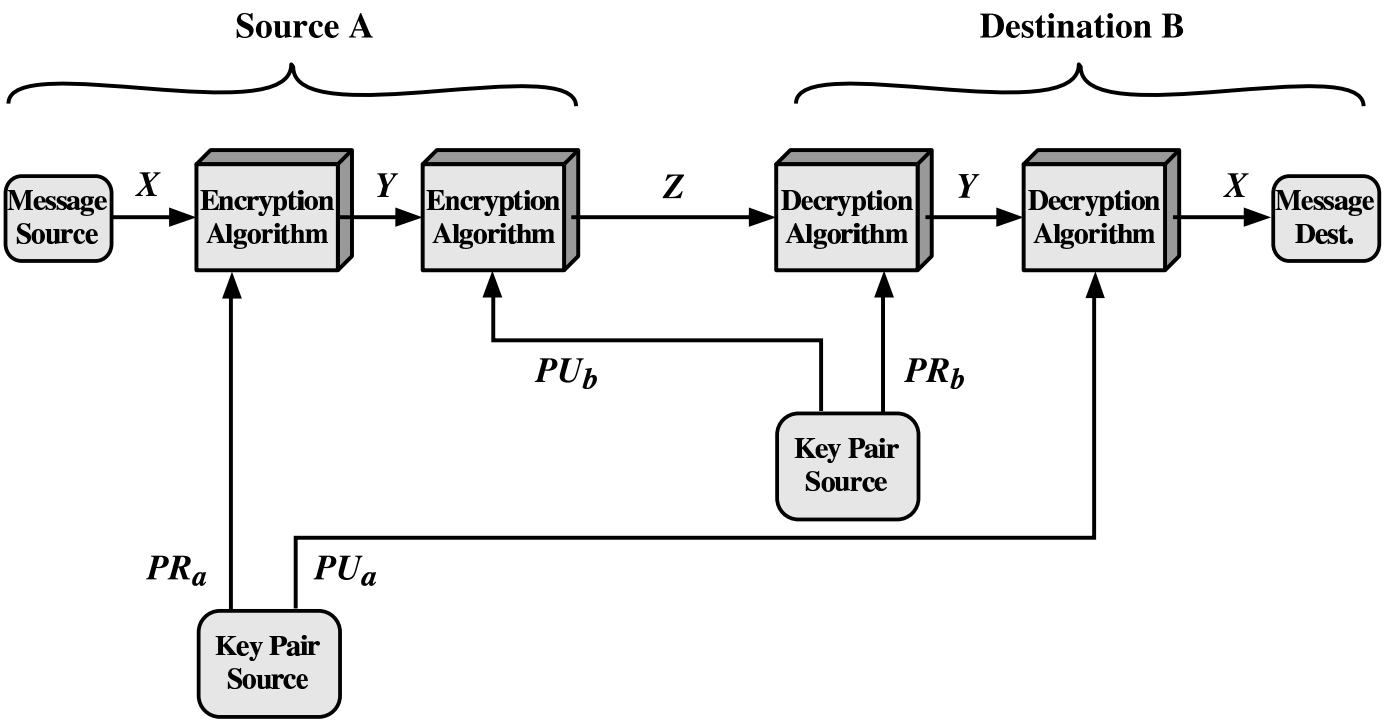

Figure 9.4 Public-Key Cryptosystem: Authentication and Secrecy

This figure is from Chapter 9 of Stallings: "Cryptography and Network Security", Fourth Edition

The Rivest-Shamir-Adleman (RSA) Algorithm for Public-Key Cryptography — The Basic Idea

- The RSA algorithm is based on the following property of positive integers: Suppose we decide to use a number n as the modulus for modular arithmetic, and suppose we choose an integer e just on the basis that it be coprime to $\phi(n)$, and then suppose we derive from e its multiplicative inverse d as stated follows:

$$
\begin{array}{lcl}
n & = & \textit{a modulus for modular arithmetic} \\
e & = & \textit{an integer that is relatively prime to the totient of } n \textit{ [This guarantees that } e \textit{ will possess a multiplicative inverse modulo the totient of } n\textit{]} \\
d & = & \textit{an integer that is the multiplicative inverse of } e \textit{ modulo the totient of } n
\end{array}
$$

