

**Foundations of Cryptography**  
**Prof. Dr. Ashish Choudhury**  
**(Former) Infosys Foundation Career Development Chair Professor**  
**Indian Institute of Technology-Bangalore**

**Lecture-12**  
**Practical Instantiations of PRG**

Hello everyone, welcome to lecture 11.

**(Refer Slide Time: 00:30)**



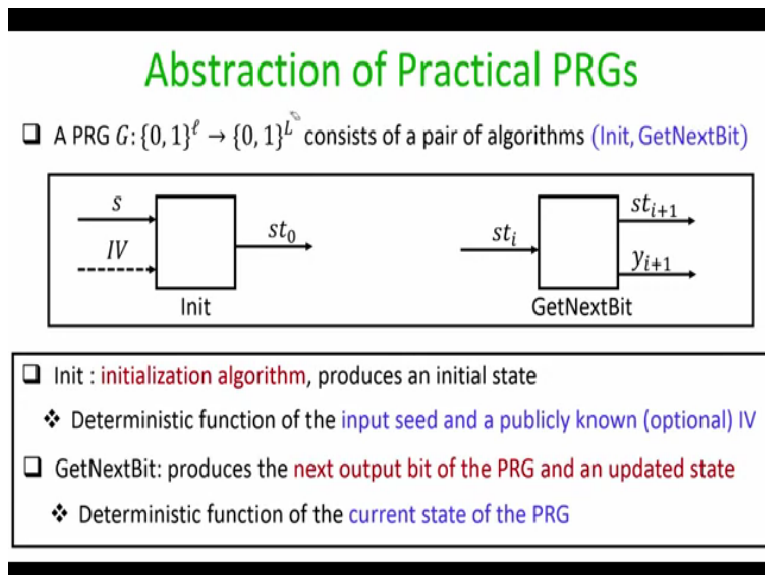
The plan for this lecture is as follows. In this lecture we will discuss about the practical instantiations of pseudo random generators, namely we will see the construction based on linear feedback shift register and RC4 and we will discuss about the recent developments in the area of practical instantiations of pseudo random generator. The reason we call these instantiations as practical is that they are super-fast, compared to the provably-secure constructions of pseudo random generators that we had discussed in the last lecture based on one way function and one way permutation.

However unfortunately, for these practical instantiations of pseudo random generators, we do not have any mathematical proof that they are indeed pseudo random generators. That means we do not have the theorem statement, which proves that hey there is no polynomial time distinguisher who can distinguish the output of this random number generator from the output of a true random number generator.

It is only because ever since the construction of these practical instantiations of PRGs we have not found any suitable attack or any suitable distinguisher, we believe that these constructions are secure, whereas for the provably-secure constructions based on one way function and hard-core predicate which we have discussed in the last lecture, we have a mathematical proof that indeed they are secure in the sense there exists no polynomial time distinguisher to distinguish the output of those algorithms from the output of the corresponding true random number generator.

So in practice, wherever you have a cryptographic construction where you want to instantiate a pseudo-random generator, we actually go for these practical instantiations.

(Refer Slide Time: 02:08)



So in all the practical instantiations of pseudo-random generators, we can follow the following abstraction. Imagine you are given a pseudo-random generator, which expands an input of  $\ell$ -bits to an output of  $L$ -bits. We can assume that the pseudo-random generator consists of a pair of algorithms, namely an initialization algorithm and GetNextBit algorithm. And what basically these algorithms do are as follows.

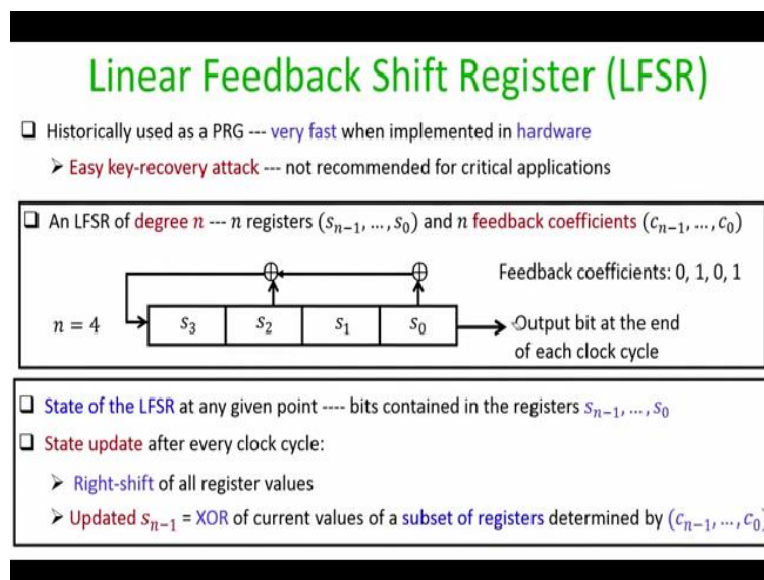
So, the Init algorithm is actually the initialization algorithm which sets the initial state of your algorithm and it is a deterministic function. And it takes the seed for the algorithm  $G$  as the input and along with that an optional initialization vector, so this initialization vector is optional. It is

not necessary that every practical instantiation of pseudo-random generator should take this initialization vector, it depends upon the underlying construction.

