Cryptography and Network Security

Prof. D. Mukhopadhyay

Department of Computer Science and Engineering

Indian Institute of Technology, IIT Kharagpur

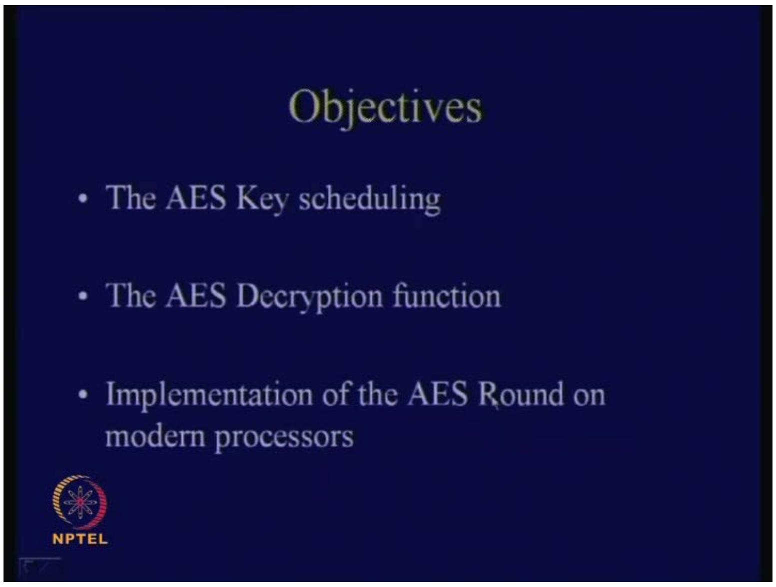
Module No. # 01

Lecture No. # 13

Block Cipher Standards (ASE) (Contd.)

We will continue with AES and discussions on the internal blocks of the standard that we were discussing in the last day's class also.

(Refer Slide Time: 00:26)

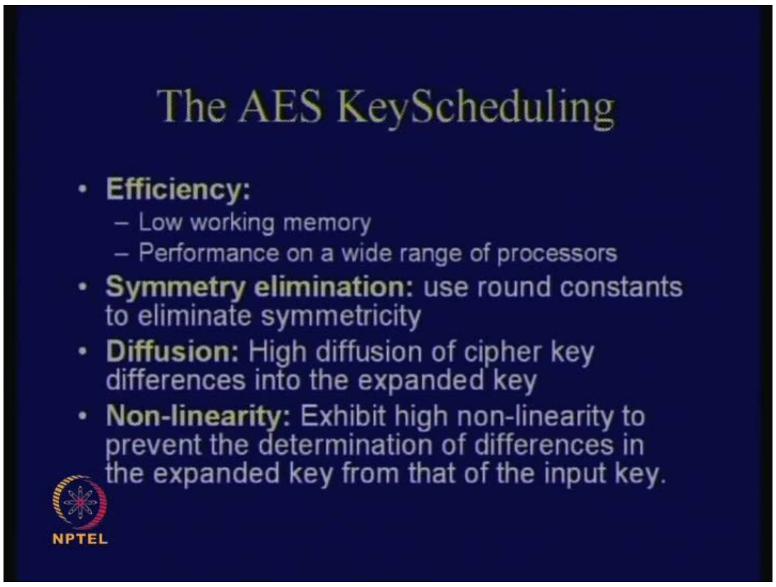


Objectives

- The AES Key scheduling
- The AES Decryption function
- Implementation of the AES Round on modern processors

We will continue with AES key scheduling. We will discuss about the internal operations inside the key-scheduling algorithm and then continue our discussions with the AES decryption function. We have seen how the encryption function looks like. We will try to see how the decryption function can be implemented similar to the encryption function and then discuss about the implementation of the AES round on modern processors. I will just try to reflect on this point, that is, how do you implement AES round on modern processors. The modern processor means the processor that has got more than 32 bits.

(Refer Slide Time: 01:05)



The AES KeyScheduling

- **Efficiency:**
  - Low working memory
  - Performance on a wide range of processors
- **Symmetry elimination:** use round constants to eliminate symmetricity
- **Diffusion:** High diffusion of cipher key differences into the expanded key
- **Non-linearity:** Exhibit high non-linearity to prevent the determination of differences in the expanded key from that of the input key.

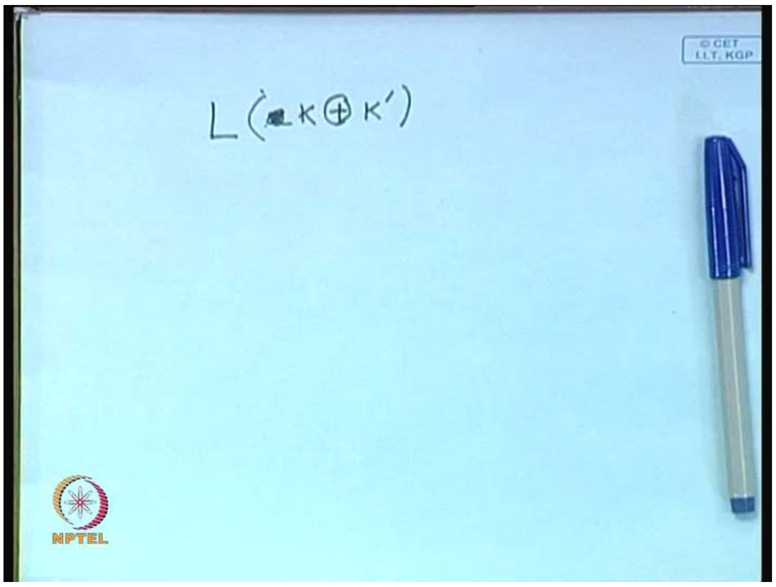
When we discuss the AES key scheduling, there are some objectives based upon which the key scheduling was designed. Primary objective was efficiency; that is, as I told you that one of the primary design criteria is that we have to support what is known as efficiency. That means the implementation have to be efficient.

The objectives were as follows. Low working memory, that is, the memory required in order to generate the next round key from the previous round key. Then, the memory requirement should be low. Also, the performance should be good on a wide range of processors. That means the processors on a wide range of data bits say for example, 8-bit processors, 32-bit processors, more than 32 bit processors. The other important criterion was symmetry elimination. That is, because there are some types of attacks, which exploit the symmetricity among the round keys, designers of AES have to eliminate symmetricity in the key scheduling. They achieve that using a concept of round constants. So, each round has a unique constant.

The rest of the algorithm was similar, but there was a different constant, which was added in every round. Essentially, that was incorporated to avoid symmetricity in the round keys. The other thing which we have discussed is that diffusion is a very important property. So, high diffusion of cipher key was expected. That is, whenever we introduce a certain amount of difference in the input cipher key, that is, the input key, it is expected that in the expanded key, there should also be a proper amount of diffusion. That means that disturbance should spread to as many bytes of the expanded key as possible. Therefore, high diffusion was required.

The other important property was non-linearity; that is, there should be some amount of non-linearity in each round key generation. Why was non-linearity kept? You know that it is precisely why S-boxes were kept in our encryption function. That is, if non-linearity is not clear and I know the input difference, then I am able to predict the output difference also. That means if the key-scheduling algorithm did not have any amount of non-linearity, then given say for example, the input key and a differential in the input key; that means if I just take two keys and I know the XOR between these two keys, then I am able to predict what will be the next round key difference much better. This is because I know that if I have got a linear transformation, then I can predict what will be the corresponding output XOR. Do you understand?

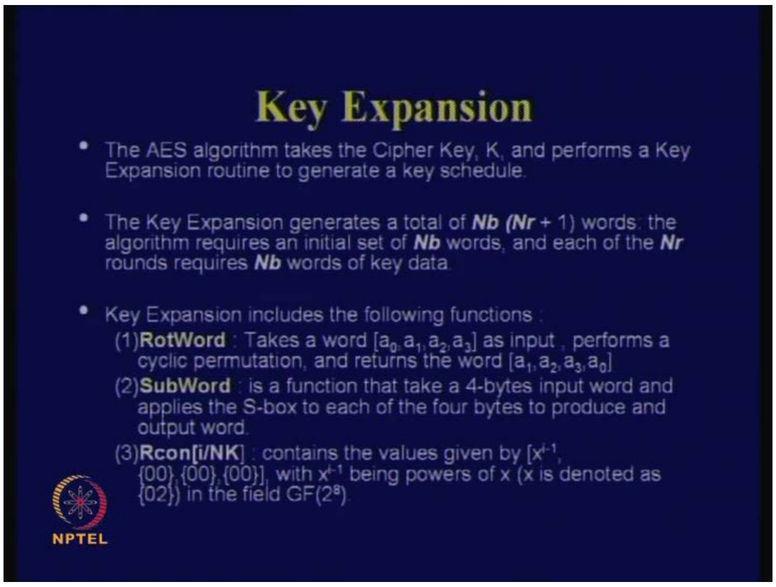
(Refer Slide Time: 03:58)



What I am saying is this - that is, suppose the input has got say for example, one of the keys and denote it by say K. You take a corresponding related key. I call that K dash. You know the difference between these two keys. Therefore, you know what is the value of K XOR K dash. So, you know the value of the difference.

Now, if you have a linear transformation, then you know that the output XOR or output difference can also be found out by doing this operation. So, if you know K XOR K dash, then you know the corresponding output difference as well because L is a linear transformation. However, the same thing does not hold if instead of L, we had non-linear transformation. So, in order to prevent that, we require that there should be some amount of non-linearity in the key-scheduling algorithm also.

(Refer Slide Time: 05:00)



Key Expansion

- The AES algorithm takes the Cipher Key, K, and performs a Key Expansion routine to generate a key schedule.
- The Key Expansion generates a total of ***Nb (Nr*** + 1) words: the algorithm requires an initial set of ***Nb*** words, and each of the ***Nr*** rounds requires ***Nb*** words of key data.
- Key Expansion includes the following functions :
  (1)**RotWord** : Takes a word $[a_0, a_1, a_2, a_3]$ as input , performs a cyclic permutation, and returns the word $[a_1, a_2, a_3, a_0]$
  (2)**SubWord** : is a function that take a 4-bytes input word and applies the S-box to each of the four bytes to produce and output word.
  (3)**Rcon[i/NK]** : contains the values given by $[x^{i-1}$, {00},{00},{00}], with x^{i-1} being powers of x (x is denoted as {02}) in the field $GF(2^8)$.

Looking little bit inside the key expansion of AES, this is the definition, that is, the AES algorithm takes the cipher key or the input key and performs the key expansion. Therefore, this routine is supposed to generate a key schedule. Therefore, it is supposed to generate the first round key, second round key; the second round key generates the third round key and so on. There is a recursive construction. Why? Because you know that the recursive construction is easily amenable to implementation. Therefore, you generate the first round key; from there you generate the second round key; from second round key, you generate the third round key and so on.

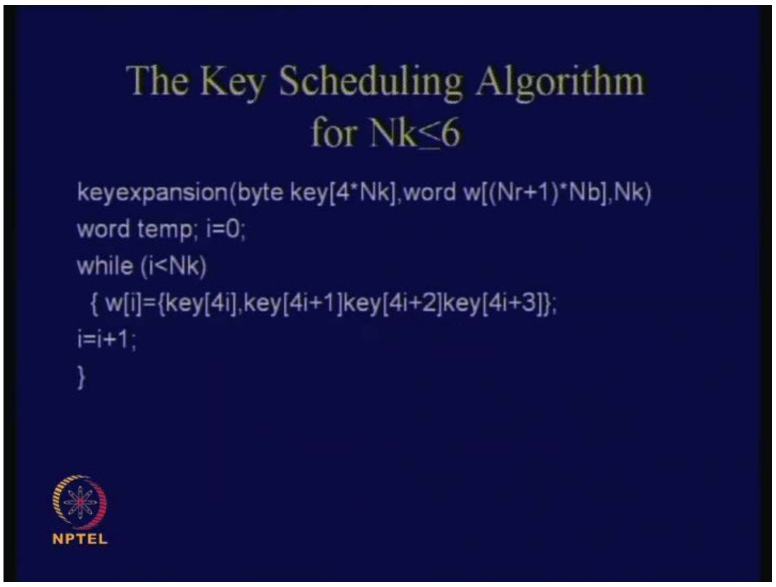
The key expansion algorithm generates a total of Nb into Nr plus 1 words. Why? What is Nb? Nb is the number of columns in the state of AES. So, that represents the number of columns. Nr was used to denote the number of rounds. There are actually Nb into Nr plus 1 words, which are being used to store the keys. Why? Because there is one key XORing at input also. After that, at the end of each round. So, there are Nr rounds, which means that there are totally Nr plus 1 number of key XORings to be done and per key XORing, there are Nb number of columns involved. So, the key expansion requires that… Nb into Nr plus 1 number of words of the keys are required to be generated from the key expansion algorithm.

Key expansion has got certain steps. The steps are as follows. It has got RotWord, which is a rotation word. It is just a cyclic rotation. Therefore, you take a word, which is like a 0, a 1, a 2 and a 3. If you do a RotWord, then that is a cyclic permutation, which means it returns a 1, a 2, a 3 and a 0. So, that is the cyclic permutation; it is just a cyclic shift. Then you do a SubWord. SubWord means nothing but the application of the sub byte operation, but you do it for each of the bytes. That is, you do it for a 1, a 2, a 3 and a 0; you do a substitute byte for each of the bytes. Collectively, I name that as SubWord.

There is also another constant, which I told you that was supposed to eliminate symmetricity in the round keys. What is it? It is called Rcon. That is a round constant and the parameter passed is i by NK. What is NK? NK is the number of columns in the key matrix; for AES-128, NK was equal to 4; for AES-192, the value of NK was equal to 6; for AES-256, the value of NK was equal to 8.

i is sort of a round number each time it is being passed. The value of Rcon i by NK is given by a 32-bit word; out of which, the last 3 bytes are essentially 0. Only the maximum significant byte is being computed and the value of that is equal to x to the power i minus 1. What is x? x is an element of GF 2 to the power 8. x being an element of GF 2 to the power 8, what we do is that we just calculate the subsequent powers of x. Using powers of x, we are generating the next round and similarly, subsequent round constants.