Now suppose we are given an integer M, $M < n$, that represents our message, then we can transform M into another integer C that will represent our ciphertext by the following modular operation

$$C \quad = \quad M^e \bmod n$$

and **recover** M back from C by the following modular operation

$$M \quad = \quad C^d \bmod n$$

The RSA Algorithm — Putting to Use the Basic Idea

- The basic idea described on the previous slide can be used in the following manner for confidential communication:

- An individual who wishes to receive messages confidentially will use the pair of integers $\{e, n\}$ has his/her public key. At the same time, this individual can use the pair of integers $\{d, n\}$ as the private key.

- Another party wishing to send a message to such an individual will encrypt the message using the public key $\{e, n\}$. Only the individual with access to the private key $\{d, n\}$ will be able to decrypt the message.

- The important theoretical question here is as to what conditions if any must be satisfied by the modulus n for this $M \rightarrow C \rightarrow M$ transformation to work?

How to Choose the Modulus for the RSA Algorithm?

- The modulus n must be selected in such a manner that the following is guaranteed:

$$\left(M^e)^d\right) \quad \equiv \quad M^{ed} \quad \equiv \quad M \pmod{n}$$

  We want this guarantee because $C = M^e \bmod m$ is the encrypted form of the message integer M and decryption is carried out by $C^d \bmod n$.

- It was shown by Rivest, Shamir, and Adleman that we have this guarantee when n is a product of two prime numbers:

$$n \quad = \quad p \times q \quad for \ \ some \ \ prime \ \ p \ \ and \ \ prime \ \ q \qquad (1)$$

- The above factorization is needed because the proof of the algorithm depends on the following two properties of primes and coprimes:

  – If two integers p and q are relatively prime to each other, the following equivalence holds for any two integers a and b:

$$\{a = b \bmod p \quad and \quad a = b \bmod q\} \quad \Leftrightarrow \quad \{a = b \bmod pq\} \qquad (2)$$

This equivalence follows from the fact '$a = b \bmod p$' implies $a - b = k_1 p$ for some integer k_1. But since we also have '$a = b \bmod q$ implying $a - b = k_2 q$, it must be the case that $k_1 = k_3 \times q$ for some k_3. Therefore, we can write $a - b = k_3 \times p \times q$, which establishes the equivalence.

– In addition to needing p and q to be coprimes, **we also want p and q to be individually primes**. It is only when p and q are individually prime that we can decompose the totient of n into the product of the totients of p and q. That is

$$\phi(n) \quad = \quad \phi(p) \times \phi(q) \quad = \quad (p-1) \times (q-1) \qquad (3)$$

The step plays a crucial role in the proof of the RSA algorithm.

- So that the cipher cannot be broken by an exhaustive search for the prime factors of the modulus n, it is important that both p and q are very large primes. Finding the prime factors of a large integer is computationally harder than determining its primality.

- We also need to ensure that n is not factorizable by one of the modern integer factorization algorithms. More on that later in this presentation.

Proof of the RSA Algorithm

- Since the integer d is the multiplicative inverse of the integer e modulo $\phi(n)$, we obviously have

$$e \times d \bmod \phi(n) \quad = \quad 1 \tag{4}$$

  This implies that there must exist an integer k so that

$$e \times d \ - \ 1 \quad = \quad k \times \phi(n) \tag{5}$$

- It must then obviously be the case that $\phi(n)$ is a divisor of the expression $e \times d - 1$. But since $\phi(n) = \phi(p) \times \phi(q)$, the totients $\phi(p)$ and $\phi(q)$ must also individually be divisors of $e \times d - 1$. That is

$$\phi(p) \mid (e \times d \ - \ 1) \quad \textit{and} \quad \phi(q) \mid (e \times d \ - \ 1) \tag{6}$$

- Focusing on the first of these assertions, since $\phi(p)$ is a divisor of $e \times d - 1$, we can write

$$e \times d \ - \ 1 \quad = \quad k_1 \phi(p) \quad = \quad k_1(p \ - \ 1) \tag{7}$$

  for some integer k_1.

