

Introduction to Cryptography

Prepared by Dr. Mohammad Alkhatib

Associate Professor

Major: Information Security

جامعة الجميع الذكية

تسعى لتكون الكيان الأول الذي يتبنى مبدأ المجتمع التعليمي
التعاوني ويقدم العلم مقابل العمل لتنمية وبناء المجتمعات.

Lecture #5: RSA Cryptosystem



جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY

RSA Cryptosystem

- Introduction to RSA Cryptosystem
- RSA Encryption and Decryption
- RSA Fast Exponentiation
- Speed-up Techniques for RSA
- *Attacks against RSA*

References:

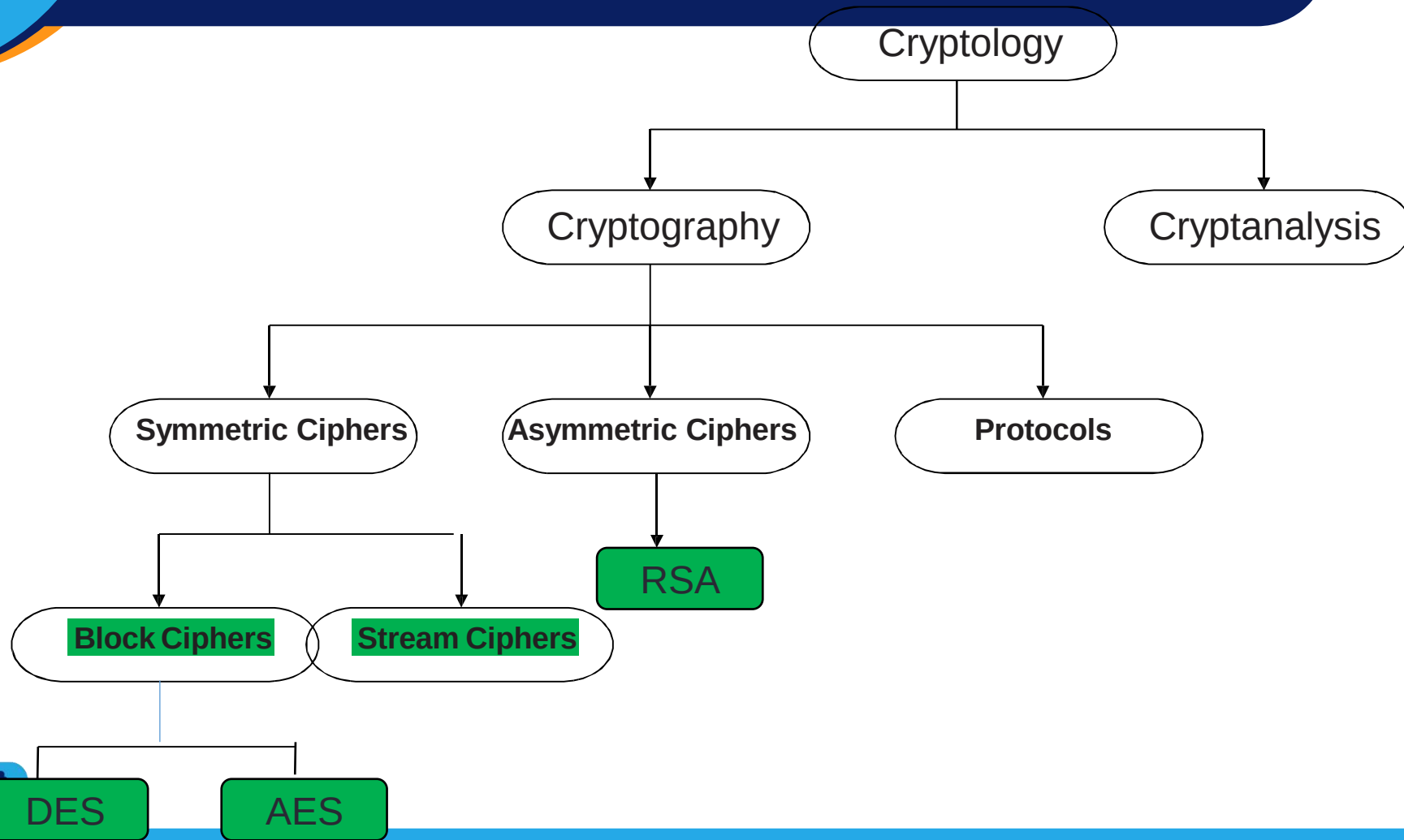
[1] Katz and Lindell, Introduction to Modern Cryptography. SECOND EDITION. Chapman & Hall/ CRC Press, 2014

