

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

Lecture – 58
Microarchitectural Attacks: Part 2 Branch Prediction Attacks (Contd.)

So, welcome back to this class on Hardware Security. So, we shall be continuing our discussion on Branch Prediction Attacks.

(Refer Slide Time: 00:21)



In particular we shall be trying to see how the branch predictions or discussions on branch predictions can be used or leverage to actually perform an attack on public key cryptosystems.

(Refer Slide Time: 00:31)

Utilizing Performance Counters for Compromising Public Key Ciphers

- Can we use the approximate model of the predictor for attacking public key encryption?
- We target 1024-bit RSA and 256-bit ECC implementations:
 - Plain square (double) and multiply (add) variants
 - Using Montgomery ladder, which is protected against timing-channels.

swayam

In particular in the last class, we saw that there was a nice correlation that we observe between the system for a branch prediction and our simple approximate model which is the 2 bit counter model. So, the question is whether we can utilize it to attack an actual implementation of cryptographic algorithm.

So, we shall be considering an array seem implementation and elliptic curve cryptographic implementation and we shall be trying to illustrate how an attack works.

(Refer Slide Time: 00:58)

Classical Algorithm for Modular Exponentiation

Inputs(M) are encrypted and decrypted by performing modular exponentiation with modulus N on public or private keys represented as n bit binary string.

Algorithm 1: Binary version of Square and Multiply Exponentiation Algorithm

```
S ← M;
for i from 1 to n - 1 do
  S ← S * S mod N;
  if  $d_i = 1$  then
    S ← S * M mod N;
  end
end
return S;
```

$S \times S \bmod N$
 $S \times M \bmod N$

Conditional execution of instruction and their dependence on secret exponent is exploited by simple power and timing side-channels.

swayam

So, let us consider the classic algorithm for modular exponentiation as our target. So, in this particular algorithm right we are the input m is encrypted or decrypted by performing modular exponentiation with modulus N on public or private keys represented as an n bit string. And this algorithm is essentially is nothing, but an implementation based on the simple square multiply exponentiation algorithm.

The idea is that here we basically perform a you know like. So, that the idea is that we basically perform a squaring; and a squaring is essentially kind of observed as S into S modulo N . So, this is something that we always do whereas, if the secret bit is one; that means, if your secret bit d_i is 1. So, this is your i th secret bit then we actually do an addition and multiplication; that means, we basically take S and we multiply S with M . So, that is your S into M modulo N operation. So, the final objective is to calculate say M power of d where d has been elaborated in the form of a binary exponent.

So, condition execution of instructions and their dependence on secret exponent is exploited as we have already seen by simple power and timing side channels. So, here we will see that how we can perform a branch mis prediction attack on these kind of implementations. So, we will start with this and we will gradually go into slightly more improvements.

(Refer Slide Time: 02:25)

Montgomery Ladder: A Balanced Exponentiation Technique

- Most popular exponentiation primitive for Asymmetric-key cryptographic implementations.

Algorithm 2: Montgomery Ladder Algorithm

```

R0 ← 1;
R1 ← M;
for i from 0 to n - 1 do
  if di = 0 then
    R1 ← (R0 * R1) mod N;
    R0 ← (R0 + R1) mod N;
  else
    R0 ← (R0 * R1) mod N;
    R1 ← (R1 + R1) mod N;
  end
end
return R0;

```

Handwritten annotations:

- $R_0 = 1, R_1 = M, R_i, R_{i+1} = M$
- Invariant:** $d_i = 0 \rightarrow R_0 = R_i \times R_1, R_1 = R_i^2$
- $\begin{pmatrix} R_0 \\ R_1 \end{pmatrix} = \begin{pmatrix} R_0 \\ R_0^2 \end{pmatrix}_{i-1}$
- $d_i = 1, R_0 = R_0 \times R_1, R_1 = R_1^2$
- $\begin{pmatrix} R_0 \\ R_1 \end{pmatrix} = \begin{pmatrix} R_0 \\ R_0^2 \end{pmatrix}_{i-1}$
- $R_0 = M^d \pmod N$

swayam

So, the improvement that we will be considering is essentially a very popular exponentiation and primitive, for asymmetric key cryptographic algorithms which is essentially called as the Montgomery ladder algorithm.