(Refer Slide Time: 08:34)

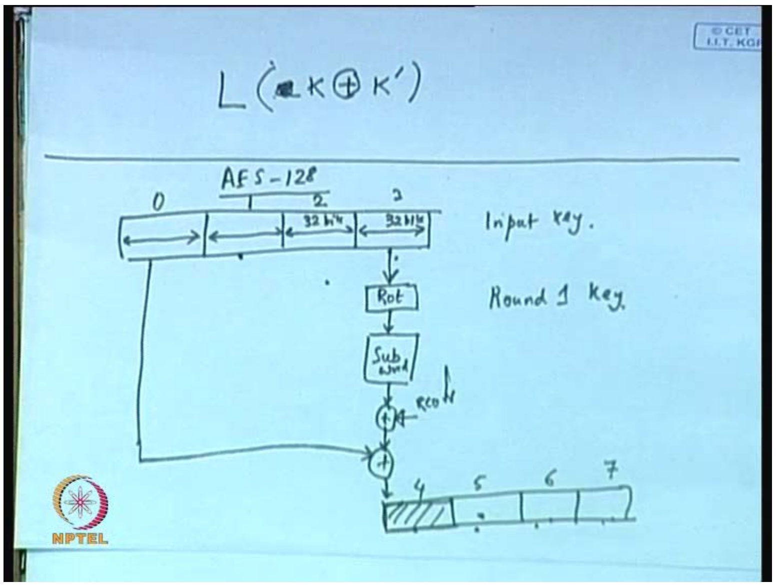


The Key Scheduling Algorithm for Nk≤6

```
keyexpansion(byte key[4*Nk],word w[(Nr+1)*Nb],Nk)
word temp; i=0;
while (i<Nk)
 { w[i]={key[4i],key[4i+1]key[4i+2]key[4i+3]};
i=i+1;
}
```

I will show you one working for that. To summarize the key-scheduling algorithm, I divide the key-scheduling algorithm into two parts. Suppose the value of NK is less than equal to 6 and suppose the NK value is greater than 6. For AES, when I say NK is less than equal to 6, there are two possibilities: NK is equal to 4 and NK is equal to 6. When I talk about NK being greater than 6, I mean that NK is equal to 8. So, just see how the key-scheduling algorithm works. It is quite simple. What it does is as follows:

(Refer Slide Time: 09:10)

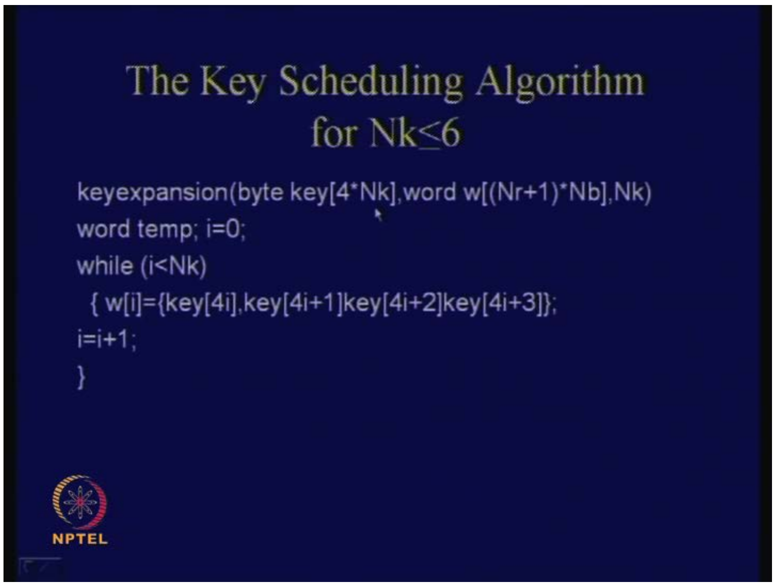


Consider it through an example. If I just see this as an example, it should be better. We have got 4 bytes in the key. So, you can imagine that the round key… Consider the example of AES-128. It can be easily generalized for the other AES versions also. So, consider AES-128. In AES-128, we have got 32 bit; there are four columns. Therefore, I can represent them as like this. We have got 32 bits and each of them is of 32 bits. Similarly, this one and this one. So, totally we have got 128 bits.

Imagine that this is your input key (Refer Slide Time: 09:55). In AES-128, what is the size of the key? 128 bits. So, from here, I have to generate the next round key, that is, have to generate the round 1 key. The question is how? It is quite simple. What you do is like this, that is, take the corresponding… If I number them like 0, 1, 2 and 3, you take the third word and pass that through certain transformations. The transformations are as follows. First, you rotate that and pass that through the SubWord operation. Then, you XOR that with something that is called Rcon. That is a round constant. XOR the output of that with the zeroth word. The corresponding output is the next word. I want to generate the next word. Therefore, what I do is I require to generate… Subsequently, I want to generate the next 4 words of the next round key. This is how you calculate this particular part, that is, the first. If I numbered them 0, 1, 2, 3, then this becomes index 4; that is the fourth word.

The next thing is that how do I generate the fifth word. You generate the fifth word by simply taking an XOR of this word; that is, the previously generated word and this word (Refer Slide Time: 11:43). That is the way how you obtain the fifth word. Now, the next question is how do I generate the sixth word? The sixth word is the XOR of the fifth word and the second word. How do you generate the seventh word? You generate by XORing the sixth word with the third word. So, that is the basic algorithm how the AES key-scheduling algorithm works.

(Refer Slide Time: 12:14)



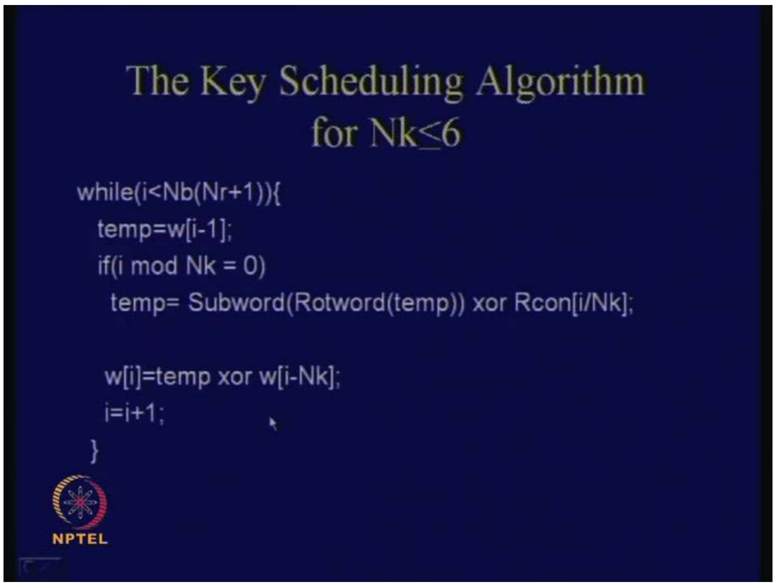
The Key Scheduling Algorithm for Nk≤6

```
keyexpansion(byte key[4*Nk],word w[(Nr+1)*Nb],Nk)
word temp; i=0;
while (i<Nk)
 { w[i]={key[4i],key[4i+1]key[4i+2]key[4i+3]};
i=i+1;
}
```

Now, you see this pseudo code; I think it should be better explained. What it does is this - it takes an input key. I have written 4 star NK. In case of AES-128, this value is equal to 4; that is, 4 into 4 means 16. Therefore, there are 16 bytes; the data type of this is byte. So, you take 16 bytes of the key and store in a word, which is supposed to be the output of the key-scheduling algorithm. Therefore, you see that the number of words, which are there is represented by Nb multiplied with Nr plus 1. Why? Because Nr plus 1 is the number of XORings required to do with the key.

There are Nb number of columns per AES key, which you are XORing. Therefore, what you do is that you take the 4 i word; that is, for example, initially, i is equal to 0. What is this value equal to? key 0, key 1, key 2 and key 3. What you do is that you just take this and dump that into an array, which is w i. So, i runs from 0, 1, 2 and 3. Therefore, what you do is that you just take the input key and dump to the initial addresses of the array. What I am doing is that I am trying to make a continuous array. So, I am storing the entire output of the key-scheduling algorithm in a linear array or in a one dimensional array. Therefore, the initial parts like w 0, w 1, w 2 and w 3 are straight away the input key, which is provided. Subsequently, I generate w 4, w 5, w 6, w 7 and so on. So, this is quite straight forward how you generate this part of the… I mean it is just that you take the input key and store that in w; as simple as that.

(Refer Slide Time: 14:08)



The Key Scheduling Algorithm for Nk≤6

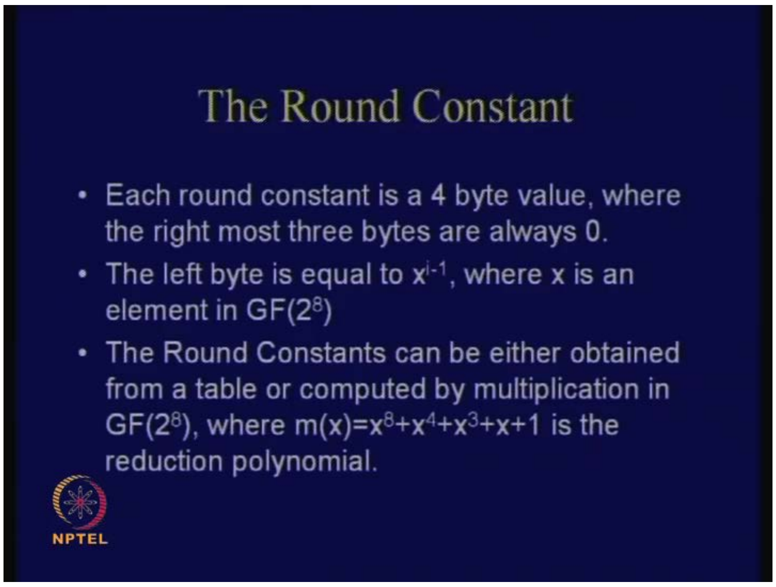
```
while(i<Nb(Nr+1)){
  temp=w[i-1];
  if(i mod Nk = 0)
    temp= Subword(Rotword(temp)) xor Rcon[i/Nk];

   w[i]=temp xor w[i-Nk];
   i=i+1;
 }
```

However, for the subsequent parts, it is required to do some operations. So, you see that if i mod Nk is equal to 0… You see that when i runs from 0, 1, 2 and 3, i becomes equal to 4. When i becomes equal to 4, then you take mod with Nk, which returns 0. That is, precisely, when you are doing this operation, you are taking temp. So, temp is equal to w i minus 1; that is the previous word. Then, you are doing some operations like doing a RotWord and followed by a SubWord and that you are XORing with a round constant.

If you do not do this; I mean if i mod Nk is not equal to 0, then you just take the previous word; you do not do any operation. Then, what do you do? You just XOR that with w i minus Nk. So, in the case of AES-128, I was taking the previous last fourth word. So, that is precisely what Nk stores, but for other values like when Nk is equal to 6, instead of taking the fourth previous word I will take the sixth previous word. Does that more or less clarify the algorithm? That is precisely what you do. Therefore, you understand that this is how you do the key-scheduling algorithm when Nk is equal to 4 and when Nk is equal to 6.

(Refer Slide Time: 15:30)

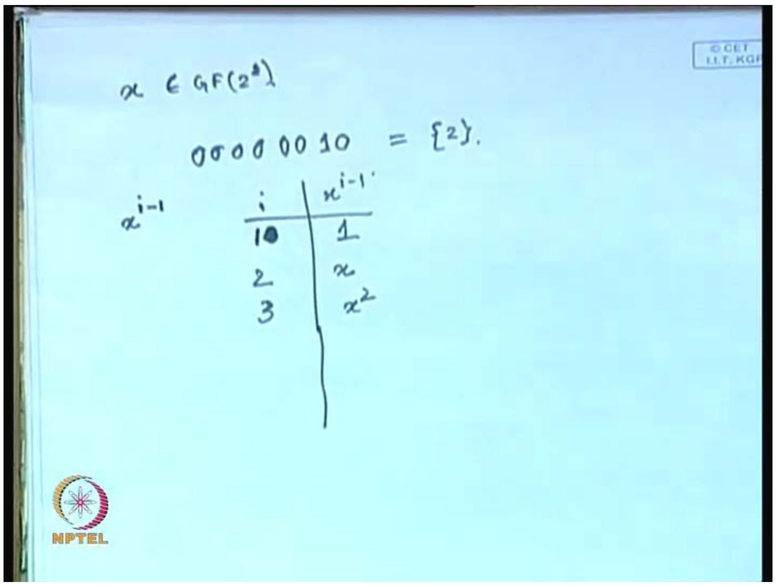


The Round Constant

- Each round constant is a 4 byte value, where the right most three bytes are always 0.
- The left byte is equal to x^{i-1}, where x is an element in $GF(2^8)$
- The Round Constants can be either obtained from a table or computed by multiplication in $GF(2^8)$, where $m(x)=x^8+x^4+x^3+x+1$ is the reduction polynomial.

Now the other thing that I will talk about is how the round constants are generated. Each round constant is a 4 byte value, where the right-most 3 bytes are always 0. Only the left byte is equal to x to the power of i minus 1, where x is an element in GF 2 to the power 8. Do you know that when I am talking about the elements of GF 2 to the power 8 and they are represented as polynomials, what does x indicate? It indicates a polynomial. So, if I consider in a decimal notation, then x would have indicated what?

(Refer Slide Time: 16:10)



$x \in GF(2^8)$

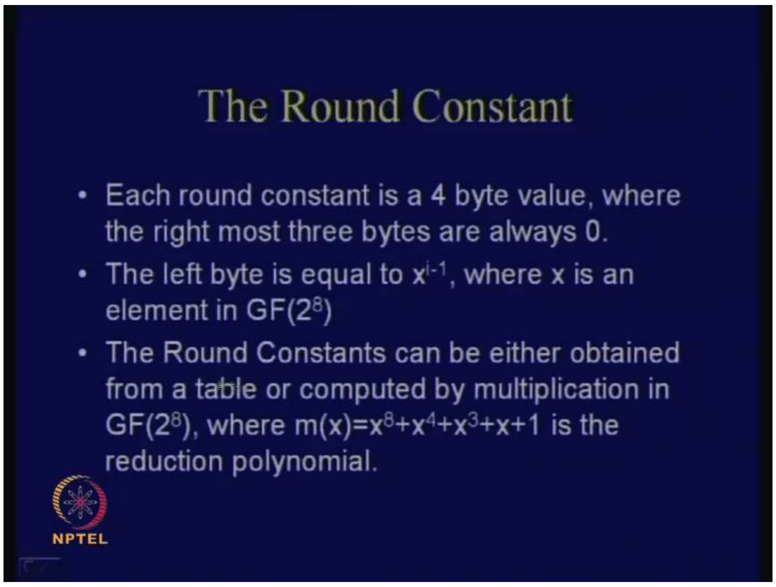
$0000\ 0010 = \{2\}.$

x^{i-1}

i	x^{i-1}
1	1
2	x
3	x^2

How could I have encoded x in x being an element in GF 2 to the power 8? How could I have denoted that? If the right one being the L S B and the left one being the M S B, then I would have represented them as 0 1 0 0 0 0 0 0; like this. That means this was equal to 2. If I just keep a binary representation, then this would have been denoted by the number 2. So, what I require is - I require subsequent powers like I require to compute x power i minus 1 when i varies. Therefore, i is equal to 1. What is the value when i is equal to 1? It is x to the power 0. So, what is x to the power 0? It is 1. Therefore, when i is equal to 1, then x to the power i minus 1 is equal to 1. What about it when i is equal to 2? It is x. What is when i is equal to 3? It is x square. Similarly, you keep on computing the powers.