[2] C. Paar and J. Pelzl. Understanding Cryptography. A textbook for Student and Practitioners. Springer 2010

[3] Trappe, Wade, and Lawrence C. Washington. 2006 (2nd Edition). Introduction to cryptography: with coding theory. Upper Saddle River, NJ: Prentice Hall



Introduction to RSA Cryptosystem



- ❑ RSA encryption and decryption is done in the integer ring Z_n and modular computations play a central role.
- ❑ RSA encrypts plaintexts x , where we consider the bit string representing x to be an element in $Z_n = \{0, 1, \dots, n-1\}$.
- ❑ As a consequence the binary value of the plaintext x must be less than n . The same holds for the ciphertext

❑ Encryption with the public key and decryption with the private key can be described as follows:

❑ **RSA Encryption:** Given the public key $(n,e) = k_{pub}$ and the plaintext x , the encryption function is:

$$y = e_{k_{pub}}(x) \equiv x^e \pmod{n}, \text{ where } x,y \in \mathbb{Z}_n.$$

❑ **RSA Decryption:** Given the private key $d = k_{pr}$ and the ciphertext y , the decryption function is:

$$x = d_{k_{pr}}(y) \equiv y^d \pmod{n}, \text{ where } x,y \in \mathbb{Z}_n$$

- ❑ In practice, x , y , n and d are very long numbers, usually 1024 bit long or more
- ❑ The value e is sometimes referred to as encryption exponent or public exponent, and the private key d is sometimes called decryption exponent or private exponent.
- ❑ If Alice wants to send an encrypted message to Bob, Alice needs to have his public key (n,e) , and Bob decrypts with his private key d .

❑ In general, there are few requirements for the RSA cryptosystem:

1. Since an attacker has access to the public key, it must be computationally infeasible to determine the private-key d given the public-key values e and n
2. Since x is only unique up to the size of the modulus n , we cannot encrypt more than l bits with one RSA encryption, where l is the bit length of n
3. It should be relatively easy to calculate $x^e \bmod n$, i.e., to encrypt, and $y^d \bmod n$, i.e., to decrypt. This means we need a method for fast exponentiation with very long numbers.
4. For a given n , there should be many private-key/public-key pairs, otherwise an attacker might be able to perform a brute-force attack. (It turns out that this requirement is easy to satisfy.)



□ The steps involved in computing the public and private-key for an RSA cryptosystem can be described as follows:

RSA Key Generation

Output: public key: $k_{pub} = (n, e)$ and private key: $k_{pr} = (d)$

1. Choose two large primes p and q .
2. Compute $n = p \cdot q$.
3. Compute $\Phi(n) = (p - 1)(q - 1)$.
4. Select the public exponent $e \in \{1, 2, \dots, \Phi(n) - 1\}$ such that

$$\gcd(e, \Phi(n)) = 1.$$

5. Compute the private key d such that

$$d \cdot e \equiv 1 \pmod{\Phi(n)}$$

□ The condition that $\gcd(e, \Phi(n)) = 1$ ensures that the inverse of e exists modulo $\Phi(n)$, so that there is always a private key d