- Therefore, we can write for any integer M:

$$M^{e\times d} \bmod p \quad = \quad M^{e\times d \,-\, 1 \,+\, 1} \bmod p \quad = \quad M^{k_1(p \,-\, 1)} \times M \bmod p \tag{8}$$

- Now we have two possibilities to consider: Since p is a prime, it must be the case that either M and p are coprimes or that M is a multiple of p.

  - Let's first consider the case when M and p are coprimes. By Fermat's Little Theorem, since p is a prime, we have

    $$M^{p \,-\, 1} \quad \equiv \quad 1 \pmod{p}$$

    Since this conclusion obviously extends to any power of the left hand side, we can write

    $$M^{k_1(p \,-\, 1)} \quad \equiv \quad 1 \pmod{p}$$

    Substituting this result in Equation (8), we get

    $$M^{e\times d} \bmod p \quad = \quad M \bmod p \tag{9}$$

  - Now let's consider the case when the integer M is a multiple of the prime p. Now obviously, $M \bmod p \;=\; 0$. This will also be true for M raised to any power. That is, $M^k \bmod p \;=\; 0$ for any integer k. Therefore, Equation (9) will continue to be true even in this case.

- From the second assertion in Equation (6), we can draw an identical conclusion regarding the other factor q of the modulus n:

$$M^{e \times d} \bmod q \quad = \quad M \bmod q \tag{10}$$

- We established on slide 8 that, since p and q are coprimes, for any integers a and b if we have '$a = b \bmod p$' and '$a = b \bmod q$', then it must also be the case that '$a = b \bmod pq$'. Applying this conclusion to the partial results shown in Equations (9) and (10), we get

$$M^{e \times d} \bmod n \quad = \quad M \bmod n \tag{11}$$

Computational Steps for Key Generation in RSA Cryptography

- The RSA scheme is a block cipher.

- One typically encodes blocks of length 1024 bits. This means that the numerical value of the message integer M will be less than 2^{1024}. If this integer is expressed in decimal form, its value would be less than 10^{309}. In other words, the message integer M will have 309 decimal digits for each block of the plaintext.

- The computational steps for key generation are

```
Select p, q                    p and q are both primes, p != q)

Calculate the modulus  n = pXq

Calculate the totient PHI(n) = (p-1)X(q-1)

Select integer e               1 < e < PHI(n),  gcd( PHI(n), e ) = 1

Calculate d                    d = e^{-1} mod PHI(n)

Public Key = [e,n]

Private Key = [d,n]
```

Computational Steps for Selecting the Primes p and q in RSA Cryptography

- You first decide upon the size of the modulus integer n. Let's say that your implementation of RSA requires a modules of size B bits.

- To generate the prime integer p;
  - You first generate a random number of size $B/2$ bits.
  - You set the lowest bit of the integer; this ensures that the number will be odd.
  - You also set the **two highest bits** of the integer; this ensures that the highest bits of n will be set.
  - Using the Miller-Rabin algorithm, you now check to see if the resulting integer is prime. If not, you increment the integer by 2 and check again. This becomes the value of p.

- You do the same thing for selecting q. You start with a randomly generated number of size $B/2$ bits, and so on.

- In the unlikely event that $p = q$, you throw away your random number generator and acquire a new one.

- For greater security, instead of incrementing by 2 when the Miller-Rabin test fails, you generate a new random number.

Choosing a Value for e in RSA Cryptography

- Recall that we want to raise the message integer M to the power e modulo n. This step is referred to as **modular exponentiation**.

- The mathematical requirement on e is that $gcd(e,\ \phi(n)) = 1$. Since $n = p \times q$, this requirement is equivalent to the two requirements $gcd(e,\ \phi(p)) = 1$ and $gcd(e,\ \phi(q)) = 1$. In other words, we want $gcd(e,\ p-1) = 1$ and $gcd(e,\ q-1) = 1$.