However, if the  $IV$  is given as an input, it is publicly known, and based on the seed and  $IV$ , the initialization algorithm produces an initial state of the algorithm which we denote by  $st_0$ . Now, the `GetNextBit` algorithm does the following. It takes the current state of your algorithm or the PRG, which I denote by  $st_i$ , and it updates the state to  $st_{i+1}$  and along with that it produces the next output bit of your algorithm  $G$ , right.

So, if you want to generate a sequence of bits, what we do is we do the initialization algorithm, get the initial state  $st_i$  and say if you are interested in getting an output of big  $L$ -bits, basically we invoke this `GetNextBit` algorithm  $L$  number of times in sequence and in each invocation the state gets updated and output bits keep on getting generated one by one. So that is the abstraction which we can use to abstract out any practical instrumentation of pseudo random generator.

(Refer Slide Time: 04:03)



So let us see, one popularly used practical instantiation of pseudo-random generator, which we use in the hardware. And this is called linear feedback shift register or LFSR. So, historically, it was used as a PRG. And it is very fast when implemented in hardware. However, it is not recommended to be used for critical applications, because it is very easy for an adversary to recover the entire key by just seeing few output bits of the LFSR.

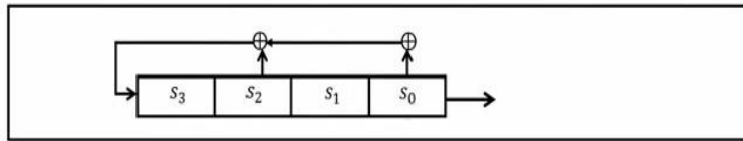
So, an LFSR of degree  $n$  basically consist of  $n$  registers denoted by  $S_0$  to  $S_{n-1}$ . And along with that, it will have  $n$  feedback coefficients  $c_0$  to  $c_{n-1}$ , where the coefficients will be Boolean values. So for example, here we have an LFSR of degree 4 consisting of 4 registers, and the feedback coefficients are 0, 1, 0, 1. So, how an LFSR operates. As far as the state of an LFSR is considered, the state is nothing but the bit values which are stored in the individual register.

So, if we take this particular example, then the state of the LFSR is nothing but a 4 bit string namely the 4 bits stored in the registers  $S_3, S_2, S_1$  and  $S_0$  and update of the state happens as follows: after every clock cycle, the bit which is present in  $S_0$  is going to be produced as the output bit and the contents of all the registers are right-shifted. As a result of that, what is going to happen is that the current  $S_1$  will become the next  $S_0$ . The current  $S_2$  will become the next  $S_1$  and so on. And as a result  $S_3$  will become empty. And the updated value of the last register, in this case  $S_3$ , will be determined by taking an XOR of the subsets of the bits of the current state. And the subsets of the registers whose XOR we take is actually determined by the feedback coefficient.

So again in this example, since the feedback coefficients are 0, 1, 0, 1, that means, after every clock cycle once we do the right shifting here, the value of  $S_3$ , is nothing, but the XOR of the current  $S_2$ , and the current  $S_0$ . If you take the XOR that will be the value which will be fed as the new value of  $S_3$ . That is why the name linear feedback shift register. In each clock cycle, we shift the contents of all the registers by one position and that is why shift register. And linear feedback, because we have a feedback loop that determines the value of the contents of  $S_{n-1}$ , in the next clock cycle. And this feedback function is a linear function of the current set of registers. That is why the name linear feedback shift register.

**(Refer Slide Time: 06:54)**

## LFSR : An Example



- ❑ Suppose the initial state is 0011

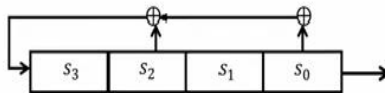


So for example, if you take this LFSR of degree 4, and suppose the initial state is 0011, then after the first clock, everything will be shifted by one position, and as a result, the bit 1 is going to be the first output bit and the feedback which will be going into the LFSR for the next iteration will be 1, namely the XOR of the bit 0 and 1 because your feedback coefficients are 0, 1, 0, 1 and as a result the next state of the LFSR will be 1001.