- ❑ Two parts of the key generation are nontrivial: Step 1, in which the two large primes are chosen, and Steps 4 and 5 in which the public and private keys are computed. The prime generation of Step 1 is quite involved and is addressed in following slides.
- ❑ The computation of the keys d and e can be done at once using the extended Euclidean algorithm (EEA).
- ❑ In practice, one often starts by first selecting a public parameter e in the range $0 < e < \Phi(n)$. The value e must satisfy the condition $\gcd(e, \Phi(n)) = 1$.
- ❑ We apply the EEA with the input parameters n and e and obtain the relationship:

$$\gcd(\Phi(n), e) = s \cdot \Phi(n) + t \cdot e$$

- ❑ If $\gcd(e, \Phi(n)) = 1$, we know that e is a valid public key. Moreover, we also know that the parameter t computed by the extended Euclidean algorithm is the inverse of e , *and thus: $d = t \bmod \Phi(n)$*
- ❑ In case that e and $\Phi(n)$ are not relatively prime, we simply select a new value for an e and repeat the process. Note that the coefficient s of the EEA is not required for RSA

- ❑ We need to distinguish between modular arithmetic in n and $\Phi(n)$ for RSA
- ❑ We always do arithmetic modula n when performing usual RSA operations, such as: encryption and decryption.
- ❑ Only for key generation, in particular: steps 4 and 5, we need to use arithmetic modula $\Phi(n)$
- ❑ At the end of the five steps, mentioned in previous slide, we can compute the following RSA parameters:
- ❑ $K_{pub} (n, e)$: Both values represent the public-key
- ❑ $K_{pri} = d$: The private key value (the inverse of e Modula $\Phi(n)$)

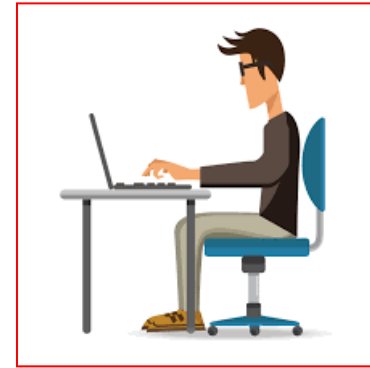
❑ **Example:** Alice wants to send an encrypted message to Bob. Bob first computes his RSA parameters in Steps 1–5. He then sends Alice his public key. Alice encrypts the message ($x = 4$) and sends the ciphertext y to Bob. Bob decrypts y using his private key



Sender (Alice)

K_{pub} : Encryption (public) key
(e, n) = (3, 33)

y : Encrypted Message (31)



Receiver (Bob)

- Encrypt a message using the public (encryption) key that was sent by the receiver

$$y = x^e \equiv 4^3 \equiv 31 \pmod{33}$$

- Note that the private and public exponents fulfill the condition $e \cdot d = 3 \cdot 7 \equiv 1 \pmod{\Phi(n)}$

❖ Bob **Decrypts** y such that:
 $y^d = 31^7 \equiv 4 = x \pmod{33}$

1. choose $p = 3$ and $q = 11$ (two large primes)
2. Compute $n = p \cdot q = 33$
3. $\Phi(n) = (3-1)(11-1) = 20$
4. choose $e = 3$, which satisfies: $\gcd(e, \Phi(n)) = 1$
5. $d \equiv e^{-1} \equiv 7 \pmod{20}$

❑ Practical RSA parameters are much, much larger. the RSA modulus n should be at least 1024 bit long, which results in a bit length for p and q of 512 bits. Here is an example of RSA parameters for this bit length:

$p = E0DFD2C2A288ACEBC705EFAB30E4447541A8C5A47A37185C5A9$
 $CB98389CE4DE19199AA3069B404FD98C801568CB9170EB712BF$
 $10B4955CE9C9DC8CE6855C6123_h$

$q = EBE0FCF21866FD9A9F0D72F7994875A8D92E67AEE4B515136B2$
 $A778A8048B149828AEA30BD0BA34B977982A3D42168F594CA99$
 $F3981DDABFAB2369F229640115_h$

$n = CF33188211FDF6052BDBB1A37235E0ABB5978A45C71FD381A91$
 $AD12FC76DA0544C47568AC83D855D47CA8D8A779579AB72E635$
 $D0B0AAAC22D28341E998E90F82122A2C06090F43A37E0203C2B$
 $72E401FD06890EC8EAD4F07E686E906F01B2468AE7B30CBD670$
 $255C1FEDE1A2762CF4392C0759499CC0ABECFF008728D9A11ADF_h$