- Obviously, if we choose a prime for e that will automatically satisfy the two conditions on e.

- For computational ease, one typically chooses a value for e that is prime, has as few bits as possible equal to 1 for fast multiplication, and, at the same time, that is cryptographically secure. Typical values for e are 3, 17, and 65537 $(2^{16}+1)$. Each of these values has only two bits set, **which makes for fast modular exponentiation**.

- Small values for e, such as 3, are considered cryptographically insecure. Let's say a sender A sends the same message M to three different receivers using their respective public keys that

have the same $e = 3$ but different values of n. Let these values of n be denoted n_1, n_2, and n_3. Let's assume that an attacker can intercept all three transmissions. The attacker will see three ciphertext messages: $C_1 = M^3 \bmod n_1$, $C_2 = M^3 \bmod n_2$, and $C_3 = M^3 \bmod n_3$. Assuming that n_1, n_2, and n_3 are relatively prime on a pairwise basis, the attacker can calculate $M^3 \bmod (n_1 \times n_2 \times n_3)$. (This assumes that $M^3 < n_1 n_2 n_3$, which is bound to be true since $M < n_1$, $M < n_2$, and $M < n_3$.) Having calculated M^3, all the attacker has to do is to figure out its cube-root.

- Having selected a value for e, it is best to **double check** that we indeed have $gcd(e,\ p-1) = 1$ and $gcd(e,\ q-1) = 1$ (since we want e to be coprime to $\phi(n)$, meaning that we want e to be coprime to $p-1$ and $q-1$ separately). Remember, with a small probability, the Miller-Rabin algorithm may declared p and/or q to be prime when in fact they are composite. If either p or q is found to not meet these two conditions on relative primality of $\phi(p)$ and $\phi(q)$ vis-a-vis e, you must discard the calculated p and/or q and start over. (It is faster to build this test into the selection algorithm for p and q.) When e is a prime and greater then 2, a **much faster** way to satisfy the two conditions is to ensure

$$\begin{aligned} p \bmod e &\neq 1 \\ q \bmod e &\neq 1 \end{aligned}$$

Calculating d from e and n in RSA Cryptography

- Once we have settled on a value for the encryption exponent e, the next step is calculate the decryption exponent d from e and the modulus n.

- Recall that $d \times e \equiv 1 \pmod{\phi(n)}$. We can also write this as

$$d = e^{-1} \bmod \phi(n)$$

  Calculating '$e^{-1} \bmod \phi(n)$' is referred to as **modular inversion**.

- Since d is the multiplicative inverse of e modulo $\phi(n)$, we can use the Extended Euclidean Algorithm for calculating d. Note we know the value for $\phi(n)$ since it is equal to $(p-1) \times (q-1)$.

- **Note that this is the main source of security in the system — keeping p and q secret and therefore also keeping $\phi(n)$ secret.** It is important to realize that knowing either will reveal the other. That is, if you know the factors p and q, you can calculate $\phi(n)$ by multiplying $p-1$ with $q-1$. And if you know $\phi(n)$ and n, you can calculate the factors p and q readily.

Modular Exponentiation for Encryption and Decryption

- As mentioned already, the message integer M is raised to the power e modulo n. That gives us the ciphertext integer C. Decryption consists of raising C to the power d modulo n.

- The exponentiation operation for encryption can be carried out efficiently by simply choosing an appropriate e. (Note that the only condition on e is that it be coprime to $\phi(n)$.) As mentioned previously, typical choices for e are 3, 17, and 65537. All these are prime and each has only two bits set.

- Modular exponentiation for decryption, meaning the calculation of $C^d \bmod n$, is an entirely different matter since we are not free to choose d. The value of d is determined completely by e and n.

