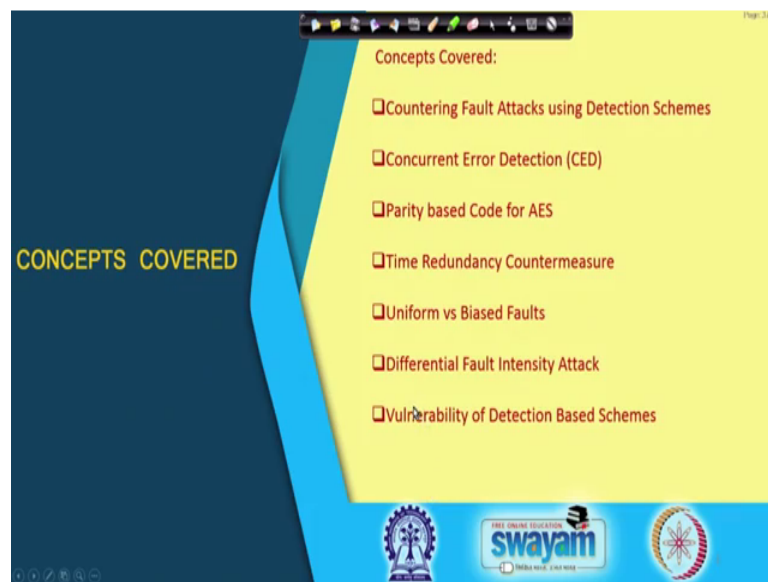
Hardware Security
Prof. Debdeep Mukhopadhyay
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 50
Redundancy Based Fault Countermeasures and Differential Fault Intensity Attacks

So, welcome to this class on Hardware Security. So, we shall be continuing our discussions on Fault Attacks.

(Refer Slide Time: 00:22)

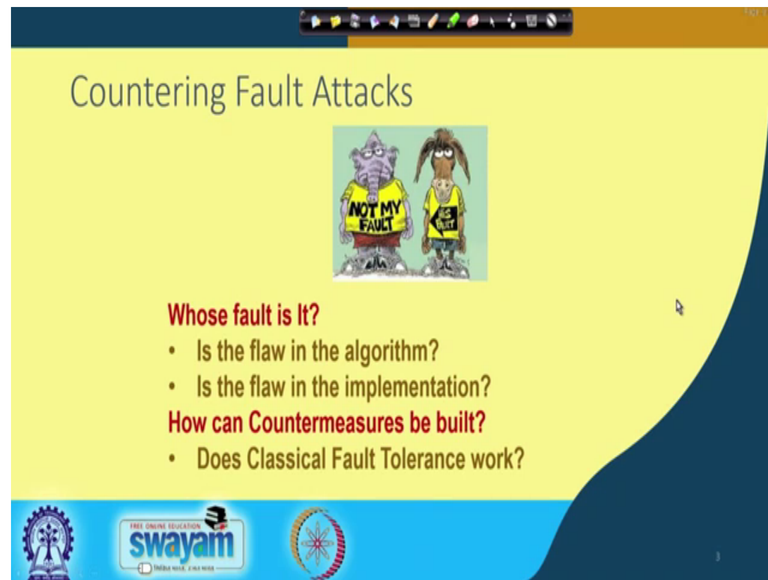


And in fact, today we shall be discussing starting our discussions about possible countermeasures against fault attacks. So, we shall be starting with countering fault attacks using detection schemes, which is the most common fault tolerance technique that is a there has been adopted for cryptography. And, it has been given a special name which is called as concurrent error detection or CED schemes.

We shall be discussing about certain forms of them and in particularly we shall be focusing about a parity based implementation for AES algorithm. And, we shall be discussing about time redundancy countermeasures and also try to kind of reflect upon the uniform fault models as opposed to something which is called as biased fault models which is very prevalent in present day fault injections.