So, the Montgomery ladder algorithm is essentially as you can observe that as supposed to the previous implementation, where there was only one register that is the register S . So, now, there are two registers. So, we have got the register R_0 and R_1 we initialize R_0 to 1 and R_1 to M . So, the idea here is that we basically make R_0 as 1 and R_1 as M and the ratio that is of R_1 and R_0 which is here M essentially remains an invariant through this entire loop.

So, these basically does not change. So, this invariance can be observed here for example, if the d_i is 0 like if the secret i 'th bit is 0, then you basically multiply R_1 in R_1 ; that means, you basically perform. So, the idea is that if d_i is 0 you basically calculate R_1 as R_1 multiplied by R_0 of course, like everything is modulo N . So, I am not writing the modulo N part. And if and also write what you do is you basically do a squaring in R_0 . So, R_0 becomes R_0 square. So, here you can observe that if you calculate R_1 by R_0 . So, that remains is R_1 into R_0 divided by R_0 square so, that remains as R_1 by R_0 , but in the previous iteration.

So, let me write this as a_{i-1} and this is your i th iteration output likewise right if your secret bit is 1 that is d_i is 1, then you perform multiplication in R_0 . So, therefore, R_0 becomes R_0 into R_1 and you perform squaring in R_1 . So, R_1 becomes R_1 square again you observe that R_1 by R_0 in the i th iteration is same as that of R_1 by R_0 in the previous iteration ok. So, therefore, right at the end of the day we basically ensure that through in through this computation, you have you return the value of R_0 and the value of R_1 is indeed equal to you know like M to the power d modulo N which is essentially what we want.

But at every iteration you are performing both squaring and multiplication. So, therefore, this is a more secured algorithm or at this it is more secured against simple power attacks or simple side channels. And it is also timing attack resistance in the sense like it is a it does not have a timing time variance as we have seen in the previous case, because you are performing a multiplication and squaring in both the cases ok. So, it is essentially

having a balanced is a more of a balanced is a more balanced exponential technique compared to the naive square multiply algorithm.

(Refer Slide Time: 05:13)

Montgomery Multiplication

Efficient way of performing multiplication mod N .
 Avoids time consuming integer division operation.
 Takes a R which is assumed to be 2^k , when N is k -bit number.
 It is based on Montgomery reduction, which given $0 \leq T < NR$, returns TR^{-1}

$R > N$
 $R = 2^k$

$m = T(-N^{-1}) \bmod R$
 $t = (T + mN) / R$
 if $(t \geq N)$
 $t = t - N$

Claim: $t = TR^{-1} \bmod N$ ✓
 Proof:
 $tR = (T + mN) \bmod N = T$
 Also note, $(T + mN) = T + T(-N^{-1})N = 0 \pmod R$
 $\therefore R \mid (T + mN) \Rightarrow t$ is an integer.
 $0 \leq t < R \Rightarrow 0 \leq mN < RN \Rightarrow 0 \leq T + mN < 2RN$
 $\therefore 0 \leq t \leq 2N$

So, now we shall be seeing right essentially of very popular way of calculating these multiplications and as you can see that multiplications are inherent to performing the public key algorithms. So, people have developed several architectures for multiplication. One of the very popular multiplication routines is essentially due to what is called as the Montgomery multiplication. So, here we illustrate it is an efficient way of performing multiplication modulo N . So, therefore, there are 2 arguments a and b and I want to calculate a into $b \bmod N$ where N is essentially a k bit number.

So, the idea is that N is essentially a k bit number and k as we know is typically large. So, it could be for example, 1024 bit or 2048 bit or even larger. So, it is based on. So, now, this multiplication is based on what is called as Montgomery reduction. So, Montgomery reduction basically chooses a large value of R , but then R essentially is something like 2 power of k and as you can see that N is a k bit number. So, you can expect that R is larger than N ok.

So, R is kind of larger than N ok. So, R is essentially also observe the form of R which is essentially 2 power of k and that makes computation for you know using R quite easy because it is a power of 2. So, in Montgomery reduction we basically calculate or return the value of TR inverse ok. So, T is my input and I would like to calculate TR inverse

when T is restricted between say NR between 0 to NR how do we do that is shown here by this algorithm. So, we basically calculate m which is equal to T into minus N to the power of minus 1 modulo R . So, therefore, we basically calculate N to the power of minus 1 and then we perform this computation and then we basically subtract this value is greater than equal to N , then we subtract N and this is my result.

