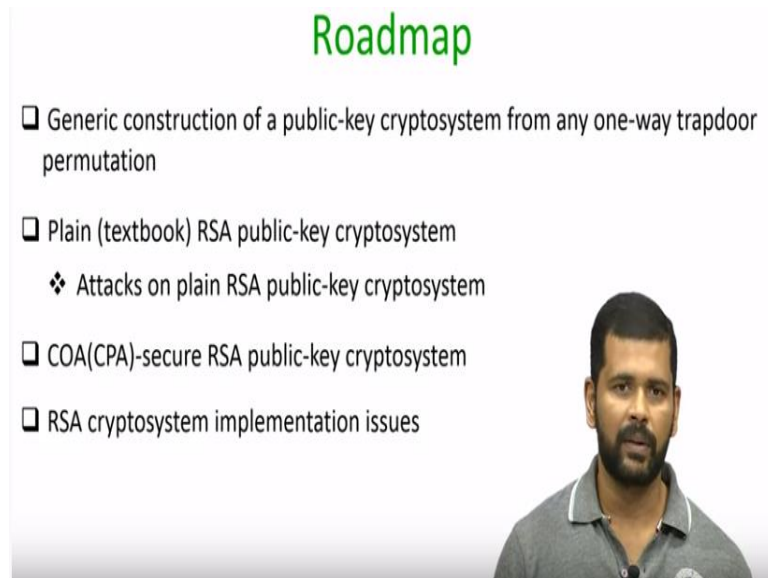


Foundations of Cryptography
Dr. Ashish Choudhury
Department of Computer Science
Indian Institute of Science– Bangalore

Lecture – 46
RSA Public Key Cryptosystem


Hello everyone. Welcome to this lecture.

(Refer Slide Time: 00:34)



Roadmap

- ❑ Generic construction of a public-key cryptosystem from any one-way trapdoor permutation
- ❑ Plain (textbook) RSA public-key cryptosystem
 - ❖ Attacks on plain RSA public-key cryptosystem
- ❑ COA(CPA)-secure RSA public-key cryptosystem
- ❑ RSA cryptosystem implementation issues



Just to recap in the last lecture, we had introduced the RSA assumption. So, we will continue our discussion with RSA assumption. Specifically, the road map for this lecture is as follows. We will see a generic construction of a public-key cryptosystem from any given One-Way Trapdoor Permutation. We will see how we can instantiate this framework whereby using the RSA Trapdoor Permutation to get the RSA public-key cryptosystem, which we also call as the plain RSA public-key cryptosystem. We will see several attacks on the plain RSA public-key cryptosystem. Then, we will see the COA or CPA-secure variant of RSA public-key cryptosystem. Finally, we will end the lecture with some implementation issues which we face when we implement RSA in practice.

(Refer Slide Time: 01:18)

Public-key Cryptosystem from Any OWTP

<p>□ Given: A OWTP scheme $T = (\text{Gen}, f, \text{Inv})$ over \mathcal{X}:</p> <ul style="list-style-type: none"> ❖ $\text{Gen}() \rightarrow (pk, sk)$ <ul style="list-style-type: none"> ➤ pk: public key ➤ sk: secret key ❖ $f_{pk}: \mathcal{X} \rightarrow \mathcal{X}$ <ul style="list-style-type: none"> ➤ $f_{pk}(x) = y$ ➤ Deterministic algorithm ❖ $\text{Inv}_{sk}: \mathcal{X} \rightarrow \mathcal{X}$ <ul style="list-style-type: none"> ➤ $\text{Inv}_{sk}(y) = x$ ➤ Deterministic algorithm <p>□ Correctness: for every (pk, sk) and $x \in \mathcal{X}$:</p> $\text{Inv}_{sk}(f_{pk}(x)) = x$ <p>□ One-Wayness: f_{pk} is a OWF, even if an adversary knows the public key pk</p>	<p>□ Goal: to construct a public-key cipher $\Pi = (\text{KeyGen}, \text{Enc}, \text{Dec})$ from T over the plaintext space $\mathcal{M} = \mathcal{X}$</p> <ul style="list-style-type: none"> ❖ Algorithm $\text{KeyGen}(n)$ <ul style="list-style-type: none"> ➤ $\text{Gen}() \rightarrow (pk, sk)$ ➤ Set pk as the encryption key and sk as the decryption key ❖ Algorithm $\text{Enc}(pk, m): m \in \mathcal{X}$ <ul style="list-style-type: none"> ➤ Compute $f_{pk}(m) = y$ ➤ Output y as the ciphertext ❖ Algorithm $\text{Dec}(sk, c)$ <ul style="list-style-type: none"> ➤ Compute $\text{Inv}_{sk}(c) = x$ ➤ Output x as the plaintext
--	--

So, let us begin our discussion with a generic framework regarding how we construct a Public-Key Cryptosystem from any One-Way Trapdoor Permutation. So, just to recall what is a One-Way Trapdoor Permutation scheme is. It is a collection of 3 algorithms. We have a Parameter Generation algorithm, which outputs a public parameter and a secret parameter, which you can call as public key and secret key.

Then, the function f is a two-input function. It takes the public key pk , and the input from the domain fancy \mathcal{X} and it gives you an output from the set fancy \mathcal{X} and it is always a Deterministic algorithm. On the other hand, an inverse function, it is a two input a function, taking the secret key sk , and input from the co-domain, which you want to invert, and it gives you an output from the domain of the function f . And also this is a deterministic algorithm.

We need two properties from any One-Way Trapdoor Permutation Scheme. The first property is the Correctness property, which requires that for every pair of keys output by the Parameter Generation algorithm and for every input x , if you compute the value of $f(x)$, and then you compute the inverse of that function with respect to the secret key sk , then you should get back the output x .