$e = 40B028E1E4CCF07537643101FF72444A0BE1D7682F1EDB553E3$
 $AB4F6DD8293CA1945DB12D796AE9244D60565C2EB692A89B888$
 $1D58D278562ED60066DD8211E67315CF89857167206120405B0$
 $8B54D10D4EC4ED4253C75FA74098FE3F7FB751FF5121353C554$
 $391E114C85B56A9725E9BD5685D6C9C7EED8EE442366353DC39_h$

$d = C21A93EE751A8D4FBFD77285D79D6768C58EBF283743D2889A3$
 $95F266C78F4A28E86F545960C2CE01EB8AD5246905163B28D0B$
 $8BAABB959CC03F4EC499186168AE9ED6D88058898907E61C7CC$
 $CC584D65D801CFE32DFC983707F87F5AA6AE4B9E77B9CE630E2$
 $C0DF05841B5E4984D059A35D7270D500514891F7B77B804BED81_h$

❑ What is interesting is that the message x is first raised to the e^{th} power during encryption and the result y is raised to the d^{th} power in the decryption, and the result of this is again equal to the message x . Expressed as an equation, this process is:

$$dk_{pr}(y) = dk_{pr}(ek_{pub}(x)) \equiv (x^e)^d \equiv x^{de} \equiv x \pmod{n}.$$

❑ Note that encryption and decryption operations are merely exponentiation

❑ To encrypt, we need to perform exponentiation using the public key e

❑ To decrypt, we need to perform exponentiation using the private key d

❑ Therefore, we need a fast method to perform exponentiation or to implement RSA encryption and decryption processes.

❑ What is also important is how to choose very large prime numbers (p, q)



❑ Question:

❑ Why the attacker cannot decrypt the message using RSA cryptosystem?

❑ He has the public key, he knows the encryption and decryption function, but what is missing, why he cannot run the Extended Euclidean Algorithm (EEA) and obtain the private key d (the inverse)

❑ Answer:

❑ He needs to compute the $\phi(n)$. We have learned that this could be done easily if we know the prime factorization p and q of n .

❑ If n is very large number $\rightarrow$ 1024 bits $\rightarrow$ 300 decimal digits $\rightarrow$ No one in the world can factor such number, even with using the latest advances in technology devices.

Even with using the factorization records, we can factor up to 600 bits or less more, but not 1024 bits

- ❑ Unlike symmetric algorithms such as AES, DES or stream ciphers, public-key algorithms are based on arithmetic with very long numbers
- ❑ Unless we pay close attention to how to realize the necessary computations, we can easily end up with schemes that are too slow for practical use.
- ❑ If we look at RSA encryption and decryption in the following equations, we see that both are based on modular exponentiation.

$$y = \text{ek}_{\text{pub}}(x) \equiv x^e \pmod{n} \text{ (encryption)}$$

$$x = \text{dk}_{\text{pr}}(y) \equiv y^d \pmod{n} \text{ (decryption)}$$



❑ A straightforward way of exponentiation looks like this:

$X: SQ \rightarrow x^2 \text{ MUL} \rightarrow x^3 \text{ MUL} \rightarrow x^4 \text{ MUL} \rightarrow x^5 \dots$

where *SQ* denotes *squaring* and *MUL* *multiplication*

❑ Straightforward exponentiation as shown above would thus require around 2^{1024} or more multiplications

Example: Let's look at how many multiplications are required to compute the simple exponentiation x^8 . With the straightforward method:

$x : \text{SQ} \rightarrow x^2 : \text{MUL} \rightarrow x^3 : \text{MUL} \rightarrow x^4 : \text{MUL} \Rightarrow x^5 : \text{MUL} : x^6 : \text{MUL} \rightarrow x^7 : \text{MUL} \rightarrow x^8$

