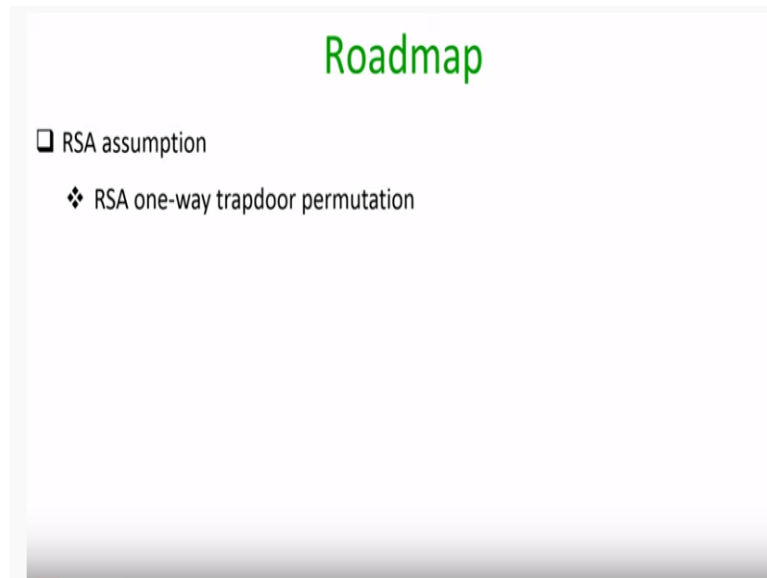**Foundations of Cryptography**
Dr. Ashish Choudhury
Department of Computer Science
Indian Institute of Science– Bangalore

Lecture – 45
RSA Assumption

Hello everyone, welcome to this lecture.

(Refer Slide Time: 00:42)

So just to recap in the last lecture, we have seen Candidate Public Encryption Scheme, CPS-secure Public Encryption Scheme, mainly El Gamal encryption scheme. In this lecture and the next lecture, we will discuss another Candidate Public Encryption Scheme based on the different heart problem, which we call as RSA assumption, which will lead us to another public encryption scheme namely the RSA Public Encryption Scheme.

(Refer Slide Time: 00:58)

The Set $\mathbb{Z}_N^*$

- $\mathbb{Z}_N^*$: set of integers modulo N, **coprime** to N

$$\mathbb{Z}_N^* \stackrel{def}{=} \{b \in \{1, \dots, N-1\} : \gcd(b, N) = 1\}$$

- Ex: $\mathbb{Z}_{10}^* = \{1, 3, 7, 9\}$
- If the modulus N is a **prime**, then $\mathbb{Z}_N^* = \{1, \dots, N-1\}$

- **Theorem (Number theory):** $(\mathbb{Z}_N^*, \cdot_N)$ constitutes a group

$(\mathbb{Z}_{10}^*, \cdot_{10})$

$\cdot_{10}$	1	3	7	9
1	1	3	7	9
3	3	9	1	7
7	7	1	9	3
9	9	7	3	1

- $O(poly(||N||))$-time algorithm exists for computing multiplicative inverse a^{-1} for any element $a \in \mathbb{Z}_N^*$

So to understand the RSA assumption, let us first try to understand the set ZN*. So ZN* basically is a set of integers modulo N, which are co-primed to the modulus N, namely ZN* is the collection of all elements b in the range {1,…,N-1} such that the element b is co-primed to the modulus N. Namely, their GCD is 1. For example, the set Z10* is nothing but the elements 1,3,7, and 9.

Because all these elements 1, 3,7, and 9, they are co-primed to the modulus 10, and it is easy to see that if modulus N is prime, then the set ZN* basically consist of all the elements 1 to N-1. So, a well-known fact from the number theory whose proof I am skipping is that the set ZN* along with the operation multiplication modulo N constitutes a group and satisfies all the axioms of the group.

