Lecture 12: Public-Key Cryptography and the RSA Algorithm

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

March 26, 2018

12:55am

Goals:

- To review public-key cryptography
- To demonstrate that confidentiality and sender-authentication can be achieved simultaneously with public-key cryptography
- To review the 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 vulnerabilities of RSA**
- **Perl and Python implementations for generating primes and for factorizing medium to large sized numbers**

CONTENTS

12.1: PUBLIC-KEY CRYPTOGRAPHY

- Public-key cryptography is also known as *asymmetric-key* cryptography, to distinguish it from the *symmetric-key* cryptography we have studied thus far.

- Encryption and decryption are 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**.

- With public key cryptography, all parties interested in secure communications publish their public keys. [As to how that is done depends on the protocol. In the SSH protocol, each server makes available through its port 22 the public key it has stored for your login id on the server. (See Section 12.10 for how an SSHD server acquires the public key that the server would associate with your login ID so that you can make a password-free connection with the server. In the context of the security made possible by the SSH protocol, the public key held by a server is commonly referred to as the server's *host key*.) When a client, such as your laptop, wants to make a connection with an SSHD server, it sends a connection request to port 22 of the server machine and the server makes its host key available automatically. On the other hand, in the SSL/TLS protocol, an HTTPS web server makes its public key available through a certificate of the sort you'll see in the next lecture.] As we will see, this solves one of the most vexing problems associated with symmetric-key cryptography — the problem of key distribution.

- 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.

- 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.

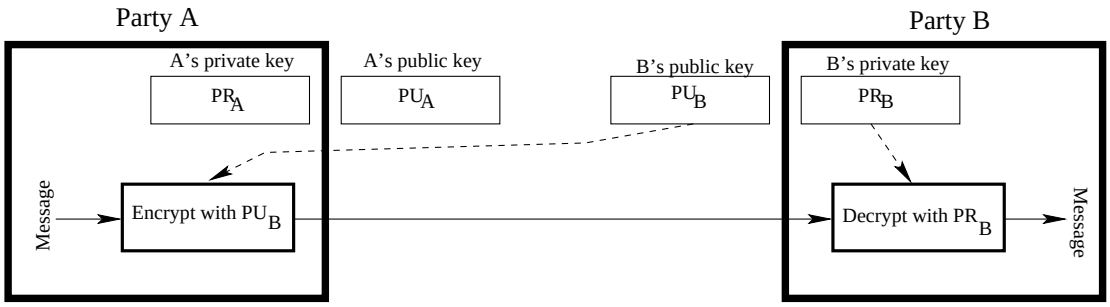
- The communication link at the bottom of Figure 1 shows how public-key encryption can be used to provide both **confidentiality and authentication** at the same time. Note again that **confidentiality** means that we want to protect a message from eavesdroppers and **authentication** means that the 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.

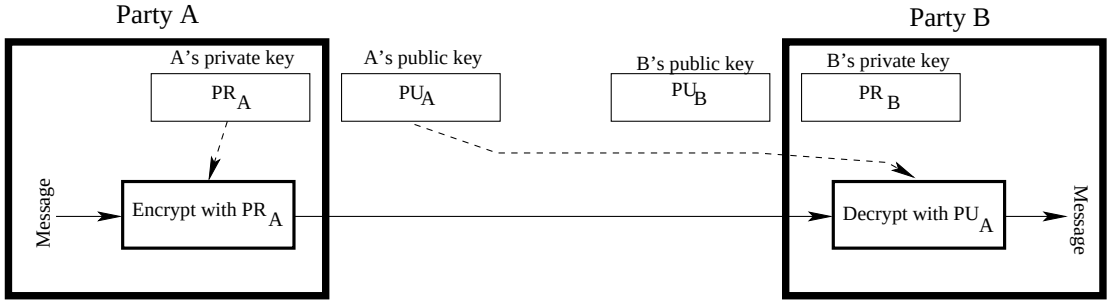
- As shown at the bottom of Figure 1, let's say that A wants to send a message M to B with both authentication and confidentiality.

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:

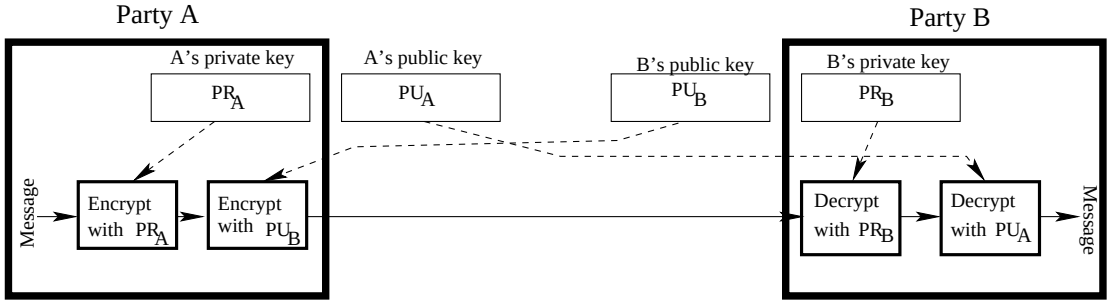


Figure 1: *This figure shows how public-key cryptography can be used for confidentiality, for digital signatures, and for both.* (*This figure is from Lecture 12 of "Computer and Network Security" by Avi Kak.*)

The processing steps undertaken by A to convert M into its encrypted form C that can be placed on the wire are:

$$C \quad = \quad E\left(PU_B,\ E\left(PR_A,\ M\right)\right)$$

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

$$M \quad = \quad D\left(PU_A,\ D\left(PR_B,\ C\right)\right)$$

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. [**DID YOU KNOW** that you are required to digitally sign the software for your app before you can market it through the official Android application store Google Play? And did you know that Apple's App Store has the same requirement?]

- 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.

- **IMPORTANT:** 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 crypto systems, symmetric-key systems continue to be widely used for content encryption. However, public-key encryption has proved indispensable for key management, for distributing the keys needed for the more traditional symmetric key encryption/decryption of the content, for digital signature applications, etc.

12.2: THE RIVEST-SHAMIR-ADLEMAN (RSA) ALGORITHM FOR PUBLIC-KEY CRYPTOGRAPHY — THE BASIC IDEA

- The RSA algorithm — named after Ron Rivest, Adi Shamir, and Leonard Adleman — is based on a property of positive integers that we describe below.

- **As a direct consequence of the Euler's Theorem of Section 11.4 of Lecture 11, we can state that when a and n are relatively prime, in exponentiation operations $a^k \; mod \; n$, the exponents behave modulo the totient $\phi(n)$ of n.** [See Section 11.3 of Lecture 11 for the definition of the totient of a number.] Euler's theorem says that $a^{\phi(n)} \equiv 1 \; (mod \; n)$ when a and n are relatively prime. Given an arbitrary exponent k, we may express it as $k = k_1\phi(n) + k_2$ for some values of k_1 and k_2. Euler's theorem implies: $a^k \; mod \; n = a^{k_2} \; mod \; n$.

- For example, consider arithmetic modulo 15. As explained in Section 11.3 of Lecture 11, since $15 = 3 \times 5$, we have $\phi(15) = 2 \times 4 = 8$ for the totient of 15. You can easily verify the following:

$$4^7 \cdot 4^4 \bmod 15 \;=\; 4^{(7+4) \bmod 8} \bmod 15 \;=\; 4^3 \bmod 15 \;=\; 64 \bmod 15 \;=\; 4$$

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

Note that in both cases the base of the exponent, 4, is coprime to the modulus 15.

- It follows from Euler's theorem that given two exponents e and d such that one is the multiplicative inverse of the other modulo $\phi(n)$, we have $M^{e\times d} \equiv M^{e\times d \ (mod\ \phi(n))} \equiv M \ (mod\ n)$.

- The result shown above, which follows directly from Euler's theorem, requires that M and n be coprime. **However, as will be shown in Section 12.2.3, when n is a product of two primes p and q, this result applies to all M, $0 \leq M < n$.** In what follows, let's now see how this property can be used for message encryption and decryption.

- 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:

$$n \quad = \quad a\ modulus\ for\ modular\ arithmetic$$

$$\begin{aligned} \phi(n) &= \textit{the totient of } n \\ e &= \textit{an integer that is relatively prime to } \phi(n) \\ &\quad [\textit{This guarantees that } e \textit{ will possess a} \\ &\quad\quad \textit{multiplicative inverse modulo } \phi(n)] \\ d &= \textit{an integer that is the multiplicative} \\ &\quad\quad \textit{inverse of } e \textit{ modulo } \phi(n) \end{aligned}$$

- Now suppose we are given an integer M, $0 \leq M < n$, 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 $0 \leq M \leq 2^{1024} - 1$.

- We can **recover** back M from C by the following modulo operation

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

since

$$(M^e)^d \ (mod\ n) \quad = \quad M^{ed\ (mod\ \phi(n))} \quad \equiv \quad M \pmod{n}$$

12.2.1: The RSA Algorithm — Putting to Use the Basic Idea

- The basic idea described in the previous subsection can be used to create a confidential communication channel in the manner described here.

- An individual A 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. The definitions of n, e, and d are as in the previous subsection.

- Another party B wishing to send a message M to A confidentially will encrypt M using A's public key $\{e, n\}$ to create ciphertext C. Subsequently, only A will be able to decrypt C using his/her private key $\{d, n\}$.

- If the plaintext message M is too long, B may choose to use RSA as a **block cipher** for encrypting the message meant for A. As explained by our toy example in Section 12.4, when RSA is used as a block cipher, the block size is likely to be half 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 512-bit blocks. [While, in principle, RSA can certainly be used as a block cipher, in practice, on account of its excessive computational overhead, it is more likely to be used just for server authentication and for exchanging a secret session key. A session key generated with the help of RSA-based encryption can subsequently be used for content encryption using symmetric-key cryptography based on, say, AES.]

- 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?

12.2.2: How to Choose the Modulus for the RSA Algorithm

- With the definitions of d and e as presented in Section 12.2, the modulus n must be selected in such a manner that the following is guaranteed:

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

  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, presented in the next subsection, 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:

$$\{a \equiv b \pmod{p} \quad and \quad a \equiv b \pmod{q}\} \quad \Leftrightarrow \quad \{a \equiv b \pmod{pq}\} \tag{2}$$

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.) [We will use this property to arrive at Equation (11) shown in the next subsection from the partial results in Equations (9) and (10) presented in the same subsection.]

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. That is

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

See Section 11.3 of Lecture 11 for a proof of this. [We will use this property to go from Equation (5) to Equation (6) in the next subsection.]

- 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.

12.2.3: Proof of the RSA Algorithm

- We need to prove that when n is a product of two primes p and q, then, in arithmetic modulo n, the exponents behave modulo the totient of n. We will prove this assertion indirectly by establishing that when an exponent d is chosen as a $mod\ \phi(n)$ multiplicative inverse of another exponent e, then the following will always be true $M^{e \times d} \equiv M\ (mod\, n)$ for all $0 \leq M < n$. [The specific derivational steps presented below do not impose the constraint that the message integer M be limited to $0 \leq M < n$. However, should it be the case that $M \geq n$, what would be returned by the operation $M^{e \times d} mod\ n$ would be the remainder of M in Z_n. Let's just say that the message integer is given by $M = n$. For this value of M, the value returned by $M^{e \times d} mod\ n$ would be 0, which is not a very useful thing to happen.]

- Using the definitions of d and e as presented in Section 12.2, since the integer d is the multiplicative inverse of the integer e modulo the totient $\phi(n)$, we obviously have

$$e \times d \equiv 1 \pmod{\phi(n)} \tag{4}$$

This implies that there must exist an integer k so that

$$\begin{aligned} e \times d - 1 &\equiv 0 \pmod{\phi(n)} \\ &= k \times \phi(n) \end{aligned} \tag{5}$$

- 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

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

  The notation '$\mid$' to indicate that its left argument is a divisor of the right argument was first introduced at the end of Section 5.1 in Lecture 5.