□ We need seven multiplications and squaring. Alternatively, we can do something faster:

$x : \text{SQ} \rightarrow x^2 : \text{SQ} \rightarrow x^4 : \text{SQ} \rightarrow x^8$

which requires only three squaring operations that are roughly as complex as a multiplication

This approach takes a total of six operations, two multiplications and four squarings

This fast method works fine but is restricted to exponents that are powers of 2, i.e., values e and d of the form 2^i . Now the question is, whether we can extend the method to arbitrary exponents? Let us look at another example:

Example: Assume we have the more general exponent 26, i.e., we want to compute x^{26} . Again, the naive method would require 25 multiplications. A faster way is as follows:

x : SQ $\rightarrow x^2$: MUL $\rightarrow x^3$: SQ $\rightarrow x^6$: SQ $\rightarrow x^{12}$: MUL $\rightarrow x^{13}$: SQ $\rightarrow x^{26}$.

□ Looking at the last example, we see that we can achieve the desired result by performing two basic operations:

1. squaring the current result,
2. multiplying the current result by the base element x .



- ❑ In the previous example, we computed the sequence SQ, MUL, SQ, SQ, MUL, SQ . However, we do not know the sequence in which the squarings and multiplications have to be performed for other exponents
- ❑ How can we obtain the right sequence to be implemented??
- ❑ One solution is the *square-and-multiply* algorithm.
- ❑ It provides a systematic way for finding the sequence in which we have to perform squarings and multiplications by x for computing x^H

□ The *square-and-multiply* algorithm works as follows:

- The algorithm is based on scanning the bit of the exponent from the left (the most significant bit) to the right (the least significant bit).
- In every iteration, i.e., for every exponent bit, the current result is squared.
- If and only if the currently scanned exponent bit has the value 1, a multiplication of the current result by x is executed following the squaring.
- Note: The first initial bit (from the left) should be ignored.

❑ Example: We again consider the exponentiation x^{26} . For the square-and-multiply algorithm, the binary representation of the exponent is crucial:

$$x^{26} = x^{11010_2} = x^{(h_4h_3h_2h_1h_0)_2}.$$

❑ The algorithm scans the exponent bits, starting on the left with h_4 and ending with the rightmost bit h_0 .

Step

#0 $x = x^{1_2}$ initial setting, bit processed: $h_4 = 1$ #1a $(x^1)^2 = x^2 = x^{10_2}$ SQ, bit processed: h_3 #1b $x^2 \cdot x = x^3 = x^{10_2}x^{1_2} = x^{11_2}$ MUL, since $h_3 = 1$ #2a $(x^3)^2 = x^6 = (x^{11_2})^2 = x^{110_2}$ SQ, bit processed: h_2

#2b

no MUL, since $h_2 = 0$ #3a $(x^6)^2 = x^{12} = (x^{110_2})^2 = x^{1100_2}$ SQ, bit processed: h_1 #3b $x^{12} \cdot x = x^{13} = x^{1100_2}x^{1_2} = x^{1101_2}$ MUL, since $h_1 = 1$ #4a $(x^{13})^2 = x^{26} = (x^{1101_2})^2 = x^{11010_2}$ SQ, bit processed: h_0

#4b

no MUL, since $h_0 = 0$

- ❑ To understand the algorithm it is helpful to closely observe how the binary representation of the exponent evolves.
- ❑ We see that the first basic operation, squaring, results in a left shift of the exponent, with a 0 put in the rightmost position.
- ❑ The other basic operation, multiplication by x , results in filling a 1 into the rightmost position of the exponent.
- ❑ Compare how the highlighted exponents change from iteration to iteration.