So, for instance, I have drawn the matrix for the elements in Z10*, namely 1, 3, 7 and 9, and the result of performing the operation multiplication modulo 10 for any pair elements from the set, and now you can see that all the axioms of the groups are satisfied and that is why this is a group. Moreover, a well-known fact from the number theory is that we have polytime algorithms for computing the multiplicative inverse, which I denote by, a power -1, for any element a in the set ZN*.

And there is a well-known algorithm for that, which we call as Extended Euclid's GCD algorithm, which we will discuss in one of our subsequent lectures, and the running time of that algorithm is polynomial in the number of builds that we need to represent the modulus N. **(Refer Slide Time: 02:47)**

Euler's Totient (phi) Function

- $\varphi(N) \stackrel{\text{def}}{=} |\mathbb{Z}_N^*|$
  - ❖ Number of elements in the set $\{1, \dots, N-1\}$, which are coprime to N

- ❑ **Theorem (Number theory):**
  - ❖ If N is a prime p, then $\varphi(p) = p - 1$
  - ❖ If N is the product of distinct primes p and q, then $\varphi(N) = (p-1)(q-1)$

- ❑ Ex: $N = 10 = 2 \cdot 5$, so $p = 2$ and $q = 5$
  - ❖ $\mathbb{Z}_{10}^* = \{1, 3, 7, 9\}$, so $\varphi(10) = 4 = (2-1)(5-1)$

- ❑ **Theorem (Number theory):** for any $a \in \mathbb{Z}_N^*$, we have $(a^{\varphi(N)} \bmod N) = 1$
  - ❖ If N is a **prime** p, then for any $a \in \{1, \dots, p-1\}$, we have $(a^{p-1} \bmod p) = 1$
  - ❖ for any $a \in \mathbb{Z}_N^*$, we have $(a^x \bmod N) = (a^{x \bmod \varphi(N)} \bmod N)$

$$a^n = \boxed{a^{\varphi(N)}}^{1} \cdot \boxed{a^{\varphi(N)}}^{1}$$

$$\boxed{a^{x \bmod \varphi(N)}} \bmod N$$

So, let us next discuss Euler's Totient Function, phi function. So the Euler's Totient Function, which is the denoted by the symbol phi(N) is basically the cardinality of the set ZN*, namely it is the number of elements in the set 1 to N-1, which are co-primed to your modulus N. Some of the well-known facts again borrowed from Number theory are as follows. If N is a prime p, then the value of phi(p) is nothing but p -1.

Because there are exactly p-1 elements, which are co-primed to p. On the other hand, if n is the product of 2 distinct primes, p and q, then phi(N) is nothing but p-1 times q-1. That means, there are these many elements, namely p-1 times q-1 elements in the range {1,…., N-1}, which will be co-primed to your modulus N. To verify that, let us take an example, say N = 10, which you can write as the product of 2 and 5, where 2 and 5 are prime.

So your p is 2 and your q is 5, then we know that there are 4 elements in the set Z10*, so phi(10) should be 4, and 4 is nothing but p-1, namely 2-1 multiplied by q-1, which is 5-1, and finally another interesting property which we will use from the Number Theory is that, for any element a in the ZN*, a to the power phi(N) modulo N is 1. That means, if N is a prime p, then this phi(N) is nothing but p-1, so we get a to the power p-1 modulo p to be 1.

This property or this result is also called as Fermat's Little Theorem, and we also get due to this fact, the corollary that if you have any element a belonging to the set ZN*, then a to the power x modulo N is exactly the same as a to the power x modulo phi(N) modulo N. That means, in the exponent you can reduce the exponent x to x modulo phi(N). So the reason this
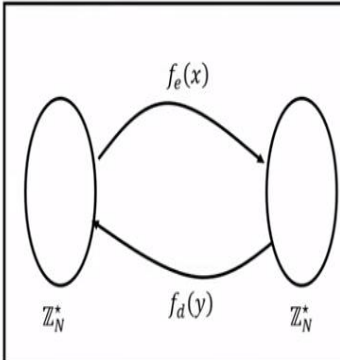
corollary issue is that, if your x is anyhow less than phi(N), then both L.H.S. and R.H.S. are same, but if your x is more than phi(N).

