

# Lecture 12: Public-Key Cryptography and RSA

## Lecture Notes on “Computer and Network Security”

by Avi Kak (kak@purdue.edu)

March 26, 2009

©2009 Avinash Kak, Purdue University

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

## 12.1: 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. This is illustrated by the top communication link in Figure 1 on page 5.

- 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. This is illustrated by the middle communication link in Figure 1 on page 5.
- The communication link at the bottom of Figure 1 shows how public-key encryption can be used to provide both **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 Figure 1, A's public and private keys are designated  $PU_A$  and  $PR_A$ . B's public and private keys are designated  $PU_B$  and  $PR_B$ .
- As shown at the bottom of Figure 1, let's say that A wants to send a message  $M$  to B with both authentication and confidentiality. The processing steps undertaken by A to convert  $M$  into its encrypted form  $C$  that can be placed on the wire are:

$$C = E(PU_B, E(PR_A, M))$$

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

$$M = D(PU_A, D(PR_B, C))$$

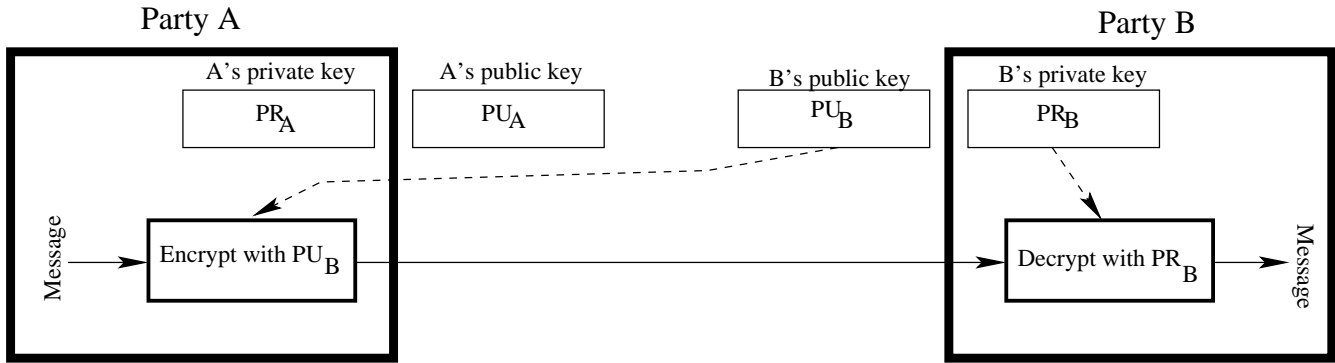
where  $D()$  stands for decryption.

- 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.

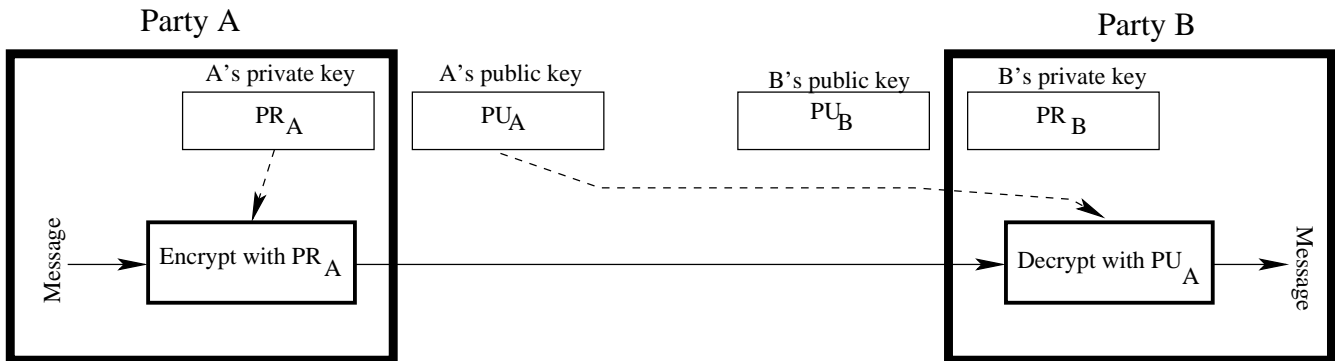
- Note that public-key cryptography does **not** make obsolete the more traditional symmetric-key cryptography. Because of the greater computational overhead associated with public-key cryptosystems, symmetric-key systems will continue to be used for the foreseeable future.
- However, it is generally agreed that public-key encryption is indispensable for key management and digital signature applications.