And this result is indeed equal to or the claim is that the result which is returned here is equal to TR to the power of minus 1 modulo N . So, let us try to understand why it works and the logic is quite simple. So, if we basically cross multiply. So, if we multiply T and R , then you should get the result right is essentially T into R which is equal to. So, you can observe here that T is nothing, but T plus mN by R . So, if I multiply small t into R then that is equal to T plus mN . So, if I now take modulo N then this basically goes to 0 therefore, I have only T and that basically establishes this fact.

Also note because it is important to observe note that, that why is this an integer because that I have say that I have done thus this division right n T plus m N divided by R . So, we observed that T plus mN is equal to T plus T into minus N to the power of minus 1 N because m is essentially equal to T into minus N to the power of minus 1 modulo R .

So, therefore, right if I do a modulo R operation then this part essentially is 1 . And therefore, or minus 1 and therefore, I get T minus T which is equal to 0 which means R divides T plus m N and therefore, right T plus m N divided by R is an integer ok. And also observed that I have done here one subtraction with N why 1 subtraction with N suffices. So, the idea is that you observe the m is essentially between 0 to R this you can easily observe because you are doing a modulo R operation. So, m is between 0 to R and therefore, if I multiply with n then I get 0 is less than equal to mN is less than RN . So, this number is less than RN and also observed T is less than RN and therefore, right when I add T plus m N then this value is less than 2 RN ; that means, right T plus mN divided by R is less than $2N$.

So, therefore, this number does not exceed N and therefore, right it does not exceed $2N$. So, therefore, in order to do or bring the result modulo N because finally, you have to bring the result modulo N . We have to do at most one subtraction with N and therefore, right is a very common way of doing the Montgomery reduction and many libraries actually does this extra reduction operation and this is exactly what we will be targeting

in our attack. So, this if statement is, what we will be targeting in our attack description. So, just to complete this part, how we can you know like use this to perform multiplication can be just observed by in this slide.

(Refer Slide Time: 09:23)

So, the idea is that what we do is as shown here that suppose I want to multiply A and B by using the Montgomery algorithm. So, what we do here is shown over here. So, the idea is that I have got a simple description here. So, what we do is we pick an R which is say equal to 2 to the power of k and the idea is that 2 to the power of k and which is R and N the gcd is 1 so; that means, like as we have already seen N to the power of minus 1 modulo R exists and we have computed that.

So, what we do is we compute a dash which is equal to a into R and we compute b dash which is equal to b into R and this is called as Montgomery Reduction; that means, with Montgomery transformation; that means, we transform a to the Montgomery domain and call it say a dash or maybe A and we transfer b into the Montgomery domain and that is called as B. So, idea is a is nothing, but a into R modulo n and B is again b into R modulo n ok. So, so you can observed that as we have in the previous case we have seen that suppose I you know like I have got a simple routine of calculating. So, if I give you an input for example, as we have seen in the previous case, where the input is for example, what we get is T into R to the power of minus 1.

So, therefore, what I can do over here is if I call that routine and I pass a and R square.

(Refer Slide Time: 10:51)

The slide is titled "Overview of the attack" and contains the following text boxes:

- Grey box:** This work shows that HPCs, which are used as **performance monitors (watchmen)** in modern computer systems can be utilized to retrieve the secret keys by reasonably modelled adversaries.
- Orange box:** The attack exploits the characteristics of branch predictor and shows that the leakage of the key increases with the ability of the attacker to model the predictor more accurately.
- Green box:** We claim that branch misses from HPCs are indeed more significant side-channels compared to timing.
- White box:** Sarani Bhattacharya, Debdeep Mukhopadhyay:
Who Watches the Watchmen?: Utilizing Performance Monitors for Compromising Keys of RSA on Intel Platforms. CHES 2015: 248-266.

At the bottom of the slide, there is a video feed of a man in a light blue shirt, and logos for "swayam" and "INDIA WISE, LEAD WISE" are visible.

So, if I pass like a and R square as 1 input, then I would get the result as a into R square. So, basically right this is my result. So, basically I would get a into R square to the power of R to the power of minus 1 and that is equal to a into R. So, then we basically also we perform the Montgomery multiplication with b and R square and I get the result which is b into R square with R to the power of minus 1 and that is b into R ok.