- 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 (presented in Section 11.2 of Lecture 11), 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

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

  – 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 in Section 12.2.2 that, when 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

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

12.3: COMPUTATIONAL STEPS FOR KEY GENERATION IN RSA CRYPTOGRAPHY

- The computational steps for key generation are

  1. Generate two different primes p and q
  2. Calculate the modulus $n = p \times q$
  3. Calculate the totient $\phi(n) = (p-1) \times (q-1)$
  4. Select for public exponent an integer e such that $1 < e < \phi(n)$ and $gcd(\phi(n), e) = 1$
  5. Calculate for the private exponent a value for d such that $d = e^{-1} \ mod \ \phi(n)$
  6. $Public\ Key = [e, n]$
  7. $Private\ Key = [d, n]$

- The next three subsections elaborate on these computational steps.

12.3.1: 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 12.4 for an explanation of why you need to set the first **two** bits.)
  - Using the Miller-Rabin algorithm described in Lecture 11, 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.

12.3.2: Choosing a Value for the Public Exponent e

- Recall that encryption consists of raising the message integer M to the power of the public exponent e modulo n. This step is referred to as **modular exponentiation**.

- The mathematical requirement on e is that $gcd(e,\ \phi(n)) = 1$, since otherwise e will not have a multiplicative inverse mod $\phi(n)$. 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 in the sense described in the next bullet. 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 use the Chinese Remainder Theorem (CRT) of Section 11.7 of Lecture 11 to reconstruct M^3 modulo $N = n_1 \times n_2 \times n_3$. (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 reconstructed M^3, all that the attacker has to do is to figure out the cube-root of M^3 to recover M. Finding cube-roots of even large integers is not that hard. (The Homework Problems section includes a programming assignment that focuses on this issue.)

- 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). 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

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

- To summarize the point made above, **you give priority to using a particular value for** e – such as a value like 65537 that has only two bits set. Having made a choice for the encryption integer e, you now find the primes p and q that, besides satisfying all other requirements on these two numbers, also satisfy the conditions that the chosen e would be coprime to the totients $\phi(p)$ and $\phi(q)$.

12.3.3: Calculating the Private Exponent d

- Once we have settled on a value for the public exponent e, the next step is to calculate the private 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 Euclid's Algorithm (see Section 5.6 of 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 the main source of security in RSA is 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.

12.4: A TOY EXAMPLE THAT ILLUSTRATES HOW TO SET n, e, d FOR A BLOCK CIPHER APPLICATION OF RSA

- As alluded to briefly at the end of Section 12.2.1, you are unlikely to use RSA as a block cipher for general content encryption. As mentioned in Section 12.12, for the moduli needed in today's computing environments, the computational overhead associated with RSA is much too high for it to be suitable for content encryption. Nevertheless, RSA (along with ECC to be presented in Lecture 14) plays a critical role in practically all modern protocols for establishing secure communication links between clients and servers. These protocols depend on RSA (and ECC) for clients and servers to authenticate each other — as you'll see in Lecture 13. In addition, RSA may also be used for generating session keys. Despite the fact that you are not likely to use RSA for content encryption, it's nonetheless educational to reflect on how it *could* be used for that purpose in the form of a block cipher.

- For the sake of illustrating how you'd use RSA as a block cipher, let's try to design a 16-bit RSA cipher for block encryption of disk files. A 16-bit RSA cipher means that our modulus will span 16 bits. [Again, in the context of RSA, an N-bit cipher means that the modulus is of

size N bits and NOT that the block size is N bits. This is contrary to not-so-uncommon usage of the phrase "N-bit block cipher" meaning a cipher that encrypts N-bit blocks at a time as a plaintext source is scanned for encryption.]

- With the modulus size set to 16 bits, we are faced with the important question of what to use for the size of bit blocks for conversion into ciphertext as we scan a disk file. Since our message integer M must be smaller than the modulus n, obviously our block size cannot equal the modulus size. This requires that we use a smaller block size, say 8 bits, and use some sort of a padding scheme to fill up the rest of the 8 bits. As it turns out, padding is an extremely important part of RSA ciphers. In addition to the need for padding as explained here, padding is also needed to make the cipher resistant to certain vulnerabilities that are described in Section 12.7 of this lecture.

- In the rest of the discussion in this section, we will assume for our toy example that our modulus will span 16 bits, but the block size will be smaller than 16 bits, say, only 8 bits. We will further assume that, as a disk file is scanned 8 bits at a time, each such bit block is padded on the left with zeros to make it 16 bits wide. We will refer to this padded bit block as our message integer M.

- 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 12.3.1, 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:

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

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

$$\begin{aligned} p &= 197 \\ q &= 211 \end{aligned}$$

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

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

As you can see, for a 16-bit RSA 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 integer 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 number 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 Section 5.6 of Lecture 5, we write

```
gcd(17, 41160)        |
  = gcd(41160, 17)    |  residue 17 =  0 x 41160  +  1 x 17
  = gcd(17, 3)        |  residue  3 =  1 x 41160  -  2421 x 17
  = gcd(3,2)          |  residue  2 =  -5 x 3     +  1 x 17
                      |             =  -5x(1 x 41160 - 2421 x 17) + 1 x 17
                      |             =  12106 x 17  - 5 x 41160
  = gcd(2,1)          |  residue  1 =  1x3  -  1 x 2
                      |             =  1x(41160 - 2421x17)
                      |                     - 1x(12106x17 -5x41160)
                      |             =  6 x 41160   - 14527 x 17
                      |             =  6 x 41160   + 26633 x 17
```

where the last equality for the residue 1 uses the fact that the additive inverse of 14527 modulo 41160 is 26633. [If you don't like working out the multiplicative inverse by hand as shown above, you can use the Python script `FindMI.py` presented in Section 5.7 of Lecture 5. Another option would be to use the `multiplicative_inverse()` method of the BitVector class.]

- 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:

$$\begin{aligned} n &= 41567 \\ e &= 17 \\ d &= 26633 \end{aligned}$$

  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.

12.5: MODULAR EXPONENTIATION FOR ENCRYPTION AND DECRYPTION

- As mentioned already, for encryption, 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, 35, 65537, etc. All these integers have only a small number of 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. Typically, d is of the same order as the modulus n.

- Computation of $C^d \bmod n$ can be speeded up by using the Chinese Remainder Theorem (CRT) (see Section 11.7 of 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 as explained in Section 11.7 of Lecture 11, 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 (presented in Section 11.2 of Lecture 11) that says that if a and p are coprimes then $a^{p-1} \; mod \; p = 1$.

- To see how Fermat's Little Theorem (FLT) 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

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

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

- When you use FLT in conjunction with CRT, you can calculate $C^d \; (mod \; n)$ in roughly quarter of the time it takes otherwise. [First note, as stated earlier in Section 12.3.1, both p and q are of the order of $n/2$ where n is the modulus. Since $V_p = C^d \; (mod \; p) = C^{d \; mod(p-1)} \; (mod \; p)$, and since d is of the order of n and $d \; mod(p-1)$ of the order of p (which itself is of the order of $n/2$), it should take no more than half the number of multiplications to calculate V_p compared to the number of multiplications needed for calculating $C^d \; (mod \; n)$ directly. The same would be true for calculating V_q. As a result, the total number of multiplications required for both V_p and V_q would be the same as in the direct calculation of $C^d \; (mod \; n)$. Note, however, the intermediate results in the modular exponentiation needed for V_p would never exceed p (and the same would never exceed q for V_q). Since integer multiplication takes time that is proportional to the square of the size of the bit fields involved, each multiplication involved in the calculation of V_p and V_q would take only one-quarter of the time it takes for each multiplication in computing $C^d \; (mod \; n)$ directly.]

- **While the speedup achieved with CRT is impressive indeed, it comes at a cost: It makes the calculation of $C^d \; (mod \; n)$ vulnerable to different types of Side Channel Attacks, such as the Fault Injection Attack and the Timing Attack. In the Fault Injection attack, for example, you can get a processor to reveal**

the values of the prime factors p and q just by deliberately causing the processor to miscalculate the value of either V_p or V_q (but not both). See Lecture 32 on "Security Vulnerabilities of Mobile Devices" for further information regarding these attacks.

12.5.1: An Algorithm for Modular Exponentiation

- After we have simplified the problem of modular exponentiation considerably by using CRT and Fermat's Little Theorem as discussed in the previous subsection, we are still left with having 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. Even when A and B consist of only a couple of digits, as in 7^{11}, the result can still be a 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 has 256 binary digits (that is 77 decimal digits) and B has 65537 binary digits. 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:

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

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:

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

- 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 roughly halves the difficulty of computing A^B because, assuming all the bits of B are set, the largest value of 2^i will be about half the largest value of B.

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

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

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

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

  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:

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

- 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 ) % n
    B = B >> 1                             # shift B by one bit to right
    A = ( A * A ) % n
return  result
```

- To see the dramatic speedup you get with modular exponentiation, try the following terminal session with Python

```
[ece404.12.d]$ => script
Script started on Mon 20 Feb 2012 10:23:32 PM EST

[ece404.12.d]$ => python
```

```
>>>
>>> print pow(7, 9633196, 9633197)
117649
>>>
>>>
>>>
>>> print (7 ** 9633196) % 9633197
117649
>>>
```

where the call to `pow(7, 9633196, 9633197)` calculates $7^{9633197-1}$ *mod* 9633197 through Python's implementation of the modular exponentiation algorithm presented in this section. This call will return instantaneously with the answer shown above. On the other hand, the second call that carries out the same calculation, but *without resorting to modular exponentiation*, may take several minutes, depending on the hardware in your machine. [You are encouraged to make similar comparisons with numbers that are even larger than those shown here. If you wish, you can record your terminal-interactive Python session with the command `script` as I did for the session presented above. First invoke `script` and then invoke `python` as shown above. Your interactive work will be saved in a file called `typescript`. You can exit the Python session by entering Ctrl-d and then exit the recording of your terminal session by entering Ctrl-d again.]

- **An important point to note is that whereas the RSA algorithm is made theoretically possible by the number property stated in Section 12.2, the algorithm is made practically possible by the fact that there exist fast and memory-efficient algorithms for modular exponentiation.**

12.6: THE SECURITY OF RSA — VULNERABILITIES CAUSED BY LACK OF FORWARD SECRECY

- A communication link possesses forward secrecy if the content encryption keys used in a session cannot be inferred from a future compromise of one or both ends of the communication link. Forward secrecy is also referred to as Perfect Forward Secrecy.

- To see why RSA lacks forward secrecy, imagine a patient attacker who is recording the encrypted communications between a server and client.

- As you will see in Lecture 13, in order to establish an encrypted session with a server (which could be an e-commerce website like Amazon.com), a client (which could be your laptop) downloads the server's certificate to, first, authenticate the server and to, then, get hold the server's RSA public key for the purpose of creating a secret session key. [As you will learn in Lecture 13, a client generates a pseudorandom number to serve as the session key. To transmit this session key to the server, the client encrypts it with the server's public key so that only the server would be able to decrypt it with its RSA private key. The client sends the encrypted session key to the server and, subsequently, the two sides engage in an encrypted conversation.]

- The attacker, who has managed to install a packet sniffer in the LAN to which the client is connected, patiently records all encrypted communications between the client and the server with the expectation that someday he will be able to get hold of the server's private keys. Obviously, if that were to happen, the attacker would be able to decrypt **the session key** that was sent encrypted by the client to the server. And, as you can imagine, after the attacker has figured out the session key, the attacker will be able to decipher all of the recorded communications between the client and the server.

- The attacker gaining access to a server's private keys is not as far fetched a scenario as one might think. Private keys may be leaked out anonymously by disloyal employees or through bugs in software. The Heartbleed bug that was discovered on April 7, 2014 is just the latest example of how private keys may fall prey to theft through bugs in software. [See Section 20.4.4 of Lecture 20 for further information on the Heartbeat Extension to the SSL/TLS protocol and the Heartbleed bug.]

- We say that the basic RSA algorithm makes it possible to carry out the exploit described above because it lacks forward secrecy. Whether or not this vulnerability in a given server-client interaction is a serious matter depends on the nature of the communications between the two — especially on the lifetime of the information exchanged between the two endpoints.

- **The solution to this problem with RSA lies in somehow creating a secret session key without putting it on the wire.** Naturally, your first reaction to this thought would be: "**but that is impossible!!!**." You are likely to add: "How can two sides share a secret without either mentioning it to the other?"

- However, as they say, never underestimate the power of human ingenuity. In Lecture 13, we will talk about an incredibly beautiful algorithm, known as the Diffie-Hellman (DH) algorithm, that makes it possible to create a session key without either party transmitting the key to the other party.

- Consequently, DH provides Perfect Forward Secrecy. However, as you will see in Lecture 13, DH does suffer from a shortcoming of its own: it is vulnerable to the man-in-the-middle attack. **By combining RSA with DH, what you get — denoted DHE-RSA — gives you perfect forward secrecy through the use of DH for exchanging the session keys and RSA for endpoint (say, server) authentication.** DHE stands for "Diffie-Hellman Exchange." Another commonly used combination protocol for creating secret session keys is ECDHE-RSA where ECDHE stands for Elliptic Curve Diffie-Hellman Exchange. The subject of elliptic curves for cryptography is presented in Lecture 14.

12.7: THE SECURITY OF RSA — CHOSEN CIPHERTEXT ATTACKS

- The ***basic*** RSA algorithm — that is, an encryption/decryption scheme whose implementation does not go beyond the mathematics of RSA as described so far — would be much too vulnerable to all kinds of attacks, simple and fancy. Regarding the simpler vulnerabilities, consider this: If we were to use the RSA algorithm only as it has been described so far, think of the following vulnerability: Let's say your public key uses the exponent 3 and that you are in the habit of sending very short messages to your business partners. If a message M is short enough, the ciphertext integer $C = M^3$ will be smaller than the modulus. Your enemies will be able to recover the plaintext integer M simply by taking the cube-root of C by using, say, the n^{th} *root algorithm*. Such attacks become unfeasible when message integers are padded, in the manner described in this section, so as to span the full length of the modulus. With appropriate padding, when the message M is raised to the power of the public exponent (even a small public exponent like 3), the result would exceed the modulus and C would now be the remainder modulo the modulus. Since n^{th} *root algorithm* do not exist for modular arithmetic, the enemy would not be able to recover M even if it is just a short message.

- Regarding the "fancier" vulnerabilities that RSA would fall prey to if it were to be implemented just in the form described so far, in this section we consider what are known as the Chosen

Ciphertext Attacks (CCA) on the RSA cipher.

- My immediate goal in this section is to convey to the reader what is meant by CCA. As to how RSA is made secure against CCA is a story of what goes into the padding bytes that are prepended to the data bytes in order to create a block of bytes that spans the width of the modulus.

- So that you understand the basic notion of CCA, a good place to start this section is to show how the data bytes are padded in Version 1.5 of the PKCS#1 scheme for RSA. This scheme is also more compactly referred to by the string "PKCS#1v1.5". [Going beyond the fundamental notions of RSA public-key cryptography presented in this lecture, how exactly those notions should be used in practice is governed by the different PKCS "schemes." The acronym PKCS stands for "Public Key Cryptography Standard." It designates a set of standards from RSA Labs for public-key cryptography.] Despite the fact that Version 1.5 was promulgated in 1993, I believe it is still the most widely used RSA scheme today. [Note that Versions 2.0 and higher of the PKCS#1 scheme are resistant to all known forms of CCA attacks. By the way, you can download all of the different versions of the PKCS#1 standard from the `http://www.rsa.com/rsalabs/` web site.]

- In PKCS#1v1.5, what is subject to encryption is a block of bytes, called, naturally, an Encryption Block (EB), that is composed of the following sequence of bytes:

```
00  ||  BT  ||  PS  ||  00  ||  D

<----------  k bytes  ---------->

k = size of modulus in bytes
```

where '||' means simple concatenation, the numeric '00' stands for a byte whose value is 0, the notation 'BT' means a one-byte integer that designates the type of EB, the notation 'PS' means a pseudorandomly generated "Padding String", and the symbol 'D' stands for the data bytes. The value of 'BT' is the integer 2 for encryption with RSA. [The values 0 and 1 for 'BT' are meant for RSA when it is used for digital signatures.]

- The PKCS#1v1.5 standard mandates that the pseudorandomly generated padding string PS contain at least 8 bytes for security reasons. Therefore, in PKCS#1v1.5, the minimum value for k, the size of the modulus, is 12 bytes. That would be accounted for by one byte for '00', one for 'BT', 8 for 'PS', one for another '00', with one leftover for the data byte 'D'.

- With that brief introduction to how an encryption block is constructed in PKCS#1v1.5, let's get back to the subject of CCA.

- Let's say you use my public key (n, e) to encrypt a plaintext message M into the ciphertext C. You send C to me, but on its way to me, the ciphertext C is picked up by someone we'll refer

to as the attacker. Through CCA, the attacker can figure out what the plaintext message M is even without having to know the decryption exponent d. The attacker's exploit would consist of the following steps:

- The attacker randomly chooses an integer s.
- The attacker constructs a new message — which hopefully would not arouse my suspicion — by forming the product $C' = s^e \times C \; mod \; n$.
- The attacker somehow lures me into decrypting C'. (I may cooperate because C' looks innocuous to me.)
- Assume that, for whatever reason, I send back to the attacker $M' = C'^d = (s^e \times C)^d \; mod \; n = s^{e \times d} \times C^d \; mod \; n = s \times M \; mod \; n$.
- The attacker will now be able to recover the original message M by $M = M' \times s^{-1} \; mod \; n$, assuming that the multiplicative inverse of s exists in Z_n. Remember, the choice of s is under the attacker's control.

- The fact that RSA could be vulnerable to such attacks was first discovered by George Davida in 1982.

- Another form of CCA was discovered by Daniel Bleichenbacher in 1998. In this attack, the attacker uses a sequence of randomly selected integers s to form a candidate sequence of ciphertexts $C' = s^e \times C \; mod \; n$. The attacker chooses the integers s one

at a time, forms the ciphertext C', and sends it to an oracle just to find out if C' is likely to have been produced by a message whose first two bytes have the integer values of 0 and 2 — in accordance with the format of the encryption block shown earlier in this section. Each positive return from the oracle allows the attacker to enlarge the size of s and make an increasingly narrower estimate for the value of the plaintext integer M that corresponds to the original C. The iterations end when the estimated value for M is just one number. [In case you are wondering about the "oracle" and as to what that would correspond to in practice, the goal here is merely to demonstrate that the attacker can recover the message integer M even with very limited knowledge that consists of some mechanism informing the attacker whether or not the chosen C' violates the structure of the encryption block that is stipulated for PKCS#1v1.5. Whether or not such a mechanism exists today is not the point. Such a mechanism *could* consist of the victim's RSA engine simply returning an error report whenever it receives a ciphertext that it believes was produced by a message that did not conform to the encryption block structure in PKCS#1v1.5.] Bleichenbacher's attack is reported in the publication "*Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS#1,*" that is available from his home page.

- These days one makes a distinction between two different types of chosen ciphertext attacks and these are referred to as CCA1 and CCA2. Under CCA1, the attacker can consult the decryption oracle mentioned above an arbitrary number of times, but only until the attacker has acquired the ciphertext C through eavesdropping or otherwise. And, under CCA2, the attacker can continue to consult the oracle even after seeing C. For obvious

reasons, in either model, the attacker cannot query the oracle with C itself. The CCA1 attack is also known as the *passive chosen ciphertext attack* and CCA2 as the *adaptive chosen ciphertext* attack. The attack by Bleichenbacher is an example of CCA2. The success of that attack implies that PKCS#1v1.5 is *not* CCA2 secure.

- RSA is made resistant to CCA2 when the padding bytes are set according to OAEP. OAEP stands for *Optimal Asymmetric Encryption Padding*. Unlike what you saw for the PKCS#1v1.5 format for encryption blocks at the beginning of this section, there is no structure in the encryption blocks under PKCS#1v2.x. The padding now involves a mask generation function that depends on a hash applied to a set of parameters. For further information, the reader is referred to the RSA Labs publication "*RSAES-OAEP Encryption Scheme*" and the references contained therein. This publication can be download from the same web site as mentioned at the beginning of this section.

12.8: THE SECURITY OF RSA — VULNERABILITIES CAUSED BY LOW-ENTROPY RANDOM NUMBERS

- Please review Section 10.8 of Lecture 10 to appreciate the significance of "Low Entropy" in the title of this section. [As explained there, the entropy of a random number generator is at its highest if all numbers are *equally likely* to be produced within the range of numbers that the output is designed for. For example, if a CSPRNG can produce 512-bit random numbers with equal probability, its entropy is at its maximum and it equals 512 bits. However, should the probabilities associated with the output random numbers be nonuniform, the entropy will be less than 512. The greater the nonuniformity of this probability distribution, the smaller the entropy. The entropy is zero for deterministic output.]

- Consider the following mind-boggling fact: **If an attacker can get hold of a pair of RSA moduli, N_1 and N_2, that share a factor, the attacker will be able to figure out the other factor for both moduli with hardly any work.** Obviously, once the attacker has acquired both factors of a modulus, the attacker can quickly calculate the private key that goes with the public key associated with the modulus. This exploit, if successfully carried out, immediately yields the private keys that go with the public keys that contain the N_1 and N_2 moduli.

- To see why that is the case, let's say that p is the common factor of the two moduli N_1 and N_2. That makes p the GCD of N_1 and N_2. Now let's denote the other factor in N_1 by q_1 and in N_2 by q_2. You already know from Lecture 5 that Euclid's recursion makes the calculation of the GCD of any two numbers extremely fast. [Using Euclid's algorithm, the GCD of two 1024-bit integers on a routine desktop can be computed in just a few microseconds using the Gnu Multiple Precision (GMP) library. More theoretically speaking, the computational complexity of Euclid's GCD algorithm is $O(n^2)$ for n bit numbers.] Therefore, the common factor p of two moduli — assuming they have a common factor — can be calculated almost instantaneously with ordinary hardware. And once you have p, the factors q_1 and q_2 are obtainable by simple integer division, which is also fast.

- You might ask: Is it really likely that an attacker would find a pair of RSA moduli that share a common factor? The answer is: It is very, very likely today. Read on for why.

- Modern port and vulnerability scanners of the sort I'll present in Lecture 23 can carry out a full SSL/TLS and SSH handshake and fetch the certificates used by the TLS/SSL hosts (these are typically HTTPS web servers) and the host keys used by the SSHD servers at a fairly rapid rate.

- In a truly landmark investigation by Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman that was presented at the 2012 USENIX Security Symposium, the authors re-

ported harvesting over 5 million TLS/SSL certificates and around 4 million RSA-based SSH host keys through scans that lasted no more than a couple of days. As you can see, these authors were able to harvest a very large number of RSA moduli in a rather short time. Subsequently they set out to find the factors that any of the moduli shared with any of the other moduli. [While the GCD of a pair of numbers can be computed very fast on a run-of-the-mill machine, it would still take a very long time to do pairwise computation for all the numbers in a set that contains a few million numbers. For further speedup, Heninger et al. used a method proposed by Daniel Bernstein. In this method, you start with calculating the product of all the moduli, multiplying two moduli at a time in what's called a *product tree*, and then reduce the product with respect to the pairwise products of the squares of the moduli in what's known as the *remainder tree*. This approach, applied to over 11 million RSA moduli from the TLS/SSL and SSH datasets, yielded the p factors in under 6 hours on a multicore PC class machine with 32 GB of RAM.] The title of the publication by Heninger et al. is "*Mining your Ps and Qs: Detection of Widspread Weak Keys in Network Devices*".

- In this manner, Heninger et al. were able to compute the private keys for 0.50% of the TLS/SSL servers (the HTTPS web servers) and for 0.03% of the SSH servers.

- The upshot of the investigation reported by Heninger et al. is that your random number generator must have high enough entropy so that each modulus is unique vis-a-vis the moduli used by any other communication device any place on the face of this earth!

- While the prescription stated above is followed for the most part by most computers of the sort we use everyday, that's not necessarily the case for a large number of what are known as headless communication devices in the internet. By headless devices we mean routers, firewalls, sever management cards, etc. As observed by Heninger et al., a very large number of such headless devices use software entropy sources for the random bytes they need as candidates for the prime numbers and the most commonly used software entropy source is `/dev/urandom` that supplies pseudorandom bytes through non-blocking reads.

- The problem with `/dev/urandom` arises at boot time when such a software entropy source is least equipped to supply high-entropy random bytes and this happens exactly when the network interface has a need to create keys for communicating with other hosts.

- See Section 10.9.4 of Lecture 10 for further information on `/dev/urandom` and its relationship to `/dev/random`.

12.9: THE SECURITY OF RSA — THE MATHEMATICAL ATTACK

- Assuming that the security issues brought up in the previous three sections are not relevant in a given application context, the security of RSA depends critically on the fact that whereas it is easy to multiply two large primes to construct a modulus, the inverse operation of factoring the modulus into its prime factors can be extremely difficult — difficult until you solve the integer factorization problem for the sizes of the numbers involved. [Functions that are easy to compute in one direction but that cannot be easily inverted without special information are known as *trapdoor functions*.] Trying to break RSA by developing an integer factorization solution for the moduli involved is known as a **mathematical attack**.

- That is, a mathematical attack on RSA 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 private 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. A detailed presentation of integer factorization is beyond the scope of this lecture. 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 [Be aware that while the methods listed below can factorize large numbers, for very large numbers of the sort used these days in RSA cryptography, you have to custom design the algorithms for each attack. Customization generally consists of making various

conjectures about the modulo properties of the factors and using the conjectures to speed up the search for the factors.]:

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 $m = p_1 p_2 p_3 \ldots p_r$ 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 follows 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 $\sqrt{n}$.

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 modulo n. In Fermat's method, we search for a single number x so that x^2 mod n is a square. But such x's are difficult to find. With quadratic sieve, we compute x^2 mod n for many x's and then find a subset of these whose product is a square.

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 randomly chosen numbers a and b so that $a \equiv b \ (mod\ d)$. Obviously, for such a and b, $a - b \equiv 0 \ (mod\ n)$, implying $a - b = kd$ for some k, further implying that d must also be a divisor of the difference $a-b$. That is, $d|(a-b)$. Since, by assumption, $d|n$, it must be the case that $gcd(a-b, n)$ is a multiple of d. We can now set d to the answer returned by gcd, assuming that this answer is greater than 1. Once we find such a factor of n, we can divide n by the factor and repeat the algorithm on the resulting smaller integer.

- This suggests the following approach to finding a factor of n: (1) Randomly choose two numbers $a, b \leq \sqrt{n}$; (2) Find $gcd(a - b, n)$; (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. [A discerning reader might say that since we know nothing about the factor d of n and since we are essentially shooting in the dark when making guesses for a and b, why should we expect a performance any better than making random guesses for the factors of n up to the square-root of n. That may well be true in general, but the beauty of searching for the factors via the differences $a - b$ is that it generalizes to the main feature of the Pollard-ρ algorithm **in which the sequence of integers you choose for b grows twice as fast as the sequence of integers you choose for a.** It is this feature that makes for a much more efficient way to look for the factors of n. This feature is implemented in lines (E10), (E11), and (E12) of the code shown at the end of this section. As was demonstrated by Pollard, letting b grow twice as fast as a in $gcd(a - b, n)$ makes for fast detection of cycles, these being two different numbers a and b that are congruent modulo some integer $d < n$.]

– In the code shown at the end of this section, the simple procedure laid out above is called `pollard_rho_simple()`; its implementation is shown in lines (D1) through (D15) of the code. We start the calculation by choosing random numbers for a and b, and computing $gcd(a - b, n)$. Assuming that this *gcd* equals 1, we now generate another candidate for b in the loop in lines (D9) through (D14). For each new candidate generated for b, its difference must be computed from all the previously generated random numbers and the *gcd* of the differences computed. In general, for the k^{th} random number selected for b, 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 function f that maps a set to itself through the equivalence of the remainders modulo n. Let's express the sequence of numbers generated through such a function by $x_{i+1} = f(x_i)\ mod\ n$. 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 $x_i \equiv x_j\ (mod\ d)$, then obviously $f(x_i) \equiv f(x_j)\ (mod\ d)$. 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 factor of n.

– The Pollard-ρ algorithm uses a function $f()$ to generate two sequence x_i and y_i, with the latter growing twice as fast as the former — see lines (E10), (E11), and (E12) of the code for an illustration of this idea. 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.

– The most commonly used function $f(x)$ is the polynomial $f(x) = x^2 + c \ \ mod \ \ n$ with the constant c not allowed to take the values 0 and -2. The code shown in lines (E4) through (E15) constitutes an implementation of this polynomial.

– Some parts of the implementation of the overall integer factorization algorithm shown below should already be familiar to you. The calculation of *gcd* in lines in (B1) through (B4) is from Section 5.4.5 of Lecture 5. The Miller-Rabin based primality testing code in lines (C1) through (C22) is from Section 11.5.5 of Lecture 11.