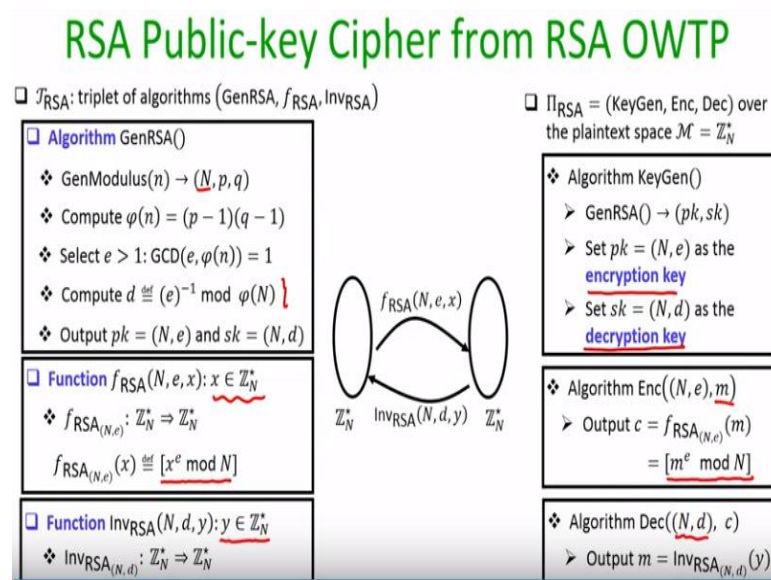
The second property which is the Security property is the one-wayness requirement. It demands that your function f_{pk} should be a one-way function against a computationally-bounded adversary even if the adversary knows the description of the public key and steps of your function f .

Now, given such One-Way Trapdoor Permutation Scheme, our goal is to construct a public-key Cryptosystem, which has a Key Generation algorithm, encryption algorithm, and decryption algorithm over the plaintext space, namely the set fancy X , and this is done as follows: the Key Generation algorithm of your encryption process will be basically to run the Parameter Generation algorithm of your Trapdoor Permutation Scheme and obtain the public key and secret key (pk, sk) , and we set the pk as the encryption key, and we set sk as the decryption key.

To encrypt a plain text m with respect to the public key pk , what we do is, we just compute the value of the function f with the key pk on the input m . So, sorry for the typo here. This x should be m because we want to encrypt the plain text m . So this is a typo here, so if we want to encrypt a plain text m , then we evaluate the value of the function f using pk as the public key on the input m , and the resultant output, which I denote as y is considered as the cipher text.

On the other hand, if we want to decrypt a cipher text using a secret key sk , then what we have to do is, we have to call the inverse function of the Trapdoor Permutation Scheme on the input c using the secret key sk to recover back the plaintext m .

(Refer Slide Time: 04:24)



So, let us instantiate this framework using the RSA One-Way Trapdoor Permutation. So, remember in the last lecture, we have seen that, RSA Permutation can be considered as a One-Way Trapdoor Permutation Scheme and the details of the RSA Trapdoor Permutation Scheme are as follows. The Parameter Generation algorithm for the RSA Trapdoor Permutation is as

follows. It runs the GenModulus algorithm and outputs primes p and q of size m -bits and modulus N , where N is the product of p and q .

Then it computes the value of $\varphi(N)$, namely the size of the set Z_N^* where $\varphi(N) = (p-1)(q-1)$. Then it selects $e > 1$ such that e is co-prime to $\varphi(N)$. And since e is co-prime to $\varphi(N)$, we can compute the multiplicative inverse of e modulo $\varphi(N)$, which we denote as d . Once we compute all these values, then the public key pk is said to be (N, e) whereas the secret key sk is said to be (N, d) .

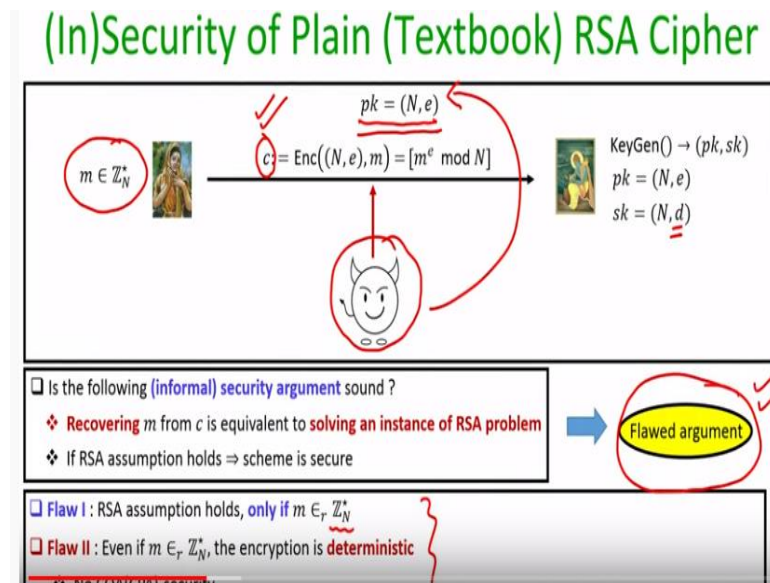
The RSA function, which takes an input x from the set Z_N^* is computed as follows. To evaluate the values of the RSA function using the public key (N, e) , we basically compute our output x^e modulo N , and x^e modulo N will also be an element of Z_N^* . On the other hand, if you want to invert the RSA function on some input $y \in Z_N^*$ using the secret keys (N, d) , then we basically output y^d modulo N .

So, that is what is the RSA One-Way Trapdoor Permutation Scheme assuming the RSA problem is difficult to solve with respect to the GenRSA function, then we have proved, we had seen in the last lecture that we can use this RSA Trapdoor Permutation as a candidate One-Way Trapdoor Permutation. Let us instantiate the framework that we had seen in the last slide by using the RSA Trapdoor Permutations.

So we get what we call as RSA Public-Key Cryptosystem over the plaintext space Z_N^* . So, the Key Generation algorithm for the RSA Public-Key Cryptosystem is we run the GenRSA algorithm of the RSA Trapdoor Permutation, and we set (N, e) to be the encryption key and we set (N, d) as the decryption key. So, the receiver will know (N, d) to decrypt a cipher text whereas the public key (N, e) will be available in the public domain.