So, then we perform a Montgomery multiplication and the Montgomery multiplication is essentially shown over here as you know like perform with A and B. So, the idea is that a is your a into R. So, we basically perform or multiply a into R with b into R and with R to the power of minus 1. So, that becomes your a b R and then we basically again call this routine a b R with 1. So, you can see that this is Montgomery multiplication is with 1. So, we basically multiply abR with 1 and with R inverse and that is equal to a into b ok.

So, we basically need a way of calculating you know like of calculating essentially a into b into R to the power of minus 1 and with that routine we essentially can calculate a into b as shown over in this part. You also observe that you know like once we have the inputs in Montgomery domain. So, if you perform the Montgomery reduction, the Montgomery reduction which a and if you just take this a dash and if you take this b dash then; if you are given this a dash into b dash, then the Montgomery reduction gives you the value of a dash into b dash R to the power of minus 1 because the moment you are as

we have seen in the previous case that if you are given T right basically get T into R to the power of minus 1 modulo n ok. So, likewise right you can perform a sequence of Montgomery reductions and you can obtain the corresponding results here and that pretty much gives you the corresponding Montgomery multiplication just by doing Montgomery reductions.

And therefore, right why it is very efficient because the moment you have got R as in the form of 2 to the power of k , the assumption is that multiplication with R division with R and modulo with R a trivial operations they are much more easier and doing modulo N operations so. So, therefore, right I mean this is essentially the working of the Montgomery operation. So, now, the question is right if you have got a kind of library, where essentially which essentially uses Montgomery multiplication there is a high probability or at least you should be careful if there is a routine as shown here where the Montgomery multiplication is using a Montgomery reduction where there is this kind of conditional statement ok.

And that is exactly what we will be targeting in our attack description. So, the overview of the attack is as follows; the work shows that the HPCs which are used as performance monitors or watchman in modern computer systems can be utilized to retrieve the secret keys by reasonably modelled adversaries. The attack exploits the characteristics of branch predictor and shows that the leakage of the key increases with the ability of the attacker to model the predictor more accurately.

For example, we have made an approximation that, there is a nice correlation with a 2 bit predictor. If I have a better model, then I can use that better model but we will show in most cases right the simple model has a 2 bit predictor. So, which will be sufficient we claim that the branch miss from the HPCs are in need more significant and they are more significant side channels compared to timing. More details of this work can be observed in this work there is shown here in this paper.

(Refer Slide Time: 14:37)

The slide is titled "Why should we consider HPCs for security analysis?". It contains the following text:

Results from HPCs are treated as an accurate representations of events occurring in hardware.

Perf subsystem has "Performance counters". We assume that the entire decryption operation is being monitored by the spy with user-level privilege.

`$perf stat -e branch_misses ./executable_name`

1. An unprivileged user residing on the same system as the privileged process can gain access to sensitive information of the privileged.
2. Increase or decrease of branch misses of the privileged process can be monitored by the spy with user-level privilege.
3. The present attack exploits the ability to measure the increase or decrease of branch misses, rather than actual values!

Process 1: User running a multiplication code and observing perf stat concurrently.

Process 2: User running a decryption code with heavy branches.

The slide also features a green monster icon, a laptop, a person at a computer, and a line graph showing a fluctuating upward trend. At the bottom, there is a Swamyam logo and a small video inset of a man speaking.

So, how we should rather why should we consider HPCs for security analysis? So, the results from HPCs that treated as an accurate representation of events occurring in hardware, performance of subsystem has already been allowed in the Linux kernel 2.6.31 onwards as performance counters for Linux.

So, we at this point assume that the adversary can observe the total number of branch misses in an entire decryption operation, by using as command like `perf stat minus e branch misses` now observe that this is the event that you would like to observe and you basically give any executable at this point ok. And so, now, there is system in the system we are considering that both the user who is my target. So, this is my victim who is essentially running a decryption code with heavy branches and also right not only every branches.

But branches as we have seen you know that which depends on the secret, and there is a spy or an attacker we which basically run say multiplication code and observe the `perf stat` concurrently. So, therefore, this executable could be the executable of the multiplication code for example. So, then the idea is that if this person like is executing a decryption code and there are branch misses as you can observe in this statistic in this in this graph, then this essentially is also observable to the attacker or to the spy.