```
#!/usr/bin/env python

##  Factorize.py
##  Author: Avi Kak
##  Date: February 26, 2011
##  Modified: Febrary 25, 2012

##  Uncomment line (F9) and comment out line (F10) if you want to see the results
##  with the simpler form of the Pollard-Rho algorithm.

import random
import sys

def factorize(n):                                                    #(F1)
    prime_factors = []                                               #(F2)
    factors = [n]                                                    #(F3)
    while len(factors) != 0:                                         #(F4)
        p = factors.pop()                                            #(F5)
        if test_integer_for_prime(p):                                #(F6)
            prime_factors.append(p)                                  #(F7)
            continue                                                 #(F8)
```

```
#        d = pollard_rho_simple(p)                                          #(F9)
         d = pollard_rho_strong(p)                                           #(F10)
         if d == p:                                                         #(F11)
             factors.append(d)                                              #(F12)
         else:                                                              #(F13)
             factors.append(d)                                              #(F14)
             factors.append(p//d)                                           #(F15)
    return prime_factors                                                    #(F16)

def test_integer_for_prime(p):                                              #(P1)
    probes = [2,3,5,7,11,13,17]                                             #(P2)
    for a in probes:                                                        #(P3)
        if a == p: return 1                                                 #(P4)
    if any([p % a == 0 for a in probes]): return 0                          #(P5)
    k, q = 0, p-1                                                           #(P6)
    while not q&1:                                                          #(P7)
        q >>= 1                                                             #(P8)
        k += 1                                                              #(P9)
    for a in probes:                                                        #(P10)
        a_raised_to_q = pow(a, q, p)                                        #(P11)
        if a_raised_to_q == 1 or a_raised_to_q == p-1: continue             #(P12)
        a_raised_to_jq = a_raised_to_q                                      #(P13)
        primeflag = 0                                                       #(P14)
        for j in range(k-1):                                                #(P15)
            a_raised_to_jq = pow(a_raised_to_jq, 2, p)                      #(P16)
            if a_raised_to_jq == p-1:                                       #(P17)
                primeflag = 1                                               #(P18)
                break                                                       #(P19)
        if not primeflag: return 0                                          #(P20)
    probability_of_prime = 1 - 1.0/(4 ** len(probes))                       #(P21)
    return probability_of_prime                                             #(P22)

def pollard_rho_simple(p):                                                  #(Q1)
    probes = [2,3,5,7,11,13,17]                                             #(Q2)
    for a in probes:                                                        #(Q3)
        if p%a == 0: return a                                               #(Q4)
    d = 1                                                                   #(Q5)
    a = random.randint(2,p)                                                 #(Q6)
    random_num = []                                                         #(Q7)
    random_num.append( a )                                                  #(Q8)
    while d==1:                                                             #(Q9)
        b = random.randint(2,p)                                             #(Q10)
        for a in random_num[:]:                                             #(Q11)
            d = gcd( a-b, p )                                               #(Q12)
            if d > 1: break                                                 #(Q13)
        random_num.append(b)                                                #(Q14)
    return d                                                                #(Q15)

def pollard_rho_strong(p):                                                  #(R1)
    probes = [2,3,5,7,11,13,17]                                             #(R2)
    for a in probes:                                                        #(R3)
        if p%a == 0: return a                                               #(R4)
    d = 1                                                                   #(R5)
    a = random.randint(2,p)                                                 #(R6)
    c = random.randint(2,p)                                                 #(R7)
```

```
    b = a                                                          #(R8)
    while d==1:                                                    #(R9)
        a = (a * a + c) % p                                        #(R10)
        b = (b * b + c) % p                                        #(R11)
        b = (b * b + c) % p                                        #(R12)
        d = gcd( a-b, p)                                           #(R13)
        if d > 1: break                                            #(R14)
    return d                                                       #(R15)

def gcd(a,b):                                                      #(S1)
    while b:                                                       #(S2)
        a, b = b, a%b                                              #(S3)
    return a                                                       #(D4)

if __name__ == '__main__':                                         #(A1)

    if len( sys.argv ) != 2:                                       #(A2)
        sys.exit( "Call syntax:  Factorize.py  number" )           #(A3)
    p = int( sys.argv[1] )                                         #(A4)
    factors = factorize(p)                                         #(A5)
    print("\nFactors of %d:" % p)                                  #(A6)
    for num in sorted(set(factors)):                               #(A7)
        print("%s %d ^ %d" % ("    ", num, factors.count(num)))    #(A8)
```

– Let's try the program on what is known as the sixth Fermat number [The n^{th} Fermat number is given by $2^{2^n}+1$. So the sixth Fermat number is $2^{64}+1$.]:

```
Factorize.py 18446744073709551617
```

The factors returned are:

```
274177 ^ 1
67280421310721 ^ 1
```

In the answer shown what comes after ^ is the power of the factor in the number. You can check the correctness of the answer by entering the number in the search window at the `http://www.factordb.com` web site. You will also notice that you will get the same in only another blink of the eye if you comment out line (F10) and uncomment line (F9),

which basically amounts to making a random guess for the factors.

– That we get the same performance regardless of whether we use the statement in line (F9) or the statement in line (F10) happens because the number we asked `Factorize.py` to factorize above was easy. As we will mention in Section 12.9, factorization becomes harder when a composite is a product of two primes of roughly the same size. For that reason, a tougher problem would be to factorize the known semiprime 10023859281455311421. Now, unless you are willing to wait for a long time, you will have no choice but to use the statement in line (F10). Using the statement in line (F10), the factors returned for this number are:

```
1308520867 ^ 1
7660450463 ^ 1
```

– For another example, when we call `Factorize.py` on the number shown below, using the statement in line (F10) for the Pollard-ρ algorithm

```
115792089237316195423570985008687907853269984665640564039
```

the factors returned are:

```
23 ^ 1
41 ^ 1
```

```
149 ^ 1
40076041 ^ 1
713526132967 ^ 1
9962712838657 ^ 1
289273479972424951 ^ 1
```

– Shown next is a Perl version of the script for factorization. Since arbitrarily sized integers are not native to Perl, this script can only handle integers that can be accommodated in 4 bytes that Perl uses for storing unsigned integers. [As mentioned previously in Lecture 11, in Perl you must import the `Math::BigInt` package for arbitrarily large integers. Later in this section I will show an implementation of the Pollard-Rho factorization algorithm that is based on the `Math::BigInt` representation of large integers.]