And motivated by the by this model of bias fault attacks we shall be defining something which is called as differential fault intensity attack, which is another class of fault attacks. And finally, we shall be discussing about vulnerabilities of detection based schemes versus DFIA kind of attacks.

(Refer Slide Time: 01:25)

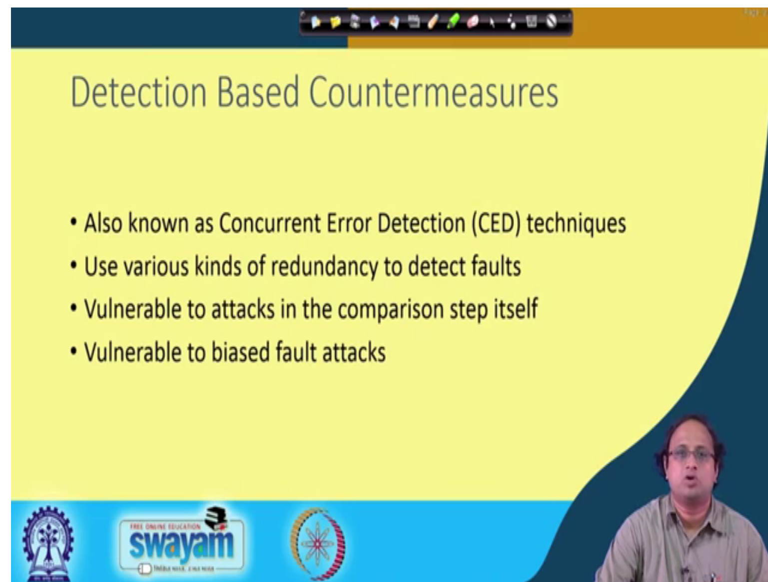


So, to start with right I mean it is very important as we have seen right to kind of address fault attacks, because fault attacks can be devastating for cryptographic implementations. So, right the question is whose fault is it? So, you know like naturally right there can be a debate about that like is it the flaw in the algorithm or is it the flaw in the implementation.

So, as opposed to like side channel attacks which is often blamed to the implementers, in context of side channel, in context of fault attacks right. I believe that there is a little bit of share for the algorithm designers also.

So, so, it is you know like. So, the question is right I mean I mean rather than you know like whose fault it is probably more what is more important is how we can build our countermeasures? So, the first thought that comes to our mind is that, we know that fault tolerance is a problem which is not only for cryptography for sure for cryptography it is it has a very devastating effect, but at the same time can we adopt classical fault tolerance in the context of psi in the context of fault attacks in the context of cryptographic scenarios.

(Refer Slide Time: 02:28)

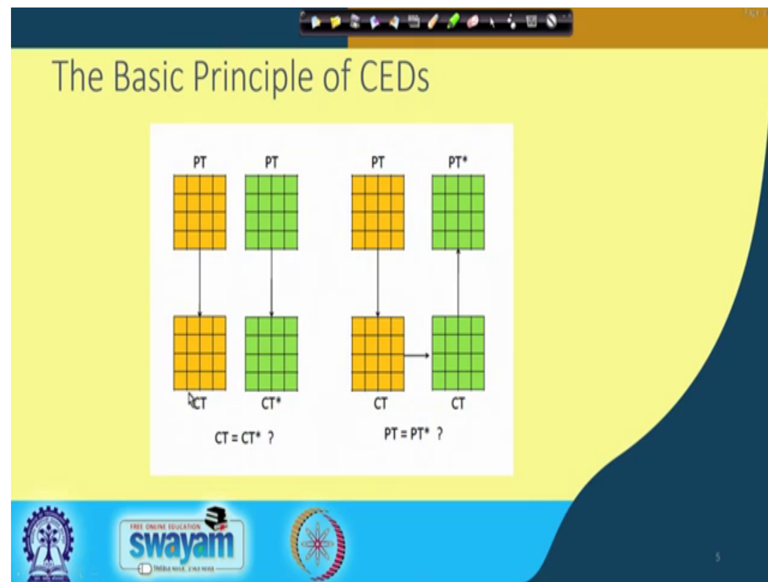


So, therefore, right first let us try to understand what is what is meant by detection based countermeasures. So, this is also known as concurrent error detections techniques or CED as it is abbreviated. It basically uses various kinds of redundancy to detect faults, redundancy is a very important you know like technique or trick to basically counter fault attacks or basically it is a very popular fault tolerance technique, which we also have tried to adopt in the context of cryptographic implementations.