- Computation of $C^d \bmod n$ can be speeded up by using the Chinese Remainder Theorem. Since the party doing the decryption knows the prime factors p and q of the modulus n, we can first carry out the easier exponentiations:

$$\begin{aligned} V_p &= C^d \bmod p \\ V_q &= C^d \bmod q \end{aligned}$$

- To apply CRT, we must also calculate the quantities

$$\begin{aligned} X_p &= q \times (q^{-1} \bmod p) \\ X_q &= p \times (p^{-1} \bmod q) \end{aligned}$$

  Applying CRT, we get

$$C^d \bmod n = (V_p X_p + V_q X_q) \bmod n$$

- Further speedup can be obtained by using Fermat's Little Theorem that says that if a and p are coprimes then $a^{p-1} = 1 \bmod p$.

- Let's see how Fermat's Little Theorem can be used to speed up the calculation of V_p and V_q. V_p requires $C^d \bmod p$. Since p is prime, obviously C and p will be coprimes. We can therefore write

  $V_p = C^d \bmod p = C^{u \times (p-1) + v} \bmod p = C^{p-1^u} C^v \bmod p = C^v \bmod p$

  for some u and v. Since $v < d$, it'll be faster to computer $C^v \bmod p$ than $C^d \bmod p$.

An Algorithm for Modular Exponentiation

- Obviously, a fundamental computational step in RSA is modular exponentiation. We want to calculate

$$A^B \bmod n$$

  for some integers A, B, and for some modulus n.

- What is interesting is that even for small values for A and B, the value of A^B can be enormous. For example, both A and B may consist of only a couple of digits, as in 7^{11}, but the result could still be very large number. For example, 7^{11} equals $1,977,326,743$, a number with 10 decimal digits. Now just imagine what would happen if, as would be the case in cryptography, A had, say, 256 binary digits (that is 77 decimal digits) and B was, say, 65537. Even when B has only 2 digits (say, $B = 17$), when A has 77 decimal digits, A^B will have 1304 decimal digits.

- The calculation of A^B can be speeded up by realizing that if B can be expressed as a sum of smaller parts, then the result is a product of smaller exponentiations. We can use the following binary representation for the exponent B:

$$B \quad \equiv \quad b_k b_{k-1} b_{k-2} \ldots b_0 \qquad (binary)$$

where we are saying that it takes k bits to represent the exponent, each bit being represented by b_i, with b_k as the highest bit and b_0 as the lowest bit. In terms of these bits, we can write the following equality for B:

$$B \quad = \quad \sum_{b_i \neq 0} 2^i$$

- Now the exponentiation A^B may be expressed as

$$A^B \quad = \quad A^{\sum_{b_i \neq 0} 2^i} \quad = \quad \prod_{b_i \neq 0} A^{2^i}$$

We could say that this form of A^B halves the difficulty of computing A^B because, assuming all the bits of B are set, the largest value of 2^i will be roughly half the largest value of B.

- We can achieve further simplification by bringing the rules of modular arithmetic into the multiplications on the right:

$$A^B \bmod n \quad = \quad \left(\prod_{b_i \neq 0} \left[A^{2^i} \bmod n \right] \right) \bmod n$$

Note that as we go from one bit position to the next higher bit position, we square the previously computed power of A.

- The A^{2^i} terms in the above product are of the following form

$$A^{2^0}, \quad A^{2^1}, \quad A^{2^2}, \quad A^{2^3}, \quad \ldots$$

As opposed to calculating each term from scratch, we can calculate each by squaring the previous value. We may express this idea in the following manner:

$$A, \quad A^2_{previous}, \quad A^2_{previous}, \quad A^2_{previous}, \quad \ldots$$

- Now we can write an algorithm for exponentiation that scans the binary representation of the exponent B from the lowest bit to the highest bit:

```
result = 1

while ( B > 0 ) :
  if ( B & 1 ) :                        # check the lowest bit of B
      result = ( result * A ) mod n
  B = B >> 1                            # shift B by one bit to right
  A = ( A * A ) mod n

return  result
```

The Security of RSA

- A particular form of attack on RSA that has been a focus of considerable attention is **the mathematical attack**.

- The mathematical attack consists of figuring out the prime factors p and q of the modulus n. Obviously, knowing p and q, the attacker will be able to figure out the exponent d for decryption.