(Refer Slide Time: 15:54)

Branches in the Target: Modelling using HPCs

- ▶ We monitor the branch misses on the square and multiply and Montgomery Ladder algorithm using Montgomery multiplication as subroutine for operations like squaring and multiplication.
- ▶ Branch miss rely on the ability of branch predictor to correctly predict future branches to be taken.
- ▶ Profiling of HPCs using performance monitoring tools provides simple user interface to different hardware event counts and are considered as side-channel.

The slide also features logos for IIT Bombay, Swayam, and another organization at the bottom, and a video inset of a man speaking in the bottom right corner.

So, therefore right I mean the idea here is that an unprivileged user residing on the same system, who has basically the capability of monitoring these events can gain access to sensitive information of the privilege that is what we will illustrate here and show that how we can get the key, which essentially the processor or the user is or the victim is utilizing. So, that the main point is that the increase or decrease of branch misses of the privileged process can be monitored by the spy, and the present attack exploits the ability to measure the increase or decrease a branch misses rather than actual value.

So, note that when we if you observe the plot that we showed in context 2 correlation, we could not you know we are not claiming or it is not that one thing we should be kept in mind is that we are not predicting you know like the actual branch misses. But what we are predicting is the relative branch misses like whether the branch misses are increasing decreasing or whether the remaining constant.

So, what is more important is the monotonicity of the graph rather than you know like whether the actual value of the branch misses. So, what we do over here is that in order to explain the attack? We basically monitor the branch misses on the square and multiply and the Montgomery ladder algorithm using Montgomery multiplication as a subroutine of for operations like squaring and multiplication and the branch miss relies on the ability of the branch predictor to correctly predict the future branch misses which should be taken.

And the profiling of the HPCs using performance monitoring tools provides a very simple user interface to different hardware events and what we are observing here is the branch misses.

(Refer Slide Time: 17:25)

The Relation of Branches and the Secret Key

Let n -bit secret scalar in ECC be denoted as $(k_0, k_1, \dots, k_i, \dots, k_{n-1})$. Trace of taken or not-taken branches as conditioned on scalar bits and expressed as $(b_0, b_1, \dots, b_{n-1})$.

- ▶ If a particular key bit k_j is 1 then the conditional addition statement in the double and add algorithm gets executed. Thus, the condition is checked first, and if the particular key bit is set then its immediate next statement i.e., addition gets performed. Since this is a normal flow of execution the branch is considered as not-taken i.e., $b_j = 0$ in this case.
- ▶ While when $k_j = 0$, the addition operation is skipped and the execution continues with the next squaring statement. Thus, in this case branch is taken i.e., $b_j = 1$.

Handwritten notes in red:

- if $(k_i = 1)$
- $b_j = 0$ Execute Addit. ↓
- or, Execute Addition ↓
- $b_j = 1$

So, in the attack we will basically considering say an n bit secret scalar. So, it could be in the either elliptic curve crypto or in the even the R S exponential as we have seen. Suppose the secret is shown here as $k_0 k_1$ and so, on till k_{n-1} . So, it is an n bit secret that we target.

So, the trace of the taken or the not taken branches as conditioned on the scalar bits are expressed as say $b_0 b_1$ so, on till b_{n-1} . So, these are basically nothing, but sequence of taken not taken and so, on. So, the idea is that if the key bit is 1; that means, if the say the k_j is 1, then the conditional additional statement in the double and add algorithm gets executed ok. So, therefore, if I get a 1, then I basically execute it and therefore, right it implies that when k_j is 1 the branch is 0 because I mean the branch the branch outcome is 0, because you are because it is essentially resulting is a not taken branch because you are not taking that branch ok. So, the what I am trying to say is that if you for example, right; like if say the k_i is or k_j is 1, then you are saying suppose that you essentially execute for example, square and multiply say you execute a multiplication step or what you in elliptic curve cryptographic right you have basically

execute. So, this is an odd you execute say double I mean the an addition operation ok. So, you have perform an addition.

So, execute multiplication or you execute an addition. So, the idea is that if the k j k j is 1 since you are going to the next instruction, that implies that there is the branch is not taken and therefore, right we would say that the b j is equal to 0. On the other hand if k j is equal to 0, then you basically jump to the next location and therefore, actually the branch takes place and so, you write b j is equal to 1 ok.