Again after the next clock cycle, the bit value 1 will be flushed out test output; that will be the second output bit of your LFSR and the feedback which will be going will be 1 and as a result your state will be updated to 1100 and so on. So, that is how an LFSR of degree  $n$  operates.

(Refer Slide Time: 07:49)

## LFSR : Security Analysis



- ❑ Can a PPT attacker predict the outcome of an LFSR, for an unknown initial state and unknown feedback coefficients, but publicly known degree  $n$ ?
- ❑ An LFSR of degree  $n$  can have at most  $2^n - 1$  non-zero states --- (an all zero state is useless)
  - ❖ Maximum-length LFSR : output sequence repeats after exactly  $2^n - 1$  non-zero states
  - ❖ Any LFSR can be set to have maximum length by setting its feedback coefficients, independent of its non-zero initial state
- ❑ A maximum-length LFSR produces all  $n$ -bit strings as output sequence with equal frequency
  - ❖ Does that mean LFSR is a secure PRG ?

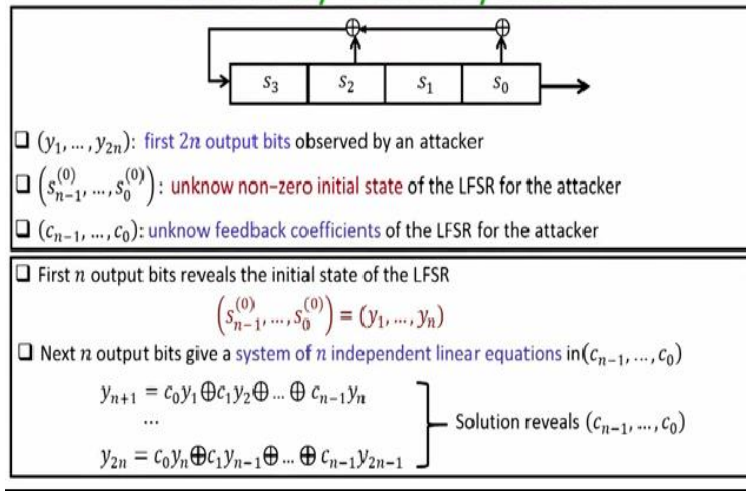
Now let us argue whether this LFSR is secure or not, that means can we consider this LFSR to be a pseudo random generator. And the requirement from a pseudo random generator is that if someone gives you the sample of an LFSR, where you are not given the initial state of the algorithm because if you are given the initial state of the LFSR then you can actually compute all the output bits of LFSR. So, imagine you are not given the input state of the LFSR and along with that imagine you are not given the feedback coefficients as well but you are given the degree of the LFSR, that means, you know the number of registers that are used in the LFSR. Then is it possible for the attacker to compute or predict outcome of LFSR. It turns out that if we have an LFSR of degree  $n$ , then it can have at most  $2^n - 1$  non-zero states. And why we are interested in non-zero states? Because once LFSR occupies the state, where the content of all the registers is 0, then after that it does not matter how many times or how many clock cycle we operate the LFSR, all the subsequent states will be 0. That means once we reach an all-zero state, we should stop generating the outputs of LFSR. So the interesting case is when we actually focus on the non-zero states of LFSR.

We define LFSR to be a maximum length LFSR, if its output sequence repeats after exactly  $2^n - 1$  number of non-zero states. And interestingly, it turns out that it does not matter with what input state you start with, if you start with any non-zero initial state, then it is always possible to set the feedback coefficients in such a way that your LFSR actually becomes a maximum length LFSR.

That means starting with that non-zero initial state, you can go through all the  $2^n - 1$  non-zero states and then only the sequence will repeat. So imagine for the moment that you have a maximum length LFSR. Intuitively, you might feel that all the  $n$ -bit strings are going to be produced with equal frequency, then does that mean that your LFSR is a secure PRG because if the output state is going to be repeated after  $2^n - 1$  number of states, that means for an attacker, it has to wait for  $2^n - 1$  number of states, which is an exponential amount of quantity. And hence it cannot distinguish the output of the LFSR from the output of a true random number generator. That could be your underlying intuition based on which you can declare your LFSR to be secure. But it turns out that that is not the case. For an LFSR of degree  $n$ , just by observing polynomial number of output bits,

**(Refer Slide Time: 10:26)**

## LFSR : Key-recovery Attack



an adversary can recover the entire key and once it recovers the entire key, it can predict all the future output bits of LFSR right. So imagine you are given an LFSR of degree  $n$ , where you do not know the feedback coefficients and you do not know the initial states of LFSR and imagine that the adversary has observed the first  $2n$  output bits of the LFSR which I denote by  $y_1, \dots, y_n$ .