Then to compute a to the power x modulo phi(N), what you can do is, you can rewrite a to the power x as several batches of a to the power phi(N) in the exponent. A to the power phi(N) depending upon the relationship between x and phi(N), and the last batch will have a to the power x modulo phi(N) in the exponent, and everything in the base modulo N. Now, we know that a to the power of phi(N) modulo N is 1, that means each of these batches, you have full a to the power phi(N) modulo N.

We are going to give you the answer 1. So, all this batches are going to give you the answer 1, 1, 1,1 and 1 multiplied by 1 is going to give you 1. So, you are finally left with a last batch, which has a to the power in the exponent, x modular phi(N). So, that is how we get this corollary.

(Refer Slide Time: 06:03)

RSA Permutation

$f_e(x)$

$f_d(y)$

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

❏ $e > 2$, such that $GCD(e, \varphi(N)) = 1$

❖ $d \stackrel{\text{def}}{=} (e)^{-1} \bmod \varphi(N)$

❖ $(ed \bmod \varphi(N)) = (de \bmod \varphi(N)) = 1$

❏ $f_e(x): \mathbb{Z}_N^* \Rightarrow \mathbb{Z}_N^*$

❖ $f_e(x) \stackrel{\text{def}}{=} (x^e \bmod N)$

❏ $f_d(y): \mathbb{Z}_N^* \Rightarrow \mathbb{Z}_N^*$

❖ $f_d(y) \stackrel{\text{def}}{=} (y^d \bmod N)$

❏ Claim: The function $f_e(x)$ is a bijection (permutation) over the set $\mathbb{Z}_N^*$

❖ Consider arbitrary $x_1, x_2 \in \mathbb{Z}_N^*$ and let $y_1 = f_e(x_1)$ and $y_2 = f_e(x_2)$, such that $y_1 = y_2$

❖ $(x_1^e \bmod N) = (x_2^e \bmod N) \Rightarrow ((x_1)^{ed} \bmod N) = ((x_2)^{ed} \bmod N) \Rightarrow x_1 = x_2$

So, with all this mathematics in place, now let us try to understand the RSA permutation. So this RSA permutation is a mapping from the set ZN* to ZN*, so the way this permutation is defined as follows. Imagine, you are given an exponent e, which is greater than 2, such that the exponent e is co-primed to your value phi(N). So, I stress it is co-primed to phi(N), not to N. Now, a well-known fact from Number Theory is that if you have an e, which is co-primed to phi(N), then you can find out the multiplicative inverse of that e modulo phi(N).

In general, if you have 2 numbers, say a and b, such that GCD of a and b is 1, that means, a and b are co-primed to each other, then you can always find the multiplicative inverse of a modulo b. So, my modulus in this case is phi(N), because e is co-primed to phi(N), and if e is co-primed to phi(N), by using the Extended Euclid algorithm, I can find d where e and d constitutes their mutual multiplicative inverses.

That means, e times d modulo phi(N), and d times e modulo phi(N) is 1. Now, the RSA permutation is defined as follows. To go in the forward direction, the function is f sub e, and to compute the value of f sub e on a value x where x is a member of ZN*, we basically compute x to the power e and do mod N. On the other hand, the reverse direction function is basically a function of d, namely, f sub d(y), and this function is basically if I want to compute f sub d(y), I will compute y to the power d, and then perform mod N.

I claim here that the function F sub d is the inverse of the function f sub e and vice versa. So, for that consider any arbitrary x here, belonging to ZN*, and say I map it to y, as per the function f sub e, so my y is nothing but x to the power e, and now let's see what we obtain by reversing this value y as per the function fd. If I reverse this y as per fd, then I obtain x to the power e mod N, and then whole raise to the power d mod N.