- Another way of stating the same as above would be that the attacker would try to figure out the totient $\phi(n)$ of the modulus n. But as stated earlier, knowing $\phi(n)$ is equivalent to knowing the factors p and q. If an attacker can somehow figure out $\phi(n)$, the attacker will be able to set up the equation $(p-1)(q-1) = \phi(n)$, that, along with the equation $p \times q = n$, will allow the attacker to determine the values for p and q.

- Because of their importance in public-key cryptography, a number that is a product of two (not necessarily distinct) primes is known as a **semiprime**. Such numbers are also called **biprimes**, **pq-numbers**, and **2-almost primes**. Currently the largest known semiprime is

$$(2^{30,402,457} - 1)^2$$

This number has over 18 million digits. This is the square of the largest known prime number.

- Over the years, various mathematical techniques have been developed for the factorization of large numbers. We will now very briefly mention some of the more prominent methods, **the goal here being merely to make the reader familiar with the existence of the methods.** For a full understanding of the mentioned methods, the reader must look up other sources where the methods are discussed in much greater detail:

Trial Division: This is the oldest technique. Works quite well for removing primes from large integers of up to 12 digits (that is, numbers smaller then 10^{12}. As the name implies, you simply divide the number to be factorized by successively larger integers. A variation is to form a product $m = p_1 p_2 p_3 \dots p_r$ of r primes and to then compute $gcd(n, m)$ for finding the largest prime factor in n. Here is a product of all primes $p \leq 97$:

$$2305567963945518424753102147331756070$$

Fermat's Factorization Method: Is based on the notion that every **odd** number n that has two non-trivial factors can be expressed as a difference of two squares, $n = (x^2 - y^2)$. If we can find such x and y, then the two factors of n are $(x-y)$ and $(x+y)$. Searching for these factors boils down to solving

$x^2 \equiv y^2 \pmod{n}$. This is referred to as a **congruence of squares**. That every odd n can be expressed as a difference of two squares from the fact that if $n = a \times b$, then

$$n = [(a + b)/2]^2 - [(a - b)/2]^2$$

Note that since n is assumed to be odd, both a and b are odd, implying that $a + b$ and $a - b$ will both be even. In its implementation, one tries various values of x hoping to find one that yields a square for $x^2 - n$. The search is begun with with the integer $x = \lceil \sqrt{n} \rceil$. Here is the pseudocode for this approach

```
x = ceil( sqrt( n ) )           # assume n is odd
y_squared = x ** 2  -  n
while y_squared is not a square
    x = x + 1
    y_squared = x ** 2  -  n    # y_squared = y_squared + 2*x + 1
return x - sqrt( b_squared )
```

This method works fast if n has a factor close to its square-root. In general, its complexity is $O(n)$. Fermat's method can be speeded up by using trial division for candidate factors up to $\sqrt{n}$.

Pollard-ρ Method: It is based on the following observations:

– Say d is a factor of n. Obviously, the **yet unknown** d satisfies $d|n$. Now assume that we have two **unequal** numbers a and b so that $a \equiv b \bmod d$. Then it must be

the case that $d|(a-b)$. Since, by assumption, $d|n$, so it must be the case that $gcd(a-b,n)$ is a multiple of d. That is, d is a factor of n.

– This suggests the following approach to finding a factor of n: (1) Randomly choose two numbers $a, b \leq \sqrt{n}$; (2) Find the $gcd(a-b,n)$; (3) If this gcd is equal to 1, go back to step 1 until the gcd calculation yields a number d greater than 1. This d must be a factor of n.

– The above procedure is logically straightforward, but runs into the following combinatorial issue concerning the gcd calculation in Step 2: Let's say we start the calculation by choosing the random numbers a and b. So we carry out the calculation of $gcd(a-b,n)$. Assuming that this gcd equals 1, we now generate another random number c. Now we must test c against both a and b. So we need to carry out 2 calculations of gcd for the second iteration of the algorithm. Assuming each of the gcd's is 1, we now choose a fourth random number d. Now we must test d against a, then against b, and then against c. So the third iteration is going to require 3 calculations of gcd. In general, for the k^{th} random number selected, you have to carry out k calculations of gcd.