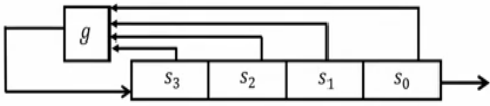
And the initial state non-zero state of the LFSR is denoted by the notation  $(s_{n-1}^{(0)}, \dots, s_0^{(0)})$ . So in this notation, in the superscript, I am putting 0 in the parenthesis that denotes the 0th state of LFSR namely the initial state, and that is also unknown for the attacker, the attacker has seen only the first  $2n$  output bits. Also we assume that the adversary is not aware of the feedback coefficients; from the viewpoint of the adversary, it could be any subset of the  $n$  registers which are actually getting XORed to decide the feedback. So the unknown coefficients  $c_{n-1}$  to  $c_0$  are also therefore the attacker. It turns out that adversary knows the relationship that the initial state of the LFSR is nothing but the first  $n$  output bits that it has seen because if you see the operation of the LFSR, after every clock cycle, the content of  $s_0$  is actually coming out as the output and after every clock cycle, the new content of  $s_0$  is actually the previous content of  $s_1$ , which in turn is actually the previous  $s_2$  and so on.

That means adversary knows that the first  $n$  output bits of your LFSR are nothing, but your initial state. So that is the first piece of information which is now available to the adversary, which is now a significant amount of information for the adversary. And it turns out that the next output

bits, namely  $y_{n+1}$  to  $y_{2n}$ , actually gives a system of linear equations in the unknowns in the feedback coefficients to the adversary. Namely adversary knows that the  $n + 1^{th}$  output bit  $y_{n+1}$  is nothing, but  $c_0y_1 \oplus c_1y_2 \oplus \dots \oplus c_{n-1}y_n$ . In the same way, the  $2n^{th}$  output bit satisfies the relation  $y_{2n} = c_0y_n \oplus c_1y_{n+1} \oplus \dots \oplus c_{n-1}y_{2n-1}$ . So adversary gets a system of  $n$  independent equations in  $n$  variables. And by solving them, it can completely recover back the feedback coefficients. So now both the keys as well as the feedback coefficients are known to the adversary. And hence it can completely determine all the subsequent outputs of your LFSR. That means just by observing  $2n$  output bits of the LFSR, adversary can completely break this LFSR. And hence no way, it is actually a pseudo random generator.

(Refer Slide Time: 13:27)

FSR : Adding Nonlinearity

<input type="checkbox"/> Idea : Prevent linear relationships between the output bits by introducing non-linearity ❖ Several ways of introducing non-linearity	
<input type="checkbox"/> Non-linear feedback : ❖ Updated value of the left-most register is a non-linear function of the current registers	➤ For $i = 0, \dots, n-2$ : $s_i^{(t+1)} = s_{i+1}^{(t)}$ ➤ $s_{n-1}^{(t+1)} = g(s_0^{(t)}, \dots, s_{n-1}^{(t)})$
	
<input type="checkbox"/> Non-linear combination generators : ❖ Variant I : Single LFSR, with output bit being a non-linear function of current registers ❖ Variant II : Several LFSRs (preferably of different degrees), with the actual output bit being a non-linear function of the output bit of the individual LFSRs	

So a method which is used to preserve the security of the LFSR is to add some kind of non-linearity. If you see the attack, or the strategy, which is used by the attacker here is to explore the system of linear equation, or namely, the attacker uses the fact that the feedback is actually a linear function of the subset of the register. So one way to get around this is to add some kind of non-linearity in the feedback shift register. And there are several ways of introducing non-linearity in the construction of linear feedback shift register.

The first method of adding the non-linearity is to make ensure that your feedback itself is nonlinear. Namely, what we assume here is that the content of the  $i^{th}$  register at the clock cycle  $t + 1$  will be the content of the  $i + 1^{th}$  register at the clock cycle  $t$ , that means everything is still shifted by one



position after every clock cycle. But the content of the last register is now a nonlinear function of the current registers. So, in the previous construction in the LFSR, the function  $g$  was actually a linear function. But the proposal here is that instead of ensuring that the feedback is a linear function, the feedback is now going to be a nonlinear function of the set of bits which are there in the current register. So, that is one way of adding non linearity which is followed in the modern constructions of pseudo random generators based on feedback shift registers.