Use the square and multiply algorithm to find $4^7 \bmod 11$

$$7 = (111)_2 \rightarrow h_1 = 1, h_2 = 1, h_3 = 1$$

For h_1 , initial state, no action is required

$h_2 = 1 \rightarrow$ we need to perform squaring and then multiplication

$$4^2 = 16 \bmod 11 = 5 \quad (\text{square})$$

$$5 * 4 = 20 \bmod 11 = 9 \quad (\text{multiply by 4})$$

$h_3 = 1 \rightarrow$ we need to perform squaring and then multiplication

$$9^2 = 81 \bmod 11 = 4 \quad (\text{square})$$

$$4 * 4 = 16 \bmod 11 = 5 \quad (\text{multiply by 4})$$



Here is the pseudo code for the square-and-multiply algorithm:

Square-and-Multiply for Modular Exponentiation**Input:**

base element x

exponent $H = \sum_{i=0}^t h_i 2^i$ with $h_i \in 0, 1$ and $h_t = 1$

and modulus n

Output: $x^H \bmod n$

Initialization: $r = x$

Algorithm:

```
1  FOR  $i = t - 1$  DOWNTO 0
1.1     $r = r^2 \bmod n$ 
        IF  $h_i = 1$ 
1.2     $r = r \cdot x \bmod n$ 
2  RETURN ( $r$ )
```

□ The modulo reduction is applied after each multiplication and squaring operation in order to keep the intermediate results small. It is helpful to compare this pseudo code with the verbal description of the algorithm above

- ❑ As we learned RSA involves exponentiation with very long numbers. Even if the low-level arithmetic involving modular multiplication and squaring as well as the square-and-multiply algorithm are implemented carefully, performing a full RSA exponentiation with operands of length 1024 bit or beyond is computationally intensive.
- ❑ Thus, people have studied speed-up techniques for RSA since its invention.
- ❑ We introduce one of the most popular general acceleration techniques in the following slide.

A surprisingly simple and very powerful trick can be used when RSA operations with the public key e are concerned.

This is in practice encryption and, as we'll learn later, verification of an RSA digital signature.

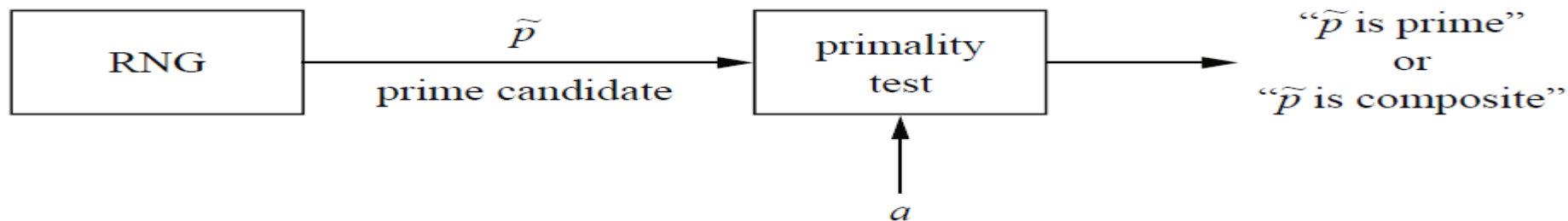
In this situation, the public key e can be chosen to be a very small value. In practice, the three values $e = 3$, $e = 17$ and $e = 2^{16} + 1$ are of particular importance.

The resulting complexities when using these public keys are given in the following table

Public key e	e as binary string	#MUL + #SQ
3	11_2	3
17	10001_2	5
$2^{16} + 1$	$1\,0000\,0000\,0000\,0001_2$	17

- ❑ Unfortunately, there is no such easy way to accelerate RSA when the private key d is involved, i.e., for decryption and signature generation.
- ❑ Hence, these two operations tend to be slow. Other public-key algorithms, in particular elliptic curves, are often much faster for these two operations.

- ❑ There is one important practical aspect of RSA which we have not discussed yet: generating the primes p and q in Step 1 of the key generation.
- ❑ Since their product is the RSA modulus $n = p \cdot q$, the two primes should have about half the bit length of n .
- ❑ For instance, if we want to set up RSA with a modulus of length $\log_2 n = 1024$, p and q should have a bit length of about 512 bit. The general approach is to generate integers at random which are then checked for primality, as depicted in the following figure, where RNG stands for random number generator.
- ❑ The RNG should be non predictable because if an attacker can compute or guess one of the two primes, RSA can be broken easily.