(Refer Slide Time: 17:35)

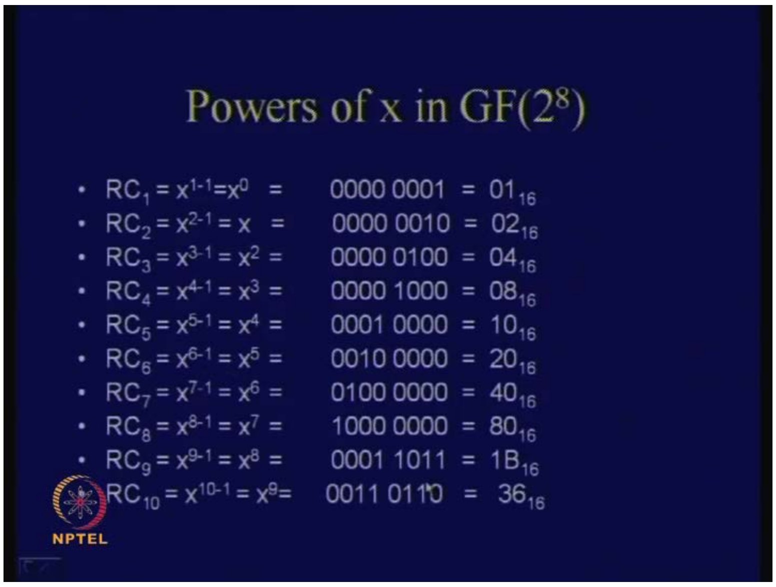


The Round Constant

- Each round constant is a 4 byte value, where the right most three bytes are always 0.
- The left byte is equal to x^{i-1}, where x is an element in $GF(2^8)$
- The Round Constants can be either obtained from a table or computed by multiplication in $GF(2^8)$, where $m(x)=x^8+x^4+x^3+x+1$ is the reduction polynomial.

Basically, what you are doing is that the round constants can be either obtained from a table, which means you can have a tabular sort of; you can remember it in the form of a table or you can do it by computing the corresponding multiplications in GF 2 to the power 8. You remember that in GF 2 to the power 8, whenever there is an overflow, it is required to do modulo with a reduction polynomial. In this case, the reduction polynomial for AES is x to the power 8 plus x to the power 4 plus x to the power 3 plus x plus 1. So, this is actually an irreducible polynomial in this field. Why? Because you cannot compute factors, which are inside the field; as simple as that.

(Refer Slide Time: 18:10)



Powers of x in $GF(2^8)$

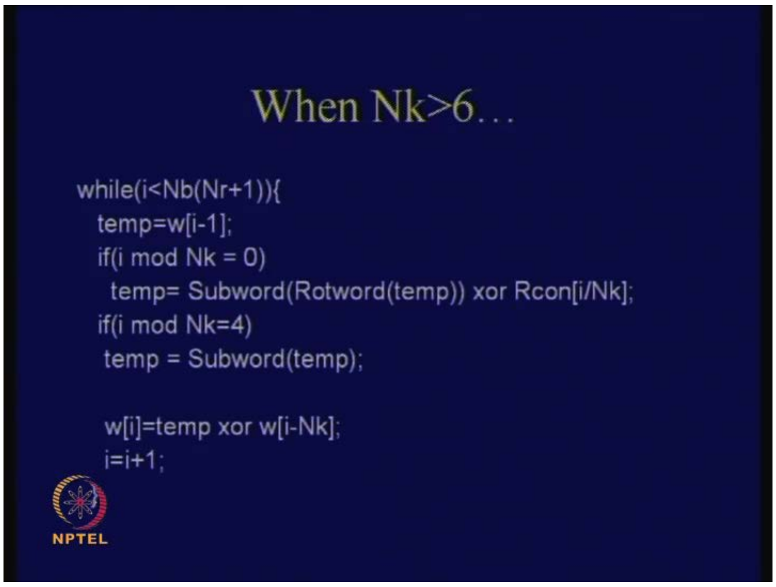
- $RC_1 = x^{1-1} = x^0 =$ 0000 0001 = 01_{16}
- $RC_2 = x^{2-1} = x =$ 0000 0010 = 02_{16}
- $RC_3 = x^{3-1} = x^2 =$ 0000 0100 = 04_{16}
- $RC_4 = x^{4-1} = x^3 =$ 0000 1000 = 08_{16}
- $RC_5 = x^{5-1} = x^4 =$ 0001 0000 = 10_{16}
- $RC_6 = x^{6-1} = x^5 =$ 0010 0000 = 20_{16}
- $RC_7 = x^{7-1} = x^6 =$ 0100 0000 = 40_{16}
- $RC_8 = x^{8-1} = x^7 =$ 1000 0000 = 80_{16}
- $RC_9 = x^{9-1} = x^8 =$ 0001 1011 = $1B_{16}$
- $RC_{10} = x^{10-1} = x^9 =$ 0011 0110 = 36_{16}

What you do is if you compute the subsequent powers, it will look like this. You compute x, x square, x cube, x to the power 4, x to the power 5, x to the power 6 and so on. So, I have stored here the 10 round constants starting from RC 1 to RC 10. You can see that in binary representations, this would have been 4 0s, 3 0s and 1. If I compute x, then this is how I denote x.

If I store in hexadecimal, this is 02 (Refer Slide Time: 18:39). Similarly, if I do power with x square, this would have been this; that is, in hexadecimal, it is 04. If I compute the next power, it is equal to like this; that is, in hexadecimal, it is equal to 8. So, actually you see that one is getting shifted. What happens after this? You know that it cannot shift beyond this, which means that there is an overflow beyond this. So, it is required to do XORing with the reduction polynomial.

You see that if you do XORing with the reduction polynomial, then you get 1s at these positions (Refer Slide Time: 19:11). Why? Because this indicates the coefficient of x to the power 4; this indicates the coefficient of x cube; this indicates the coefficient of x; this indicates the constant 1. The reduction polynomial was equal to (Refer Slide Time: 19:28) x to the power 8 plus x to the power 4 plus x to the power 3 plus x plus 1. So, subsequently after this, (Refer Slide Time: 19:33) again when you are computing the next power, it is just a shift of this like you get the one shifted here, one shifted here and so on. So, these are the constants, which are named as the round constants. When you are implementing this, you can either store this in the form of a table or you can compute them; that is, you can compute the powers of x and take a modulo if there is an over flow.

(Refer Slide Time: 20:01)



When Nk>6…

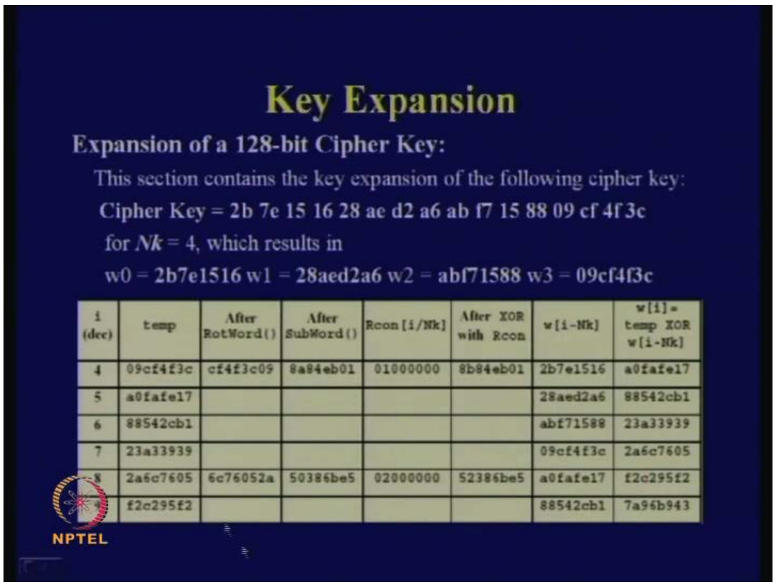
```
while(i<Nb(Nr+1)){
  temp=w[i-1];
  if(i mod Nk = 0)
    temp= Subword(Rotword(temp)) xor Rcon[i/Nk];
  if(i mod Nk=4)
   temp = Subword(temp);

   w[i]=temp xor w[i-Nk];
   i=i+1;
```

What happens when Nk is greater than 6? Entire thing remains the same except for one particular additional conditional statement. It says that if i mod Nk is equal to 4, then you do not do RotWord, you just do SubWord; that is, you just take the temp and do SubWord. So, that means when I am considering Nk greater than 6, that is, Nk equal is to 8, then when shall the value of i mod Nk is equal be 4? For example, when i is equal to 4, this is equal to 0. Therefore, this same thing happens here. I mean when i is equal to 8, then 8 mod Nk is equal to 0. So, the same thing happens here (Refer Slide Time: 20:41).

However, what happens when i becomes equal to 4? Then, it is required to do an additional step; that is, instead of doing RotWord or XORing with the round constant, just do a substitute word. The other things remain the same. So, the rest of the things are precisely as what we have discussed for Nk less than equal to 6. So, there is an additional step that is required to take care of.

(Refer Slide Time: 21:10)



Key Expansion

Expansion of a 128-bit Cipher Key:

This section contains the key expansion of the following cipher key:

Cipher Key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

for *Nk* = 4, which results in

w0 = 2b7e1516 w1 = 28aed2a6 w2 = abf71588 w3 = 09cf4f3c

i (dec)	temp	After RotWord()	After SubWord()	Rcon [i/Nk]	After XOR with Rcon	w[i-Nk]	w[i]= temp XOR w[i-Nk]
4	09cf4f3c	cf4f3c09	8a84eb01	01000000	8b84eb01	2b7e1516	a0fafe17
5	a0fafe17					28aed2a6	88542cb1
6	88542cb1					abf71588	23a33939
7	23a33939					09cf4f3c	2a6c7605
8	2a6c7605	6c76052a	50386be5	02000000	52386be5	a0fafe17	f2c295f2
9	f2c295f2					88542cb1	7a96b943

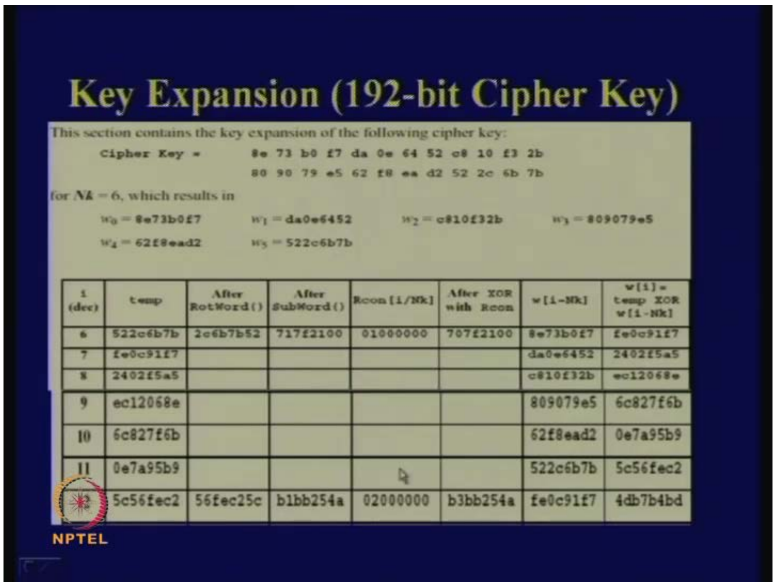
This is an example. I have given you some workouts as follows. That is, if it is required to do a 128-bit cipher key generation, then you see that the input cipher keys for example, is this as stored here. When Nk is equal to 4, you know that to do the initial part, you just dump from the input key; that is, you take w 0, w 1, w 2 and w 3. So, what you are doing is that you are just storing from the key. That is, 09cf4f3c comes straight away to w 3 and the rest of the words are just copied into the w array. So, the key scheduling actually starts after this.

What you do after this is that when i becomes equal to 4, then you know that you take the previous word, that is, 09cf4f3c. Since i is equal to 4 and you are considering AES 128, 4 mod 4 is equal to 0. Therefore, you have to do those subsequent steps; that is, you have to do RotWord, which means you have to rotate this word (Refer Slide Time: 22:09). You see that this is a rotation of the previous word. Follow that up with the substitute; that is, you have to operate the S-boxes and then XOR that with the round constant. So, what is the round constant in this case? It is 1. So, what you do is that you take this and XOR this (Refer Slide Time: 22:24). After the XOR, you get this and subsequently you can compute the next word of your round key.

For the other cases, you do not do these operations (Refer Slide Time: 22:40), you just take this and XOR with w i minus Nk and obtain the next word. This is how you repeat these steps because you can see that the round constant from here (Refer Slide Time:

22:49) has gone up to here. So, this is in hexadecimal notation. You are storing this in hexadecimal means that this is 1 and this is 2. Subsequently, here are the next round constants. So, this is how the expansion for 128-bit cipher key works.