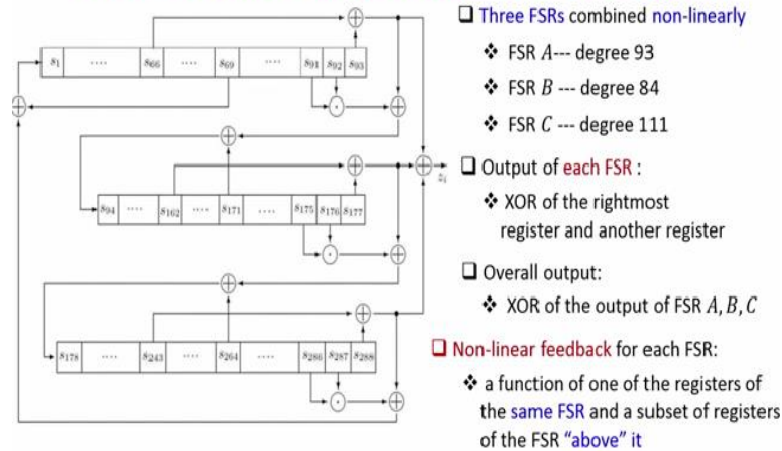
The other way of adding nonlinearity is to add non linearity in the output itself. So till now we are discussing the case where output is actually the content of the current  $s_0$ , where  $s_0$  is the value of  $s_1$  in the previous clock cycle and so on and every clock cycle everything gets shifted by one position. But I could have another way of determining the output, where output is actually a nonlinear function.

And there are 2 ways to determine nonlinear combination generators. Variant one is the following: we still use single LFSR, where everything get shifted by one position and we have a linear feedback. But instead of just ensuring that  $s_0$  is the output bit, the output bit turns out to be a nonlinear function of the current registers. And the variant two is, instead of using one LFSR, we will now have several LFSR, preferably of different degrees. And the overall output bit of the combined LFSR will be a complicated nonlinear function of the output of the individual LFSRs. So that is the second variant.

So, that means to add non linearity you have 2 options, non-linearity in the feedback and non linearity in the output, where non linearity in the output can be achieved in 2 ways. Just use 1 LFSR, with output being a complicated nonlinear function and option two is to use several LFSRs, with the output being a complicated nonlinear function of the output of the individual LFSRs.

**(Refer Slide Time: 16:31)**

## PRG Trivium : Hardware Stream Cipher



And it turns out that the modern constructions of PRGs based on LFSR indeed use these principles. So here is a candidate construction based on LFSR, which is called Trivium. And it is a highly popular instantiation of pseudo random generator. If you see pictorially, it is actually a combination of 3 feedback shift registers. So you have the first LFSR consisting of 93 registers, which we denote as the feedback shift register A. Then you have the next LFSR, which we denote as B consisting of 84 registers. And then we have the next feedback shift registers C, consisting of 111 registers. Now, the reason why it is using 93, 84, 111 are not clearly known; they are the design principle used by the designers of the trivium. However, there are some well-known principles which are used to select the value of FSR A, FSR B, FSR C like that.

But otherwise in general there are no fixed guidelines which are used to select the size of FSR A, FSR B, FSR C to be like this. And now, you see that each FSR has an individual output. So, if I consider the output of FSR A, so, it is basically the XOR of the rightmost register, in this case, the 93rd register and another register of the same FSR. So this is the first difference in the construction of Trivium compared to the regular LFSR.

In the regular LFSR, the 93rd output, the content of the 93rd register will be considered as the output bit after every clock cycle. But now after every clock cycle, it is the XOR of the 93rd register and a 66<sup>th</sup> register in the FSR A, which will be considered as the output of the FSR A

after the individual clock cycles. And the same holds for the FSR B as well as the FSR C. That is the first way of adding nonlinearity here.

And the overall output of the FSR is basically the XOR of the output of the FSR A, FSR B, FSR C. So that is the second way of adding nonlinearity here. As far as the feedback is considered here, if you see for example, the FSR A here, now what is going as the feedback? So the feedback is nothing but basically a function of one of the registers in the same FSR and a subset of registers of the FSR above it. So what I mean “above it” here is the following: as I said the sequence of the first 93 registers is the FSR A and the next 84 register is FSR B and the next 1011 register is FSR C. You can imagine this construction as some kind of circular construction, where above the FSR A, we have the FSR C, above the FSR B we have the FSR A and above the FSR C we have the FSR B. That is what I mean by “above” in this context.

And the feedback of the FSR A is basically an XOR of one of the registers of the FSR A, along with some of the registers of the FSR C. In the same way, the feedback of the FSR B is an XOR of some of the register of the FSR B along with some of the registers in the FSR A, and in the same way the feedback for the FSR C is an XOR of some of the registers of the FSR C along with some of the registers in the FSR B.

That is how we are introducing nonlinearity. So, the idea here is by making this complicated construction, we are actually completely removing the linearity which was present in the original construction of the LFSR. And this is how the Trivium is designed. And this is considered to be a secure a PRG because, after the development of this construction, we have not got any practical attack. That means no polynomial time algorithm has been reported, which can actually predict outcome of the trivium if the initial state of the trivium is not known to you.

So, I am not going into the full details of the construction, what design principles are used, why the 3 FSRs used, their degrees are constructed like this and so on. If you want to know more about the details of such constructions you can refer to any of the references that we are following in this course.