```
#!/usr/bin/env perl

##  Factorize.pl
##  Author: Avi Kak
##  Date: February 19, 2016

##  Uncomment line (F12) and comment out line (F13) if you want to see the results
##  with the simpler form of the Pollard-Rho algorithm.

use strict;
use warnings;

die "\nUsage:   $0  <integer> \n" unless @ARGV == 1;                          #(A1)
my $p = shift @ARGV;                                                          #(A2)

die "Your number is too large for factorization by this script. " .
    "Instead, try the script 'FactorizeWithBigInt.pl'\n"
    if $p > 0x7f_ff_ff_ff;                                                    #(A3)

my @factors = @{factorize($p)};                                               #(A4)
my %how_many_of_each;                                                         #(A5)
map {$how_many_of_each{$_}++} @factors;                                       #(A6)
print "\nFactors of $p:\n";                                                   #(A7)
foreach my $factor (sort {$a <=> $b} keys %how_many_of_each) {                #(A8)
    print "    $factor ^ $how_many_of_each{$factor}\n";                        #(A9)
}
```

```
sub factorize {                                                          #(F1)
    my $n = shift;                                                       #(F2)
    my @prime_factors = ();                                              #(F3)
    my @factors;                                                         #(F4)
    push @factors, $n;                                                   #(F5)
    while (@factors > 0) {                                               #(F6)
        my $p = pop @factors;                                            #(F8)
        if (test_integer_for_prime($p)) {                                #(F9)
            push @prime_factors, $p;                                     #(F10)
            next;                                                        #(F11)
        }
#        my $d = pollard_rho_simple($p);                                 #(F12)
        my $d = pollard_rho_strong($p);                                  #(F13)
        if ($d == $p) {                                                  #(F14)
            push @factors, $d;                                           #(F15)
        } else {
            push @factors, $d;                                           #(F16)
            push @factors, int($p / $d);                                 #(F17)
        }
    }
    return \@prime_factors;                                              #(F18)
}

sub test_integer_for_prime {                                             #(P1)
    my $p = shift;                                                       #(P2)
    my @probes = qw[ 2 3 5 7 11 13 17 ];                                 #(P3)
    foreach my $a (@probes) {                                            #(P4)
        return 1 if $a == $p;                                            #(P5)
    }
    my ($k, $q) = (0, $p - 1);                                           #(P6)
    while (! ($q & 1)) {                                                 #(P7)
        $q >>= 1;                                                        #(P8)
        $k += 1;                                                         #(P9)
    }
    my ($a_raised_to_q, $a_raised_to_jq, $primeflag);                    #(P10)
    foreach my $a (@probes) {                                            #(P11)
        my ($base,$exponent) = ($a,$q);                                  #(P12)
        my $a_raised_to_q = 1;                                           #(P13)
        while ((int($exponent) > 0)) {                                   #(P14)
            $a_raised_to_q = ($a_raised_to_q * $base) % $p
                                          if int($exponent) & 1;         #(P15)
            $exponent = $exponent >> 1;                                  #(P16)
            $base = ($base * $base) % $p;                                #(P17)
        }
        next if $a_raised_to_q == 1;                                     #(P18)
        next if ($a_raised_to_q == ($p - 1)) && ($k > 0);                #(P19)
        $a_raised_to_jq = $a_raised_to_q;                                #(P20)
        $primeflag = 0;                                                  #(P21)
        foreach my $j (0 .. $k - 2) {                                    #(P22)
            $a_raised_to_jq = ($a_raised_to_jq ** 2) % $p;               #(P23)
            if ($a_raised_to_jq == $p-1) {                               #(P24)
                $primeflag = 1;                                          #(P25)
                last;                                                    #(P26)
            }
```

```
        }
        return 0 if ! $primeflag;                                   #(P27)
    }
    my $probability_of_prime = 1 - 1.0/(4 ** scalar(@probes));      #(P28)
    return $probability_of_prime;                                   #(P29)
}

sub pollard_rho_simple {                                            #(Q1)
    my $p = shift;                                                  #(Q2)
    my @probes = qw[ 2 3 5 7 11 13 17 ];                            #(Q3)
    foreach my $a (@probes) {                                       #(Q4)
        return $a if $p % $a == 0;                                  #(Q5)
    }
    my $d = 1;                                                      #(Q6)
    my $a = 2 + int(rand($p));                                      #(Q7)
    my @random_num = ($a);                                          #(Q8)
    while ($d == 1) {                                               #(Q9)
        my $b = 2 + int(rand($p));                                  #(Q10)
        foreach my $a (@random_num) {                               #(Q11)
            $d = gcd($a - $b, $p);                                  #(Q12)
            last if $d > 1;                                         #(Q13)
        }
        push @random_num, $b;                                       #(Q14)
    }
    return $d;                                                      #(Q15)
}

sub pollard_rho_strong {                                            #(R1)
    my $p = shift;                                                  #(R2)
    my @probes = qw[ 2 3 5 7 11 13 17 ];                            #(R3)
    foreach my $a (@probes) {                                       #(R4)
        return $a if $p % $a == 0;
    }
    my $d = 1;                                                      #(R5)
    my $a = 2 + int(rand($p));                                      #(R6)
    my $c = 2 + int(rand($p));                                      #(R6)
    my $b = $a;                                                     #(R7)
    while ($d == 1) {                                               #(R8)
        $a = ($a * $a + $c) % $p;                                   #(R9)
        $b = ($b * $b + $c) % $p;                                   #(R10)
        $b = ($b * $b + $c) % $p;                                   #(R11)
        $d = gcd($a - $b, $p);                                      #(R12)
        last if $d > 1;                                             #(R13)
    }
    return $d;                                                      #(R14)
}

sub gcd {                                                           #(S1)
    my ($a,$b) = @_;                                                #(S2)
    while ($b) {                                                    #(S3)
        ($a,$b) = ($b, $a % $b);                                    #(S4)
    }
    return $a;                                                      #(S5)
}
```

– If you call the above script with the argument shown below

```
Factorize.pl 1844674407
```

the script will return the answer shown below:

```
Factors of 1844674407:

    3 ^ 2
    204963823 ^ 1
```

– On the other hand, if you call this script for a large integer, as in

```
Factorize.pl 18446744073709551617
```

the script will come back with the error message:

```
Your number is too large for factorization by this script.
Instead, try the script 'FactorizeWithBigInt.pl'
```

This error message is triggered by the statement in line (A3) of the script where we compare the user-supplied integer with the largest integer that can be stored in 4 bytes.

– That brings me to a `Math::BigInt` variant of the Perl script shown above **in order to deal with arbitrarily large integers**. Although the `Math::BigInt` library is now a part of the Perl core, it is somewhat awkward to use unlike what is the case with Python where transitioning to the big-number representation happens under the hood. When using `Math::BigInt`, all operations — addition, multiplication, exponentiation, modular multiplication, modular exponentiation, and so on — require calls to this module's API.

– In the script that is shown below, we immediately convert the user supplied integer as a command-line argument into its `Math::BigInt` representation in line (A5). As stated in my introduction to the Pollard-Rho algorithm, the algorithm requires randomly generated integers whose differences, if found coprime to the integer that is being factorized, then become the factors you are looking for. For the script `Factorize.pl` shown above, we could call on Perl's native `rand()` function to supply us with those random numbers. [**Since we upper-bounded the integers to be factorized in that script to the largest that can be stored in 4 bytes and since that is also the upper bound on the numbers that `rand()` can return, the behavior of `rand()` is consistent with what the script `Factorize.pl` is capable of.**] However, when you are dealing with arbitrarily large integers, you need a random number generator commensurate with such numbers. That is the reason for importing the `Math::BigInt::Random::OO` in line (A2).

```
#!/usr/bin/env perl

##  FactorizeWithBigInt.pl
##  Author: Avi Kak
##  Date: February 21, 2016

##  Uncomment line (F13) and comment out line (F14) if you want to see the results
##  with the simpler form of the Pollard-Rho algorithm.

use strict;
use warnings;
use Math::BigInt;
use Math::BigInt::Random::OO;

############################  class FactorizeWithBigInt  ############################
package FactorizeWithBigInt;

sub new {                                                                    #(A1)
    my ($class, $num) = @_;                                                  #(A2)
```

```
    bless {                                                              #(A3)
        num  =>  int($num),                                              #(A4)
    }, $class;                                                           #(A5)
}

sub factorize {                                                          #(F1)
    my $self = shift;                                                    #(F2)
    my $n = $self->{num};                                                #(F3)
    my @prime_factors = ();                                              #(F4)
    my @factors;                                                         #(F5)
    push @factors, $n;                                                   #(F6)
    while (@factors > 0) {                                               #(F7)
        my $p = pop @factors;                                            #(F8)
        if ($self->test_integer_for_prime($p)) {                         #(F9)
            my $pnum = $p->numify();                                     #(F10)
            push @prime_factors, $p;                                     #(F11)
            next;                                                        #(F12)
        }
#       my $d = $self->pollard_rho_simple($p);                           #(F13)
        my $d = $self->pollard_rho_strong($p);                           #(F14)
        if ($d->copy()->bacmp($p->copy()) == 0) {                        #(F15)
            push @factors, $d;                                           #(F16)
        } else {                                                         #(F17)
            push @factors, $d;                                           #(F18)
            my $div = $p->copy()->bdiv($d->copy());                      #(F19)
            push @factors, $div;                                         #(F20)
        }
    }
    return \@prime_factors;                                              #(F21)
}

sub test_integer_for_prime {                                             #(P1)
    my $self = shift;                                                    #(P2)
    my $p = shift;                                                       #(P3)
    return 0 if $p->is_one();                                            #(P4)
    my @probes = qw[ 2 3 5 7 11 13 17 ];                                 #(P5)
    foreach my $a (@probes) {                                            #(P6)
        $a = Math::BigInt->new("$a");                                    #(P7)
        return 1 if $p->bcmp($a) == 0;                                   #(P8)
        return 0 if $p->copy()->bmod($a)->is_zero();                     #(P9)
    }
    my ($k, $q) = (0, $p->copy()->bdec());                               #(P10)
    while (! $q->copy()->band( Math::BigInt->new("1"))) {                #(P11)
        $q->brsft( 1 );                                                  #(P12)
        $k += 1;                                                         #(P13)
    }
    my ($a_raised_to_q, $a_raised_to_jq, $primeflag);                    #(P14)
    foreach my $a (@probes) {                                            #(P15)
        my $abig = Math::BigInt->new("$a");                              #(P16)
        my $a_raised_to_q = $abig->bmodpow($q, $p);                      #(P17)
        next if $a_raised_to_q->is_one();                                #(P18)
        my $pdec = $p->copy()->bdec();                                   #(P19)
        next if ($a_raised_to_q->bcmp($pdec) == 0) && ($k > 0);          #(P20)
        $a_raised_to_jq = $a_raised_to_q;                                #(P21)
        $primeflag = 0;                                                  #(P22)
```

```
        foreach my $j (0 .. $k - 2) {                                          #(P23)
            my $two = Math::BigInt->new("2");                                  #(P24)
            $a_raised_to_jq = $a_raised_to_jq->copy()->bmodpow($two, $p);      #(P25)
            if ($a_raised_to_jq->bcmp( $p->copy()->bdec() ) == 0 ) {           #(P26)
                $primeflag = 1;                                                #(P27)
                last;                                                          #(P28)
            }
        }
        return 0 if ! $primeflag;                                              #(P29)
    }
    my $probability_of_prime = 1 - 1.0/(4 ** scalar(@probes));                 #(P30)
    return $probability_of_prime;                                              #(P31)
}

sub pollard_rho_simple {                                                       #(Q1)
    my $self = shift;                                                          #(Q2)
    my $p = shift;                                                             #(Q3)
    my @probes = qw[ 2 3 5 7 11 13 17 ];                                       #(Q4)
    foreach my $a (@probes) {                                                  #(Q5)
        my $abig = Math::BigInt->new("$a");                                    #(Q6)
        return $abig if $p->copy()->bmod($abig)->is_zero();                    #(Q7)
    }
    my $d = Math::BigInt->bone();                                              #(Q8)
    my $randgen = Math::BigInt::Random::OO->new( max => $p );                  #(Q9)
    my $a = Math::BigInt->new();                                               #(Q10)
    unless ($a->numify() >= 2) {                                               #(Q11)
        $a =  $randgen->generate(1);                                           #(Q12)
    }
    my @random_num = ($a);                                                     #(Q13)
    while ($d->is_one()) {                                                     #(Q14)
        my $b = Math::BigInt->new();                                           #(Q15)
        unless ($b->numify() >= 2) {                                           #(Q16)
            $b =  $randgen->generate(1);                                       #(Q17)
        }
        foreach my $a (@random_num) {                                          #(Q18)
            $d = Math::BigInt::bgcd($a->copy()->bsub($b),$p);                  #(Q19)
            last if $d->bacmp(Math::BigInt->bone()) > 0;                       #(Q20)
        }
        push @random_num, $b;                                                  #(Q21)
    }
    return $d;                                                                 #(Q22)
}

sub pollard_rho_strong {                                                       #(R1)
    my $self = shift;                                                          #(R2)
    my $p = shift;                                                             #(R3)
    my @probes = qw[ 2 3 5 7 11 13 17 ];                                       #(R4)
    foreach my $a (@probes) {                                                  #(R5)
        my $abig = Math::BigInt->new("$a");                                    #(R6)
        return $abig if $p->copy()->bmod($abig)->is_zero();                    #(R7)
    }
    my $d = Math::BigInt->bone();                                              #(R8)
    my $randgen = Math::BigInt::Random::OO->new( max => $p );                  #(R9)
    my $a = Math::BigInt->new();                                               #(R10)
    unless ($a->numify() >= 2) {                                               #(R11)
```