So, we have already seen you know like I mean as we will see right gradually that these kind of techniques are vulnerable to attacks in the comparison step itself, because there would be an explicit comparison step and that can be often you know like targeted by the attacker. So, when we discuss about such kind of detection based countermeasures therefore, tacitly we will assume that the comparison step is flawless right. I mean we will assume that it basically gives you a correct comparison.

So, we will as we proceed right, we will see that detection based countermeasures are also vulnerable to what are called as biased fault injections or biased fault attacks.

(Refer Slide Time: 03:41)



And so, what is the basic principle of series. So, the basic principle of series very straightforward, it as this diagram shows it is essentially based on redundancy. So, like usually what we do is we are given a plaintext we get a cipher text and we were releasing the cipher text in the network, but here before we release that we will do a double check.

So, for example, you know like we can have a duplication with respect to hardware and we can kind of do this encryption in two different devices and then we compare the corresponding cipher text. If, they match then I will give you the output. The idea of the assumption is that you know like it is basically right as you we are assuming kind of tacitly then it is kind of improvable, that I will be inducing the same fault in both the cases which we need to the same cipher text.

So, later on we will see you know like this probability is not very low in many real life scenarios, but based on the assumption that you know like that all faults are equally probable, because you know like for example, suppose you have a 128 bits state in AES and if I assume that all the faults; that means, there are 2 to the power of 128 possible faults or maybe 2 to the power of 128 minus 1 possible faults because you know like 0 is not a fault.

So, if I just exclude that even if even if I exclude that right there are huge number of possible faults, and if I assume that all of the faults are equally likely then the probability that 2 fault injections will essentially lead to the same fault is really small ok.

And therefore, write this as an assumption is based on this fact; likewise right we can also do not only the for redundancy with respect to encryption, but we can also do redundancy with respect to the opposite operation, that is with respect to you know like we can do in this way like we get a plaintext, we apply a cipher text and then we take the same cipher text, and then we apply decryption. And then we make a check in the plaintext right, which with the check that whether the same plain text as we know obtained back.

So, you can you know like apply different forms of redundancies and we will see that there are huge number of possibilities, I will be trying to talk about some of the important ones in today's class.

(Refer Slide Time: 05:44)

So, first right the most obvious countermeasure would be a hardware redundancy. And as we know right hardware redundancy means, that I will duplicate hardware and I will detect fault by comparing the output of two copies. So, there are two copies and I would like to kind of kind of redo the operation and I will check naturally right. There is a 200 percent area over it, because I am kind of duplicating the hardware , but at the same time right because there is a 2 x like if the it was x now, it is 2 x basically requirement of the hardware.

So, now, there is an another alternative you know like strategy which can be useful particularly right to prevent bypassing by injecting the same fault twice. Suppose right

the adversity is trying to inject the same fault and is able to inject the same fault twice, then he would be able to kind of bypass such kind of countermeasure.

So, another possible or you know interesting solution which was proposed is depicted or illustrated in this diagram. So, the idea is that I take you know like I have got two data paths, data path one and data path two or data path A and data path B.

And then what I do is? I basically do the encryption in data path A and data path B by kind of doing by doing intermixing. For example, what I can do is I can in the classic redundancy right basically I take these states and I do an operation, I then in a second data path, I kind of take this and do the operation and make a check here, but here what I will do is that I will kind of take some states from this configuration, and I will take some states of this configuration, and then I will mix them.