**(Refer Slide Time: 20:56)**

## RC4 : A Software Stream Cipher

❑ Very fast and simple. Several **vulnerabilities** reported recently.

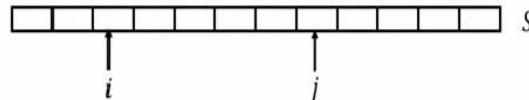
❖ Used earlier in WEP (Wired Equivalent Privacy) encryption standard 802.11

❑ A **state** in RC4 consists of:

❖ 256-byte array  $S$  --- always a **permutation of the set  $\{0, 1, \dots, 255\}$**

➤ Initialized to a **key-dependent** pseudorandom permutation of  $\{0, \dots, 255\}$

❖ Two index pointers  $i, j \in \{0, 1, \dots, 255\}$



❑ In each iteration, the bytes of  $S$  **shuffled around** and one of the bytes is output

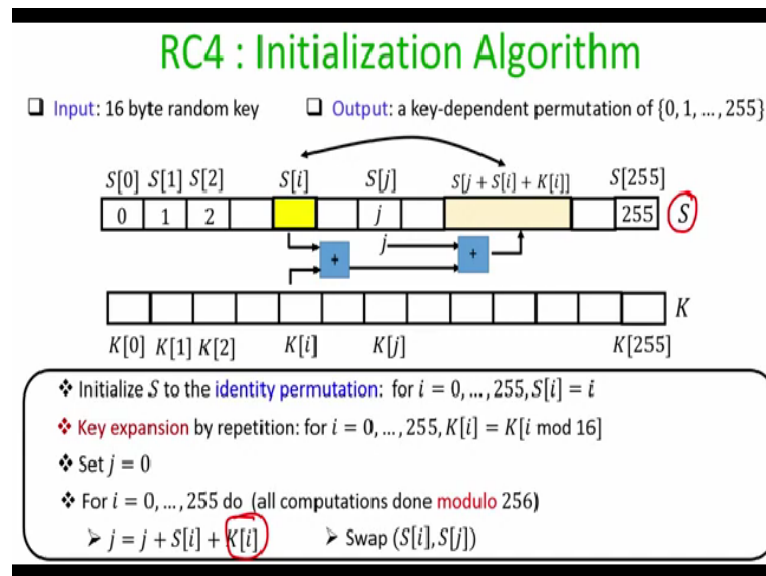
Now we will consider another popular instantiation of pseudo random generator, namely RC4 which is super fast when implemented in software. And even though it was highly popular till some years back, recently several vulnerabilities have been reported. And that is why it is no longer recommended to be used for critical purpose. In fact, it was used as one of the standards in WEP. And after vulnerabilities were reported in RC4 it is no longer used in the standard.

So, recall that any practical instantiation of pseudo random generator consist of 2 algorithms, namely an initialization algorithm and a state update algorithm. And in the initialization algorithm, the state is initialized and within the state updation algorithm, the state is updated and the next output bits generated. So as far as the state in RC4 is concerned, the state basically consist of an array, consisting of 256 bytes. And throughout the algorithm, it will be ensured that these 256 bytes actually consist of a permutation of the set 0 to 255. That means each of the values 0 to 255 will occur as one of the bytes among these 256 bytes. And that is why it is a permutation of the set 0 to 255.

Now the initialization algorithm for this RC4 is as follows. The initialization algorithm creates a key dependent pseudo random permutation of the set 0 to 255. And along with that, initialize 2 index pointers  $i$  and  $j$  in the range 0 to 255. That is the way initialization happens for RC4, we will go into the details of the initialization very soon. And once the initialization is done, in the state update algorithm in each iteration the bytes of the  $S$ , which are actually a key dependent pseudo

random permutation of the set 0 to 255 are shuffled around, and after shuffling one of the bytes is output, that is the way state is updated.

(Refer Slide Time: 23:02)