(Refer Slide Time: 23:07)



Key Expansion (192-bit Cipher Key)

This section contains the key expansion of the following cipher key:

```
Cipher Key =      8e 73 b0 f7 da 0e 64 52 c8 10 f3 2b
                  80 90 79 e5 62 f8 ea d2 52 2c 6b 7b
```

for $Nk = 6$, which results in

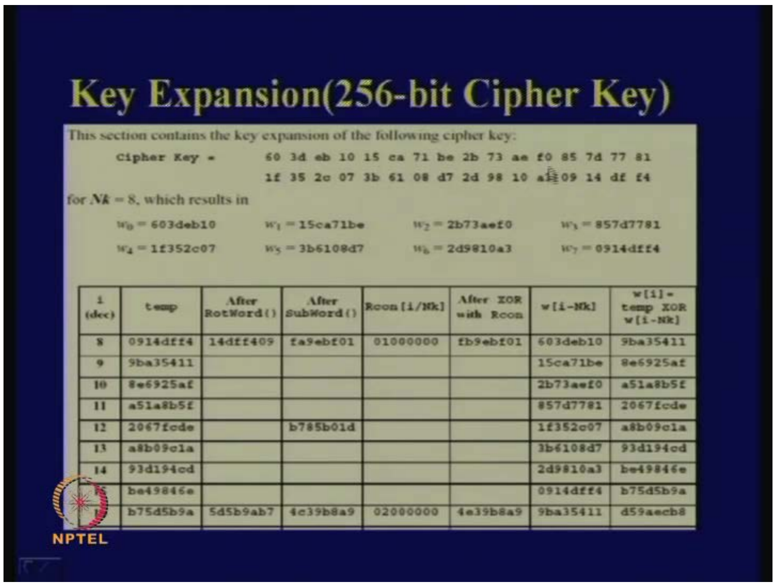
w_0 = 8e73b0f7 w_1 = da0e6452 w_2 = c810f32b w_3 = 809079e5

w_4 = 62f8ead2 w_5 = 522c6b7b

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i-Nk]	w[i] = temp XOR w[i-Nk]
6	522c6b7b	2c6b7b52	717f2100	01000000	707f2100	8e73b0f7	fe0c91f7
7	fe0c91f7					da0e6452	2402f5a5
8	2402f5a5					c810f32b	ec12068e
9	ec12068e					809079e5	6c827f6b
10	6c827f6b					62f8ead2	0e7a95b9
11	0e7a95b9					522c6b7b	5c56fec2
12	5c56fec2	56fec25c	b1bb254a	02000000	b3bb254a	fe0c91f7	4db7b4bd

For 192-bit cipher key, you see that since in this case, i minus Nk will be i minus 6, you have to take the sixth previous word and do XORing. Similarly, for your temp also, it works only when i becomes equal to 6. That is, you have to do RotWord and SubWord when XORing with the round constant only when the index of i becomes equal to 6, but not when it becomes equal to 8 because now, the value of the Nk is 6.

(Refer Slide Time: 23:40)



Key Expansion(256-bit Cipher Key)

This section contains the key expansion of the following cipher key:

```
Cipher Key =    60 3d eb 10 15 ca 71 be 2b 73 ae f0 85 7d 77 81
                1f 35 2c 07 3b 61 08 d7 2d 98 10 a3 09 14 df f4
```

for *Nk* = 8, which results in

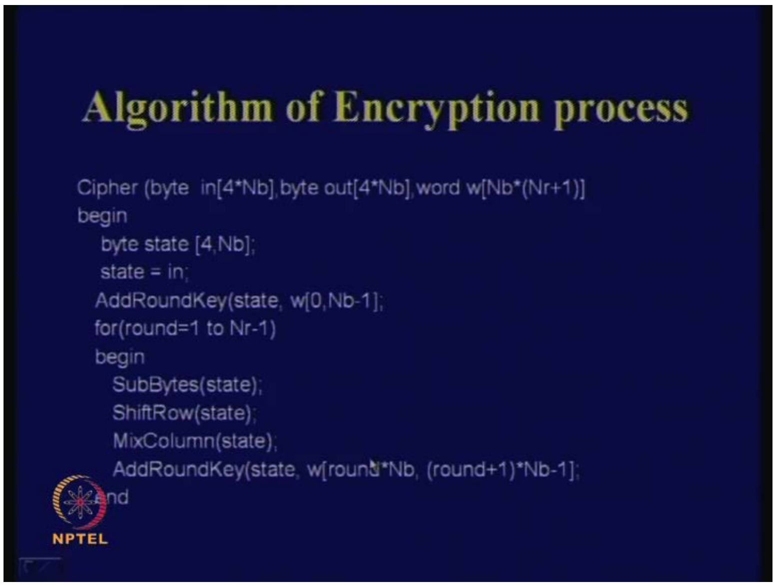
w_0 = 603deb10 w_1 = 15ca71be w_2 = 2b73aef0 w_3 = 857d7781

w_4 = 1f352c07 w_5 = 3b6108d7 w_6 = 2d9810a3 w_7 = 0914dff4

i (dec)	temp	After RotWord()	After SubWord()	Rcon [i/Nk]	After XOR with Rcon	w[i-Nk]	w[i] = temp XOR w[i-Nk]
8	0914dff4	14dff409	fa9ebf01	01000000	fb9ebf01	603deb10	9ba35411
9	9ba35411					15ca71be	8e6925af
10	8e6925af					2b73aef0	a51a8b5f
11	a51a8b5f					857d7781	2067fcde
12	2067fcde		b785b01d			1f352c07	a8b09c1a
13	a8b09c1a					3b6108d7	93d194cd
14	93d194cd					2d9810a3	be49846e
15	be49846e					0914dff4	b75d5b9a
16	b75d5b9a	5d5b9ab7	4c39b8a9	02000000	4e39b8a9	9ba35411	d59aecb8

Similarly, for 256-bit cipher key, it works only when i is equal to 8. So, when you do i equal to 8, you have do these operations. However, additionally when i is equal to 12, since 12 mod 8 is what? 4, do some other operation; that is, do not do RotWord or XORing with the round constant, but straight away do a substitute word. You XOR that with the 8th previous word because in this case, Nk is equal to 8. So, this is how your key expansion works.

(Refer Slide Time: 24:11)



Algorithm of Encryption process

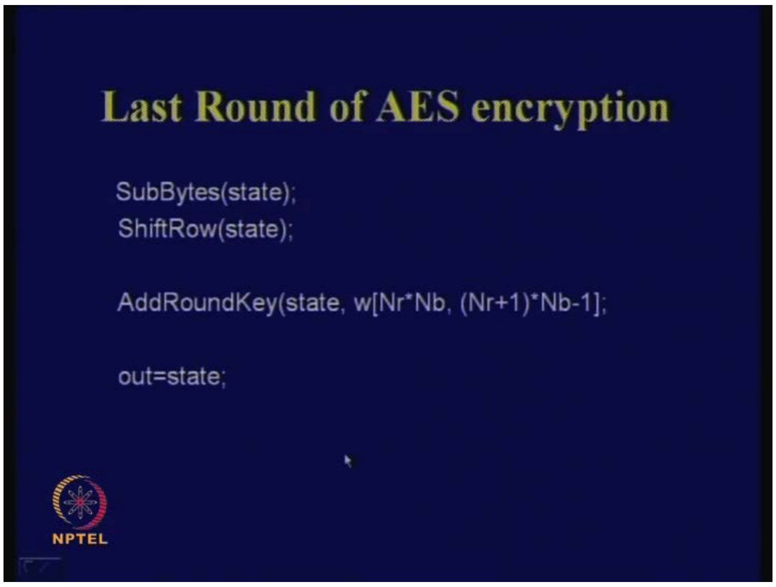
```
Cipher (byte  in[4*Nb],byte out[4*Nb],word w[Nb*(Nr+1)]
begin
   byte state [4,Nb];
   state = in;
  AddRoundKey(state, w[0,Nb-1];
  for(round=1 to Nr-1)
  begin
    SubBytes(state);
    ShiftRow(state);
    MixColumn(state);
    AddRoundKey(state, w[round*Nb, (round+1)*Nb-1];
  end
```

That is all about the AES key scheduling. So, summarizing whatever we have talked about the round constructions of the AES algorithm. I told you that there are Nr number of rounds in AES, where Nr can either equal to 10, 12, 14. There are four operations per round. What are the operations? They are substitute word, shift row, mix column and addition with the round key.

If I summarize this in the form of a pseudo algorithm, this is how it will look like. You take the in and it is required to produce the out and you store the entire key in the linear array. For example, the name of the linear array is w. Who is giving you these keys? The key-scheduling algorithm. So, you get only a small key and from there you generate a big key, which supplies the keys per round. What you do is that you take this state and you see that in this case, state is denoted as [4,Nb]. So, I have represented this as a 2-dimensional array. So, state is a 2-dimensional array of 4 rows and Nb columns. So, what you do is that you take the input, which is the plain text and you store that into a state 2-dimensional array. Then, you do an add round key with the input key. Therefore, you know that w 0 to Nb minus 1 holds only the input key.

The subsequent keys are being generated by the key-scheduling algorithm. So, what you do is that you do an add round key and the output is stored again in the state variable. Then, from round 1 to round Nr minus 1, you do these operations; (Refer Slide Time: 26:11) that is, do a sub byte, do a shift row, follow that with mix column, then do an add round key with the next round key. You see that we have excluded the last round key. Why? Because in the last round, there is a small change. What is that? Mix column step is not there.

(Refer Slide Time: 26:50)



Last Round of AES encryption

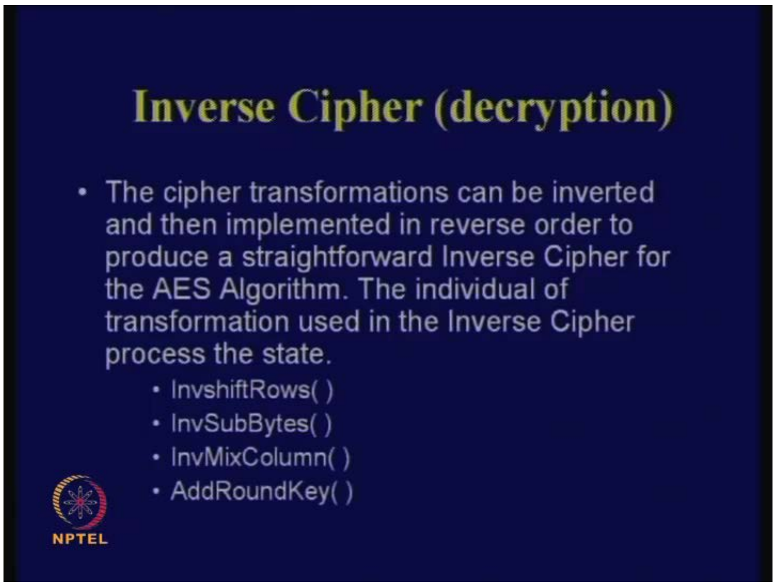
```
SubBytes(state);
ShiftRow(state);

AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1];

out=state;
```

Why? Because we discussed that in the last round, if we keep a linear layer, then it does not add to the security. So, unnecessarily we would not keep any step because this will impose performance penalty. So, without providing any extra security, we exclude that in the last round of the AES encryption. That means we have only the sub byte, shift row and the add round key with the last round key. The corresponding output gets its value from the final value in the state variable. So, this is how the AES encryption works. So, for summarizing, this is what we do.

(Refer Slide Time: 27:10)



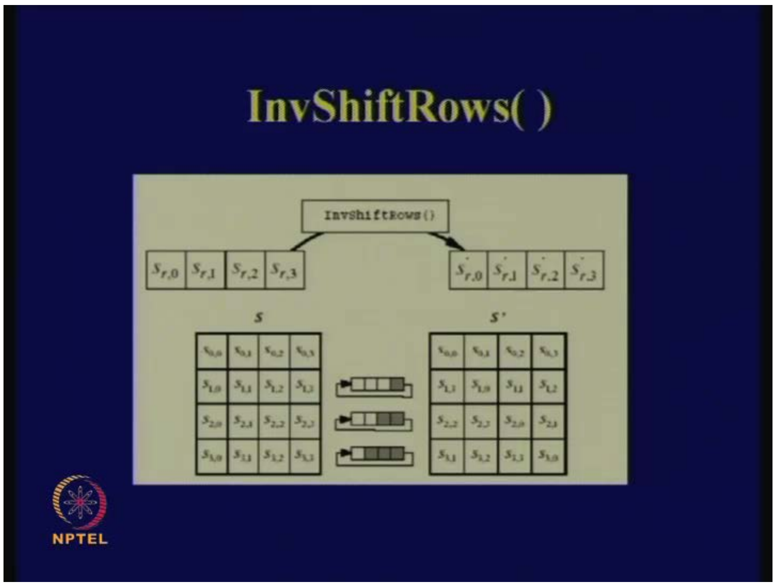
Inverse Cipher (decryption)

- The cipher transformations can be inverted and then implemented in reverse order to produce a straightforward Inverse Cipher for the AES Algorithm. The individual of transformation used in the Inverse Cipher process the state.
    - InvshiftRows( )
    - InvSubBytes( )
    - InvMixColumn( )
    - AddRoundKey( )

Talking about decryption, you understand that each step has a decryption function like if I talk about shift row, there is an inverse shift row. That is, precisely from the definition of the operations itself like shifting was just doing a left shift and obviously, the right shift would have been the inverse. Similarly, I could have done the inverse sub byte also because our substitution byte was a bijective mapping. We could have obtained the inverse also. Similarly, mix column also. There is a matrix; that is, the 2 3 1 1 kind of matrix also has an inverse matrix in the Galois fields. So, we can do an inverse mix column as well. Add round keys are self-inverse operator.

The cipher transformations can be inverted and then implemented in reverse order. So, that is quite straight forward. So, from the input, you obtain output and you start going backward. You can easily obtain the plain text from the cipher text. What is the problem in that? You require an extra hardware to do that or write a piece of code; that is, the same block cannot function both as an encryption or a decryption. Therefore, the objective that we will try to inspect is that whether we can write the decryption function similar to the encryption function. That means whether we have the same sequence of steps and can we do a decryption?

(Refer Slide Time: 28:32)

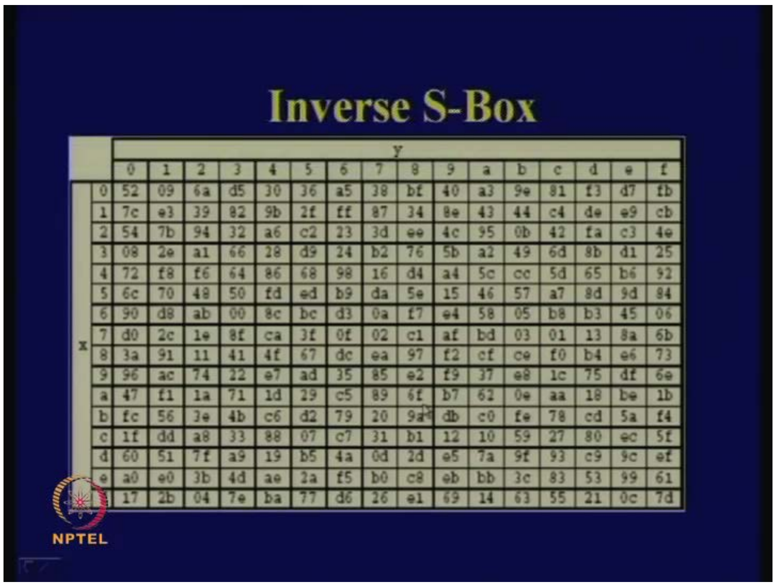


InvShiftRows( )

InvShiftRows()
$s_{r,0}$ $s_{r,1}$ $s_{r,2}$ $s_{r,3}$
$s'_{r,0}$ $s'_{r,1}$ $s'_{r,2}$ $s'_{r,3}$
s
s'
NPTEL

First of all, let us quickly go through the inverse operations. You know that shift row was defined; we have discussed how shift row was defined. The inverse shift row that you see is just the opposite transformation. So, if you are doing a left circular shift in shift row, in inverse shift row, you are required to do a right circular shift. The number of steps are same; that is, for the first row, you do not shift anything; for the second row, you shift by 1 byte; you shift by 2 bytes in the third row; you shift by 3 bytes in the fourth row.