If there is a sender, which wants to encrypt a plaintext m using the public key (N, e) then it has to basically compute the RSA function on the input m , which is nothing but the computing the value of m^e modulo N . On the other hand, if a receiver receives cipher text c , then using the secret key (N, d) , it can decrypt a cipher text c . It has to basically compute the inverse RSA function and output y^d modulo N . So, the correctness of this RSA cryptosystem follows from the correctness of the generic framework that we have discussed in the last slide.

(Refer Slide Time: 07:33)



Now, let us focus on the security of the RSA cipher that we have just constructed and imagine that we have a receiver, which runs the Key Generation algorithm of the RSA cipher that we have discussed, obtains the public parameter (N, e) and a secret parameters (N, d) , and it makes its public key (N, e) available in the public domain, and say there is a sender, which has a plain text $m \in \mathbb{Z}_N^*$ which it wants to communicate to the receiver.

Then as per the syntax of the RSA Encryption Scheme, it computes a cipher text, which is m^e modulo N and cipher text is communicated to the receiver and assume we have a polytime or computationally bounded eavesdropper who has eavesdrop the cipher text. Now, one might be wondering that whether the following security argument is sound or not.

We can say that if there is an adversary who has eavesdrop the cipher text c and its goal is to recover the underlying plain text m , which is encrypted in c . Then basically it has to solve an instance of RSA problem because this adversary only knows that description of (N, e) and he does not know the underlying m , and he does not know the secret key d . So, recovering m from c is equivalent to solving an instance of RSA problem.

And hence if you assume that the RSA assumption holds with respect to the GenRSA algorithm that the receiver has run, then we can argue that this scheme should be secure, but it turns out that this informal argument is completely flawed here. There is intuition that recovering the underlying plain text from the cipher text is equivalent to solving an instance of RSA problem is completely a flawed argument and there are many reasons for doing that.


There are many reasons behind this flaw. The first major flaw is that if you look closely into the RSA assumption, it holds only if the underlying m , which is encrypted in c , $m \in_r \mathbb{Z}_N^*$, which may not be the case in practice. In practice, sender might have any kind of plaintext, $m \in \mathbb{Z}_N^*$, which might be encrypting and producing the cipher text c whereas for RSA assumption, we need that underlying $m \in_r \mathbb{Z}_N^*$.

The second flaw here is that, even if you assume for the moment that sender is encrypting messages, $m \in_r \mathbb{Z}_N^*$, and hence the RSA assumption hold, the whole encryption process that the sender is using now is deterministic. There is no internal randomness, which is there as part of the encryption algorithm. That means, if the same sender wants to encrypt the same message m multiple times using the same public key (N, e) , then it will end up getting the same cipher text c . Throughout this course, we have rigorously analyzed that how unsafe it is to use a deterministic encryption process. We can never hope that an encryption process can be CPA-secure if we are using the deterministic encryption process. These are the two security flaws, which are there with the so-called RSA cipher that we have designed.


(Refer Slide Time: 10:51)

More Attacks on Plain RSA Cipher

$pk = (N, e)$


 $m \in \mathbb{Z}_N^*$

$c := [m^e \bmod N]$


 $KeyGen() \rightarrow (pk, sk)$
 $pk = (N, e)$
 $sk = (N, d)$

❑ Assume we use plain RSA cipher for **encrypting random messages** from \mathbb{Z}_N^* (thus RSA assumption holds) and we aim to **not achieve indistinguishability-based (semantic) security**

❑ **Encrypting short messages using small encryption-exponent e**

- ❖ Several **practical instantiations** of RSA cipher set $e = 3$, to ensure that **encryption process is fast**
- ❖ Consider an application where $m \in \mathbb{Z}_N^*$, but $m < N^{1/e}$
 - **Hybrid RSA encryption**: $||N|| \approx 1024$ bits, $e = 3$ and $m \in_r \{0, 1\}^{300}$
- ❖ $c := [m^e \bmod N] = m^3 \dots$ **integer cube** and not modulo N cube

It turns out that apart from this short coming that your RSA encryption process is deterministic, we can show several attacks on the Plain RSA Cipher. Assume for the moment, we are using the RSA Cipher where the sender wants to encrypt random $m \in_r \mathbb{Z}_N^*$, and we do not want a semantic security, namely, we do not aim for indistinguishability-based security. It is suffice for us if adversary does not learn the entire plaintext by eavesdropping upon the cipher text. That is a notion of secrecy that we are aiming for right now, but for the moment assume that.

It turns out even if we go for this weaker notion of secrecy, namely all or nothing kind of secrecy, then there are several attacks which can be still launched on the Plain RSA Cipher. So, the first class of attacks is basically what we call as encrypting short messages using small encryption-exponent e .

So, it turns out that for many practical instantiations of RSA, we set the public exponent or the encryption exponent e to be 3 to ensure that our encryption process is fast. Because if you see the encryption, encryption is basically computing the e th power of plaintext modulo N , and if I set e to be 3, then my encryption process will be very fast. It is fine that if I set e to be 3, then my d will be very large and my decryption process running time will be slow. But we can have some kind of tradeoff and for most of the many practical instantiation, this is a common choice, which is used to make the encryption process fast.

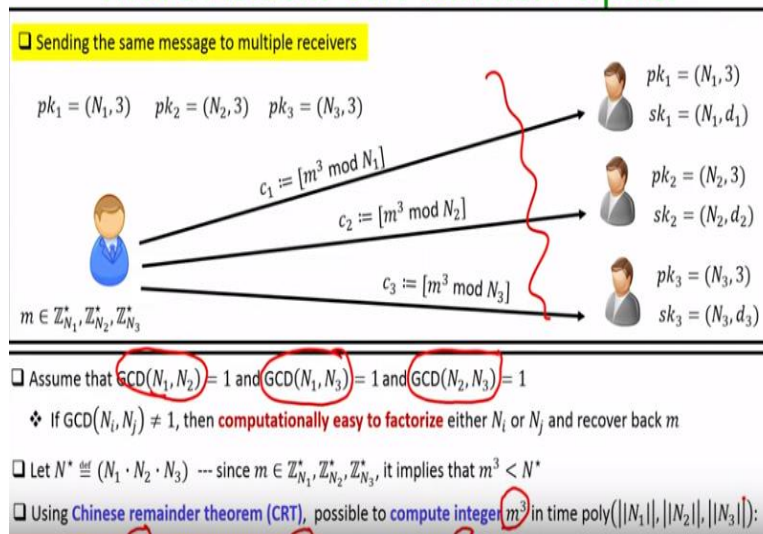
Now, assume we are using an instantiation of RSA, where we have set e to be 3, and say we are consider an application where the underlying messages $m \in_r \mathbb{Z}_N^*$, but the magnitude of m is strictly less than the e th root of the modulus N . And if you are wondering that what kind of applications encounters such kind of m , if later on we will see the Hybrid RSA encryption. Then the size of the modulus that we typically use is of size 1024 bits, and the message that we would like to encrypt to the Hybrid RSA encryption will be roughly a 300-bit uniformly random bit string and we will set e to be 3.