So I can take the overall mod outside, and I obtain that this is the same as x to the power e times d modulo N, and by using the corollary that we have stated in the last slide, x to the power ed modulo N in the exponent, I can reduce this exponent e times d to e times d modulo phi(N). I know that e times d modulo phi(N), which is there in the exponent, gives me value 1, because I have chosen e and d such that they are multiplicative inverse of each other.

So, in the exponent, it is e times d modulo phi(N) reduces to 1, so hence I obtain that this x times ed modulo N is nothing but x modulo N, and since I have chosen my x to belong to the set ZN*. That means x is anyhow less than N, and if x is less than N, then x modulo N gives you x (()) (09:18) happens at all. So this prove that your function fd and fe are mutually inverse of each other.

I also prove here now that my function fe(x) is a bijection, and that means, it is a one-to-one onto mapping. For that, let us consider an arbitrary pair of inputs x1, x2, from the set ZN*, and say the resultant images of x1 and x2 as per the function f sub e, are y1 and y2. Now, if

y1 and y2 are same, that means, x1 to the power e modulo N, is same as x to the power e modulo N, that means, if I raise both the sides to the exponent d, I get x1 times x1 to the power ed modulo N is same as the x to the power ed modulo N.

This basically means that x1 is equal to x2, because x1 to the power ed modulo N, is nothing but x1 and x2 to the power ed modulo N is nothing but x2. So that means, if I take an arbitrary x1, x2 mapping to the same y, then basically I end up showing that x1 is equal to x2, and hence I overall get that my function fe is a permutation here.

(Refer Slide Time: 10:32)

Now, let us discuss Resulted Assumption, which will be related to the RSA assumption, which we will finally discuss. This is called a Factoring Assumption, and the intuition behind this assumption is that, if you are given the product of two large arbitrary prime numbers, it turns out that finding the prime factors is indeed a difficult task. This is a very widely known well-studied problem studied over several centuries.

And till now, we do not have efficient polytime algorithms for factorizing a product of two arbitrary prime numbers, if those prime numbers are chosen arbitrarily. So, this intuition or the hardness of this problem is going to be formalized by an experiment, and for that experiment, let us first define an algorithm, which we call as GenModulus algorithm. It basically picks two random n-bit prime numbers, say p and q, and there are well-known algorithms to pick random prime numbers.

I am not going into the exact details at how exactly you pick those two random prime numbers, and poly of n time. You can refer to the book by (()) (11:38) for the exact algorithm. Now, once p and q are chosen, you compute the modulus, namely N, which is the product p and q. Now, the Factoring Assumption, with respect to the GenModulus algorithm, is modeled as experiment, which we call as Factor Experiment, and the rules of that experiment are as follows.

The challenger runs the GenModulus algorithm, generates the parameter N, p, and q, and the challenge is thrown to the adversary, namely N, and the challenge for the adversary is to come up with its prime factorization, namely p and q. So, it submits a pair of numbers, p dash and q dash, and we say that the output of the experiment is 1, namely adversary has won the experiment.

If indeed, the pair p dash, q dash is exactly the same as the prime factorization of your modulus N, and we say that the Factoring Assumption holds with respect to our algorithm GenModulus. If for every polytime adversary, the probability that the adversary could come up with correct factorization of the modulus N, is upper bounded by some negligible function. Notice that, there is always a guessing strategy by an adversary, it can guess some p dash and q dash, and with non-zero probability, it may indeed with the correct factorization of N.

(Refer Slide Time: 13:00)

What we want is basically that no polytime adversary should be able to do anything better than that. That is basically the intuition of this experiment. Now, finally let us see the RSA

assumption here. So, it turns out that Factoring Assumption, even though it looks like a candidate One-Way Function, that means that if I give you the product of two arbitrary large prime numbers and challenge you to come up with the factorization.