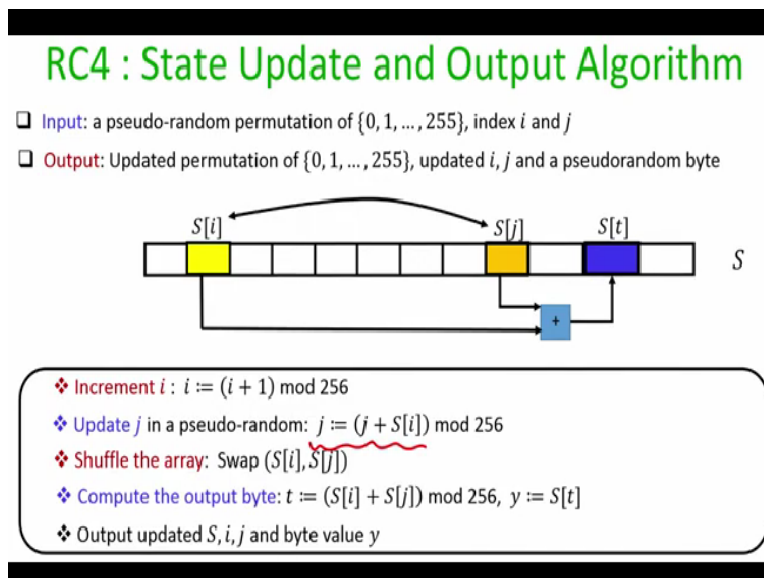
So now let us go into the details of the initialization algorithm. So the key or the seed for the RC4 algorithm is basically a 16 byte key, which is going to be a random 16 byte key. So we denote it by  $k_0$  to  $k_{15}$ . And the output is going to be a pseudo random permutation of the set 0 to 255. So that will be the array  $S$ . So we initialize the array  $S$  consisting of values 0 to 255 in sequence. Namely, the first byte is set to be 0. The next byte is set to be 1 and the last byte is said to be the value 255. This is an identity permutation. And now we have to somehow reshuffle the contents of the array  $S$  based on the value of the key array  $K$ . That means depending upon the contents of the key bytes, we have to shuffle the contents of the array  $S$ , ensuring that after shuffling, the modified  $S$  still represents a permutation of the set 0 to 255.

So to do that, we actually repeat the values of the key and ensure that the key array becomes of size 256. And this is done by performing the operation,  $K[i] = K[i \bmod 16]$ , that means we take the first 16 bytes, and repeat it again and again, to ensure that we have now an expanded key array of size 256. That is the way we do the key expansion. And then we set the initial index pointer  $j$  to be 0. And once the pointer  $j$  is set to 0, for the next 256 iterations, we do the following. We do the shuffling and to do the shuffling, we actually change the value of  $j$  like this: we set  $j$  to be the

summation of current  $j$  and current contents of the  $i^{th}$  byte of  $S$  and the  $i^{th}$  key byte. Namely,  $j = j + S[i] + K[i]$ . And once we have the updated index  $j$ , we swap the contents of  $S[i]$  and  $S[j]$ .

That is how we do the shuffling. So, intuitively what you can imagine here is that in each iteration, the index  $i$  gets incremented by 1. And in each iteration the index  $j$  is randomly shuffled, depending upon the byte of the keys. And once the pointer  $j$  is updated, we go to that location in the array  $S$ . And swap the contents  $S[i]$  with the contents of that location. That is how we actually generate a key-dependent pseudo random permutation of  $S$ . Why this is a key-dependent pseudo random permutation of  $S$  is because in each iteration, the value of  $j$  depends upon the contents of the key array. That's how the resultant permutation is a key-dependent permutation.

(Refer Slide Time: 26:01)



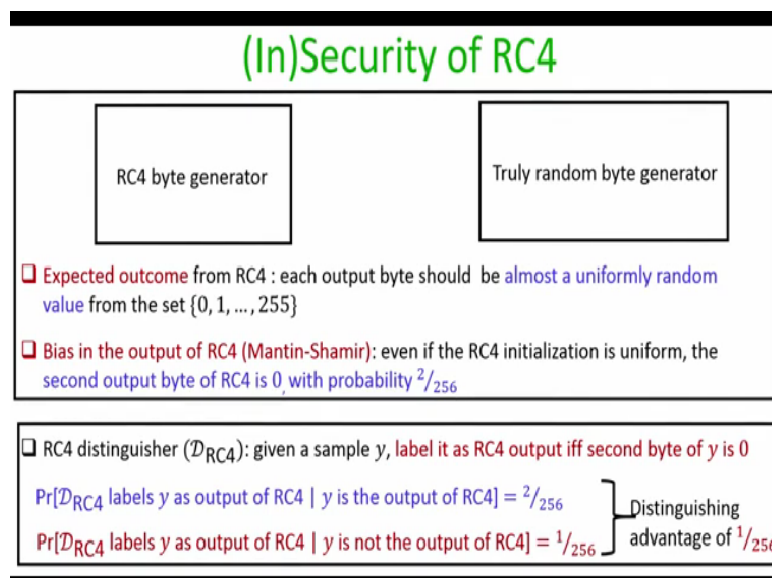
Now once the state has been initialized, we go to the state update and output algorithm. So, in the state update algorithm, in each invocation, the current contents of the  $S$  will be shuffled around and one of the bytes is going to be output. So, the way it is generated is as follows.