## Party A wants to send a message to Party B

When only confidentiality is needed:



When only authentication is needed:



When both confidentiality and authentication are needed:

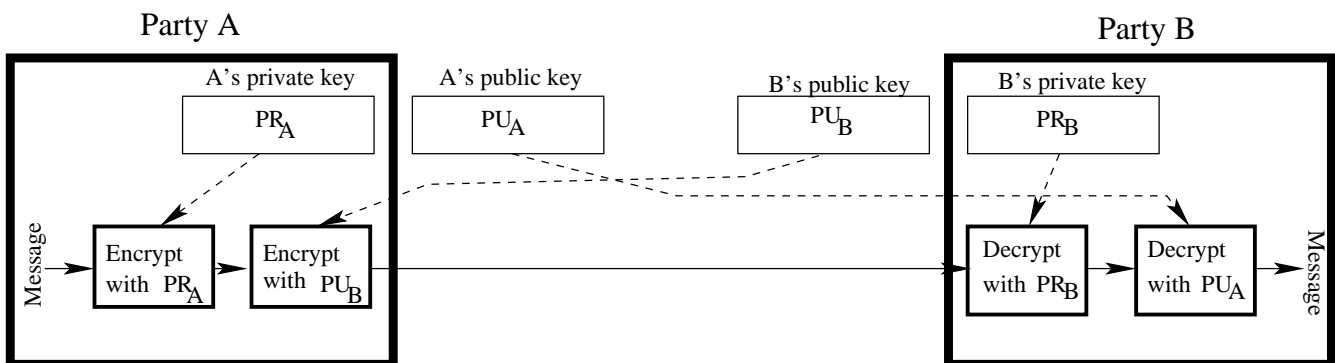


Figure 1: *This figure is from Lecture 12 of "Computer and Network Security" by Avi Kak*

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

- The RSA algorithm is based on the following properties of positive integers.
- In arithmetic operations modulo  $n$ , the exponents behave modulo the totient  $\phi(n)$  of  $n$ . [See Lecture 11 for the definition of the totient of a number.] For example, consider arithmetic modulo 15. We have  $\phi(15) = 8$  for the totient. You can easily verify the following:

$$5^7 \cdot 5^4 \bmod 15 = 5^{(7+4) \bmod 8} \bmod 15 = 5^3 \bmod 15 = 125 \bmod 15 = 5$$

$$(4^3)^5 \bmod 15 = 4^{(3 \times 5) \bmod 8} \bmod 15 = 4^7 \bmod 15 = 4$$

- Again considering arithmetic modulo  $n$ , let's say that  $e$  is an integer that is coprime to the totient  $\phi(n)$  of  $n$ . Further, say that  $d$  is the multiplicative inverse of  $e$  modulo  $\phi(n)$ . These definitions of the various symbols are listed below for convenience:

$n$  = *a modulus for modular arithmetic*

$\phi(n)$  = *the totient of  $n$*

$e$  = *an integer that is relatively prime to  $\phi(n)$*   
*[This guarantees that  $e$  will possess a*  
*multiplicative inverse modulo  $\phi(n)$ ]*

$d$  = *an integer that is the multiplicative*  
*inverse of  $e$  modulo  $\phi(n)$*

- 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 modulo exponentiation:

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

At this point, it may seem rather strange that we would want to represent any arbitrary plaintext message by an integer. But, it is really not that strange. Let's say you want a block cipher that encrypts 1024 bit blocks at a time. Every plaintext block can now be thought of as an integer  $M$  of value  $0 \leq M \leq 2^{1024} - 1$ .

- As you will soon see, we can **recover**  $M$  back from  $C$  by the following modulo operation

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

### 12.3: 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  $\{e, n\}$  as 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.
- As explained by an example in Section 12.10, RSA could be used as a **block cipher** for the encryption of the message. The block size would equal the number of bits required to represent the modulus  $n$ . If the modulus required, say, 1024 bits for its representation, message encryption would be based on 1024-bit blocks.

- 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?