In order to make this approach work, we have to answer two questions:

1. How many random integers do we have to test before we have a prime? (If the likelihood of a prime is too small, it might take too long.)
2. How fast can we check whether a random integer is prime? (Again, if the test is too slow, the approach is impractical.)

It turns out that both steps are reasonably fast, as is discussed in the following slides.

- ❑ There is no known serious attack against RSA cryptosystem.
- ❑ The majority of known attacks exploit weaknesses in the way RSA is implemented or used rather than the RSA algorithm itself

❑ **Mathematical Attacks:**

- ❑ The best mathematical cryptanalytical method we know is factoring the modulus
- ❑ An attacker, Oscar, knows the modulus n , the public key e and the ciphertext y
- ❑ His goal is to compute the private key d *which has the property that* $e \cdot d \equiv 1 \pmod{\Phi(n)}$.
- ❑ It seems that he could simply apply the extended Euclidean algorithm and compute d .
- ❑ *However, he does not know the value of $\Phi(n)$.*
- ❑ At this point factoring comes in: the best way to obtain this value is to decompose n *into its primes p and q .*
- ❑ *If Oscar can do this, the attack succeeds in three steps:*

$$\Phi(n) = (p - 1)(q - 1)$$

$$d^{-1} \equiv e \pmod{\Phi(n)}$$

$$x \equiv y^d \pmod{n}.$$

- ❑ There is no known serious attack against RSA cryptosystem.
- ❑ The majority of known attacks exploit weaknesses in the way RSA is implemented or used rather than the RSA algorithm itself

❑ Mathematical Attacks:

- ❑ The best mathematical cryptanalytical method we know is factoring the modulus
- ❑ An attacker, Oscar, knows the modulus n , the public key e and the ciphertext y
- ❑ His goal is to compute the private key d *which has the property that* $e \cdot d \equiv 1 \pmod{\Phi(n)}$.
- ❑ It seems that he could simply apply the extended Euclidean algorithm and compute d .
- ❑ *However, he does not know the value of $\Phi(n)$.*
- ❑ At this point factoring comes in: the best way to obtain this value is to decompose n *into its primes p and q .*
- ❑ *If Oscar can do this, the attack succeeds in three steps:*

$$\Phi(n) = (p - 1)(q - 1)$$

$$d^{-1} \equiv e \pmod{\Phi(n)}$$

$$x \equiv y^d \pmod{n}.$$

- ❑ Intelligence organizations might be capable of doing it possibly even earlier.
- ❑ Hence, it is recommended to choose RSA parameters in the range of 2048–4096 bits for long-term security

❑ Side-Channel Attacks:

- ❑ Side-channel attacks exploit information about the private key which is leaked through physical channels such as the power consumption or the timing behavior
- ❑ In order to observe such channels, an attacker must typically have direct access to the RSA implementation, e.g., in a cell phone or a smart card
- ❑ Even though side-channel attacks are a large and active field of research in modern cryptography
- ❑ Example: The attacker may trace the time/power consumed by the crypto-processor during the decryption process and hence reveal the bit representation of the private key d .

The main computational load of RSA is the squarings and multiplication during the Exponentiation.

Then, it can be concluded that the high-activity intervals correspond to those two operations.

If we look more closely at the power trace, we see that there are high activity intervals which are short and others which are longer.

This behavior is explained by the square-and-multiply algorithm. If an exponent bit has the value 0, only a squaring is performed. If an exponent bit has the value 1, a squaring together with a multiplication is computed.

But this timing behavior reveals immediately the key: A long period of activity corresponds to the bit value 1 of the secret key, and a short period to a key bit with value 0.

Therefore, by simply looking at the power trace we can identify the secret exponent. Thus we can learn the following bits of the private key by looking at the trace:

Module 2

Attacks Against RSA

operations: *S SM SM S SM S S SM SM SM S SM*
private key: 0 1 1 0 1 0 0 1 1 1 0 1