So, as you can see right here I have kind of mix them and remaining thing. For example here there are some white bytes, which has been taken from this below state matrix. And, there are some dark state bytes, which has been taken from the top state matrix. And I and I develop you know like the top matrix, which is kind of a mix of the top these two state matrices.

And, likewise the remaining things like the you know the things which are not taken from here and the ones which are not taken from here constitute this state matrix ok. And, then I do a usual operation, you can already easily understand that if these and these are always matched, then I should again you know like get the same encryption or same operations for both of them.

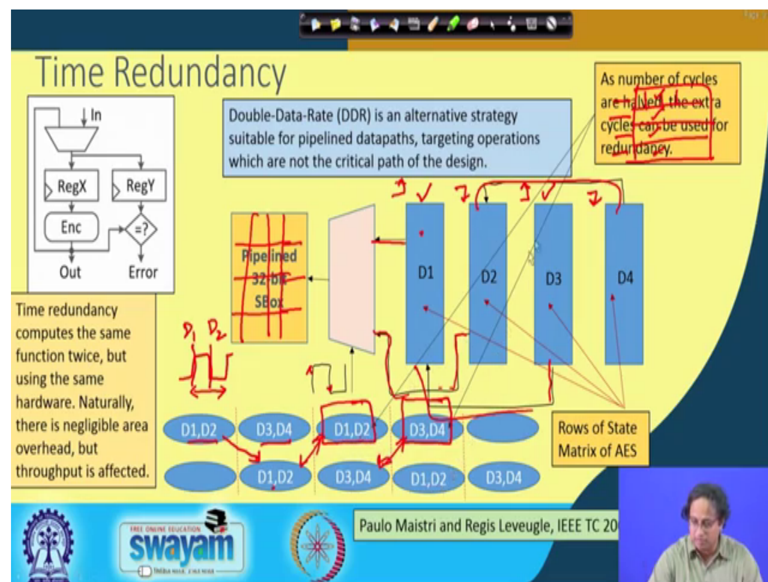
But, what is the you know like advantage of doing that remember that, that the fault attacker right is even trying to kind of bypass it. And therefore, right if it kind of you know like is has knowledge like ok; I will be trying to corrupt a specific byte for example, where the chance of corruption is more. The chance of the probability that I kind of inject the same fault in the other pair is also probable right it is probably doable, but at the same time here it becomes more difficult, because the because of this mixing ok.

So, it is be it is kind of more difficult for the attacker to control his fault injection to bypass such a strategy ok. And therefore, right this is an interesting possible

countermeasure, but at the same time right this also requires a significant amount of overhead in terms of resources.

So, therefore, right I mean the so, this is you know like a depiction of how the hardware redundancy looks like. So, as you can understand right there are two encryption bodies and I am kind of comparing the results from two encryption. So, therefore, the hardware over it is differently because of two encryption blocks that I am posing in my circuit.

(Refer Slide Time: 09:05)



So, so, the other possible strategy would be what is called as time redundancy? So, in time redundancy, we basically compute the same function twice. So, therefore, the duplication is not in with respect to resource or hardware, but with respect to time. So, therefore, right I explained time and therefore, I do it in say two time steps and you know again I do the check. For example, here I take the input, I obtain the encryption and therefore, the previous output right is kind of stored is basically stored in a register, you basically stored the previous output in a register and then you repeat the encryption ok.

So, you basically repeat the encryption and then you get the same cipher text and then you kind of compare that whether the previous cipher text is same as the present cipher text. If, they do not match then you raise a flag naturally right there is less or negligible area overhead, but the throughput is affected.

Now in order to alleviate both the throughput as well as the resource requirement, and get the advantage of time in redundancy. There was an interesting proposal which was published in I EEE transactions on computers in 2008. This technique is what is called as a Double-Data-Rate or DDR. Now, DDR is an alternative strategy which is suitable often for pipeline data paths, targeting operations which are not the critical path of your design.