If that is the case, then the cipher text will be m^3 modulo N . But since my magnitude of m is strictly less than the cube root of N , m^e or m^3 modulo N will be equivalent to the integer cube and the effect of mod N will not take place at all. And if adversary is aware of the fact that we are using an application with underlying plain text, its magnitude is strictly less than the cube root of modulus N , then it also will be aware that the underlying m , which has been encrypted.

Its cube is actually contained in the cipher text c , namely the integer cube not the modulo cube, and it is very easy to recover the underlying unknown m from the c by just finding the integer cube root of the cipher text c , which can be computed in poly of the size of modulus N amount of computation. So that is our first class of attack.

(Refer Slide Time: 14:14)

More Attacks on Plain RSA Cipher



We can have many other kinds of attack possible on Plain RSA Cipher, so imagine for the moment that we are considering an application where the same message needs to be encrypted and sent to multiple receivers. So, imagine if we say we have 3 receivers, receiver 1, 2, and 3 each of it has his own modulus N_1 , N_2 , and N_3 respectively.

But each of them is using a same public exponent say 3, and of course a different decryption exponent d_1 , d_2 , and d_3 where d_1 is the multiplicative inverse of 3 modulo N_1 , d_2 is the multiplicative inverse of 3 modulo N_2 and so on. So, the public key, which is available in the public domain are the respective public keys of the 3 receivers and imagine that we have a sender, which has a random message $m \in_r \mathbb{Z}_{N_1}^*$, as well as $\mathbb{Z}_{N_2}^*$, as well as $\mathbb{Z}_{N_3}^*$ randomly chosen and say it wants to encrypt and send a same message to the 3 receivers.

So, it will compute c_1 which will be the m^3 modulo N_1 , c_2 which will be the m^3 modulo N_2 and so on. Now, assume that this modulus N_1 , N_2 , and N_3 , they are pair-wise prime. If not suppose for instance, the GCD of N_i and N_j is not 1, they are not co-prime to each other. That means there exist a common factor of N_i and N_j , then using that common factor, it is computationally easy to factorize either the modulus N_i or N_j and say for instance if N_i is factorizable, and since e_i is also publicly known and if we know N_i 's factors, we can compute the value of $\varphi(N_i)$.

And once we know $\varphi(N_i)$ and e_i , we can compute the value of d_i in polynomial amount of time. If d_i is computable, then any encryption which is intended for the i^{th} receiver can be decrypted by the adversary. So, it is safe to assume for the moment that pair wise, all the modulus N_1 , N_2 , and N_3 , they are co-prime to each other.

Now, let me define a bigger modulus N^* , which is the product of all the three modulus here, namely N_1 , N_2 , and N_3 , and since my underlying plaintext m , which is the sender has encrypted $\in Z_{N_1}^*$, as well as $Z_{N_2}^*$, as well as $Z_{N_3}^*$. That means m is strictly less than N_1 , it is strictly less than N_2 , it is strictly less than N_3 . That means the integer m^3 will be strictly less than the bigger modulus N^* and now what I can do is since my modulus N_1 , N_2 , and N_3 and N_1 , N_3 they are co-prime to each other.

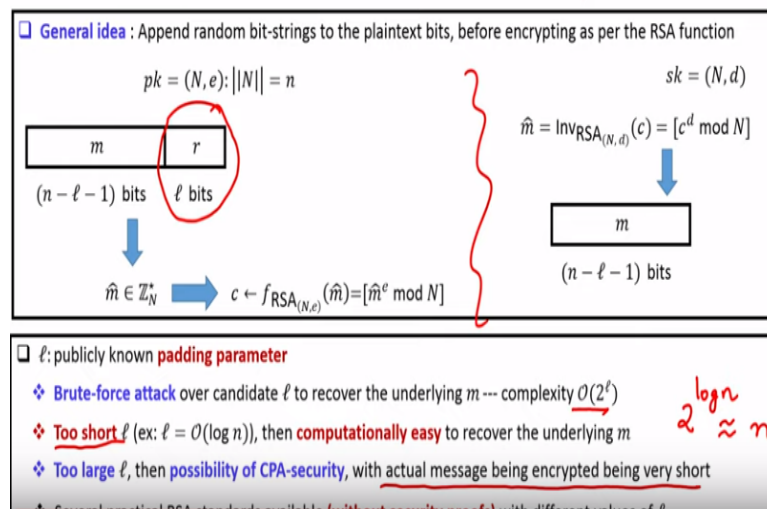
I can invoke a very nice result from the Number Theory, which we call as the Chinese Remainder Theorem, which says that if your individual modulus N_1 , N_2 , and N_3 they are co-prime. Then it is possible to compute the integer value m^3 in polynomial amount of time, polynomial in the size of the respective modulus N_1 , N_2 , and N_3 such that the recovered m^3 basically satisfies the system of following modular equations.

So, you are given the value c_1 , which is some integer m^3 modulo N_1 , where m^3 is not known, and you are given c_2 , which is same unknown m^3 modulo N_2 and you are given the value of c_3 , which is the same unknown m^3 modulo N_3 , and if your N_1 , N_2 , and N_3 , the respective modulus they are co-prime to each other. Then what basically the Chinese Remainder Theorem says is it is possible to recover that unknown m^3 in polynomial amount of computation.