Then that looks like a candidate One-Way Function. But just based on this candidate One-Way Function, we cannot directly get a practically public-key cryptosystem. So, to get around that, we introduce a related problem, which we call as an RSA problem, whose difficulty is related to the hardness of the factoring problem. So, this RSA problem is described with respect to another Parameter Generation algorithm, which we call as GenRSA.

So, what does GenRSA algorithm does is, it first runs the GenModulus algorithm and pick up random primes p and q of size n-bits each, and multiples them to get the modulus N, and now it computes the value phi(N) and it is computable because phi(N) is nothing but the product of p-1 and q-1. Then this GenRSA algorithm picks an exponent e such that e is relatively prime or co-prime to phi(N).

And since e is co-prime to phi(N), by running the Extended Euclid's algorithm, the multiplicative inverse d(e) modulo phi(N) can be computed. Overall, this GenRSA algorithm now outputs N, p, q, e, and d. Intuitively, the RSA problem is, if you are given just a public modulus N and the public exponent e, and the random element y from the ZN* set, then the challenge for you is to compute the eth root of y modulo N without actually knowing the actual value of d or without knowing the prime factorization of N.

I stress that the challenge is to compute eth root modulo N because if I just challenge you to compute the eth root modulo N, because if I just challenge you to compute the eth root without modulo N, that is very easy to do that. The challenge here is to compute the value of eth root of the random y modulo N, which is formalized by this experiment which I call as RSA inverse experiment.

The experiment is as follows, the challenger runs the GenRSA algorithm to generate the parameters N, p, q, e, d are random-wise chosen from the set ZN*. Again, if you are wondering that how picking up a random element from the set ZN* is possible in poly(n) time, well it is possible, you can see the book by (()) (15:29) where the algorithm is given of how to pick random elements.

Now the challenge for the adversary is to find out the eth root of this random y just based on the knowledge of public modulus N and the public exponent e. So, it has to output a group element say x and we say that adversary has won the experiment or the output of the experiment is won, if and only if, x is indeed the eth root of random y modulo N, namely if x to the power e modulo N is y. We say that RSA assumption holds with respect to this GenRSA algorithm, if for every polytime algorithm participating in this experiment, the probability that it can come up with the correct eth root of a random y is upper bounded by some negligible probability.

(Refer Slide Time: 16:23)

Now, let's try to compare the RSA assumption and Factoring Assumption, because on a high level, they might look similar to you. So, on your left hand side you have the factoring experiment where the challenge for the adversary is to come up with the prime factorize of the public modulus N. On the right hand side, you have the experiment modeling the RSA problem where adversaries are now given some extra information as part of its challenge, namely, it is given the public exponent e and the random y from the set ZN*, and its goal is to come up with eth root of random y modulo N.

It turns out that we can prove that if RSA assumption holds, then Factoring Assumption also holds and this can be proved by contrapositive, namely, if you assume that factoring is computationally easy, that means it is possible for a polytime adversary to come up with a prime factorization p, q of a modulus N. Then, once you factorize your modulus N into its

prime factorization of p and q, then you can easily compute phi(N) because phi(N) is nothing but p-1 times q-1.

And if phi(N) is computable in polytime and anyhow e is given to you, then by running the Extended Euclid algorithm, you can yourself compute the multiplicative inverse of e, namely d, and once you compute d, the eth root of y is nothing but y to the power d modulo N. That means, this implication is indeed true. On the other hand, let's try to consider the relationship between the Factoring

Assumption and RSA, in the sense, can we say that if Factoring Assumption holds, then RSA assumption holds and we do not have any answer for this. That means, we cannot prove that if RSA assumption does not hold and the Factoring Assumption does not hold. It is a big open problem because there might be a way to efficiently solve the RSA problem without actually factoring your modulus.