```
        $a =  $randgen->generate(1);                                    #(R12)
    }
    $randgen = Math::BigInt::Random::OO->new( max => $p );              #(R13)
    my $c = Math::BigInt->new();                                        #(R14)
    unless ($c->numify() >= 2) {                                        #(R15)
        $c =  $randgen->generate(1);                                    #(R16)
    }
    my $b = $a->copy();                                                 #(R17)
    while ($d->is_one()) {                                              #(R18)
        $a->bmuladd($a->copy(), $c->copy())->bmod($p);                  #(R19)
        $b->bmuladd($b->copy(), $c->copy())->bmod($p);                  #(R20)
        $b->bmuladd($b->copy(), $c->copy())->bmod($p);                  #(R21)
        $d = Math::BigInt::bgcd( $a->copy()->bsub($b), $p );            #(R22)
        last if $d->bacmp(Math::BigInt->bone()) > 0;                    #(R23)
    }
    return $d;                                                          #(R24)
}

################################    main    #####################################
package main;

unless (@ARGV) {                                                        #(M1)
    1;                                                                  #(M2)
} else {                                                                #(M3)
    my $p = shift @ARGV;                                                #(M2)
    $p = Math::BigInt->new( "$p" );                                     #(M3)
    my $factorizer = FactorizeWithBigInt->new($p);                      #(M4)
    my @factors = @{$factorizer->factorize()};                          #(M5)
    my %how_many_of_each;                                               #(M6)
    map {$how_many_of_each{$_}++} @factors;                             #(M7)
    print "\nFactors of $p:\n";                                         #(M8)
    foreach my $factor (sort {$a <=> $b} keys %how_many_of_each) {      #(M9)
        print "    $factor ^ $how_many_of_each{$factor}\n";              #(M10)
    }
}
```

– To demonstrate the script shown above in action, if you call

`FactorizeWithBigInt.pl 123456789123456789123456789123456789123456789123456789`

the script returns the following factorization:

```
Factors of 123456789123456789123456789123456789123456789123456789:

    3 ^ 3
    7 ^ 1
    11 ^ 1
    13 ^ 1
```

```
19 ^ 1
757 ^ 1
3607 ^ 1
3803 ^ 1
52579 ^ 1
70541929 ^ 1
14175966169 ^ 1
440334654777631 ^ 1
```

– The Pollard-ρ algorithm is based on John Pollard's article "*A Monte Carlo Method for Factorization*," BIT, pp. 331-334. A more efficient variation on Pollard's method was published by Richard Brent: "*An Improved Monte Carlo Factorization Algorithm*," in the same journal in 1980.

12.10: 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 =
  27997833911221327870829467638722601621070446786 95
  54285375600099293261284001076093456710529553608 56
  06182235191095136578863710595448200657677509858 05
  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
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
3107418240490043721350750035888567930037346022 8
427275457201619488232064405180815045563468296 71
723286782437916272838033415471073108501919548 52
900733772482278352574238645401469173660247765 23
46609
```

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

- RSA-768, shown below, was factored in December 2009 by T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thome, J Bos, P. Gaudry, A. Kruppa, P. Montgommery, D. Osvik, H. te Riele, A. Timofeev, and P. Zimmerman. This is the largest modulus

that has been factored to date. This factorization resulted from a multi-year effort in distributed computing.

```
Name:          RSA-768
Prize:         $50000      (retracted)
Digits:        232
Digit Sum:     1018
123018668453011775513049495838496272077285356959533479219732245215172640050726365751874520219978646938995647494277406384592519255732630345373154826850791702612214291346167042921431160222124047927473779408066535141959745985690214341 3
```

12.10.1: The Old RSA Factoring Challenge: Numbers Not Yet Factored

```
Name:          RSA-896
Prize:         $75000       (retracted)
Digits:        270
Digit Sum:     1222
412023436986659543855531365332575948179811699 84
432798284545562643387644556524842619809887042 31
618418792614202471888694925609317763750334211 30
982397485150944909106910269861031862704114880 86
697056490290365365886743373172081310410519086 42
547932826013912576240339463732693 91
```

```
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
655137144791393234716956698806 9
```

```
Name:         RSA-2048
Prize:        $200000      (retracted)
Digits:       617
Digit Sum:    2738
251959084756578934940271832400483985714292821262 0
403202777713783604366202070759555626401852588078 4
406918290641249515082189298559149176184502808489 1
200728449926873928072877767359714183472702618963 7
501497182469116507761337985909570009733045974880 8
428401797429100642458691817195118746121515172654 6
322822168699875491824224336372590851418654620435 7
679842338718477444792073993423658482382428119816 3
815010674810451660377306056201619676256133844143 6
038339044149526344321901146575444541784240209246 1
651572335077870774981712577246796292638635637328 9
912154831438167899885040445364023527381951378636 5
643912120103971228221207203 57
```

12.11: THE RSA ALGORITHM: SOME OPERATIONAL DETAILS

- The main goal of this section is to explain how the public and the private keys — which theoretically speaking are merely pairs of integers $[n, e]$ and $[n, d]$, respectively, — are actually represented in the memory of a computer. As you will see, the representation used depends on the protocol. The key representation in the SSH protocol is, for example, very different from the key representation in the TLS/SSL protocol. However, before getting to the key representation issues, what follows are some very important general comments about the RSA algorithm.

- The size of the key in the RSA algorithm typically refers to the size of the modulus integer in bits. *In that sense, the phrase "key size" in the context of RSA is a bit of a misnomer.* As you now know, the actual keys in RSA are the public key $[n, e]$ and the private key $[n, d]$. In addition to depending on the size of the modulus, the key sizes obviously depend on the values chosen for e and d.

- 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 (and that translates into a 256-character hex string if we had to print out the 128 bytes for 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
263232007163440584083618711941939199 99
```

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 ..... ff7fffda5f
```

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 (meaning the size of the modulus) 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 eight. 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 they can be

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 rsa
```

The public and the private keys 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 with a password.** [You can also replace 'rsa' with 'dsa' in the above call. The flag 'dsa' refers to the Digital Signature Algorithm that typically uses the ElGamal protocol (see Section 13.6 of Lecture 13 for ElGamal) for generating the key pairs. 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_rsa` of the home account in your laptop. The public key will be deposited in a file that will be named `.ssh/id_rsa.pub`. **Execute the above command on your laptop and then copy the public key that is generated into the file `.ssh/authorized_keys` in your own account of the remote machine to which you want SSH access without the bother of having to log in with a password.**]

- I will now show an example of a private key generated by the command "`ssh-keygen -t rsa -f mykeyfile`" where the option "`-f`" causes the private key to be written out to the file `mykeyfile` and the corresponding public key to be written out to the file `mykeyfile.pub`, both in the directory in which you execute the command. [As mentioned in the previous bullet, without the "`-f`" option, the private key would be written out to the file `.ssh/id_rsa` and the public key to

the file `.ssh/id_rsa.pub`, both in your home directory.] The private key shown below is Base64 encoded:

```
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAt/lgFURxL351WHaOhLzRHeWBfgt2vomE4nEl08CDtscn0yl7
UMj08B0T896QGuSPvMhFu/I5aGCM4SS7W4wPMxYlnpohKQNCOFFClLxh8AusF5NC
sZSDoRpsG02EBFtUVvzRoQcBpDQoEbbrZ60/cZEve/59z6tIexBbw19LlWAnWqO2
Q33p12mOmuNlcP+StE55Dlz828cCSStyiOtru+Z3DyMhc8D7uBDbrsXchV2dE8tf
6zpIorheFwO4iJ+0YQSaKNc7GpPJEXayBslpVeb5Dp1uZcSDWhJD5dkvpgngmGfJ
FjMuz2FhpCG7msojn1OMOG0JkrpMjMjDnRs7KwIDAQABAoIBAQCfywcskbzprqIH
yznwivvrRruZmiVyS4nTEMn1SUJeEE9D9sS4HOO4BtmcuRRbd2htZVpEa5h3U10p
9LiTRGyzR6o3PtJMiBsNRW9aNuGuGMVJ2MtV6JQH6yY49LQVAKqZv4/omGhRkke3
UzTWYUE4yA0BWsM2DCSxVz0MzgmdXGJ/s3ARcj2hetfN4FUjBg7TenSoIcb+2XA+
0B6l0sr5R3DvStl/erqwlywkA+IeX2Bvqn31yHwoSYqcxZ/eKYxn5BnRTjtXrnNE
W9LBb6Au4Ch1yzkpgCos/kGVb2nzQIKvlFbISD3FhL9hNrL9u2Wnlm9ee9umgLC8
qTiK6QFBAoGBAPM4q700n7nOFtB/zoZn9bO63dfKdZ/QuOudrgJjhygRVXIQlPyi
ZzRjdTDtfTY9DlMk6GqTpeTgVjKARE2C72Uip0Ba5s4/8lhdlvlgJjJ9w1S8LcXm
fySVHEd6vJSZ9CiaA75jybwYIlETitP3d92/nPWY7iOi67ctXaREPkphAoGBAMGj
1BzH2qcEsX/flOMMv3ou0lnq2cfw9tdw8B4DFrfeBdO2z4yOO18hi9sPSO/P7Zzb
1hfgK3RRxJJcHYVwuibObvj4n3LHAV0LuCYlRoEIt7IzbuTTQbOsWqxtHz5sGH+T
G4Vpe8iLHdEMsA2Gm911xjwURYshTFqfdKtX1KkLAoGBANaPScGRqM4qpimsda5Q
C4pP8OAqdHVVlawU36qvzk1kbTJAuo3bXpvymTFecDTmy7pBNt69/XzZAnFugdK3
DST02wKErlOISev2M/cuAMRN+YDIuMB6Q/Mrr1THS5Dz91XR+Dd+pDpQOAW57aBs
EMwH+xkVng7F7JcdaBw/L4xBAoGAM4wPHRI+rJNdRPMZyU9NcZMhP/p6uvT0YOmZ
ogOkepHJ71AM9BewymQ9vLTW76/kSwtidLyiFLbnoyaOXUVi6I2vkOtuVrmPLVu/
S7hEinjtnax/ar6qENBi2t+5n35bDyrz+pHX98zAxTOhchhRSaTefoPO93iHwOAN
yMb6v30CgYALPSHHlRCgWSd/0rqykPwzGACobm9a4Pth8YtuOSdHHfnkf2CoU3Go
y6gDXV/0k0NL9HF33A9mgXa3H0uj5hYLswRcWwmlWex6ytavFbsRwykKHRHMp9w5
iLz2nnDdZ9DwhBtsSjq948TZYoD9mGg/PZabLjBsicTtjSvCIrP1FQ==
-----END RSA PRIVATE KEY-----
```

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

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC3+WAVRHEvfnVYdo6EvNEd5YF+C3
a+iYTicSXTwIO2xyfTKXtQyPTwHRPz3pAa5I+8yEW78jloYIzhJLtbjA8zFiWemiEp
A0I4UUKUvGHwC6wXk0KxlIOhGmwbTYQEW1RW/NGhBwGkNCgRtutnrT9xkS97/n3Pq0
h7EFvDX0uVYCdao7ZDfenXaY6a42Vw/5K0TnkOXPzbxwJJK3KI62u75ncPIyFzwPu4
ENuuxdyFXZ0Ty1/rOkiiuF4XA7iIn7RhBJoo1zsak8kRdrIGyWlV5vkOnW5lxINaEk
Pl2S+mCeCYZ8kWMy7PYWGkIbuayiOfU4w4bQmSukyMyMOdGzsr kak@pixie
```