(Refer Slide Time: 29:11)

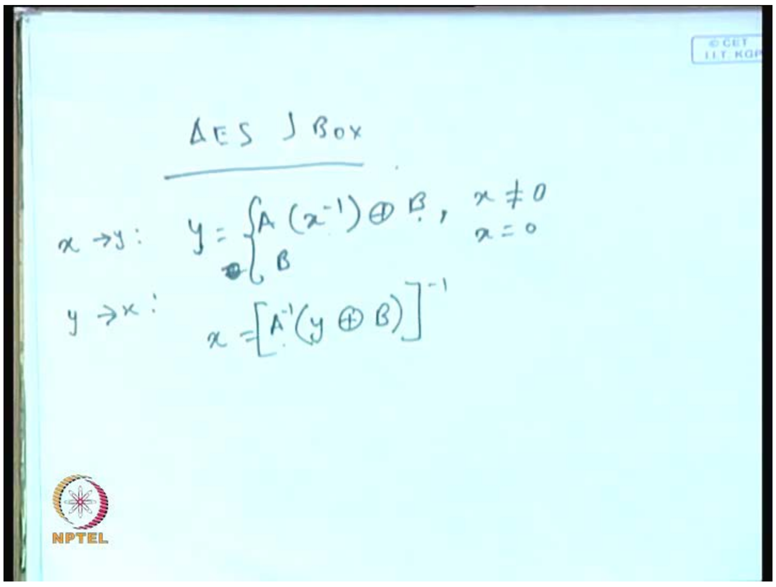


Inverse S-Box

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

You obtain an inverse S-box that is also obtainable. I am not going into the steps, but you know that inverse S-box is invertible.

(Refer Slide Time: 29:23)



AES S Box

$x \to y:$ $y = \begin{cases} A(x^{-1}) \oplus B, & x \neq 0 \\ B & x = 0 \end{cases}$

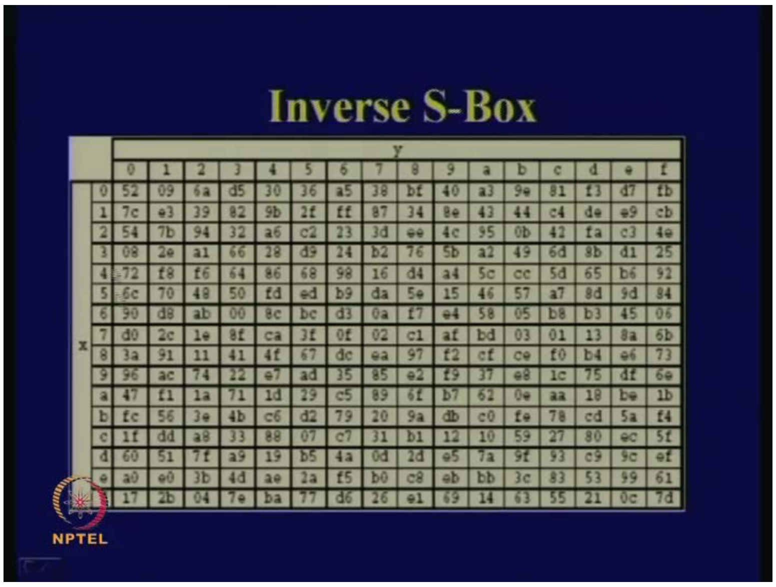
$y \to x:$ $x = \left[A^{-1}(y \oplus B)\right]^{-1}$

You can get it straight away because of this fact that the AES S-box was defined as y equal to A x inverse XORed with B. So, that was how the AES S-box was working. In order to decrypt, that is, in order to obtain, this was your mapping from x to y. Suppose I want from y to x. That is easily obtainable because we have the inverse operator working. So, this was when x was not equal to 0.

When x was equal to 0, we had y equal to B because x inverse was 0 in that case. So, you see that when it is required to do an inverse, you can do an inverse because you can do like this; that is, y XORed with B. Then, you bring it here (Refer Slide Time: 30:21) and multiply it with A inverse. Then, you take the inverse of this operation; that is, you take a finite field inverse. So, what I have done is that I have just taken y and XORed that with B.

You know that you get A x inverse. So, we pre-multiply with the matrix A inverse and A inverse does exist. You can easily understand that A inverse exists because you remember that how A was constructed; A was constructed by taking 1 0 1 vector and then we shifted that cyclically. So, that means all the rows were linearly independent and we had a full length matrix. Therefore, the inverse of that matrix exists. Therefore, if we pre-multiply A inverse and take a finite field inverse, then we can obtain back x. Therefore, AES S-box really is an invertible mapping. We already saw that its input and output were the same; that is, we have the same number of bits, but still it is a permutation. Therefore, if you know AES S-box, from there you can store or find out the inverse table as well.

(Refer Slide Time: 31:40)

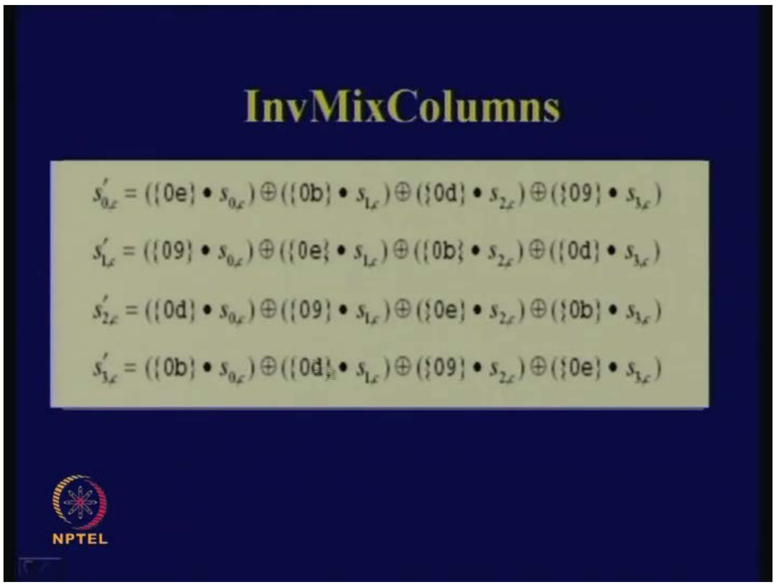


Inverse S-Box

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
x	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

The inverse S-box table looks like this. That is, it is stored here and you can check that whether it is really an inverse or not.

(Refer Slide Time: 31:47)



InvMixColumns

$$s'_{0,c} = (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c})$$

$$s'_{1,c} = (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{3,c})$$

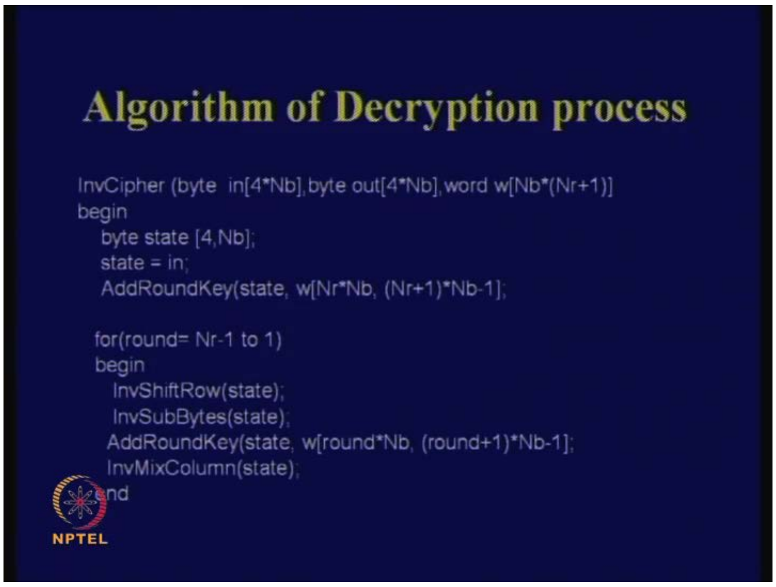
$$s'_{2,c} = (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0e\} \bullet s_{3,c})$$

Similarly, the inverse mix column is also computable, but the elements are not as simple as 2 3 1 1 that we had for mix column. We have slightly complicated elements and all the elements are infinite fields like in GF 2 to the power 8. What are the elements?

We have 0e, 09, 0d and 0b. You see that you can also store this in the form of a matrix multiplied with the vector. So, you can do this operation similar to the mix column operation, but only you have slightly more complicated multiplications to do. That is one of the reason why the inverse in AES performs poorer compared to encryption. Encryption is much more optimize and much better in terms of implementation.

(Refer Slide Time: 32:32)



Algorithm of Decryption process

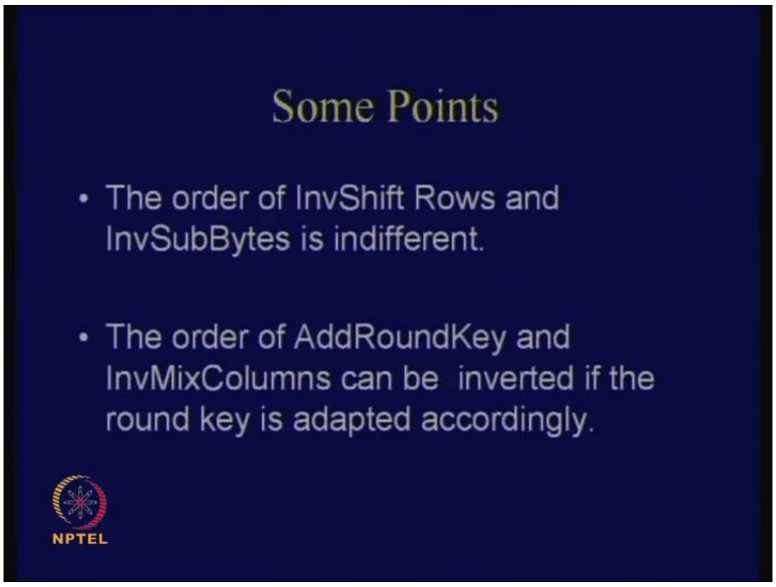
```
InvCipher (byte  in[4*Nb],byte out[4*Nb],word w[Nb*(Nr+1)]
begin
   byte state [4,Nb];
   state = in;
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1];

  for(round= Nr-1 to 1)
  begin
     InvShiftRow(state);
     InvSubBytes(state);
    AddRoundKey(state, w[round*Nb, (round+1)*Nb-1];
    InvMixColumn(state);
   end
```

Therefore, if I do a straight forward thing, the algorithm of decryption process would have been this. That is, you take the input, store it, and then you do an add round key with the last round key. Then, you follow that up with the inverse shift row, inverse sub byte, add round key and inverse mix column. So, you see that these ordering of the operations are changed.

Even if I design these blocks, I require a completely different piece of hardware to do the decryption compared to the encryption. Therefore, we will see that whether we can try to exploit any properties in these transformations and implement the decryption quite similar to the encryption function.

(Refer Slide Time: 33:18)

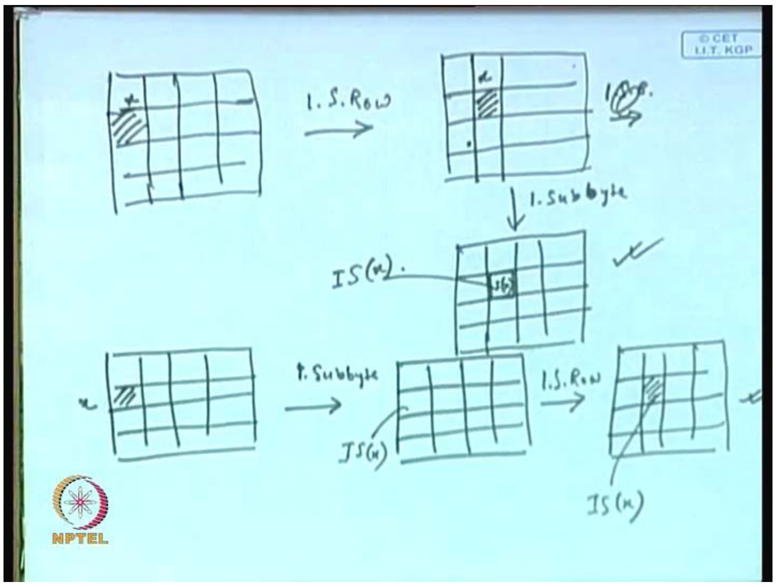


Some Points

- The order of InvShift Rows and InvSubBytes is indifferent.
- The order of AddRoundKey and InvMixColumns can be inverted if the round key is adapted accordingly.

There are two observations that we make. You see that the order of inverse shift row and inverse sub byte is actually indifferent; it does not matter in which we do. Why? Can you see it why? Because the inverse shift row; what you are doing? It is not taking place among the bytes. Rather, it is taking place among the bytes, but the inverse sub byte is a byte-wise operation. Do you see what I am trying to say? What do you do here? What you do is that you do an inverse shift row and follow that up with a sub byte operation. You do it in the other way round; that is, you do an inverse sub byte and then do an inverse shift row; both of them are exactly the same. Why?