So, for example, what I am is letting let me kind of illustrate this by this particular picture, suppose you have got a pipeline 32 bit S box ok. So, remember that every state right of the AES, AES matrix essentially is constituted of say 4 columns or 4 rows of 32 bits.

And, what I would like is that I would basically try to kind of you know like perform S box on each of the bytes of the state matrix right. So, therefore, right what I do is basically I take my I basically take my state matrix and try to do them one by one. So, therefore, this is my state matrix if you if you remember right, this is my state matrix of AES 128.

And, I kind of take each of the rows of the state matrix separately. Now, each of the rows again are made up of 4 bytes and that is why I write 4 into 8 we have got 32 bit S box. So, therefore, the S box essentially is also at the same time pipeline, which means right you can perform several S box computations simultaneously ok. And, again there are like 4 4 individual S boxes S box operations ok.

So, this is kind of pictographical over view on how the S S box looks like here. So, now, the key point of double data rate computation is the, is that let us you know like try to perform the AES round, you know you know like rather than. So, in the usual pipeline I will I will basically perform the AES round in one single clock ok.

So, there will be like you know like for say what I what I will do is rather you know like I will be expending 4 clocks, because I will be you know like taking 1 clock for this maybe, another clock for this, another clock for this, another clock for this. So, now, what I will do is rather than you know like I will basically try to reduce the number of clocks, by operating this at both the positive edge as well as the negative edge.

So, that would imply that what I will try to do is for example, I will try to kind of operate these two, that is alternative you know like stages in the pipeline in a specific clock edge

maybe you know the positive clock edge, but on the other hand right the other two I will try to operate them at the negative clock edges ok.

So, so, I will basically try to kind of interpose them in this fashion ok. So, if you observe this diagram here so, this is my you know like state row say D 1. So, this is a first row. I basically operate them you know like here as this arrow shows maybe you know like I just operate them here at the positive edge. And in the negative edge, I basically start operating this as you can see that there is a forwarding here from D 2.

So, therefore, right in the first stage I basically do both D 1 and D 2 ok. So, I am basically because you know like in the first clock cycle, if you see right the first clock cycle is made up of this, here I am doing D 1 and here I am doing D 2.

So, basically I am performing D 1 here and I am performing D 2 here. So, that kind of you know like helps me, that to do you know like both D 1 and D 2 in 1 clock cycle. So, what is the advantage of this the advantage is that, I am basically reducing the number of clock cycles ok.

Of course, right there will be a cost or a you know like an effect on this will be on the critical will be on the frequency of your clock that will be affected, but as I say that assuming that this is not you know like exactly in the critical path. So, I would expect that there would be a kind of like a small effect on the critical on your frequency, but not a significant 1.

So, because you know like it is not in the I am assuming that this part of the circuitry is not in my critical path. So, therefore, right I mean if that is so, then the advantage here is quite evident, because now since you have basically reduce the number of clock cycles, you have now some extra clock cycles. Now, what we will do is that in those extra clock cycles, we will be doing a redundant computation.

So, this basically will have a good effect on you know like redundancy as well as throughput. So, you can see here what I will do next is in the next clock cycle; I will start processing of D 3. So, I will kind of forward D 3 to this block and I will forward D 4 to this block and again I will repeat this process ok.

So, I will be doing D 3 D 4 and remember that D 1 D 2 is going into the next stage of the pipeline, because these S box is pipelined ok. So, now, what happens is in the next clock cycle, I basically start doing a redundant computation of D 1 D 2. So, therefore, right I mean if I and again in the next clock cycle I do a redundant computation of D 3 D 4, and then I kind of verify whether it kind of matches with my previous computation. So, here I basically do a match and here also I do a match.

So, basically I am kind of doing you know like redundancy, but at the same time you know like you are essentially doing it simultaneously ok. So, therefore, you know like what we so, therefore, this is essentially you know like, the redundant blocks and as you can observe that you know like there is so, you can observe that as a number of cycles are half the extra cycle can be used for redundancy ok.