- In general, the format used for storing a key is specific to each protocol. The public key shown above is for the SSH protocol as described in RFC 4253. An SSH public key stores the following three fields separated by white space: (1) the key type; (2) a chunk of Base64 encoded data; and (3) A comment. In the public key that I showed above, the first and the third fields are, respectively, the strings 'ssh-rsa' and 'kak@pixie'. What is in-between the two is the Base64 encoded data that holds the public exponent and the modulus integers. After you Base64-decode this string, you end up with a stream of bytes for three `<length data>` records. These three records hold the following three pieces of information: (1) Algorithm name (which would be the same as the key-type you would have seen in the first field of the public key; (2) the RSA public exponent; and (3) the RSA modulus. *In each record, the length value is stored in the first four bytes in the Big-endian form.* [Therefore, in order to extract the (e, n) integers from the key shown above, we must scan the byte stream that we get after Base64 decoding of the middle field shown above. We look at the first four bytes to see how many subsequent bytes hold the name of the algorithm. After we have read off those bytes, we again look at the next four bytes to find out how many subsequent bytes hold the public exponent; and so on for extracting the modulus integer.] Shown below is a Python script that extracts the public exponent and the modulus stored in an SSH RSA public key. In line with the note in blue, the script first separates the three field in the key by splitting it on white space. It then applies Base64 decoding to the middle field since that's where the public exponent and the modulus are stored. Subsequently, it scans the stream of decoded bytes for the `<length,value>` records under the assumption that the length of the value is always placed in the first four bytes of each record.

```
#!/usr/bin/env python

##  extract_sshpubkey_params.py
##  Author:  Avi Kak
##  Date:    February 11, 2013

import sys
import base64
import BitVector
if len(sys.argv) != 2:
    sys.stderr.write("Usage: %s   <public key file>\n" % sys.argv[0])
    sys.exit(1)
keydata = base64.b64decode(open(sys.argv[1]).read().split(None)[1])
bv = BitVector.BitVector( rawbytes = keydata )
parts = []
while bv.length() > 0:
  bv_length = int(bv[:32])               # read 4 bytes for length of data
  data_bv = bv[32:32+bv_length*8]        # read the data
  parts.append(data_bv)
  bv.shift_left(32+bv_length*8)          # shift the starting BV and
  bv = bv[0:-32-bv_length*8]             #    and truncate its length
public_exponent = int(parts[1])
modulus = int(parts[2])
print "public exponent: ", public_exponent
print "modulus: ", modulus
```

- If I invoke the above script on my public SSH RSA key in `~/.ssh/id_rsa.pub` by

```
extract_sshpubkey_params.py  ~/.ssh/id_rsa.pub
```

I get the following output:

```
public exponent:  35

modulus:  28992239265965680130833686108835390387986295644147105350109222053494471862488069515097328563379
          83891022841669525585184878497657164390613162380624769814604174911672498450880421371197440983388
          47257142771415372626026723527808024668042801683207069068148652181723508612356368518824921733281
          43920627731421841448660007107587358412377023141585968920645470981284870961025863780564707807073
          26000355974893593324676938927020360090167303189496460600023756410428250646775191158351910891625
          48335568714591065003819759709855208965198762621002125196213207135126179267804883812905682728422
          31250173298006999624238138047631459357691872217
```

- The SSL/TLS public and private keys, as also the SSH RSA private keys, are, on the other hand, stored using a more elaborate procedure: The key information is first encoded using Abstract Syntax Notation (ASN) according to the ASN.1 standard and the resulting data structure DER-encoded into a byte stream. (DER standards for 'Distinguished Encoding Rules' — it's a part of the ASN.1 standard.) Finally, the byte stream thus generated is turned into a printable representation by Base64 encoding. [The ASN.1 standard, along with one of its *transfer encodings* such as DER, accomplishes the same thing for complex data structures in a binary format that the XML standard does in a textual format. You can certainly convert XML representations into binary formats, but the resulting encoding will, in general, be much longer than those produced by ASN.1. Let's say you wish to represent all of your assets in a manner that would be directly readable by different computing platforms and different programming languages. A record of your assets is likely to consist of the names of the financial institutions and the value of the assets held by them, a listing of your fixed assets, such as real estate properties and their worth, etc. In general, such data will require a tree representation in which the various nodes may stand for the names of the financial institutions or the names of the assets and the children of the leaf nodes would consist of asset values. The values for some of the nodes may be in the form of ordered lists, unordered lists (sets), key-value pairs, etc. ASN.1 creates compact byte level representations for such structures that is portable across platforms and languages. *Just to give you a small taste of the flexibility of ASN.1 representation, it places no constraints on the size of any of the symbolic entities or any of the numerical values.* And to also give you a taste of the secret to the sauce, when ASN.1 is used with BER (Basic Encoding Rules) encoding, each node of the tree is represented by three blocks of bytes: (1) Identification block of an unlimited number of bytes; (2) Length block of an unlimited number of bytes; and (3) Value block of an unlimited number of bytes.The important thing to note here is there are no constraints on how many bytes are taken up by each of the thee blocks. How does ASN.1 accomplish that? It's all done by using high-end bytes to carry information about bytes further downstream. For example, if the length is to be represented by a single byte, then the value of length must not exceed 128. However, if the value of length is 128 or greater, then the most significant bit of the first byte must be set to 1 and the trailing bits must tell

us how many of the following bytes are being used for storing the length information. Similar rules are used for the other blocks to permit them to be of arbitrary length.]

- To generate the private and public keys for the SSL/TLS protocol you can use the OpenSSL library in the following manner:

```
openssl genrsa -out myprivate.pem 1024

openssl rsa -in myprivate.pem -pubout > mypublic.pem
```

  where the first command creates a private key for a 1024 bit modulus and the second then gives you the corresponding public key. The private key will be deposited in the file `myprivate.pem` and the public key in the file `mypublic.pem`.

- If you want to see the modulus and the public exponent used in the public key, you can execute

```
openssl rsa -pubin -inform PEM -text -noout < mypublic.pem
```

- As mentioned earlier, SSL/TLS keys are stored (and transmitted) by first encoding them with the abstract notation of ASN.1, turning the resulting structure into a byte stream with DER encoding, and, finally, making this byte stream printable with Base64 encoding. [The standards documents that address the formatting of such keys are RFC 3447 and 4716.] The ASN.1 representation of a public RSA key is given by:

```
SEQUENCE {
```

```
        SEQUENCE {
            OBJECT IDENTIFIER rsaEncryption (1 2 840 113549 1 1 1),
            NULL
        }
        BIT STRING {
            RSAPublicKey ::= SEQUENCE {
                modulus            INTEGER,                -- n
                publicExponent     INTEGER                 -- e
            }
        }
    }
```

where "1 2 840 113549 1 1 1" is the ANS.1 specified object ID for **rsaEncryption** and where n is the modulus and e the public exponent. [In symbolic depictions of ASN.1 data structures, what comes after a double hyphen is a comment.] The terms **SEQUENCE**, **BIT STRING**, etc. are some of the ASN.1 keywords. [Note that the ASN.1 keywords, such as 'SEQUENCE', 'BIT STRING', etc., that you see in the data structure above determine how the data bearing bytes are laid out in the byte-stream representation of the object. These keywords themselves do not appear directly in their symbolic forms in the byte level representation of a key. An agent receiving such a key would know its "schema" from the object identifier and would thus be able to decode the bytes.]

- The ASN.1 representation for a private key is given by (where we have suppressed ancillary information related to the object identity, etc.):

```
RSAPrivateKey ::= SEQUENCE {
    version           Version,
    modulus           INTEGER,              -- n
    publicExponent    INTEGER,              -- e
```

```
        privateExponent   INTEGER,              -- d
        prime1            INTEGER,              -- p
        prime2            INTEGER,              -- q
        exponent1         INTEGER,              -- d mod (p-1)
        exponent2         INTEGER,              -- d mod (q-1)
        coefficient       INTEGER,              -- (inverse of q) mod p
        otherPrimeInfos   OtherPrimeInfos OPTIONAL
    }
```

where `n` is the modulus, `e` the public exponent, `d` the private exponent, `p` and `q` the two primes whose product is the modulus. The rest of the fields are used in the modular exponentiation that is carried out for decryption.

- In Perl, you can use the `Convert::ASN1` module for creating an ASN.1 encoded representation of a data structure and for its transformation into a byte stream with BER or DER encodings. In Python, you can do the same with the `pyasn1` library.

12.12: IN SUMMARY ...

- Assuming that you are using the best possible random number generators to create candidates for the primes that are needed and that you also use a recent version of the RSA scheme that is resistant to the chosen ciphertext attacks, 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 a key whose length is — or, to speak more precisely, a modulus whose size is — shorter than 1024 bits for RSA. Some people recommend 2048 or even 4096 bit keys. The following table vividly illustrates how the key sizes compare for symmetric-key cryptography and RSA-based public-key cryptography for the same level of cryptographic security [Values taken from NIST Special Publication 800-57, *Recommendations for Key Management — Part 1*," by Elaine Barker et al.]

Symmetric Key Algorithm	Key Size for the Symmetric Key Algorithm	Comparable RSA Key Length for the Same Level of Security
2-Key 3DES	112	1024
3-Key 3DES	168	2048
AES-128	128	3072
AES-192	192	7680
AES-256	256	15360

- 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.

12.13: HOMEWORK PROBLEMS

1. The necessary condition for the encryption key e is that it be coprime to the totient of the modulus. But, in practice, what is e typically set to and why? (Obviously, now the burden falls on ensuring the selected primes p and q are such that the necessary condition on e is still satisfied.)

2. On the basis of the material presented in Sections 12.6, 12.7, 12.8 and 12.9, make your own assessment of the security vulnerabilities of RSA that are important today, that could become important in the next decade, and that could be important over the very long term.

3. From the public key, we know the modulus n and the encryption integer e. If a bad guy could figure out the totient of the modulus, would that amount to breaking the code?

4. Following the steps outlined in Section 12.4, create an RSA block cipher with 16 bits of encryption (implying that you will use a 16-bit number for the modulus n in your cipher). Do NOT use the same primes for p and q that I used in my example in Section

12.4. Use the n and e part of the cipher for block encryption of the 6-byte word "purdue". Print out the encrypted word as a 12-character hex string. Next use the n and d part of the cipher to decrypt the encrypted string.

5. Assume for the sake of argument that your RSA scheme is as simple as the one outlined in the toy example of Section 12.4. How do you think it is possible for an attacker to figure out the message bytes from the ciphertext bytes without access to the private exponent?

6. As you now know, in the RSA algorithm a message M is encrypted by calculating:

$$C \quad = \quad M^e \;\; (mod\, n)$$

where n is the modulus.

Assume that you are using a 1024-bit RSA algorithm (meaning that the modulus is of size 1024 bits) for encrypting your messages. Now let's say that your enemy knows that your business partners are in the habit of communicating with you with very short messages — messages that involve very small values of M compared to the size of the $n = p \times q$ modulus.

Since the enemy will know your public key, he will know that what your business partner has sent you is $C = M^e$ where e is the public exponent that the enemy would know about. Assuming

for the sake of convenience that $e = 3$, why can't the enemy decrypt the confidential message intended for you by just taking the cube-root of C?

7. Programming Assignment:

To better understand the point made in Section 12.3.2 that a small value, such as 3, for the encryption integer e is cryptographically unsafe, assume that a party A has sent the same message $M = 10$ to three different recipients using the following three public keys:

```
[29, 3]        [37, 3]        [41, 3]
```

In each public key, the first integer is the modulus n and the second the encryption integer e. Now use the Chinese Remainder Theorem of Section 11.7 in Lecture 11 to show how you can reconstruct M^3, which in this case would be 1000, from the three ciphertext values corresponding to the three public keys. [HINT: If you are using Python, the ciphertext value in each case is returned by the built-in 3-argument function *pow*(). For example, $pow(M, 3, 29)$ will return the ciphertext integer C_1 for the first public key shown above. For each public key, we have $C_i = M^3 \ mod \ n_i$ where the three moduli are denoted $n_1 = 29$, $n_2 = 37$, and $n_3 = 41$. Now to solve the problem, you can reason as follows: Since n_1, n_2, and n_3 are pairwise co-prime, CRT allows us to reconstruct M^3 modulo $N = n_1 \times n_2 \times n_3$. This will require that you find $N_i = N/n_i$ for $i = 1, 2, 3$. And then you would need to find the multiplicative inverse of each N_i modulo its corresponding n_i. Let N_i^{inv} denote this multiplicative inverse. You can use the Python multiplicative-inverse calculator shown in Section 5.7 of Lecture 5 to calculate the N_i^{inv} values. Then, by CRT, you should be able to recover M^3 by $(C_1 \times N_1 \times N_1^{inv} + C_2 \times N_2 \times N_2^{inv} + C_3 \times N_3 \times N_3^{inv}) \ mod \ N$.]