## 12.4: How to Choose the Modulus for the RSA Algorithm?

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

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

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

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

$$n = p \times q \quad \text{for some prime } p \text{ and prime } q \quad (1)$$

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

1. If two integers  $p$  and  $q$  are coprimes (meaning, relatively prime to each other), the following equivalence holds for any two integers  $a$  and  $b$ :

$$\{a \equiv b \pmod{p} \text{ and } a \equiv b \pmod{q}\} \Leftrightarrow \{a \equiv b \pmod{pq}\} \quad (2)$$

This equivalence follows from the fact  $a \equiv b \pmod{p}$  implies  $a - b = k_1 p$  for some integer  $k_1$ . But since we also have  $a \equiv b \pmod{q}$  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. (Note that this argument breaks down if  $p$  and  $q$  have common factors other than 1.)

2. 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  $q$  (see Lecture 11). That is

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

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  $q$  be 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 these notes.

## 12.5: Proof of the RSA Algorithm

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

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

This implies that there must exist an integer  $k$  so that

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

- 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

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

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

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

for some integer  $k_1$ .

- Therefore, we can write for any integer  $M$ :

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

- 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

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

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

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

Substituting this result in Equation (8), we get

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

– 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  $n$ :

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

- We established in Section 12.4 that, since  $p$  and  $q$  are coprimes, for any integers  $a$  and  $b$  if we have  $a \equiv b \pmod{p}$  and  $a \equiv b \pmod{q}$ , then it must also be the case that  $a \equiv b \pmod{pq}$ . Applying this conclusion to the partial results shown in Equations (9) and (10), we get

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

## 12.6: 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

Generate two different primes  $p$  and  $q$

Calculate the modulus  $n = p \times q$

Calculate the totient  $\text{PHI}(n) = (p-1) \times (q-1)$

Select integer  $e$   $1 < e < \text{PHI}(n), \text{gcd}(\text{PHI}(n), e) = 1$

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

Public Key =  $[e, n]$

Private Key =  $[d, n]$

## 12.7: 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 modulus of size  $B$  bits.
- To generate the prime integer  $p$ ;
  - Using a high-quality random number generator (See Lecture 10 on random number generation), you first generate a random number of size  $B/2$  bits.
  - You set the lowest bit of the integer generated by the above step; 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. (See Section 12.10 for an explanation of why you need to set the first **two** bits.)
  - 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.

## 12.8: 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  $\gcd(e, q - 1) = 1$ .
- 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**. But don't forget the basic requirement on  $e$  that it must be relatively prime to  $p - 1$  and  $q - 1$  simultaneously. Whereas  $p$  is prime,  $p - 1$  definitely is not since it is even. The same goes for  $q - 1$ . So even if you wanted to, you may not be able to use a small integer like 3 for  $e$ .

- 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 declare  $p$  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 than 2, a **much faster** way to satisfy the two conditions is to ensure

$$\begin{array}{l} p \bmod e \neq 1 \\ q \bmod e \neq 1 \end{array}$$

## 12.9: 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 to 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

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

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

- Since  $d$  is the multiplicative inverse of  $e$  modulo  $\phi(n)$ , we can use the Extended Euclid's Algorithm (see Lecture 5) for calculating  $d$ . Recall that 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.

## 12.10: A Toy Example That Illustrates How to Set $n$ , $e$ , $d$ for a Block Cipher Application of RSA

- For the sake of illustrating how you'd use RSA as a block cipher, consider a toy example in which the block size is 16. What that means is that we want to scan a disk file and convert each 16-bit plaintext block into a 16-bit ciphertext block.
- So our first job is to find a modulus  $n$  whose size is 16 bits. Recall that  $n$  must be a product of two primes  $p$  and  $q$ . Assuming that we want these two primes to be roughly the same size, let's allocate 8 bits to  $p$  and 8 bits to  $q$ .
- So the issue now is how to find a prime suitable for our 8-bit representation. Following the prescription given in Section 12.7, we could fire up a random number generator, set its first two bits and the last bit, and then test the resulting number for its primality with the Miller-Rabin algorithm presented in Lecture 11. But we don't need to go to all that trouble for our toy example. Let's use the simpler approach described below.
- Let's assume that we have an as yet imaginary 8-bit word for  $p$

whose first two and the last bit are set. And assume that the same is true for  $q$ . So both  $p$  and  $q$  have the following bit patterns:

$$\begin{aligned} \text{bits of } p & : & 11 & - & - & - & - & 1 \\ \text{bits of } q & : & 11 & - & - & - & - & 1 \end{aligned}$$