(Refer Slide Time: 34:05)

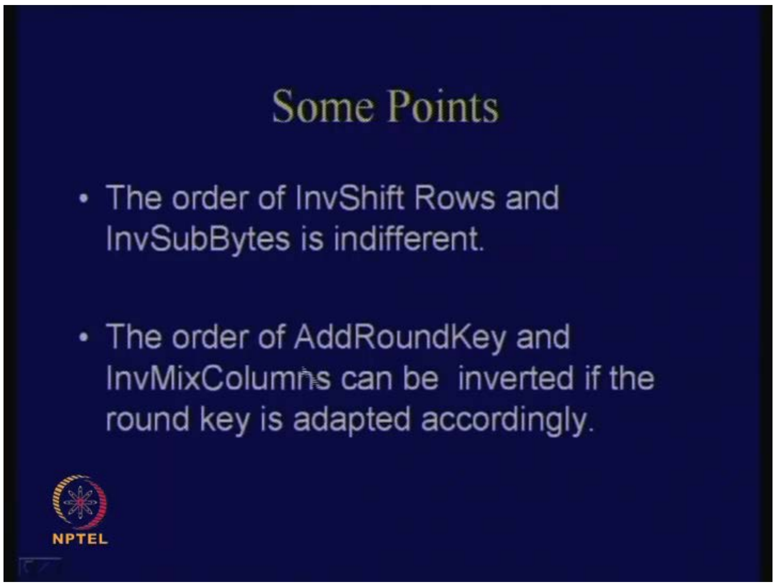


You see what we are doing here. Let us consider a simple straight matrix diagram. So, what you do here? Suppose I just do the inverse shift row, what you do here is that you take the first row. The first row does not shift. What about the second row? It shifts circularly to the right by 1 byte. This row shifts by 2 blocks and this one by 3 blocks to the right. That you follow by an inverse sub byte operation, which means what you do is that consider for example, this shaded block (Refer Slide Time: 34:46). So, when you are doing an inverse shift row, where will this get shifted to? It will get shifted to one place to the right. So, it will come here.

What about the inverse sub byte? What happens if you do an inverse sub byte here? (Refer Slide Time: 35:04) Here, you look into the S-box table and the output of that is stored in this corresponding byte location. Therefore, if this value was equal to x, here (Refer Slide Time: 35:28) you get x and here you get S of x. Maybe it is not so visible; it is actually equal to S of x. What if you do in the other way round?

What you do next is that you take this straight matrix (Refer Slide Time: 35:43) and you consider this element. Suppose this value is x, you do a inverse sub byte; I call it inverse sub byte. So, here you have got I S x. Then, you do an inverse shift row. That means this byte (Refer Slide Time: 36:22) moves to this location and here you have got I S x. So, you see that this and this are exactly the same. So, that means if you commute these two operations, it is permissible. That is, I do an inverse shift row and then follow it by an inverse sub byte. It is same thing as doing inverse sub byte and then follow that with an inverse shift row operation. That is one observation that we make.

(Refer Slide Time: 36:54)



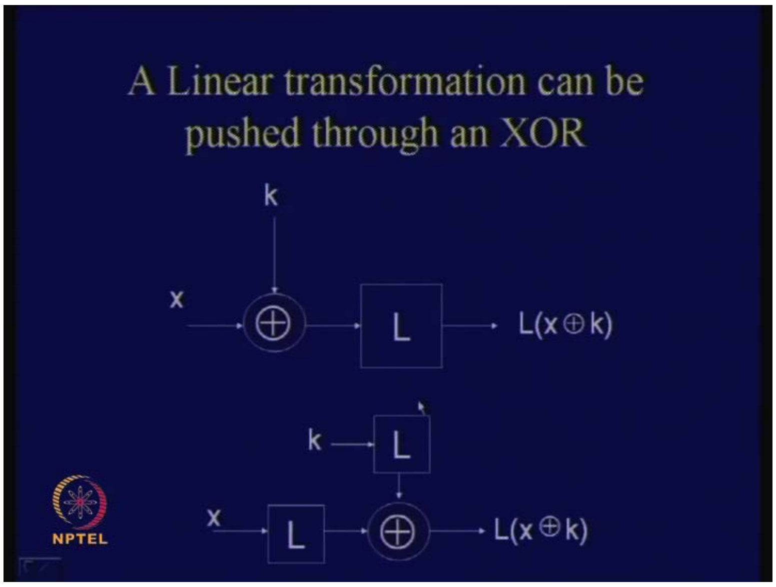
Some Points

- The order of InvShift Rows and InvSubBytes is indifferent.
- The order of AddRoundKey and InvMixColumns can be inverted if the round key is adapted accordingly.

Now, what about the last point? The last point says that the order of add round key and inverse mix columns can also be inverted if the round key is adapted accordingly. This is actually not so straight forward. What it says is that the order of the add round key and the inverse mix columns can also be swapped.

(Refer Slide Time: 37:16)

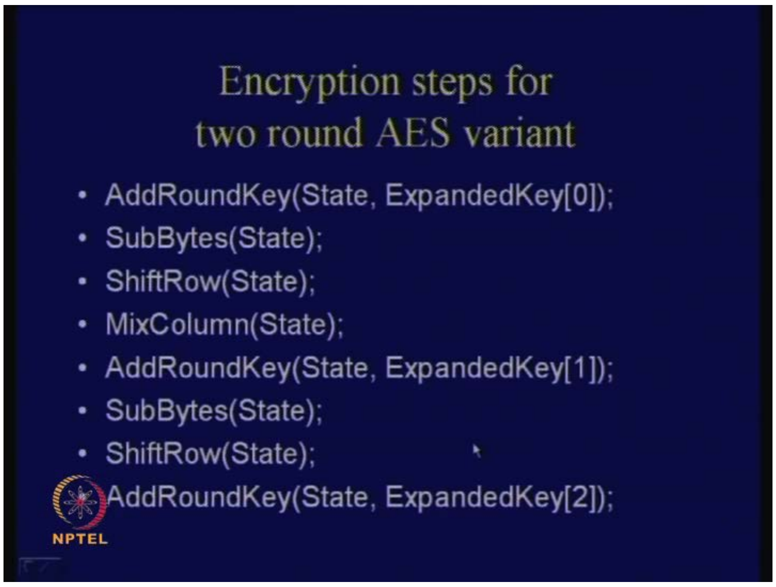
A Linear transformation can be pushed through an XOR



Probably, we can understand this if we do this small experiment. What we do is this; that is, we take x, XOR that with k. So, that is the add round key. When you are doing mix column or an inverse mix column; that is also linear transformation, which means that you take L and obtain L x XOR k.

The question is can you swap these operations? (Refer Slide Time: 37:40) That is, can you do L first and then follow it with the XOR operation. You see that you can actually push the L operator through the XOR operator. Why? Because when you push it back, you get L, but it is required to do a similar transformation on the key as well. That means what is required to do is that in order to obtain the same output L x XOR k, that is equal to L x XORed with L k. So, that means x gets transformed with L and the key also gets transformed with L. So, if you want to push back L, then it is required to do a similar modification or adaptation on the key as well.

(Refer Slide Time: 38:20)



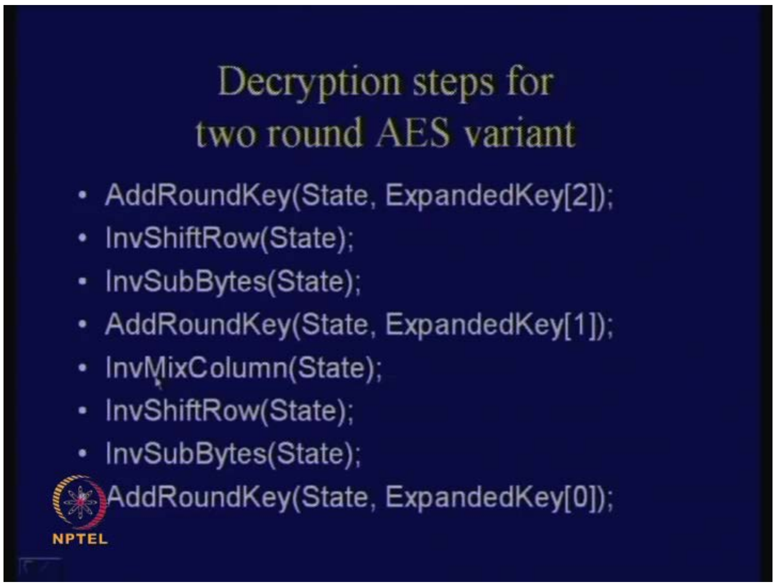
Encryption steps for two round AES variant

- AddRoundKey(State, ExpandedKey[0]);
- SubBytes(State);
- ShiftRow(State);
- MixColumn(State);
- AddRoundKey(State, ExpandedKey[1]);
- SubBytes(State);
- ShiftRow(State);
- AddRoundKey(State, ExpandedKey[2]);

Keeping these two things in mind, we just run through a two round AES variant. AES instead of being a 10 round cipher, suppose I consider a two round AES cipher. When I say that I have a two round AES cipher means that the first round has got all the four steps, but the last round does not have a mix column operation. So, in this case, the second round does not have a mix round step.

What are the operations for a two round variant? These are as follows. You see that you have got an add round key followed with a sub byte, then a shift row, mix column and again an add round key. However, now, you have a sub byte shift row, but no mix column and you have got an add round key. So, this is just a two round variant of AES.

(Refer Slide Time: 39:16)



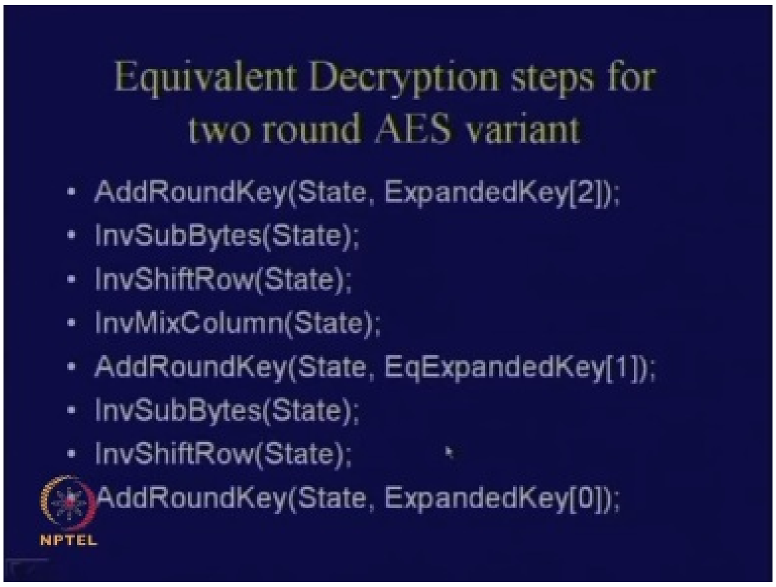
Decryption steps for two round AES variant

- AddRoundKey(State, ExpandedKey[2]);
- InvShiftRow(State);
- InvSubBytes(State);
- AddRoundKey(State, ExpandedKey[1]);
- InvMixColumn(State);
- InvShiftRow(State);
- InvSubBytes(State);
- AddRoundKey(State, ExpandedKey[0]);

What would have been the straight forward decryption of this? Straight forward decryption would have been like this - you take the second key first, follow that with inverse shift row, inverse sub byte, again an add round key, inverse mix column, inverse shift row, inverse sub byte and add round key.

Now, based upon the observation that we do, we know that we can actually swap these two operations (Refer Slide Time: 39:37). So, we can bring this up and bring this down. Similarly, we can also swap these two operations (Refer Slide Time: 39:44) with a slight change to your expanded key.

(Refer Slide Time: 39:53)



Equivalent Decryption steps for two round AES variant

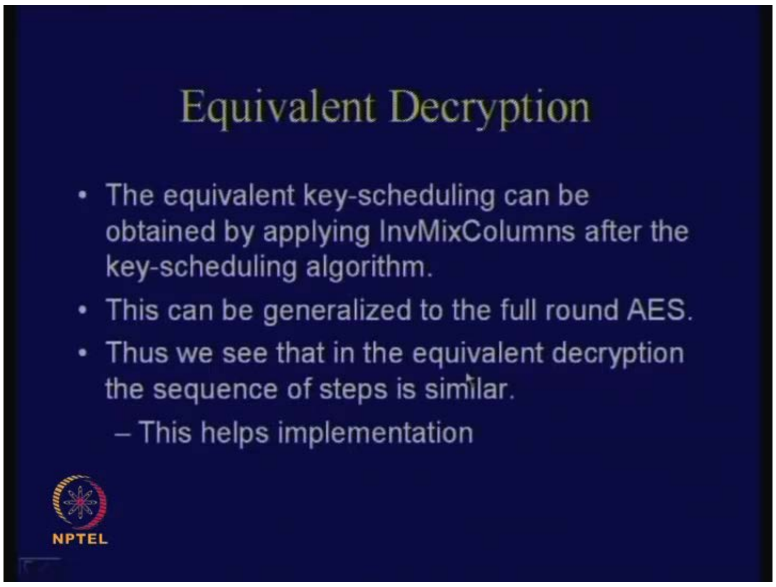
- AddRoundKey(State, ExpandedKey[2]);
- InvSubBytes(State);
- InvShiftRow(State);
- InvMixColumn(State);
- AddRoundKey(State, EqExpandedKey[1]);
- InvSubBytes(State);
- InvShiftRow(State);
- AddRoundKey(State, ExpandedKey[0]);

If you do these two operations, then what you will get is this; that is, you do an add round key and you see that you have swapped these two steps (Refer Slide Time: 39:58), and you have swapped this. Therefore, instead of writing expanded key, you have written here equivalent expanded key. So, basically, it is just a simple multiplication with the inverse mix column matrix.

Similarly, in this case also, you see that the last round (Refer Slide Time: 40:15) had a inverse shift row and inverse sub byte; we have actually made it inverse sub byte and inverse shift row. So, we have also swapped these two steps (Refer Slide Time: 40:23). We do not require any swapping here because there is no mix column in the last step. Therefore, this actually remains expanded key.

Now, you see that these sequences of operations are exactly the same as that of the encryption function. We also had an add round key, sub byte, shift row, mix column, add round key, sub byte, shift row and an add round key. Therefore, sequencing of the steps in exactly the same, but it is required to do two modifications. That is, instead of the expanded key, it is required to find out the equivalent key schedule output, which means it is required to do a small amount of change in the key scheduling function. However, the rest of the things are more or less the same.