Imagine we have the current  $i$  and the current  $j$ , what we do is we increment the value of  $i$ . And then we randomly decide the value of  $j$ , depending upon the current contents of  $S$  as follows. So, the value of  $j$  is updated in a pseudo-random fashion by setting :  $j := (j + S[i]) \bmod 256$ . That means whatever is the current index of  $j$ , to that we add the byte value that is stored at the  $i^{th}$  location of the array  $S$ , and to ensure that we do the wraparound, all the operations are done

modular 256. So, by doing this operation we update the value of the index counter  $j$  in a pseudo-random fashion. And then what we do is, we swap the contents of  $i^{th}$  location of array  $S$  and  $j^{th}$  location of the array  $S$ . That is how we actually do the state update.

To determine the output, what we do is we determine a new index  $t$ , which is nothing but summation of the contents of the  $i^{th}$  location of the array  $S$  and the  $j^{th}$  location of the array  $S$ . That is,  $t := (S[i] + S[j]) \bmod 256$ . And we go to that location, that is the  $t^{th}$  location and whatever is the byte value which is stored there, that is the byte value which we are going to output.

(Refer Slide Time: 27:46)



So now let us discuss about the security of RC4 algorithm. So if you see the pseudocode of the state-update algorithm and initialization algorithm, you can see that we are not doing any complicated operation. That is why this algorithm is super-fast when you implement it in software. And that is why it was highly popular. Then people started reporting vulnerabilities in the security of RC4. So, we want to analyze here whether indeed RC4 is a candidate pseudo-random number generator or not. So if you recall, in each alteration of the state update, RC4 outputs a byte. So we have to compare the RC4 byte generator algorithm with a true random byte generator algorithm.

A true random byte generator algorithm will produce any byte in the set 0 to 255, uniformly randomly. And the expected outcome from the RC4 is that in each invocation of the state update algorithm, the output byte could be almost like a uniformly random value from the set 0 to 255. It

turns out that in one of the previous works, Mantin and Shamir showed that even if we assume that initialization of algorithm of the RC4 is a uniform initialization (that means it does not depend upon the key algorithm), if we start running the state update algorithm, then the second output byte of RC4 is more likely to be 0 that means, the probability that the second output byte of RC4 is 0 is  $\frac{2}{256}$ , compared to  $\frac{1}{256}$ . And this can be proved formally. So, if you want to see the exact formal details that how exactly this probability is derived you can refer to one of the references that we are following. So, assuming that this is the case, then here is a very simple RC4 distinguisher who can distinguish apart a sample byte generated by the RC4 byte generator, from a sample which is generated by a truly random byte generator.

So imagine the distinguisher is given a sample  $y$ , and it has to determine whether it is produced by the RC4 byte generator or by a random byte generator. So what the distinguisher does, since it knows the result of Mantin and Shamir, it just checks the second byte of  $y$ , and then if the second byte of  $y$  turns out to be 0, then it labels that the sample  $y$  is generated by the RC4 byte generator. Otherwise it labels the sample  $y$  to be generated by a truly random output. Now let us calculate the distinguishing advantage of the distinguisher that we have designed. If indeed the sample  $y$  which is a sequence of bytes is generated by the RC4 by generator is given to the distinguisher then the probability that indeed the second byte is 0 is  $\frac{2}{256}$ . So the probability that our distinguisher when given a random sample generated by RC4 byte generator labels it as the outcome of RC4 is  $\frac{2}{256}$ . Whereas if the sequence of bytes is generated by a truly random byte generator is given to the distinguisher, then the probability that its second byte is 0 is actually 1 over 256. That means in that case only, with that much probability only, our distinguisher will end up labeling a true random sequence of bytes to be an outcome of a RC4. So what is the distinguishing advantage of the attacker in this case? Well, the absolute difference is  $\frac{1}{256}$ , which is a significant probability. That means we can no longer claim that the sequence of bytes which are generated by the RC4 byte generator is close to the sequence of bytes, which a truly random byte generator would have produced and that is why this RC4 is no longer considered as secure.

To conclude, in this lecture we have seen on a very high level some of the practical instantiations of pseudo random generator. Then we have seen a hardware based construction, which we call as



the linear feedback shift register. The original linear feedback shift register is no longer used in the form it was proposed, because by observing polynomial number of outputs, an adversary can find out the entire state as well as the feedback coefficients and hence get can predict all the subsequent output bits of LFSR. So that is why the modern instantiations of pseudo random generators based on the feedback shift register introduce some kind of non-linearity which can be done by several ways, such as nonlinearity in the form of nonlinear feedback coefficients, nonlinear output bits and so on.

We also discussed a construction called Trivium, based on that principle, and a second construction that we saw is the software construction which can be implemented in software and it is super-fast, namely RC4. Unfortunately, we also saw some of the vulnerabilities which have been reported in RC4 due to which it is no longer recommended to be used.