where ‘-’ denotes the bit that has yet to be determined. As you can verify quickly from the three bits that are set, such an 8-bit integer will have a minimum decimal value of 193. [Here is a reason for why you need to manually set the first two bits: Assume for a moment that you set only the first bit. Now it is theoretically possible for the smallest values for  $p$  and  $q$  to be not much greater than  $2^7$ . So the product  $p \times q$  could get to be as small as  $2^{14}$ , which obviously does not span the full 16 bit range desired for  $n$ . When you set the first two bits, now the smallest values for  $p$  and  $q$  will be lower-bounded by  $2^7 + 2^6$ . So the product  $p \times q$  will be lower-bounded by  $2^{14} + 2 \times 2^{13} + 2^{12}$ , which itself is lower-bounded by  $2 \times 2^{14} = 2^{15}$ , which corresponds to the full 16-bit span. With regard to the setting of the last bit of  $p$  and  $q$ , that is to ensure that  $p$  and  $q$  will be odd.]

- So the question reduces to whether there exist two primes (hopefully different) whose decimal values exceed 193 but are less than 255. If you carry out a Google search with a string like “first 1000 primes,” you will discover that there exist many candidates for such primes. Let’s select the following two

$$p = 197$$

$$q = 211$$

which gives us for the modulus  $n = 197 \times 211 = 41567$ . The bit pattern for the chosen  $p$ ,  $q$ , and modulus  $n$  are:

$$\begin{aligned} \text{bits of } p & : 0Xc5 & = & 1100\ 0101 \\ \text{bits of } q & : 0Xd3 & = & 1101\ 0011 \\ \text{bits of } n & : 0Xa25f & = & 1010\ 0010\ 0101\ 1111 \end{aligned}$$

**As you can see, for a 16-bit block cipher, we have a modulus that requires 16 bits for its representation.**

- Now let's try to select appropriate values for  $e$  and  $d$ .
- For  $e$  we want an integer that is relatively prime to the totient  $\phi(n) = 196 \times 210 = 41160$ . Such an  $e$  will also be relatively prime to 196 and 210, the totients of  $p$  and  $q$  respectively. Since it is preferable to select a small prime for  $e$ , we could try  $e = 3$ . But that does not work since 3 is not relatively prime to 210. The value  $e = 5$  does not work for the same reason. Let's try  $e = 17$  because it is a small prime and because it has only two bits set.

- With  $e$  set to 17, we must now choose  $d$  as the multiplicative inverse of  $e$  modulo 41160. Using the Bezout's identity based calculations described in Lecture 5, we write

$$\begin{array}{l|l}
 \gcd(17, 41160) & \\
 = \gcd(41160, 17) & \text{residue } 17 = 0 \times 41160 + 1 \times 17 \\
 = \gcd(17, 3) & \text{residue } 3 = 1 \times 41160 - 2421 \times 17 \\
 = \gcd(3, 2) & \text{residue } 2 = -5 \times 3 + 1 \times 17 \\
 & = -5 \times (1 \times 41160 - 2421 \times 17) + 1 \times 17 \\
 & = 12106 \times 17 - 5 \times 41160 \\
 = \gcd(2, 1) & \text{residue } 1 = 1 \times 3 - 1 \times 2 \\
 & = 1 \times (41160 - 2421 \times 17) \\
 & \quad - 1 \times (12106 \times 17 - 5 \times 41160) \\
 & = 6 \times 41160 - 14527 \times 17 \\
 & = 6 \times 41160 + 26633 \times 17
 \end{array}$$

where the last equality for the residue 1 uses the fact that the additive inverse of 14527 modulo 41160 is 26633.

- The Bezout's identity shown above tells us that the multiplicative inverse of 17 modulo 41160 is 26633. You can verify this fact by showing  $17 \times 26633 \bmod 41160 = 1$  on your calculator.
- Our 16-bit block cipher based on RSA therefore has the following numbers for  $n$ ,  $e$ , and  $d$ :

$$\begin{array}{l}
 n = 41567 \\
 e = 17 \\
 d = 26633
 \end{array}$$

Of course, as you would expect, this block cipher would have no security since it would take no time at all for an adversary to factorize  $n$  into its components  $p$  and  $q$ .