Then, by applying the Chinese Remainder Theorem, an adversary who has eavesdrop these three cipher text c_1 , c_2 , c_3 , can apply the Chinese Remainder Theorem, and recover the unknown m^3 . Once the unknown integer m^3 is recovered, it can find out the exact m . So, again this is another attack, which is possible on the Plain RSA Cipher even if we assume that a sender's message $m \in Z_N^*$ set. It turns out that there could be several other possible attacks, which can be launched on the Plain RSA encryption process.

(Refer Slide Time: 18:48)

Padded RSA Public-key



So, this brings us to the concept of what we call as padded RSA public-key cryptosystem and basically the general idea here is that, we have seen already that the Plain RSA encryption process is a deterministic encryption process. There is no internal randomness hidden and if there is no internal randomness, then we can never hope to achieve CPA security.

So, the idea behind the padded RSA encryption process is that we try to append some random bit-strings to the plaintext bits, which we want to encrypt and convert everything into a group element in the set \mathbb{Z}_N^* , and then we apply the RSA function for encryption. On the other hand, the decryption end, we will chop off the random bit strings, which we have added to the plain bit strings before encryption and then that will be the recovered plaintext.

So, pictorially what we are trying to do here is the following. Imagine that a receiver has done the public key setup. It has made its public key (N, e) publicly available where the size of the modulus N is n -bits. Then, the idea here is that the padded RSA encryption process, it will take another parameter l and the size of the plain text, which is padded RSA encryption process will support will be of size $n-l-1$ bits.

That means, you can encrypt any message in the set $\{0,1\}$, raise to the power $n-l-1$ bits, and the remaining l bits are reserved for randomness. Before encryption, imagine you have a message m , along with that you pad l bits of randomness, which will be uniform randomness and you will be left with 1 more bit, which we by default keep it as 0. If we do padding like that, and if we ensure that the resulted bit string, which will be of now length l bits, which we denote as \hat{m} , we interpret it as an element of \mathbb{Z}_N^* .

So, it might be possible that the resultant m concatenated with the randomness that we have appended, may not lead you to an element of Z_N^* and if the resultant element is not an element of Z_N^* , then it might look like that the RSA encryption function and RSA decryption function, they need not be mutually an inverse function of each other.

However, it turns out that even if the resultant \hat{m} is not an element of Z_N^* , but it $\in Z_N$. It is suffice for us to apply the RSA function, but the encryption end and to apply the RSA decryption function at the decryption end, the Correctness property will still hold if the padded m when interpreted as a group element does not belong Z_N^* , but rather belong to Z_N . The Correctness property that is there with respect to the RSA forward direction function and inverse function will still hold.

We will prove that at the end of this lecture, but for the moment assume that the padded message N , which I denote by \hat{m} is an element of Z_N^* . Then once we have \hat{m} , we encrypt the message \hat{m} or the group element \hat{m} , we apply the RSA function, compute \hat{m} raise to the power e modulo N . At the decryption end, the receiver who has the secret key (N, d) , it will apply that RSA inverse function, namely it will compute c^d modulo N and it will recover the group element \hat{m} .

And since it knows the value of l , l here is a system parameter, it will be known both to the sender as well as the receiver and it will be publicly available. So, what the receiver can do is, it can parse the recovered \hat{m} as 0 followed by a plain text of length $n-l-1$ bits followed by a randomness of l bits and it can simply ignore the randomness part. And the remaining $n-l-1$ bits is considered as the recovered plain text. So, as I said here, l is the publicly known public padding parameter, which is available both to the sender, receiver, and to any third party.

Now, it turns out that an adversary who is aware of the value l can launch a Brute-force attack over the amount of randomness or over the actual randomness, which has been padded at the sender's end before encrypting the message. Complexity of the Brute-force attack is basically $\mathcal{O}(2^l)$ because there could be 2^l number of candidate values of r , and overall those candidate randomness adversary has to perform a computation of order 2^l .

That automatically implies that if your l is too short, that means the amount of randomness slot that is available in your version of padded RSA is too short. For instance, it is of order \log of l , then it is computationally easy to recover the underlying m because a Brute-Force of the order $2^{\log n}$ is equivalent to a computation of order n , which is polynomial in the security parameters. So too short size of l is dangerous.

On the other hand, if you reserve a huge amount of slot for the randomness, but little slot for the actual plain text, which you want to encrypt, then you can hope for a possible CPA security for the resultant instantiation of the padded RSA. But the disadvantage will be that the actual message that we are going to encrypt, namely the actual bit string of length $n-l-1$ will be very short.

So, we have a kind of trade-off here and it turns out that there several practical RSA standards, which are available in the literature with different values of l . But, unfortunately most of them do not have any kind of security proofs. However, we will use this blue print and we will later return to the CCA-secure versions of RSA public-key cryptosystem, where we will retain this blue print and they are also the idea will be that we will take the actual plain text, which is a bit string appended with some random pad.

Then convert everything either as an element of \mathbb{Z}_N^* , or an element \mathbb{Z}_N , and encrypt it as per the RSA function, and the decryption end, we do the reverse operation. The kind of padding that we use that will ensure that we get a CCA-secure version of RSA public-key cryptosystem.

(Refer Slide Time: 25:18)

Fermat's Little Theorem : RSA Message Space \mathbb{Z}_N

□ If p is a **prime**, then for every integer a , such that $[a \bmod p] \neq 0$, we have:

$$[a^{p-1} \bmod p] = 1$$

□ **Theorem:** Let $N = p \cdot q$, where p, q are **distinct primes** (hence $\phi(N) = (p-1) \cdot (q-1)$). Let e, d be integers, such that $[ed \bmod \phi(N)] = 1$. Then **for all** $x \in \mathbb{Z}_N$:

$$[x^{ed} \bmod N] = x \quad \approx \quad [x^{ed} - x] \text{ divisible by } N$$

❖ **Case I:** Let $[x \bmod p] = 0$

$$[x^{ed} - x] \text{ divisible by } p$$

$x^{ed} - x$ divisible by
distinct primes p, q

❖ **Case I:** Let $[x \bmod q] = 0$

$$[x^{ed} - x] \text{ divisible by } q$$

❖ **Case II:** Let $[x \bmod p] \neq 0$

$$[x^{p-1} \bmod p] = 1$$

❖ **Case II:** Let $[x \bmod q] \neq 0$

$$[x^{q-1} \bmod q] = 1$$

Finally, let me end this lecture with something related to the message space of Z_N and for that let me first state a property from the well-known theorem from Number Theory, which I will not prove, which we also call as Fermat's Little Theorem. What Fermat's Little theorem basically says that if p is a prime, then for every integer a , which is not divisible by p , namely for every integer a such that a modulo p is not 0, then a^{p-1} modulo p is 1. This holds for every integer a , which is not divisible by the prime p .