(Refer Slide Time: 41:10)

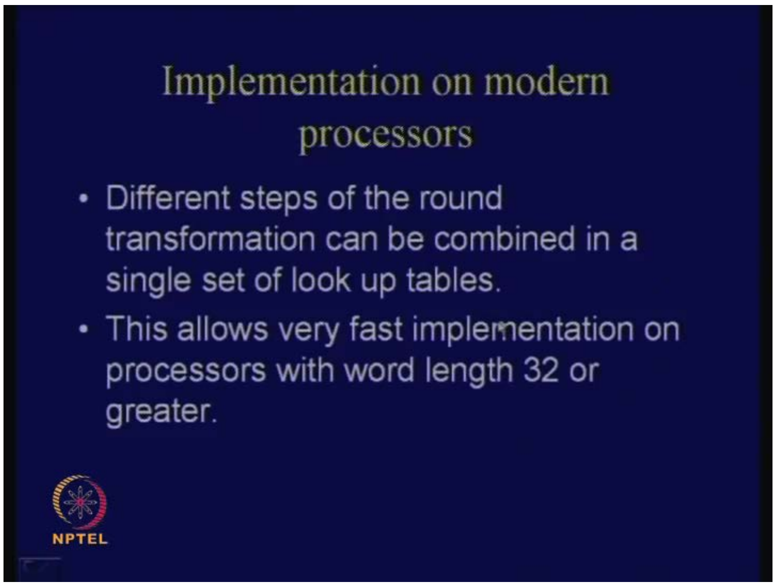


Equivalent Decryption

- The equivalent key-scheduling can be obtained by applying InvMixColumns after the key-scheduling algorithm.
- This can be generalized to the full round AES.
- Thus we see that in the equivalent decryption the sequence of steps is similar.
  - This helps implementation

The equivalent key scheduling can be obtained by applying inverse mix columns after the key-scheduling algorithm. This can be generalized to the full round AES. So, you see that the same logic can be extended to the full round AES as well. Thus, we see that in the equivalent decryption, the sequence of steps is similar. This actually helps in implementation. So, as I told you, throughout, the designers of AES had kept an objective that they wanted to make the implementation easy. That is one of the prime reasons why AES became the AES.

(Refer Slide Time: 41:45)



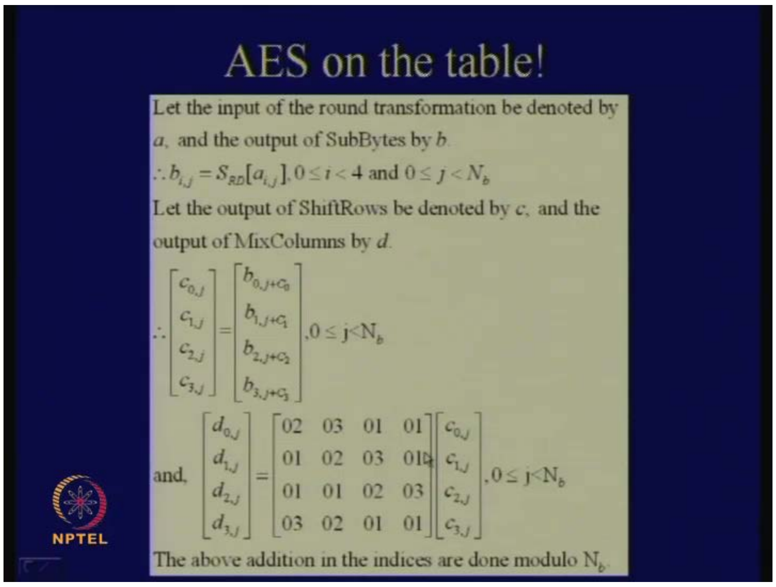
Implementation on modern processors

- Different steps of the round transformation can be combined in a single set of look up tables.
- This allows very fast implementation on processors with word length 32 or greater.

Talking about implementations will slightly conclude our talk with reflection about how do you implement an AES algorithm on modern processors. Suppose I tell you that write a code, which is quite optimized in today's processors. The first thing that we will probably try to do is a normal functional coding. I take the algorithm; I know how to right a piece of c code for example, I described that in that fashion, but there is a problem. As I told you that encryption is an overrate. Therefore, our implementations also should be good which means that should be fast. At the same time, it should be resistant against certain class of attacks, which we call side channel attacks. That is, for example, there are class of attacks, which tries to exploit the amount of timing that is required by an AES function. That is, suppose for two keys, I require 2 different timings, then immediately from the timing an attacker can try to understand where the key was say, k 1 or k 2. So, you have to make it timing resistant as well.

When you are writing a software routine; I am not talking about the hardware aspect; even then, it is required to take care of so many other factors apart from the fact that your code has to be functionally correct. You have to take care of other factors as well. You have to take care of the security and you have to take care of the implementation also. I will just try to talk about the small aspect on that. That is, I try to see that how we can derive a fast implementation on processors, whose word length is 32 or even greater.

(Refer Slide Time: 43:21)



AES on the table!

Let the input of the round transformation be denoted by a, and the output of SubBytes by b.

$\therefore b_{i,j} = S_{RD}[a_{i,j}], 0 \leq i < 4$ and $0 \leq j < N_b$

Let the output of ShiftRows be denoted by c, and the output of MixColumns by d.

$$\therefore \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j+C_0} \\ b_{1,j+C_1} \\ b_{2,j+C_2} \\ b_{3,j+C_3} \end{bmatrix}, 0 \leq j < N_b$$

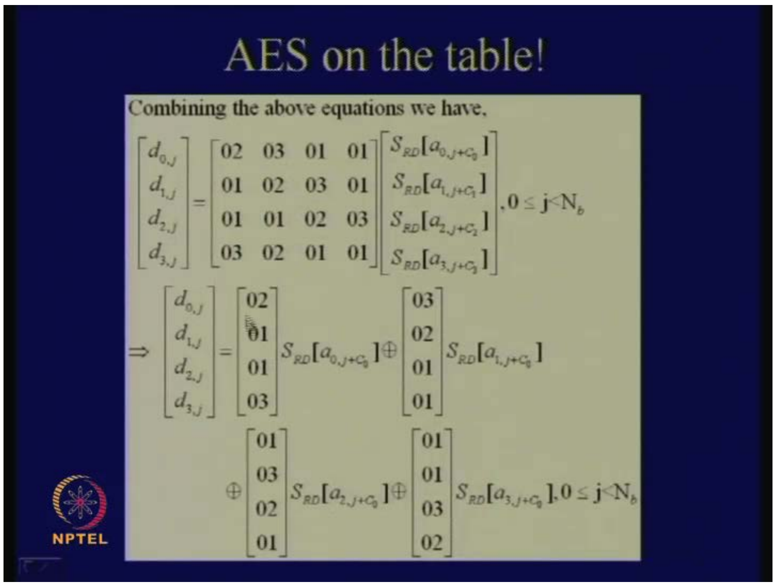
and, $$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 02 & 01 & 01 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix}, 0 \leq j < N_b$$

The above addition in the indices are done modulo N_b.

This is how I call it. So, AES solved on the table. Therefore, in order to make the AES computation fast, what we try to do is that we try to write the AES round transformations in the form of a table. Since table look up is quite fast in software, this implementation is supposed to be working much faster. What you do is this; that is, let the input of the round transformation be denoted by a and the output of the sub bytes by b. Then, you know that what I do is that given the value of a, I look up into the S-box. I denote that by S Rijndael; S RD, that is, the S-box of the Rijndael algorithm and I obtain the corresponding output b. When you see i comma j, it means that this is the row index and the column index of the matrix. The row index runs from 0 to 4, whereas the column index runs from 0 to Nb; it is actually 0 to 3 and 0 to Nb minus 1. Then, you obtain the S-box output.

The next step that you do is a shift row. Here (Refer Slide Time: 44:29) we have stored b 0 comma j plus c 0. Therefore, 0, 1, 2, 3 are the row numbers and the column numbers are indicate by j plus c 0, j plus c 1, j plus c 2 and j plus c 3. So, this is to accommodate the shift row operation. So, you know that c 0 is equal to 0, c 1 is equal to 1, c 2 is equal to 2, and c 3 is equal to 3. So, when you are doing addition operation on the index, then you know that you have to do a modulo operation also. That means you have to bring it back to the state; I mean the total data weight that you have got. For example, you cannot go beyond 3. So, what you do is that you just store them and this is the output of the shift row (Refer Slide Time: 45:14). You store them in this corresponding vector form c 0 j, c 1 j, c 2 j, and c 3 j. You store the mix column output in a similar matrix denoted by the variable d. This is your mix column matrix.

(Refer Slide Time: 45:32)



AES on the table!

Combining the above equations we have,

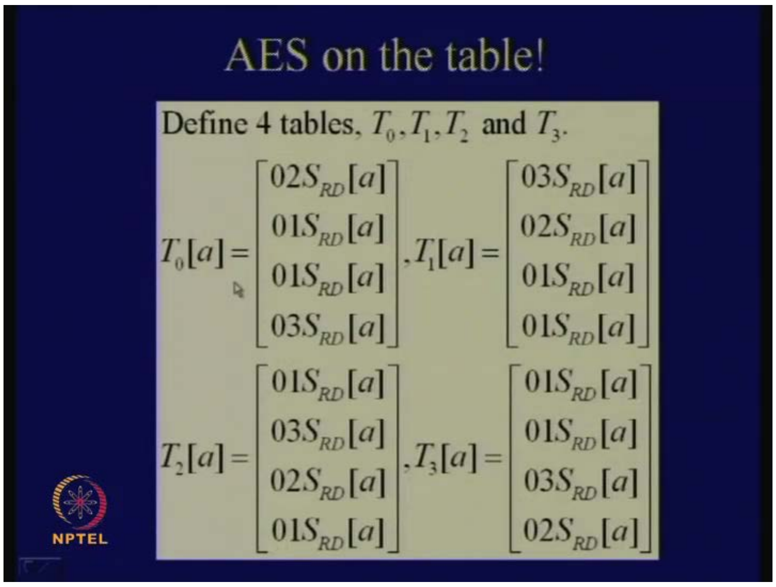
$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 02 & 01 & 01 \end{bmatrix} \begin{bmatrix} S_{RD}[a_{0,j+C_0}] \\ S_{RD}[a_{1,j+C_1}] \\ S_{RD}[a_{2,j+C_2}] \\ S_{RD}[a_{3,j+C_3}] \end{bmatrix}, 0 \le j < N_b$$

$$\Rightarrow \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} S_{RD}[a_{0,j+C_0}] \oplus \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} S_{RD}[a_{1,j+C_1}]$$

$$\oplus \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} S_{RD}[a_{2,j+C_2}] \oplus \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} S_{RD}[a_{3,j+C_3}], 0 \le j < N_b$$

Now, when you combine all these operations, (Refer Slide Time: 45:31) you would get this. This is how we combine all the operations together. Therefore, you see that you can represent each of these vectors as a linear combination of 4 table look ups. You see that you have got one table here, second table here, third table here, and fourth table here (Refer Slide Time: 45:50). Thus, that is how you derive all the 4 bytes; that is, you derive the entire column. What is the width of one column? It is 32 bits.

(Refer Slide Time: 46:07)



AES on the table!

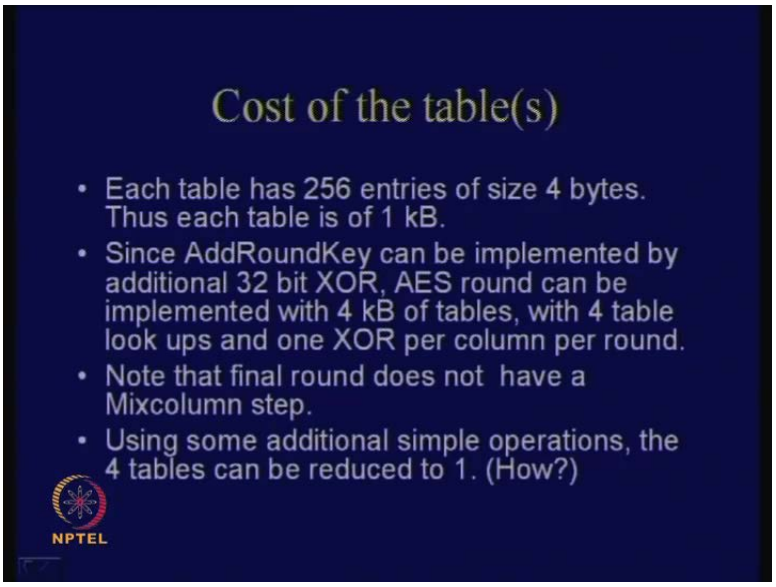
Define 4 tables, T_0, T_1, T_2 and T_3.

$$T_0[a] = \begin{bmatrix} 02S_{RD}[a] \\ 01S_{RD}[a] \\ 01S_{RD}[a] \\ 03S_{RD}[a] \end{bmatrix}, T_1[a] = \begin{bmatrix} 03S_{RD}[a] \\ 02S_{RD}[a] \\ 01S_{RD}[a] \\ 01S_{RD}[a] \end{bmatrix}$$

$$T_2[a] = \begin{bmatrix} 01S_{RD}[a] \\ 03S_{RD}[a] \\ 02S_{RD}[a] \\ 01S_{RD}[a] \end{bmatrix}, T_3[a] = \begin{bmatrix} 01S_{RD}[a] \\ 01S_{RD}[a] \\ 03S_{RD}[a] \\ 02S_{RD}[a] \end{bmatrix}$$

If I store them in the form of table, each table would be like this. I denote them as T 0, T 1, T 2 and T 3. How many elements will each of the elements in the table have? Will have 256 elements because a is actually a byte. What is the width of each table? It is 32 bits. So, there are 256 entries each of 32 bits.

(Refer Slide Time: 46:37)



Cost of the table(s)

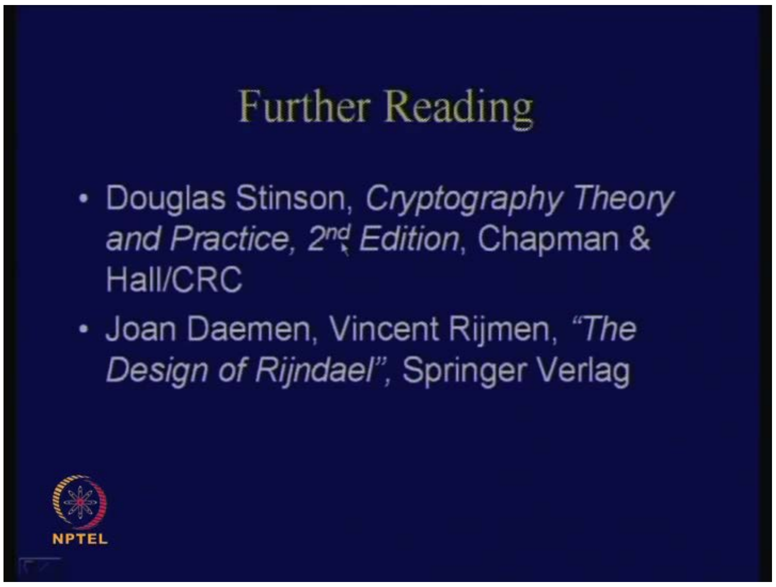
- Each table has 256 entries of size 4 bytes. Thus each table is of 1 kB.
- Since AddRoundKey can be implemented by additional 32 bit XOR, AES round can be implemented with 4 kB of tables, with 4 table look ups and one XOR per column per round.
- Note that final round does not have a Mixcolumn step.
- Using some additional simple operations, the 4 tables can be reduced to 1. (How?)