So, this is essentially where you are doing two redundant computation. So, these are the redundant computations. So, we can generalize this process of course, and you know like a pretty much this can be generalized for any pipelines where this condition is satisfied.

(Refer Slide Time: 15:53)

So, there is another interesting improvement on redundancy and that is called as hybrid redundancy. So, in hybrid redundancy right there are generally two forms of proposals, in one form they kind of assume that in a chip right we both have encryption and decryption, because normally a chip would probably work as a transceiver or as a part of transceiver. So, there will be a you know like a in both an encryptor as well as decryptor.

So, if it be so, then what we can do is that we can leverage that and we can use the encryption block and the decryption block to make a check. For example, right I will take an input x, I will encrypt it I will give it y and then I will kind of put y to the decryption engine, and I will kind of decrypt it, and I will verify that whether it matches with my first plaintext ok. Now, there is another interesting proposal which is called as recomputing with permuted operands or repo.

So, this particular style of design right is essentially based on again suppose we target the inside in the encryption process. We basically try to kind of study a useful permutation for the in for encryption ok. For example, you can see that this is an example that we illustrate here with respect to AES. So, if you for example, consider the state matrix of AES, suppose the plaintext of AES and apply a permutation as shown here ok.

So, in this permutation you can see what has been done is that so, you know like it is kind of an you know so, basically you if you observe here that this is your state matrix like showing the 0 th row, 0 th column, first column, second column and third column.

And, the permutation basically identifies an invertible permutation is basically where you do a you know like cyclic left shift. So, basically as you can observe here this 0 1, you know like basically kind of shifts over here, the 0 2 shifts over here, the 0 3 shifts over here, and a 0 1 comes over here ok. So, you basically do a corresponding shift ok.

Now, if you do this shift right then what the property is that and if you denote it by say the permutation P for example, then if you basically apply take a state matrix x S for example, and if we apply the AES round. And, then we apply this permutation it is the same as if you know take the state S apply permutation and then apply the AES round ok. So, this would give you the same result in both the cases.

So, the advantage here is that you can easily kind of once you have this kind of property, then you can take this and you know like use it to kind of develop an architecture, where rather than having two encryption and decryption blocks. Now, you have got two parallel paths; in one you basically just do a normal operation. In the other case you basically perform this permit permuted operation ok; that means, you basically perform a redundant operation with this permutation.

So, you take the state x you basically permute it you apply encryption and then we apply the inverse permutation. So, you should get back because you are applying p inverse on AE AES round and PS, you should get the same as that of your original output ok, and then you match that and you see whether there is an error.

So, this design right apparently has got a very good throughput with respect to most of the attacks that we would we that we have exploited like single byte faults for example, and it has got an area overhead of around 20 percent for an iterated AES implementation ok.

So, therefore, it is a very nice tradeoff between overhead and fault tolerance. So, this paper was published in that in 2012 and these are reference corresponding to this work.

(Refer Slide Time: 19:28)

So, now there is another very important class of redundancy and that is called as information redundancy and it is based on error detection codes ok. So, the idea here is that I take an input text and I perform an operation. So, I basically have got some kind of you know like check bit generator. And the check bit generator basically you know like is used to predict that some check bits that would happen after this computation.

So, therefore, I basically take the check bit again apply the check bit generator after the operation, and then I basically make a match whether my prediction predicted check bit matches with my with the output of the check bit generator after the actual operation has

been done. So, therefore, right I mean one of the very common forms of such kind of checks right is basically what is called as parity based checks ok. Where we basically perform parity the other common form of checks is what is called as residue base checks.

For example, we have heard of CRC and stuff like that ok. And it can be applied at different levels you can kind of adopt it for word level, byte level or nibble level as your requirement so. So, therefore, right I mean this kind of gives a very very optimized design, because as you can understand that if you want to for example, calculate the parity you can have very efficient design architectures for doing so, but at the same time right you do not get 100 percent fault coverage, you essentially get much lesser compared to that.