Now, what we are going to prove using this Fermat's Little theorem is that imagine if you have a modulus N , which is the product of two distinct primes p and q , that means your $\varphi(N) = (p-1) * (q-1)$. Say you have two integers e, d such that e and d are multiplicative inverse of each other modulo $\varphi(N)$. That means, ed modulo $\varphi(N)$ is 1.

Then what we are going to show is that for every x in the set Z_N , x^{ed} modulo N is going to give you x . Equivalently this means that $x^{ed} - x$ is completely divisible by N . That is what we are going to prove. It is easy to show that these two conditions are equivalent. If indeed, x^{ed} modulo N is x , that means, when I divide x^{ed}/N , I get the remainder x .

That means, if I subtract the remainder x from x^{ed} , then the resultant number will be completely divisible by N . Before I go into the proof of this theorem, what exactly is the significance of this theorem. The significance here is the RSA function and its corresponding inverse function will work even if I have an x , which is not a member of Z_N^* , but rather a member of Z_N .

That means if you go and see the padded RSA where we are padding the bit string, which we want to encrypt with a random pad and trying to interpret it as an element of Z_N^* , I said that it may not be an element of Z_N^* , because to be an element of Z_N^* , the resultant element should be co-prime to your modulus N , but that need not be the case here. Your m^{\wedge} need not be your co-prime to N , but that is fine.

This theorem says that even if you encrypt such m^{\wedge} using the RSA function and try to decrypt it using the RSA function, you will get back the right m^{\wedge} . Of course, remember that we have reserved the first bit in the padded RSA to be 0, which automatically ensures that m^{\wedge} is definitely less than the modulus N and hence it is an element of Z_N . So, let us proceed to prove this theorem.

So we take case 1, so what basically we are going to do in this proof is we are going to prove certain properties with respect to x modulo p , and symmetrically we are going to prove similar properties with respect to x modulo q , and then we will combine these two properties to arrive at the proof of this theorem.

So, let us first consider case 1, where we are going to see the properties of x modulo p . And we can have two subcases here, if x modulo p is 0, that means if x is completely divisible by p , then so is x^{ed} , and if x^{ed} is divisible by p and if x is also divisible by p , then it implies automatically that $(x^{ed} - x)$ is also divisible by p . So, that is sub case 1 and sub case 2 here is when x modulo p is not zero, that means x is not completely divisible by p .

And in this case, I can invoke my Fermat's Little theorem, and I can say that x^{p-1} modulo p is 1, and now what I am going to do is, I am going to use the fact that ed modulo $\varphi(N)$ is 1. Remember, my ed modulo $\varphi(N)$ is 1, and also my $\varphi(N) = (p-1) * (q-1)$. That means, I can rewrite my $e*d = k * (p-1) * (q-1) + 1$.

Because ed gives the remainder 1 when divided by $\varphi(N)$, and now what I can do is if I want to find out the value of x^{ed} modulo p , I know that $e*d = k * (p-1) * (q-1) + 1$. So, I can rewrite x^{ed} , modulo p as x^{p-1} whole raise to the power K times $q-1$, and this $+1$ will contribute another x modulo p , so that is what x^e modulo p will be and I know that x^{p-1} modulo p is 1.

That means this thing is nothing but 1, and 1 raise to the power K times $q-1$ is going to give me 1 only. That means, this entire first bracket is going to give me 1, so that means I can say that x^{ed} modulo p is nothing but x modulo p . That means with respect to x modulo p in subcase 1, if x is already divisible by p , then I can say x^{ed} minus x is completely divisible by p .

And in the second sub case also, I am establishing that even if x modulo p is not 0, I have established that x^{ed} modulo p , is same as x modulo p . So, that means both x^{ed} and as well as x gives us the same remainder on divided by p , and as a result I can say that their difference, namely $x^{ed} - x$ is completely divisible by p . It does not matter whether x is divisible by p or not, I have established here that $x^{ed} - x$ is completely divisible by p .

And symmetrically, I can do the same argument for x modulo q as well. So, with respect to x and q , I have 2 subcases. If x is already divisible by q , then so is x^{ed} , and hence I can easily conclude that $x^{\text{ed}} - x$ is completely divisible by q . On the other hand, for the case when x is not divisible by q , I can apply Fermat's Little Theorem here, and I can say that x^{q-1} modulo q is 1.

And then using the similar argument that I have used for the case of module x and p , namely I can use the fact that $e \cdot d = k \cdot (p-1) \cdot (q-1) + 1$, and then I can rewrite x^{ed} as x^{q-1} whole raise to the power k times $p - 1$, multiplied by x modulo q . I know that since x^{q-1} , I know that overall it turns out to only x modulo q , that means that x^{ed} , as well as x gives me the same remainder on divided by q .

And hence I can say that $x^{\text{ed}} - x$ is completely divisible by q . So, these are the properties that I have established now, that means I have shown that $x^{\text{ed}} - x$ is individually divisible by prime p as well as by prime q . Since my primes p and q are distinct, it automatically implies that $x^{\text{ed}} - x$ is also going to be divisible by N , because N is my product of p and q and that establishes this theorem.