What is the total size of one table? It is equal to 1 kilobyte because you have got 256 entries and each of them is of 32 bits.

So, you can see that it will work to 2 to the power 10; 2 to the power 8 multiply with 2 to the power 2. So, it is equal to 1 kilobyte. You will find that each table requires 1 kilobyte of storage. Now, if you have got 4 tables, you would have required 4 kilobytes of table. However, one observation you can see that if you allow some additional simple operation for example, rotation, then you can squeeze 4 tables into 1 table. So, you can reduce them into 1 table because you see that all the tables (Refer Slide Time: 47:22) are nothing but rotations of each other. Therefore, if you allow cyclic rotations, then instead of having a 4 kilobyte storage, you can reduce it to 1 kilobyte. Therefore, there is always a cost that you need to pay. That depends upon what is your objective.

Note that the final round does not have a mix column step. That means in final round, we just have a sub byte look up; you do not have a mix column operation.

(Refer Slide Time: 47:51)

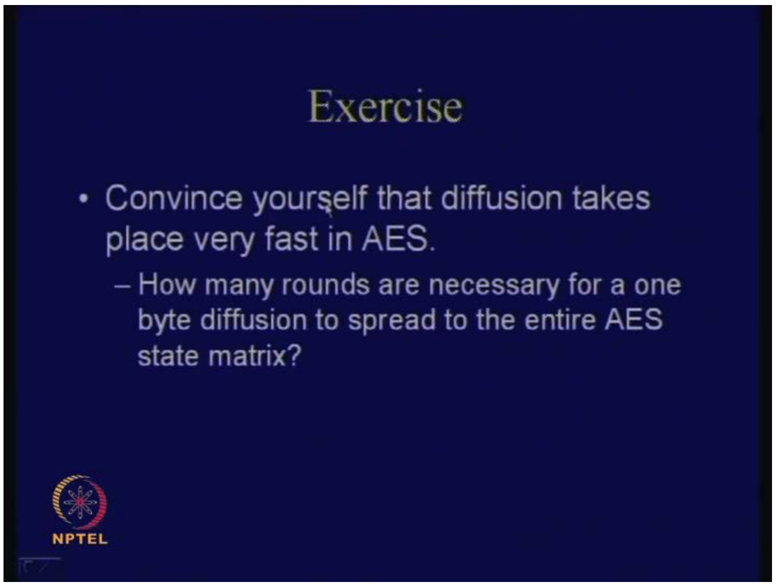


Further Reading

- Douglas Stinson, *Cryptography Theory and Practice, 2nd Edition*, Chapman & Hall/CRC
- Joan Daemen, Vincent Rijmen, *"The Design of Rijndael"*, Springer Verlag

You can do further reading from this book by Douglas Stinson. There is a book is written by Daemen and Rijmen themselves from which I have taken certain excepts. It is called The Design of Rijndael. This book is not available so freely. So, you can look into this book (Refer Slide Time: 48:06). This gives you a quite nice detailing.

(Refer Slide Time: 48:12)



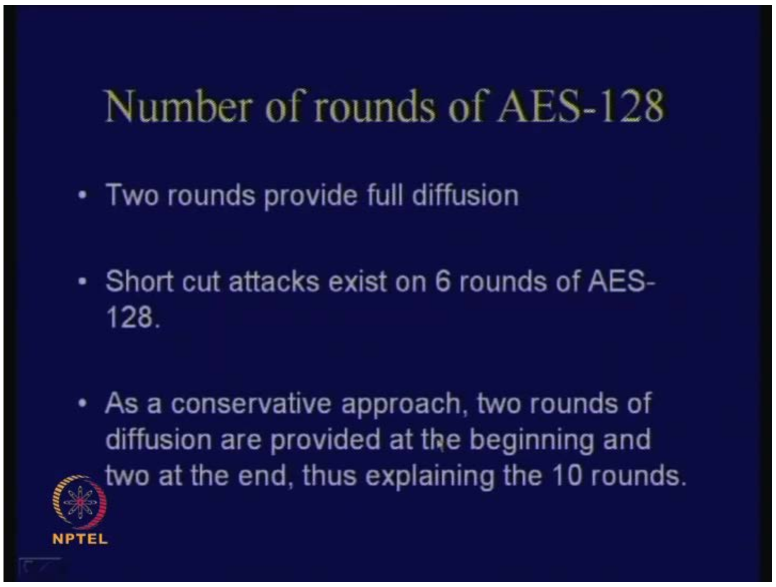
Exercise

- Convince yourself that diffusion takes place very fast in AES.
  - How many rounds are necessary for a one byte diffusion to spread to the entire AES state matrix?

Some exercises; that is, first of all, convince yourself that the diffusion in AES takes place very fast; That is, what you require to do is that to sort of (( )) out that how many rounds are necessary for a one byte diffusion to spread to the entire AES state matrix.

This is given to you as an exercise not supposed to submit, but at least you can work at your own place. Just see that how many rounds are necessary for a one byte diffusion to spread to the entire AES state matrix. So, you can just take it as an example and really appreciate the fact that the AES diffusion is quite fast.

(Refer Slide Time: 48:47)



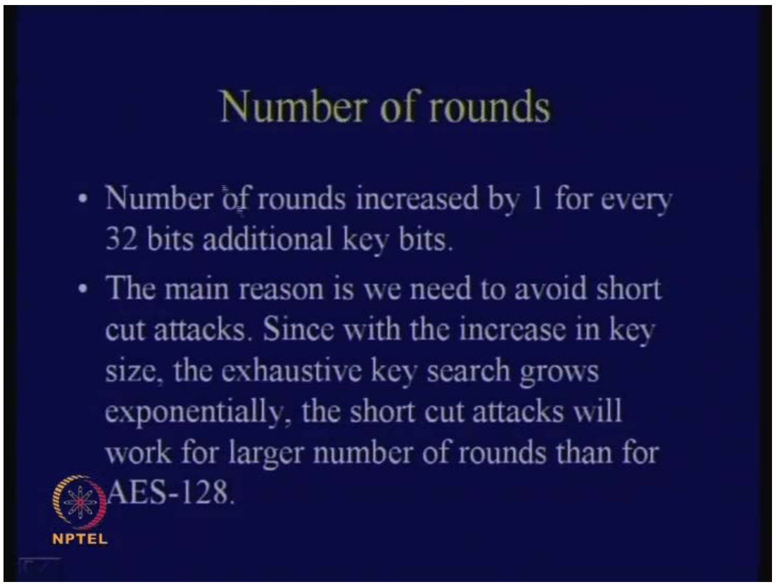
Number of rounds of AES-128

- Two rounds provide full diffusion
- Short cut attacks exist on 6 rounds of AES-128.
- As a conservative approach, two rounds of diffusion are provided at the beginning and two at the end, thus explaining the 10 rounds.

Before I conclude I will just try to reflect because I had some questions. That is, how were the number of rounds have a cipher being fixed? Although we have not really talked about cryptanalysis, I will just conclude with an idea of view for that. For example, I have given answer to the previous exercise. We will see that in AES-128, two rounds are sufficient to provide full diffusion. This means that if you just consider two rounds of AES, each output bit will be a function of all the input bits two rounds previous. So, that means you obtain full diffusion in two rounds.

People have found out that six rounds of AES 128 has got short cut attacks, which means that there are attacks that are better than an exhaustive search. As a conservative approach, it was thought that we will keep two rounds of diffusion at the beginning and two rounds of diffusion at the end. So, you have got six rounds of AES-128; for safe guard, you keep two rounds at the beginning and two rounds at end. These two rounds are supposed to provide full diffusion. It was thought that obviously I would not have kept six rounds, but I require more than six rounds because we have attacks for six rounds. So, what we do is that we keep two rounds at the beginning and two rounds at the end and totally, we have got 10 rounds for AES-128.

(Refer Slide Time: 50:20)



Number of rounds

- Number of rounds increased by 1 for every 32 bits additional key bits.
- The main reason is we need to avoid short cut attacks. Since with the increase in key size, the exhaustive key search grows exponentially, the short cut attacks will work for larger number of rounds than for AES-128.

However, for the other variants, that is, the number of rounds increased by 1 for every 32 bits additional key bits. So, we have seen that for every 32 bits of additional key, we increase the number of rounds by 1. For example, in AES-192, we had a 32 bit increase we increased the number of… What is the difference between 192 and 128? 64. So, there are two 32 bits and we have increased the number of rounds by 2. Similarly, for 256, we have again increased it by 2. Why? Because you see that although you are giving large number of keys means you are providing an additional guarantee of security. So, what you are saying is that if AES-192 is really secured, then there are no short cut attacks against AES-192. That means that there is no attack, which is better than a 2 to the power 192 search.

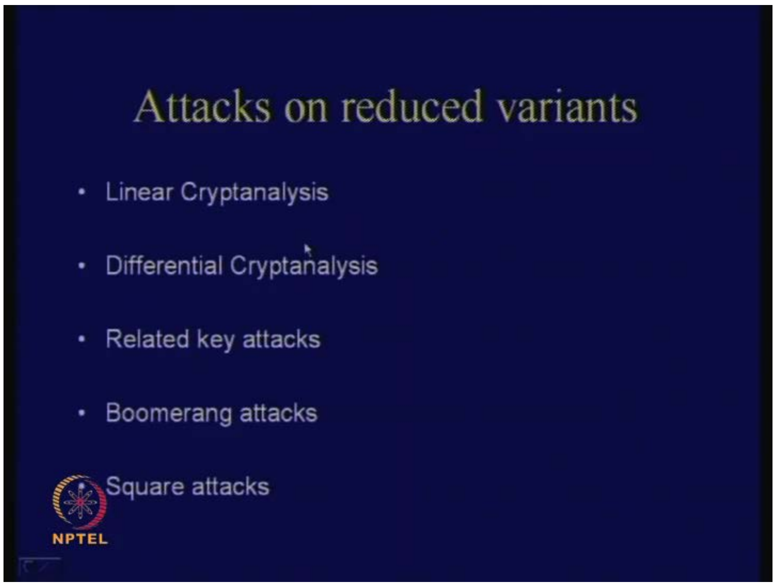
The main reason is that we need to avoid short cut attacks. Since with the increase in key size, the exhaustive key search grows exponentially, the short cut attacks will work for larger number of rounds than for AES 128. So, the last lines means that suppose we have attacks for six round AES 128, it is quite likely that if you use similar kind of properties, then you will have attacks for maybe seven rounds of AES 192. Why? Because when I am talking about an attack better than a short cut attack means what? Means that I am guarantying that there should be an attack that is better than 2 to the power 192.

When I am saying that six rounds of AES can be attacked, it means that there are attacks that are better than 2 to the power 128. However, when I am saying that for several rounds of AES 128 there are no attacks better than a short cut attack, it means that I can device an attack, which has got a complexity of work force more than 2 to the power 128. For example, it can be 2 to the power 150, but 2 to the power 150 for AES-192 would be considered an attack. Therefore, you see that for AES-192, I require larger number of rounds.

Similarly, if you consider data words like if you expand the data words… You know that in Rijndael, you can actually expand the data words also. That is, from 128, you can have 192 data; you can process on 256 bit blocks as well. For these kind of ciphers, that is, which are actually not AES, but Rijndael to be more particular; in those cases, the diffusion is quite slow. You can see that for 192, the diffusion takes place in three rounds, but not in two rounds. That means what? That means you have to provide extra number of rounds. So, that is precisely the reason why AES-192 or AES-256 or Rijndael algorithm, which work on larger blocks of plain text requires more number of rounds than AES-128.

If I summarize, AES 128 is still considered to be a more secured algorithm compared to AES-192 or AES-256. In crypto this year, there has been a paper, which cryptanalysis AES 192 and AES 256. There, people have actually been able to find out ways better than doing 2 to the power 192 tries and 2 to the power 128 tries using trails for AES-192 and AES-256. However, AES-128 is still an open problem.

(Refer Slide Time: 54:02)

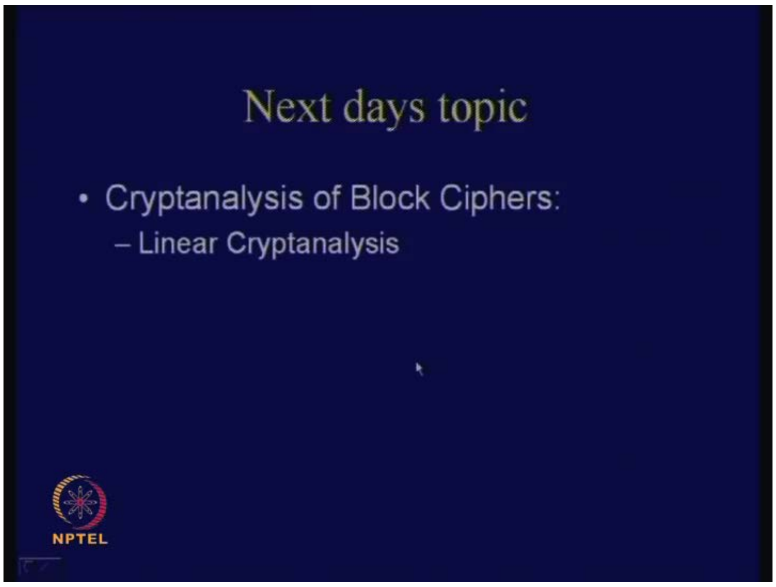


Attacks on reduced variants

- Linear Cryptanalysis
- Differential Cryptanalysis
- Related key attacks
- Boomerang attacks
- Square attacks

People have combined various kinds of attacks strategies like linear cryptanalysis, differential cryptanalysis, related key attacks; there are some attacks called boomerang attacks, square attacks, slide attacks and so on. So, we will have actually devised lot of kind of attacks. You have to basically choose something of these or the other and tailor out a really good attack on a cipher.

(Refer Slide Time: 54:26)



Next days topic

- Cryptanalysis of Block Ciphers:
  - Linear Cryptanalysis

We will talk in the next day’s class on one such class of attacks, which is called linear cryptanalysis. So, we will discuss about how linear cryptanalysis works against block ciphers.