– The above mentioned ever-increasing number of gcd calculations for each iteration of the algorithm is avoided by what

is the heart of the Pollard-ρ algorithm. The candidate numbers are generated pseudorandomly using a random function f that maps a set to itself. The sequence of numbers generated can be expressed as $x_{i+1} = f(x_i) \bmod n$. Again assuming the **yet unknown** factor d of n, suppose we discover a pair of indices i and j, $i < j$, for this sequence such that $x_i \equiv x_j \bmod d$, then obviously $f(x_i) \equiv f(x_j) \bmod d$. This implies that each element of the sequence after j will be congruent to each corresponding element of the sequence after i modulo the unknown d.

– So let's say we can find two numbers in the sequence x_i and x_{2i} that are congruent modulo the unknown factor d, then by the logic already explained $d|(x_i - x_{2i})$. Since $d|n$, it must be case that $gcd((x_i - x_{2i}), n)$ must be a multiple d.

– For a more efficient implementation of the above idea, the Pollard-ρ algorithm uses the function $f()$ to generate two sequence x_i and y_i, with the latter twice as fast as the former. That is, at each iteration, the first sequence corresponds to $x_{i+1} \leftarrow f(x_i)$ and $y_{i+1} \leftarrow f(f(y_i))$. This would cause each (x_i, y_i) pair to be the same as (x_i, x_{2i}). If we are in the cycle part of the sequence, and if $x_i \equiv x_{2i} \pmod d$, then we must have a $d = gcd((x_i - y_i), n)$, $d \neq 1$ and we are done.

Sieve Based Methods: Sieve is a process of successive cross-

ing out entries in a table of numbers according to a set of rules so that only some remain as candidates for whatever one is looking for. The oldest known sieve is the **sieve of Eratosthenes** for generating prime numbers. In order to find all the prime integers up to a number, you first write down the numbers successively (starting with the number 2) in an array-like display. The sieve algorithm then starts by crossing out all the numbers divisible by 2 (and adding 2 to the list of primes). Next you cross out all the entries in the table that are divisible by 3 and you add 3 to the list of primes, and so on. Modern sieves that are used for fast factorization are known as **quadratic sieve**, **number field sieve**, etc. The quadratic sieve method is the fastest for integers under 110 decimal digits and considerably simpler than the number field sieve. Like the principle underlying Fermat's factorization method, the quadratic sieve method tries to establish congruences module n. In Fermat's method, we search for a single number x so that $x^2 \bmod n$ is a square. But such x's are difficult to find. With quadratic sieve, we compute $x^2 \bmod n$ for many x's and then find a subset of these whose product is a square.

Factorization of Large Numbers: RSA Factorization Challenge

- Since the security of the RSA algorithm is so critically dependent on the difficulty of finding the prime factors of a large number, RSA Labs (`http://www.rsasecurity.com/rsalabs/`) sponsors a challenge to factor the numbers supplied by them. The challenges are denoted

  `RSA-XXX`

  where XXX stands for the **number of bits** needed for a binary representation of the number in the new round of challenges starting with $RSA-576$.

- Let's look at the factorization of the number in the RSA-200 challenge (200 here refers to the number of decimal digits):

  ```
  RSA-200 =
    27997833911221327870829467638722601621070446786955428537560009929326128400107609345671052955360856
    06182235191095136578863710595448200657677509858057613579098734950144178863178946295187237869221823983
  ```

  Its two factors are

  ```
  35324619344027701212726049781984643686711974001976250
  23649303468776121253679423200058547956528088349

  79258699544783330333470858414800596877379758573642
  19960734330341455767872818152135381409304740185467
  ```

- RSA-200 was factored on May 9, 2005 by Bahr, Boehm, Franke, and Kleinjung of Bonn University and Max Planck Institute.