8. **Programming Assignment:**

Using the Python or the Perl version of the `PrimeGenerator` class shown below and the multiplicative-inverse finding scripts presented earlier in Section 5.7 of Lecture 5, write a script that would constitute a "complete" implementation of a 64-bit RSA algorithm. (As you now know from Section 12.7, a truly complete implementation of RSA involves serious security considerations related to padding, etc., that are beyond the scope of a homework assignment. All you are being asked to do in this homework is to address the basic mathematics of RSA.)

```
#!/usr/bin/env python

##  PrimeGenerator.py
##  Author: Avi Kak
##  Date: February 18, 2011
##  Modified Date:  February 28, 2016

##  Call syntax:
##
##        PrimeGenerator.py  width_desired_for_bit_field_for_prime
##
##  For example, if you call
##
##        PrimeGenerator.py 32
##
##  you may get a prime that looks like 3262037833.  On the other hand, if you
##  call
##
##        PrimeGenerator.py 128
##
##  you may get a prime that looks like 338816507393364952656338247029475569761
##
##  IMPORTANT:  The two most significant are explicitly set for the prime that is
##              returned.

import sys
import random

############################  class PrimeGenerator  ##############################
```

```
class PrimeGenerator( object ):                                                  #(A1)

    def __init__( self, **kwargs ):                                              #(A2)
        bits = debug = None                                                      #(A3)
        if 'bits' in kwargs  :     bits = kwargs.pop('bits')                     #(A4)
        if 'debug' in kwargs :     debug = kwargs.pop('debug')                   #(A5)
        self.bits            =     bits                                        #(A6)
        self.debug           =     debug                                       #(A7)
        self._largest        =     (1 << bits) - 1                             #(A8)

    def set_initial_candidate(self):                                             #(B1)
        candidate = random.getrandbits( self.bits )                              #(B2)
        if candidate & 1 == 0: candidate += 1                                    #(B3)
        candidate |= (1 << self.bits-1)                                          #(B4)
        candidate |= (2 << self.bits-3)                                          #(B5)
        self.candidate = candidate                                               #(B6)

    def set_probes(self):                                                        #(C1)
        self.probes = [2,3,5,7,11,13,17]                                         #(C2)

    # This is the same primality testing function as shown earlier
    # in Section 11.5.6 of Lecture 11:
    def test_candidate_for_prime(self):                                          #(D1)
        'returns the probability if candidate is prime with high probability'
        p = self.candidate                                                       #(D2)
        if p == 1: return 0                                                      #(D3)
        if p in self.probes:                                                     #(D4)
            self.probability_of_prime = 1                                        #(D5)
            return 1                                                             #(D6)
        if any([p % a == 0 for a in self.probes]): return 0                      #(D7)
        k, q = 0, self.candidate-1                                               #(D8)
        while not q&1:                                                           #(D9)
            q >>= 1                                                              #(D10)
            k += 1                                                               #(D11)
        if self.debug: print("q = %d  k = %d" % (q,k))                           #(D12)
        for a in self.probes:                                                    #(D13)
            a_raised_to_q = pow(a, q, p)                                         #(D14)
            if a_raised_to_q == 1 or a_raised_to_q == p-1: continue              #(D15)
            a_raised_to_jq = a_raised_to_q                                       #(D16)
            primeflag = 0                                                        #(D17)
            for j in range(k-1):                                                 #(D18)
                a_raised_to_jq = pow(a_raised_to_jq, 2, p)                       #(D19)
                if a_raised_to_jq == p-1:                                        #(D20)
                    primeflag = 1                                                #(D21)
                    break                                                        #(D22)
            if not primeflag: return 0                                           #(D23)
        self.probability_of_prime = 1 - 1.0/(4 ** len(self.probes))              #(D24)
        return self.probability_of_prime                                         #(D25)

    def findPrime(self):                                                         #(E1)
        self.set_initial_candidate()                                             #(E2)
        if self.debug:  print("    candidate is: %d" % self.candidate)            #(E3)
        self.set_probes()                                                        #(E4)
        if self.debug:  print("    The probes are: %s" % str(self.probes))       #(E5)
        max_reached = 0                                                          #(E6)
```

```
        while 1:                                                              #(E7)
            if self.test_candidate_for_prime():                               #(E8)
                if self.debug:                                                #(E9)
                    print("Prime number: %d with probability %f\n" %
                          (self.candidate, self.probability_of_prime) )       #(E10)
                break                                                         #(E11)
            else:                                                             #(E12)
                if max_reached:                                               #(E13)
                    self.candidate -= 2                                       #(E14)
                elif self.candidate >= self._largest - 2:                     #(E15)
                    max_reached = 1                                           #(E16)
                    self.candidate -= 2                                       #(E17)
                else:                                                         #(E18)
                    self.candidate += 2                                       #(E19)
                if self.debug:                                                #(E20)
                    print("    candidate is: %d" % self.candidate)             #(E21)
        return self.candidate                                                 #(E22)

####################################  main  ######################################
if __name__ == '__main__':

    if len( sys.argv ) != 2:                                                  #(M1)
        sys.exit( "Call syntax:  PrimeGenerator.py  width_of_bit_field" )     #(M2)
    num_of_bits_desired = int(sys.argv[1])                                    #(M3)
    generator = PrimeGenerator( bits = num_of_bits_desired )                  #(M4)
    prime = generator.findPrime()                                             #(M5)
    print("Prime returned: %d" % prime)                                       #(M6)
```

If you make the following call to this script:

```
PrimeGenerator.py 64
```

the script will return a full-width 64-bit prime that will look like:

```
Prime returned: 17828589080991197309
```

On the other hand, a call like

```
PrimeGenerator.py 128
```

will return something like:

```
Prime returned: 290410362853346697538147183843312052911
```

For those of you will be doing this homework in Perl, here is a Perl version of the above script:

```
#!/usr/bin/env perl

##  PrimeGenerator.pl
##  Author: Avi Kak
##  Date: February 26, 2016

##  Call syntax:
##
##        PrimeGenerator.pl  width_desired_for_bit_field_for_prime
##
##  For example, if you call
##
##        PrimeGenerator.pl 32
##
##  you may get a prime that looks like 3340094299.  On the other hand, if you
##  call
##
##        PrimeGenerator.pl 128
##
##  you may get a prime that looks like 333618953930748159614512936853740718827
##
##  IMPORTANT:  The two most significant are explicitly set for the prime that is
##              returned.

use strict;
use warnings;
use Math::BigInt;

############################  class PrimeGenerator  ##############################
package PrimeGenerator;

sub new {                                                                    #(A1)
    my ($class, %args) = @_;                                                 #(A2)
    bless {                                                                  #(A3)
        _bits  =>  int($args{bits}),                                         #(A4)
        _debug =>  $args{debug} || 0,                                        #(A5)
        _largest => (1 << int($args{bits})) - 1,                             #(A6)
    }, $class;                                                               #(A7)
}

sub set_initial_candidate {                                                  #(B1)
    my $self = shift;                                                        #(B2)
    my @arr = map {my $x = rand(1); $x > 0.5 ? 1 : 0 } 0 .. $self->{_bits}-4;#(B3)
    push @arr, 1;                                                            #(B4)
    unshift @arr, (1,1);                                                     #(B6)
    my $bstr = join '', split /\s/, "@arr";                                  #(B7)
#    $self->{candidate} = oct("0b".$bstr);                                   #(B8)
```

```
    $self->{candidate} = Math::BigInt->from_bin($bstr);                          #(B8)
}

sub set_probes {                                                                 #(C1)
    my $self = shift;                                                            #(C2)
    $self->{probes} = [2,3,5,7,11,13,17];                                        #(C3)
}

# This is the same primality testing function as shown earlier
# in Section 11.5.6 of Lecture 11:
sub test_candidate_for_prime_with_bigint {                                       #(D1)
    my $self = shift;                                                            #(D2)
    my $p = $self->{candidate};                                                  #(D3)
    return 0 if $p->is_one();                                                    #(D4)
    my @probes = @{$self->{probes}};                                             #(D5)
    foreach my $a (@probes) {                                                    #(D6)
        $a = Math::BigInt->new("$a");                                            #(D7)
        return 1 if $p->bcmp($a) == 0;                                           #(D8)
        return 0 if $p->copy()->bmod($a)->is_zero();
    }
    my ($k, $q) = (0, $p->copy()->bdec());                                       #(D9)
    while (! $q->copy()->band( Math::BigInt->new("1"))) {                        #(D10)
        $q->brsft( 1 );                                                          #(D11)
        $k += 1;                                                                 #(D12)
    }
    my ($a_raised_to_q, $a_raised_to_jq, $primeflag);                            #(D13)
    foreach my $a (@probes) {                                                    #(D14)
        my $abig = Math::BigInt->new("$a");                                      #(D15)
        my $a_raised_to_q = $abig->bmodpow($q, $p);                              #(D16)
        next if $a_raised_to_q->is_one();                                        #(D17)
        my $pdec = $p->copy()->bdec();                                           #(D18)
        next if ($a_raised_to_q->bcmp($pdec) == 0) && ($k > 0);                  #(D19)
        $a_raised_to_jq = $a_raised_to_q;                                        #(D20)
        $primeflag = 0;                                                          #(D21)
        foreach my $j (0 .. $k - 2) {                                            #(D22)
            my $two = Math::BigInt->new("2");                                    #(D23)
            $a_raised_to_jq = $a_raised_to_jq->copy()->bmodpow($two, $p);        #(D24)
            if ($a_raised_to_jq->bcmp( $p->copy()->bdec() ) == 0 ) {             #(D25)
                $primeflag = 1;                                                  #(D26)
                last;                                                            #(D27)
            }
        }
        return 0 if ! $primeflag;                                                #(D28)
    }
    my $probability_of_prime = 1 - 1.0/(4 ** scalar(@probes));                   #(D29)
    $self->{probability_of_prime} = $probability_of_prime;                       #(D30)
    return $probability_of_prime;                                                #(D31)
}

sub findPrime {                                                                  #(E1)
    my $self = shift;                                                            #(E2)
    $self->set_initial_candidate();                                              #(E3)
    print "       candidate is:  $self->{candidate}\n" if $self->{_debug};       #(E4)
    $self->set_probes();                                                         #(E5)
    print "       The probes are: @{$self->{probes}}\n" if $self->{_debug};      #(E6)
```

```
    my $max_reached = 0;                                                    #(E7)
    while (1) {                                                             #(E8)
        if ($self->test_candidate_for_prime_with_bigint()) {                #(E9)
            print "Prime number:  $self->{candidate} with probability: " .
                  "$self->{probability_of_prime}\n" if $self->{debug};      #(E10)
            last;                                                           #(E11)
        } else {                                                            #(E12)
            if ($max_reached ) {                                            #(E13)
                $self->{candidate} -= 2;                                    #(E14)
            } elsif ($self->{candidate} >= $self->{_largest} - 2) {         #(E15)
                $max_reached = 1;                                           #(E16)
                $self->{candidate} -= 2;                                    #(E17)
            } else {                                                        #(E18)
                $self->{candidate} += 2;                                    #(E19)
            }
        }
    }
    return $self->{candidate};                                              #(E20)
}

1;
################################  main    ########################################
package main;

unless (@ARGV) {                                                            #(M1)
    1;                                                                      #(M2)
} else {                                                                    #(M3)
    my $bitfield_width = shift @ARGV;                                       #(M4)
    my $generator = PrimeGenerator->new(bits => $bitfield_width);           #(M5)
    my $prime = $generator->findPrime();                                    #(M6)
    print "Prime returned: $prime\n";                                       #(M7)
}
```

A call such as the one shown below for generating a 256 bit prime

```
PrimeGenerator.pl 256
```

comes back with

```
Prime returned: 110683214729271322144990809842795090895043970651486233118696734813266440218909
```

9. **Programming Assignment:**

This assignment is also about implementing the RSA algorithm, but now you are allowed to use modules from open-source libraries

for some of the work. Because these libraries sit on top of highly efficient C code, you should be able to test your implementation for much larger moduli than what you used in the previous programming assignment. Write Perl or Python scripts that implement the RSA encryption and decryption algorithms. Do NOT use the key-generator functions implemented in the modules of the Perl/Python toolkits to find d for a given e. On the other hand, you must use either the Python implementation shown in Section 5.7 of Lecture 5 or your own implementation of the Extended Euclidean Algorithm to find the multiplicative inverses you need. Feel free to use any other modules in the toolkits listed below, or, for that matter, any other modules of you choice. However, you must list the modules used and where you found them in the reference section of your code.

Python Cryptography Toolkit: `http://www.amk.ca/python/code/crypto`

Perl Crypt-RSA Toolkit: `http://search.cpan.org/~vipul/Crypt-RSA-1.57/lib/Crypt/RSA.p`