(Refer Slide Time: 33:01)

Using Chinese Remainder Theorem (CRT) for Fast RSA Encryption and Decryption

- ❑ Both RSA encryption and decryption requires **modular exponentiation**

$$f_{\text{RSA}_{(N,e)}}(x) \equiv [x^e \bmod N] \quad \text{Inv}_{\text{RSA}_{(N,d)}}(y) \equiv [y^d \bmod N]$$

$$\left. \begin{array}{l} f_{\text{RSA}_{(N,e)}}(x) \equiv [x^e \bmod N] \\ \text{Inv}_{\text{RSA}_{(N,d)}}(y) \equiv [y^d \bmod N] \end{array} \right\} c \cdot n^3, \text{ where } |N| = n$$
- ❑ **Time complexity** for computing modular exponentiation modulo an ℓ -bit integer $\sim c \cdot \ell^3$
- ❑ **(Chinese Remainder Theorem)**: Let $N = p \cdot q$, where p and q are **primes**

$\mathbb{Z}_N^* \xleftrightarrow{\text{CRT}} \mathbb{Z}_p^* \times \mathbb{Z}_q^*$

Bijection

$[y^d \bmod N] \leftrightarrow (y_p, y_q)$

❑ **(Number Theory)**: $\text{GCD}(y, p) = \text{GCD}(y, q) = 1$

❖ $[y^d \bmod p] = [y^{d \bmod (p-1)} \bmod p] = y_p$

❖ $[y^d \bmod q] = [y^{d \bmod (q-1)} \bmod q] = y_q$

❑ Using CRT, $y^d \bmod N$ can be computed in $\frac{c \cdot n^3}{8} + \frac{c \cdot n^3}{8}$

Handwritten notes on the slide:

- $N = p \cdot q$
- $|N| = n$
- $|N| \approx \frac{n}{2}$

Now, let us see how we can apply the Chinese Remainder Theorem for fast RSA Encryption and Decryption because this is a common trick which we use during the real-world instantiation of RSA function. So, if you see the RSA encryption function and decryption function, then both of them requires modular exponentiation. To encrypt, we need to perform x^e modulo N , and to decrypt we need to perform cipher text raise to the power c^d modulo N .

Now even though we have not yet discussed the best-known algorithms for computing modular exponentiation, if time permits, we will discuss one of such algorithms in our subsequent lectures. It turns out the best-known algorithm for performing modular exponentiation modulo l -bit integer requires amount of time, which is c times l^3 , where c is some constant.

That means, if my modulus N is of size n bits, then to perform both RSA encryption as well as decryption function, I need to perform c times n^3 operation. Now, what Chinese Remainder theorem says is since N is a product of 2 primes p and q , where the two factors p and q are co-prime to each other, then basically the Chinese Remainder Theorem gives you a bijection namely a one-to-one, on-to mapping from the set Z_N^* to the Cartesian product of the set Z_p^* and Z_q^* .

Remember Z_N^* is basically the set of all integers in the range 1 to $N-1$, which are co-prime to N . In the same way Z_p^* is basically the set 1 to $p-1$. Basically, the set of integers $\{1, \dots, p-1\}$ which are co-prime to p , but since p is prime, the set Z_p^* is nothing but the entire set $\{1, \dots, p-1\}$. And in the same way, the set Z_q^* is the entire set $\{1, \dots, q-1\}$. So, what a Chinese Remainder Theorem does is, it defines a very nice bijection from the set $Z_N^* \rightarrow Z_p^* \times Z_q^*$, where any element $a \in$ the set Z_N^* , it is mapped to the following pair of elements.

You take the remainder of a modulo p , and the remainder of a modulo q . That will be the representation of the element $a \in Z_N^*$ in the set Z_p^* and the set Z_q^* respectively. And not only that what Chinese Remainder Theorem basically shows is, tells you that if you are performing some computation in the set Z_N^* , say if you are evaluating the value of $f(a)$ modulo N , then the same computation you can carry over individually Z_p^* and Z_q^* and you will get a representation of the result of $f(a)$ modulo N .

That means instead of performing $f(a)$ modulo N , you can perform $f(a)$ modulo p and $f(a)$ modulo q , and that resultant pair of elements can be considered as the representation of your result $f(a)$ modulo N . So, that is how you can interpret this Chinese Remainder Theorem bijection.

That means, if I consider a decryption function for the moment, I am assuming that my public exponent e is 3, that takes care of the encryption part of the RSA function. RSA encryption process will be very fast, if I set my e to be 3, but my problem will be my decryption. I want

to speed up my decryption process. Since, my decryption function requires me to compute y^d modulo N , by applying the Chinese Remainder Theorem the element y^d modulo N , which is an element of Z_N^* can be interpreted as a pair of elements, one from Z_p^* and another from Z_q^* , where the representation from Z_p^* will be y^d modulo p and representation from Z_q^* will be y^d modulo q .

Now, since y is relatively prime to p , as well as relatively prime to q , and if indeed that is the case, it turns out that we can prove that y^d modulo p is same as $y^{(d \bmod p-1)}$ modulo p . That means, in the exponent, I can reduce my exponent, d to $d \bmod p-1$, and I call this value to be y_p , and the same holds for y to the power $d \bmod q-1$ as well. That is, in the exponent, I can reduce the exponent e from d to $d \bmod q-1$ and let me call this value to be y_q . That means, this y^d modulo N can be written as or expressed as the pair of elements y_p, y_q as per the Chinese Remainder Theorem.

That means, if you know how to compute y_p and y_q , then using the Chinese Remainder Theorem, you can obtain the result y^d modulo N . That is what Chinese Remainder Theorem ensures for you. Now let us try to analyse how fast it is to compute y_p and y_q . Before going into that, let me tell you that since d is known to the receiver who is going to run the decryption algorithm, and $p-1$ is also known to the receiver because it knows the factors p and q .

So $d \bmod p-1$ can be pre-computed by the receiver and can be stored once for all, and same holds for $d \bmod q-1$ as well. So d and q are also known to the receiver or to the person who is going to run the decryption algorithm and hence $d \bmod q-1$ can be pre-computed in advance by the receiver.