(Refer Slide Time: 20:55)

So, here is an example of trying to do a parity based code for AES and I will not go into all the details, I will leave few things for you to kind of think over and as an kind of exercise. For example, the shift row is very easy, because you know that if you have got a parity bit of the state matrix. So, parity is you know like very easy to implement, because what I can do is I can for a it is very easy to kind of adopt it for AES. Because what I can do is I can adopt the parity for every bit of the AES matrix ok.

So, what I can do is basically I can take the AES state and for every bit of the AES state matrix like for this bit or this bit or so on, I basically can have a corresponding parity bit. So, I can have a parity bit like p i j ok. So, so, therefore, right I mean as you can

understand that the, for shift row right is very easy because if you have got a sparity state matrix.

So, let me call this state matrix with parity as parity state matrix. If I apply a shift row then this parity state matrix state values will just get shifted accordingly ok. So, it is pretty easy to adopt. Likewise for, but the hardest thing right is probably for the S box, because of the S box of the fact is that the S box is a non-linear transformation.

So, therefore, right what is done is that we basically kind of kind of elevate this by kind of increasing the storage remember that the S box right, if basically a storage where I have got 8 bits of input right, I have got 8 bits of input and I have got 8 bits of output that was my usual AES S box right. So, now, what I do is basically I keep an extra check or extra bit and this is one extra bit that I keep and this is a parity bit ok.

So, again I have got 256 possible locations, because is 8 bits cannot do 256 possible values, but then right the table becomes 256 cross 9. So, previously it was 256 cross 8. And, now because of this parity storage it becomes 256 cross 9 ok. And, that is not you know like I mean probably it can be afforded. And therefore, right we are expanding the S box to do 256 cross 9 and by adding this output parity bit we can propagate in incoming errors rather than having to check explicitly ok.

So, but; however, right it is interesting to see how it works for the mix column operation ok. Because in mix columns as we know that we have basically performed the Galois field operations ok and, here are some equations for doing so, for mix columns of AES.

So, let me try to kind of work this out and see whether we can understand how these equations are derived.

(Refer Slide Time: 23:52)

So, basically right I mean what I will do is basically I will take the AES state matrix ok. So, imagine that I will be just considering you know like 1 column of this and the same thing will be applicable for all other things. So, let me consider these as S 0 S 1 S 2 and S 3 ok. And, we will be applying the matrix on this so, remember the mix column matrix, it has the S 2 part of elements 2 3 1 1 2 3 1 1 2 3 1 1.

So, now so, let us consider the effect of so, so basically like if I multiply this right and just consider this column, then it is 2 S 0 plus 3 S 1 plus S 2 plus S 3 like why this S 0 plus 2 S 1 plus 3 S 2 plus 3 S 2 plus S 3. And, likewise this is S 0 plus S 1 plus 2 S 2 plus 3 S 3 and this is 3 of S 0 plus S 1 plus S 2 plus 2 S 3 right.

So, now, if you just consider 2 into S 0, let us just observe this with respect to parity. So, therefore, right let us consider that S 0 is denoted as the bite a 7 x power of 7 plus a 6 x to the power of 6 plus a 5 x to the power of 5 plus a 4 x to the power of 4 plus a 3 x cube plus a 2 x square plus a 1 x plus a 0 ok.

And, the parity of S 0 is nothing, but a 7 plus so, all these pluses are XORs that we have to operate additions a 7 plus a 6 plus a 5 plus so, on till a 0. So, now, if I calculate 2 into S 0 so, in polynomial notation this is nothing, but x multiplied with a 7 x to the power of 7 plus so, on till a naught.

So, therefore, I will have a 7 x to the power of 8 plus a 6, x to the power of 7 plus a 5, x to the power of 6 plus a 4, x to the power of 5 plus a 3 x to the power of 4 plus a 2 x cube plus a 1 x square plus a 0 x right. So, now, remember that this x power of 8 has to be reduced and our polynomial was x power of 8 plus x power of 4 plus x power of 3 plus x plus 1.