## 12.11: 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 (see Lecture 11 for CRT). 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

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

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

## 12.12: 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  $B$ :

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

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  $B$ :

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

- Now the exponentiation  $A^B$  may be expressed as

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

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:

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

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

$$A^{2^0}, A^{2^1}, A^{2^2}, A^{2^3}, \dots$$

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:

$$A, A_{previous}^2, A_{previous}^2, A_{previous}^2, \dots$$

- 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

```

## 12.13: 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 solving the **integer factorization** problem involving large numbers. We will now 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 than  $10^{12}$ ). 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- $\rho$  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  $a \equiv b \pmod{d}$ . 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- $\rho$  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  $x_i \equiv x_j \pmod d$ , then obviously  $f(x_i) \equiv f(x_j) \pmod d$ . 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- $\rho$  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 crossing 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  $x^2 \pmod{n}$  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.

## 12.14: Factorization of Large Numbers: The Old RSA Factoring 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/>) used to sponsor a challenge to factor the numbers supplied by them.
- The challenge generated a lot of excitement when it was active. Many of the large numbers put forward by RSA Labs for factoring have still not been factored and are not expected to be factored any time soon.
- Given the historical importance of this challenge and the fact that many of the numbers have not yet been factored makes it interesting to review the state of the challenge today.
- The challenges are denoted

**RSA-XXX**

where XXX stands for the **number of bits** needed for a binary representation of the number to be factored in the 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 =

```
2799783391122132787082946763872260162107044678695
5428537560009929326128400107609345671052955360856
0618223519109513657886371059544820065767750985805
57613579098734950144178863178946295187237869221823983
```

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  
656996221305168759307650257059

- 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  
31074182404900437213507500358885679300373460228  
42727545720161948823206440518081504556346829671  
72328678243791627283803341547107310850191954852  
90073377248227835257423864540146917366024776523  
46609

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

## 12.15: The Old RSA Factoring Challenge: Numbers Not Yet Factored

Name: RSA-704  
Prize: \$30000 (prize retracted)  
Digits: 212  
Digit Sum: 1009  
74037563479561712828046796097429573142593188889  
23128908493623263897276503402826627689199641962  
51178439958943305021275853701189680982867331732  
73108930900552505116877063299072396380786710086  
096962537934650563796359

Name: RSA-768  
Prize: \$50000 (retracted)  
Digits: 232  
Digit Sum: 1018  
12301866845301177551304949583849627207728535695  
95334792197322452151726400507263657518745202199  
78646938995647494277406384592519255732630345373  
15482685079170261221429134616704292143116022212  
40479274737794080665351419597459856902143413

Name: RSA-896  
Prize: \$75000 (retracted)  
Digits: 270  
Digit Sum: 1222

41202343698665954385553136533257594817981169984  
43279828454556264338764455652484261980988704231  
61841879261420247188869492560931776375033421130  
98239748515094490910691026986103186270411488086  
69705649029036536588674337317208131041051908642  
54793282601391257624033946373269391

Name: RSA-1024  
Prize: \$100000 (retracted)  
Digits: 309  
Digit Sum: 1369

135066410865995223349603216278805969938881475605  
667027524485143851526510604859533833940287150571  
909441798207282164471551373680419703964191743046  
496589274256239341020864383202110372958725762358  
509643110564073501508187510676594629205563685529  
475213500852879416377328533906109750544334999811  
150056977236890927563

Name: RSA-1536  
Prize: \$150000 (retracted)  
Digits: 463  
Digit Sum: 2153  
184769970321174147430683562020016440301854933866  
341017147178577491065169671116124985933768430543  
574458561606154457179405222971773252466096064694  
607124962372044202226975675668737842756238950876  
467844093328515749657884341508847552829818672645  
133986336493190808467199043187438128336350279547  
028265329780293491615581188104984490831954500984  
839377522725705257859194499387007369575568843693  
381277961308923039256969525326162082367649031603  
6551371447913932347169566988069

Name: RSA-2048  
Prize: \$200000 (retracted)  
Digits: 617  
Digit Sum: 2738  
2519590847565789349402718324004839857142928212620  
4032027777137836043662020707595556264018525880784  
4069182906412495150821892985591491761845028084891  
2007284499268739280728777673597141834727026189637  
5014971824691165077613379859095700097330459748808  
4284017974291006424586918171951187461215151726546  
3228221686998754918242243363725908514186546204357  
6798423387184774447920739934236584823824281198163

8150106748104516603773060562016196762561338441436  
0383390441495263443219011465754445417842402092461  
6515723350778707749817125772467962926386356373289  
9121548314381678998850404453640235273819513786365  
64391212010397122822120720357

## 12.16: The RSA Algorithm: Some Operational Details

- **The size of the key in the RSA algorithm typically refers to the size of the modulus integer in bits.**
- Consider the case of an RSA implementation that provides 1024 bits of security. So we are talking about an implementation of the RSA algorithm that uses a 1024 bit modulus. It is interesting to reflect on the fact that 1024 bits can be stored in only 128 bytes in the memory of a computer (that translates into a 256-character hex string if we had to print out the 128 bytes for a visual display), yet the decimal value of the integer represented by these 128 bytes can be monstrously large. Here is an example of such a decimal number:

```
896648260163177445892450830685346881485335435
598887985722112773321881386436681238522440572
201181538908178518569358459456544005330977672
121582110702985339908050754212664722269478671
818708715560809784221316449003773512418972397
715186575579269079705255036377155404327546356
26323200716344058408361871194193919999
```

There are 359 decimal digits in this very large integer. [It is trivial to generate arbitrarily large integers in Python since the language places no limits on

the size of the integer. I generated the above number by simply setting a variable to a random 256 character hex string by a statement like

```
num = 0x7fafdbff7fe0f9ff7.... 256 hex characters ..... ff7ffda5f
```

and then just calling 'print num'.] The above example should again remind you of the exponential relationship between what it takes to represent an integer in the memory of a computer and the value of that integer.

- RSA Laboratories recommends that the two primes that compose the modulus should be roughly of equal length. So if you want to use 1024-bit RSA encryption, that means that your modulus integer will have a 1024 bit presentation, and that further means that you'd need to generate two primes that are roughly 512 bits each.
- Doubling the size of the key will, in general, increase the time required for public key operations (as needed for encryption or signature verification) by a factor of four and increase the time taken by private key operations (decryption and signing) by a factor of 8. Public key operations are not as affected as the private key operations when you double the size of the key is because the public key exponent  $e$  does not have to change as the key size increases. On the other hand, the private key exponent  $d$  changes in direct proportion to the size of the modulus. The key generation time goes up by a factor of 16 as the size of the key (meaning the

size of the modulus) is doubled. But key generation is a relatively infrequent operation. (Ref.: <http://www.rsa.com/rsalabs>)

- The public and the private keys are stored in particular formats specified by various protocols. For the public key, in addition to storing the encryption exponent and the modulus, the key may also include information such as the time period of validity, the name of the algorithm used for key generation, etc. For the private key, in addition to storing the decryption exponent and the modulus, the key may include additional information along the same lines as for the public key, and, additionally, the corresponding public key also. Typically, the formats call for the keys to be stored using Base64 encoding so that displayed using printable characters. (See Lecture 2 on Base64 encoding.) To see such keys, you could, for example, experiment with the following function:

```
ssh-keygen -t dsa
```

[The public and the private key returned by this call, when stored appropriately, will allow your laptop to establish SSH connections with machines elsewhere from virtually anywhere in the world (unless a local firewall blocks SSH traffic) without you having to log in explicitly. In the above call, 'dsa' refers to the Digital Signature Algorithm that typically uses the ElGamal protocol for generating the key pairs. You can use the same call to create RSA keys by simply replacing 'dsa' with 'rsa'. (A call such as above will ask you for a passphrase, but you can ignore it if you wish.) The above call will store the private key in the file `.ssh/id_dsa` of your home account. If you generate the RSA keys, the private key will be stored in `.ssh/id_rsa` file. The public key will be

deposited in a file that will be named either `.ssh/id_dsa.pub` or `.ssh/id_rsa.pub`. Now all you have to do is to copy the public key into the file `.ssh/authorized_keys` of any of the remote machines to which SSH access without the bother of logins.]

- Here is an example of a private key in the `.ssh/id_rsa` file of a now retired machine. Note that it is in Base64 encoding.