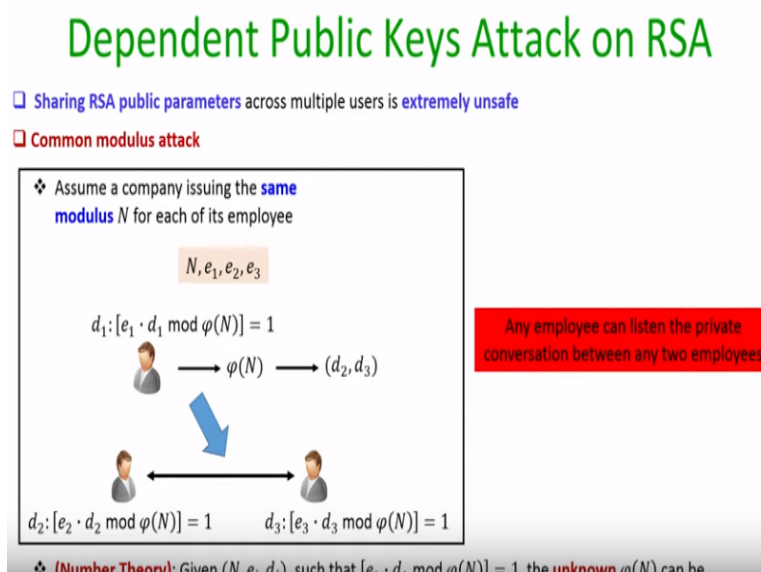
Now, it turns out that using the Chinese Remainder Theorem, computing y^d modulo N requires you to perform two times $(c * n^3)/8$ or $n^3/4$ times computation. This is because, while N is the product of two primes p and q , and my size of N is roughly of n bits, since N is the product of p and q , that means the size of prime factors p and q that we are using here is roughly of size $n/2$ bits each. That means, to compute y to the power $d \bmod p-1$ modulo p or the value y_p , I need to perform a modular exponentiation, where the size of the modulus is n over two bits.

And by using the best known algorithm that will require me to perform computations of order c times $n^3/8$. And similarly to compute y_q , I need to perform computation of order c and $n^3/8$,

If I sum these two things, I get that the total computation that I need to perform to compute y_p and y_q will be two times $(c * n^3)/8$, which is basically c times $n^3/4$, and this is simply 25% saving compared to the naïve way of performing or calculating y^d modulo N directly.

If instead of computing y^d modulo N directly, we use this indirect approach where we compute y_p and y_q , in Z_p^* , and Z_q^* . Then combine it to obtain the result y^d modulo N , basically end up saving 25% of computation and this is a very common thing that is used in the practical instantiations of the RSA decryption algorithm.

(Refer Slide Time: 40:25)



Now, let me end this lecture by showing you some more attacks on RSA, namely I want to show you that how dangerous it could be to share public parameters across multiple users in the context of RSA cryptosystem. So, I am going to show you one of the simplest possible attack, which we call as common modulus attack, there are other sophisticated attacks, which are also possible.

But I am not going to discuss them, and this is kind of very different compared to El Gamal encryption scheme. We have seen in the context of El Gamal encryption scheme, sharing public parameters, namely sharing the description of the group, description of the generator across multiple receivers, it is fine. That means If multiple receivers use the same cyclic group to instantiate their El Gamal encryption scheme, the same generator to instantiate your El Gamal encryption scheme and so on.

But when it comes to RSA, it might be extremely unsafe to share public parameters across multiple users. So, the attack here is as follows. Assume a company, which is using the same modulus N for each of its employee. That means, every time a new employee joins the company, you can imagine that the company retains the same modulus N for the employee, but what it does is, it generates a fresh independent public exponent e_i for that user, corresponding decryption d_i for that user such that (e_i, d_i) are co-prime modulo N .

So for the moment, assume we have 3 employees in the organization, the first employee has a secret decryption key d_1 . The second employee has its secret decryption key d_2 , and in the same way, the third employee has a secret decryption key d_3 , and we know that property wise e_i and d_i , they are multiplicative inverse of each other modulo $\varphi(N)$. Now, it turns out that again by using a well-known fact from Number Theory, we can prove that, if we are given the modulus N , and (e_i, d_i) such that e_i and d_i are multiplicative inverse of each other.

So I beg your pardon here, it should not be e_1 times d_1 here, it should be e_i times d_i , that is a typo here. So, if you know the modulus N , if you know the value of e_i , and we know the value of d_i such that e_i and d_i are multiplicative inverse of each other modulo $\varphi(N)$ where $\varphi(N)$ is not known to us, then we can compute a value of $\varphi(N)$ in poly of size of modulus N time.

That means, for instance employee 1 is considered here. Since it knows its public exponent e_1 , as well as decryption exponent d_1 , but it does not know the unknown value of $\varphi(N)$. By using this fact from the Number Theory, and using that algorithm, it can compute in polynomial amount of time, the value of $\varphi(N)$. Once $\varphi(N)$ is computed by the employee 1, since it knows the value of e_2 , namely the public exponent public key of the second employee, and it knows the modulus N , then by using the same result from Number Theory, it can compute d_2 .

And in the same way it can compute the value of the decryption exponent of the third employee as well. And once it knows the value of d_2 and d_3 , then any private communication which is happening between the second and the third employee using RSA encryption scheme, even if it is a padded RSA encryption scheme, which is randomized for the moment assume that, since the value of d_2 and d_3 is known completely to the employee 1, it can completely find out what exactly is happening between employee 2 and 3.

So, that shows that how unsafe it might be if we end up having multiple users having the same modulus N . It is highly, highly dangerous. So, that brings me to the end of this lecture.

Just to summarize, in this lecture, we have seen that how we can instantiate a public-key encryption scheme from any One-Way Trapdoor Permutation. And we have shown that how we can instantiate using the RSA Trapdoor Permutation, and get what we call as Plain RSA Cipher, but Plain RSA Cipher cannot be used in practice because first of all it is not randomized, and secondly we cannot directly reduce its security to the RSA problem. We have seen how we can make RSA randomized encryption process by doing padding, and we have seen that how we can speed up the RSA decryption function using Chinese Remainder Theorem. Thank you!