- Here is a description of RSA-576:

```
Name:         RSA-576
Prize:        $10000
Digits:       174
Digit Sum:    785
188198812920607963838697239461650439807163563379
417382700763356422988859715234665485319060606504
743045317388011303396716199692321205734031879550
65699622130516875930765025705 9
```

- RSA-576 was factored on Dec 3, 2003 by using a combination of lattice sieving and line sieving by a team of researchers (Franke, Kleinjung, Montgomery, te Riele, Bahr, Leclair, Leyland, and Wackerbarth) working at Bonn University, Max Planck Institute, and some other places.

- Here is a description of RSA-640:

```
Name:         RSA-640
Prize:        $20000
Digits:       193
Digit Sum:    806
310741824049004372135075003588856793003734602 28
427275457201619488232064405180815045563468296 71
723286782437916272838033415471073108501919548 52
```

```
900733772482278352574238645401469173660247765234660
9
```

- RSA-640 was factored on November 2, 2005 by the same team that solved RSA-576. Took over five months of calendar time.

RSA Factorization Challenge: Numbers Not Yet Factored

Name: RSA-704
Prize: $30000
Digits: 212
Digit Sum: 1009

```
74037563479561712828046796097429573142593188889
23128908493623263897276503402826627689199641962
51178439958943305021275853701189680982867331732
73108930900552505116877063299072396380786710086
09696253793465056379635 9
```

Name: RSA-768
Prize: $50000
Digits: 232
Digit Sum: 1018

```
12301866845301177551304949583849627207728535695
95334792197322452151726400507263657518745202199
78646938995647494277406384592519255732630345373
15482685079170261221429134616704292143116022212
40479274737794080665351419597459856902143413
```

Name: RSA-896
Prize: $75000
Digits: 270
Digit Sum: 1222

```
412023436986659543855531365332575948179811699844327982845455626433876445565248426198098870423161841879261420247188869492560931776375033421130982397485150944909106910269861031862704114880866970564902903653658867433731720813104105190864254793282601391257624033946373269391
```

Name: RSA-1024
Prize: $100000
Digits: 309
Digit Sum: 1369

```
135066410865995223349603216278805969388814756056670275244851438515265106048595338339402871505719094417982072821644715513736804197039641917430464965892742562393410208643832021103729587257623585096431105640735015081875106765946292055636855294752135008528794163773285339061097505443349998111500569772368909275630
```

Name: RSA-1536
Prize: $150000
Digits: 463
Digit Sum: 2153

```
184769970321174147430683562020016440301854933866
341017147178577491065169671116124985933768430543
574458561606154457179405222971773252466096064694
607124962372044202226975675668737842756238950876
467844093328515749657884341508847552829818672645
133986336493190808467199043187438128336350279547
028265329780293491615581188104984490831954500984
839377522725705257859194499387007369575568843693
381277961308923039256969525326162082367649031603
655137144791393234716956698806 9
```

Name: RSA-2048
Prize: $200000
Digits: 617
Digit Sum: 2738

```
251959084756578934940271832400483985714292821262 0
403202777713783604366202070759555626401852588078 4
406918290641249515082189298559149176184502808489 1
200728449926873928072877767359714183472702618963 7
501497182469116507761337985909570009733045974880 8
428401797429100642458691817195118746121515172654 6
322822168699875491824224336372590851418654620435 7
679842338718477444792073993423658482382428119816 3
```

81501067481045166037730605620161967625613384414360383390441495263443219011465754445417842402092461651572335077870774981712577246796292638635637328991215483143816789988504044536402352738195137863656439121201039712282212072035 7

In Summary ...

- The security of RSA encryption depends critically on the difficulty of factoring large integers.

- As integer factorization algorithms have become more and more powerful over the years, RSA cryptography has had to rely on increasingly larger values for the integer modulus and, therefore, increasingly longer encryption keys.

- These days you are unlikely to use keys shorter than 1024 bits for RSA. Some people recommend 2048 or even 4096 bit keys.

- As you'd expect, the computational overhead of RSA encryption/decryption goes up as the size of the modulus integer increases.

- This makes RSA inappropriate for encryption/decryption of actual message content for high data-rate communication links.

- However, RSA is ideal for the exchange of secret keys that can subsequently be used for the more traditional (and much faster) symmetric-key encryption and decryption of the message content.