```
-----BEGIN RSA PRIVATE KEY-----
MIIEogIBAACAQEA5amriY96HQS8Y/nKc8zu3z0y1vp0n3vzMmWwrtyDy+aBvns4
UC1RXoaD9rDKqNNMCBAQwWdsYwCAFsrBzbxRQONHePX81RWgM87MseWGlU6WPzWG
iJMclTA09CTknp1G9w1NzLQBJ3dP1M895iLF6jvJ7GR+V3CRU6UUbMmRvgPcsfv6
ec9RRPm/B8ftUuQICLOjt4tKdPG45PBJUy1Hs71FuE9FJNp01hrj1EMF0bNTcsy9
zuis0YPyZArTYS0UsGg1leExAQYi7iLh17pAa+y6fZrGLsptgqryuftN9Q4NqPuT
gsB/AoGBAPudYPoCvHMEI4V0d1EcALUIIaxFKKSAkXzIzb0sxbj699SR1VHdyot
vIkRm+8aWStwJSfB+fSUE/U2014pvoCIHSyiDccPC4gzveHSrwd7GLU4R2Hxh837
Pn/hUtTDQXQ1yGDDFH84bhszuUh+L8KZ3m5rt0g7/EsntzIc0qTHAoGBA0mqWUsw
VdrOK483uvTjdYiQchF/zJhXfD3ywn4IFtvKo/nsKb/TsxWZkMmR03m0qBShhESP
orW2wch22QK/lrQot1oTkezLRNZ06YfyhqKf6P3tu25Yp3+g6+ogvi4I14zY7+wX
m71YYIQZ/G3Z3LcTvv9ySShbvyH1/ggIzDJfAoGBAIFm4WqiHaNhNtbX5ZdtfLtf
iFjlqRowCDU7sSxKDgU7bzhshyVx3+pzX04D2QIBIwKCAQB8rJBR/W4tApInpNud
8ugSxESBgJEUv6FHPoR8LpCwhJRdhdBd6/UOmT1AOMLMOAho1I9F1vAtyD2bhFv
r19PHete6++Eia69CdzVmdtZP7C1+Hw7gw+EMAgeIqf+UzUnBQz6GJMh/vDSnGnu
TWQgEdQD/AoSNCs8CShYUCqLt+y2Bmm451M+P2Pf8ieiUYsm8ebixnrHK6LfU1+
KRkgEVgk51SXi4qEYPcL4Ja9k96ickIuE1HUFW1LABJBCaHT4mwwmJRleJ2/UaeV
BiW25fyr5MdQNVQPaljY3kY06209zL/33z1k9dI5WyshwNA0VZt6t/3LHEu54mDj
nEn3roB8opfyPexC6dpmlpAbr6gzYdstdudoJE8U3WbL9dnuuxARo90yI4+DLsXC
WCWVK6gzn4fgGILEH4AwrZ8HACO+C1P9jdt8BwMput03BSB1YNB17Y326Gf+04j
PzF50bAe2Yw8p1uZy2qAoGBANWjD8+3KeyfPcTFPterZCUWwamZapnplioChe+S
XgrH5mDX66gSAtHrfrAQTFIEQeb6EofTx/aYdqinLM9QFMH5Vy3Iv+5ws/dGUdth
ZSb4moHdaYl1oHSwYqoskJ8eBucsvhmVLOpfbi+iuugXpTmrp0/zdhZFQQkba+oW
rBDLAoGACEEjZnRkxKogIobZcmLZF1rJEUnpaezuXp5dWjh1CBUqjJfxGKeSR7VH
WCqx21GvA5ipwZp0HuCaWvWNQ/tdx14fTG4aES2/uurZBs0umzJZPJIC25shJLa+
TOCKIDY3afvDdVsktwzLnCybMOWQZVTGX1k6sttROH0swshX4A=
-----END RSA PRIVATE KEY-----
```

- And here is an example of the public key that goes with the above private key

```
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA5amriY96HQS8Y/nKc8zu3z0y1vp
```

On3vzMmWwrtyDy+aBvns4UC1RXoaD9rDKqNNMCBAQwWdsYwCAFsrBzbxRQONHeP  
X81RWgM87MseWglu6WPzWGiJMc1TA09CTknplG9w1NzLQBj3dP1M895iLF6jvJ7  
GR+V3CRU6UUbMmRvgPcsfv6ec9RRPm/B8ftUuQICLOjt4tKdPG45PBJUylHs71F  
uE9FJNp01hrj1EMF0bNTcsy9zuis0YPyzArTYSOUSGg1leExAQYi7iLh17pAa+y  
6fZrGLsptgqryuftN9Q4NqPuTiFjlqRowCDU7sSxKDgU7bzhshyVx3+pzX04D2Q  
== kak@pixie

## 12.17: 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.

## **12.18: Acknowledgments**

I'd like to thank Matheux Matheux for bringing to my attention the fact that the RSA Factoring Challenge is no longer active.