Because one of the ways of solving the RSA problem might be to factorize your N and then once the factorization is known, come up with value of d and so on, but that need not be the only way to solve the RSA problem. There might be other ways to compute to solve the RSA problem in polynomial amount of time without actually factorizing your modulus, and in that sense, making the RSA assumption is a very strong assumption compared to making the factoring assumption because difficulty wise, the factoring problem looks like a more challenging problem, more difficult problem than solving the RSA problem.

(Refer Slide Time: 19:05)

RSA Permutation as a Trapdoor Permutation

- One-way trapdoor permutation (OWTP)

  $X \xrightarrow{f} X$

  - Easy to compute for any input from the domain
  - Difficult to invert any random input from codomain
  - Easy to invert any input from the codomain, given the knowledge of the trapdoor information

- A trapdoor permutation scheme $\mathcal{T}$ over a finite set X is a triplet of algorithms (Gen, f, Inv):
  - Gen() $\rightarrow$ (pk, sk)
    - pk: public key   - sk: secret key
  - $f_{pk}: X \rightarrow X$
    - $f_{pk}(x) = y$
    - Deterministic algorithm
  - Inv$_{sk}: X \rightarrow X$
    - Inv$_{sk}(y) = x$
    - Deterministic algorithm

- Correctness property --- for every possible outputs (pk, sk) of Gen and for all $x \in X$:

$$\text{Inv}_{sk}\left(f_{pk}(x)\right) = x$$

So, finally let us discuss how exactly you can utilize your RSA Permutation as a Trapdoor Permutation because in our next lecture, we are going to treat our RSA Permutation as our Trapdoor permutation, and we will see that how we can formulate an instantiation of public encryption scheme from this RSA Permutation. So, for that recall the definition of One-Way Trapdoor Permutation.

So, it is a function from set fancy X to the set fancy X, which is easy to compute for any input from the domain, but the difficult to invert from any random input from the co-domain, until and unless you are given with special trapdoor information. So, this is formalized by saying that more formally we have trapdoor permutation scheme over the set fancy X, which consists of a Key Generation algorithm, a trapdoor function f, and its inverse.

Where the Parameter Generation algorithm will output a public parameter and a secret parameter, the function f is basically, a keyed function, keyed by the public key pk and it takes value from the set fancy X and it gives you an output from the fancy X set and it is a Deterministic algorithm and the corresponding inverse algorithm is operated by a secret key, so your secret key is basically acting as a trapdoor information here and using this trapdoor information.

You can correctly invert any y from the codomain, namely the set fancy X. The correctness property that we require from the trapdoor permutation scheme is that for every pair of parameter generated by Generation algorithm, for every x from your domain, if you compute the value of f with a public key on that input x, and say obtain the y output, and now if you invert that y with the secret key sk.

You should get back the x, and a second requirement is the one in its requirement, which is basically states that your function f should behave like a one-way function, even if an adversary knows the value of the public parameter pk.

(Refer Slide Time: 21:09)

RSA Permutation as a Trapdoor Permutation

❏ RSA trapdoor permutation $\mathcal{T}_{RSA}$ is a triplet of algorithms $(GenRSA, f_{RSA}, Inv_{RSA})$

❏ **Algorithm** GenRSA(n)

- ❖ Generate uniformly random n-bit primes p and q
- ❖ Compute $N = pq$ and $\varphi(n) = (p-1)(q-1)$
- ❖ Select $e > 1$, such that $GCD(e, \varphi(n)) = 1$
- ❖ Compute $d \stackrel{\text{def}}{=} (e)^{-1} \bmod \varphi(N)$
- ❖ Output $pk = (N, e)$ and $sk = (N, d)$

❏ **Function** $f_{RSA}(N, e, x)$

- ❖ $x \in \mathbb{Z}_N^*$
- ❖ $f_{RSA_{(N,e)}}: \mathbb{Z}_N^* \Rightarrow \mathbb{Z}_N^*$
- $f_{RSA_{(N,e)}}(x) \stackrel{\text{def}}{=} [x^e \bmod N]$