So, what will do is basically we will write will basically substitute x power of 4 plus x cubed plus x plus 1 instead of x power of 8. And that will basically leave us with you know like with this right we will basically start with a 6 x to the power of 7. And, this will basically affect only the fourth degree polynomial the third degree polynomial and this basically right.

So, therefore, I will have a 6 a 6 x to the power of 7 plus a 5 x power of 6 plus a 4 x to the power of 5 plus right. We will have a 3 plus a 7 x to the power of 4 plus a 2 plus a 7 x to the power of 3 plus a 1 x square plus a 0 plus a 7 x right plus a 7, because there is a constant plus a 7.

So, therefore, what is the parity of 2 S 0? The parity of 2 S 0 you can see is a 6 plus a 5 plus a 4 plus a 3 plus a 7 plus a 2 plus a 7 plus a 1 plus a 0 plus a 7 plus a 7. So, we can do a bit of canceling and we can see that for example, a 7 will get cancelled here with a 7 ok, likewise this a 7 will get cancelled with this a 7. So, will have pretty much all the terms a 0 a 1 a 2 a 3 a 4 a 5 a 6 but only not a 7.

So, I can write this as the parity of a 0 plus a 7 right. So, therefore, the parity of this column can be easily written in this way and also note that the parity of 3 S 0 can be easily derived as the parity of S 0 plus the parity of 2 S 0, because parity is linear ok. And therefore, I can write this as parity of S 0 plus parity of S 0 plus a 7 and that is just a 7 right.

So, therefore, right if I want to calculate the parity of this column. So, let me just write here as a big parity. So, then right every element here we basically for example, have like 2 S 0 if I just considered this for example, 2 S 0 this will be parity of S 0.

So, let me write this as you know like parity of S 0 plus since I am doing 2 S 0. So, there will be an S 0 7 ok. So, there is the 7 bit of S 0 plus 3 S 1 will be the 7 bit of S 1 plus parity of S 2 plus parity of S 3 ok.

(Refer Slide Time: 29:47)

And, this side is if you just a little bit of rearrange. So, similarly we can will get similar stuff for the other 3 bytes. So, this will be nothing, but parity of S 0 plus parity of S 2 plus parity of S 3 plus S 0 7 plus S 1 7 ok. So, now, if you just you know like go back. So, remember this equation.

So, similarly we will have for the other possible things here. So, you see you get this right. So, this is the 3 parity bites plus the 7 bits which has been XORd. So, likewise you get for the other three things also 3 bites also. Another very interesting thing is that if I you know like kind of XOR the parity of these 4 bytes.

(Refer Slide Time: 30:41)

For example, if I take p 0. So, I am just leaving out the j for simplicity p 0 plus p 1 plus p 2 plus p 3 ok. Then you observe that this is equal to if I just add up right then this S 0 7 will get canceled up here, S 1 7 will get canceled up here S 2 j will get cancelled up this will get cancelled out. So, pretty much this entire stuff will get canceled up.

Here also you can see that p 0 j and p 0 j and there will be like another p 0 j for example, the two of them will cancelled up and only one of them will stay. So, therefore, right and similarly for all the other three things so, the parity right of this column does not change actually ok. And that is a property of the mixed column I mean and you can verify this very easily without performing any complicated (Refer Time: 31:28) multiplication and so on ok.

And that can leap to an very efficient check using parity. So, this kind of depicted here that you can use this parity prediction logic and then you basically you know like predict the parity output and then you can compare after using this logic. So, now, what is again you know like because of parity I you know that this is not giving you like hundred percent coverage.

(Refer Slide Time: 31:55)

And therefore, right it can lead to you know like kind of reduction with respect to the overall coverage that you can obtain. So, what we will see is the next right is how we will kind of you know like improve this, by using the using the technique of non-linear codes. And, also right we will see about the bass fault injections and also like how to perform what is called as differential fault intensity attack in the next class so.

Thank you.