❏ **Function** $Inv_{RSA}(N, d, y)$

- ❖ $y \in \mathbb{Z}_N^*$
- ❖ $Inv_{RSA_{(N,d)}}: \mathbb{Z}_N^* \Rightarrow \mathbb{Z}_N^*$
- $Inv_{RSA_{(N,d)}}(y) \stackrel{\text{def}}{=} (y^d \bmod N]$

❏ **Correctness property** --- for **every possible outputs** (N, e, d) of GenRSA and **for all** $x \in \mathbb{Z}_N^*$:

❏ **One-Waynes:** f_{RSA} is a one-way function OWF, **even if an adversary knows the public key** $pk = (N, e)$

❖ True, if the RSA assumption holds for GenRSA

So, let us see how we can visualize the RSA permutation from the set ZN* to ZN* as an instantiation of the Trapdoor Permutation. So, basically RSA Trapdoor Permutation consist of three algorithms, the Parameter Generation algorithm is nothing but GenRSA algorithm, namely, it picks uniformly random n-bit prime numbers, p and q, compute the modulus N, and then compute the size of ZN*, namely phi(N).

It picks up the public exponent e, such that e is co-prime to phi(N). Since e is co-prime to phi(N), by running the Extended Euclid algorithm, multiplicative inverse of e modulo phi(N) is computed and the public parameters are N, e, and the secret parameters are N, d. The forward direction function, namely the function f sub RSA, and operated with the public key N, e is as follows.

To compute the value of this function f sub RSA on an input x basically you just output x to the power e modulo N, and x to the power e modulo N can be computed in poly(n)time. We will see later on how exactly this group exponentiation can be computed in polynomial in n amount of time. On the other hand, if you have a trapdoor namely a secret parameter n, d and if you want to invert any given y from a set ZN*.

Basically you have to compute y to the power d modulo N. So, let's see whether this RSA Trapdoor Permutation indeed satisfies the requirement of a generic Trapdoor Permutation Scheme. So the correctness requirement states that, it should hold that x to the power e followed by raising it to d modulo N, should give me by x and we already have approved that

indeed that is the case when we showed that the function f and the inverse RSA function are inverse of each other.

On the other hand, the One-Waynes property from this RSA permutation, requires that anyone who knows just the public parameter, namely say N, e, but does not know the trapdoor information, say d, without the knowledge of the trapdoor information, it is difficult to compute the eth root of any random y, and its turns out that indeed that is the case if the RSA assumption holds with respect to your Parameter Generation algorithm that means.

If indeed the RSA problem is difficult to solve with respect to this RSA Parameter Generation algorithm, then this so called RSA permutation can be viewed as an instantiation of your Trapdoor Permutation. So, recall that in one of our previous lecture, we had seen that how generically we can convert Trapdoor Permutation Scheme into a key agreement protocol with weak secrecy notion.

And now if you instantiate Trapdoor Permutation by an RSA Trapdoor Permutation, then by using the RSA Trapdoor Permutation, we actually obtain an RSA key-efficient protocol, which gives you a weak secrecy notion, namely the resultant key is going to be known to the sender and the receiver, and the adversary will not know the full key in its entirety. On the other hand, there exist the discrete lock function that we had discussed earlier, it cannot be reviewed as an instantiation of the Trapdoor Permutation.

Because we know the forward-direction function, namely say G to the power x can be treated as your forward direction function, but we do not know what should be the corresponding trapdoor information which we can use and invert back this function. That is kind of big challenging problem which is there. On the other hand, in the context of RSA permutation, we have the corresponding trapdoor associated namely the exponent d, which is the multiplicative inverse of e, modulo phi(N).

So that brings me to the end of this lecture. In this lecture, we have introduced the RSA assumption, and we have seen the relationship between the RSA problem and the factoring problem. Thank you.