(Refer Slide Time: 19:23)




Effect of Compiler Optimization on Branching

► We validate our understanding for conditional branching and observe the effect of optimization options in gcc:

- 1 `.LC3: .string hello`
- 2 `.LC4: .string hi`

without Optimization	O1	O2	O3
<pre> .L3: movl -56(%rbp), %eax cmpl \$19, %eax movl -32(%rbp,%eax), %eax cmpl \$19, %eax jnc .L5 movl \$LC3, %edi call puts jmp .L4 .L5: movl \$LC4, %edi call puts </pre>	<pre> .L5: cmpl \$19, (%rbp,%eax) jnc .L3 movl \$LC3, %edi call puts jmp .L4 .L2: movl \$LC4, %edi call puts </pre>	<pre> .L5: movl \$LC4, %edi call puts .L5: call puts .L5: ... jnc .L3 movl \$LC3, %edi call puts ... </pre>	<pre> .L2: movl \$LC4, %edi call puts .L5: ... jnc .L3 movl \$LC3, %edi call puts q ... </pre>

Figure: Assembly generated using various optimization options in gcc

So, now, with this background you can observe right I mean and its also important observe that if you take this kind of codes and run it to several compiler optimizations like O1, O2 and O3 then you will still find that these kind of instructions will remain in your final target code.

And therefore, right the idea is that the assembly generated even using various optimization levels would probably retain these conditional statements that you are writing in your high level program.

(Refer Slide Time: 19:51)

Threat Model of the Attack

- ▶ The attacker knows first i bits of the private key and wants to determine next unknown bit d_i of the key $(d_0, d_1, \dots, d_i, \dots, d_{n-1})$.
- ▶ Generate a trace of branches as $(t_{m,1}, t_{m,2}, \dots, t_{m,i})$ for conditional reduction of Montgomery multiplication at every squaring step.
- ▶ Under the assumption of d_i having value j , where $j \in \{0, 1\}$, appropriate value of $t_{m,i+1}^j$ is simulated.

The adversary can only simulate the branches using its model for any partially known bit.

The adversary does not have granular control on the HPC values of any sub-simulation. This is a practical assumption, as the adversary may not have such fine grained control on the HPCs, which may require high privileges, kernel patch etc.

swayam

And that means right that understanding and evaluating them with respect to security is of paramount importance. So, what we try to do in the attack can be shown here is as in the threat models. So, again this is an iterative attack algorithm. So, we assume that we know say you know like d_0 to d_i , and what we would like to know is or we know the first i bits of the private key and we want to determine the next unknown bit which is d_{i+1} of the key.

So, we generate a trace of branches. So, what we do is that, we basically target the underlying Montgomery multiplication of may be the, you know like of may be the of the squaring step because remember the squaring takes place always so. So, therefore, right we basically target the Montgomery reduction if statement as I said which was the target, and we basically kind of target it and basically stimulate the trace; that means, you know like in several the whenever you like that the d_0 is if a d_0 means the first key bit is coming into play, then the next key bit which is d_1 which is coming to play and so, on at every step like $d_0 d_1 d_2 d_3$ and so, on.

The corresponding if statement in the Montgomery multiplication or the Montgomery reduction is essentially generating a sequence of branches; like taken not taken and so on ok. So, this essentially or this history is essentially denoted as the trace of branches and is essentially is what we basically denote by this tuple $t_{m,1} t_{m,2}$ and so on till $t_{m,i}$ ok.

So, therefore, right for if so, this is the this is for the conditional reduction of the Montgomery multiplication at every squaring operation.

So, then we basically make an guess of the i ,th bit. So, remember that the i ,th bit can have the value either 0 or 1 and based upon that we basically make a prediction. We basically kind of simulate an appropriate value of $t_{m,i+1}$; that means, the next corresponding you know like branch taken or branch not taken for the conditional reduction of the Montgomery multiplication of again of the squaring step. So, the adversary remember can only simulate the branches using its model for any partially known bit. So, this is partial simulation can only be performed by using a simple model essentially which is the 2 bit predictor model that I described.

And the adversary does not have any granular control on the HPC values of any sub simulation. So, we basically for the actual system like for the actual target we basically can get a you know like a hard count for the branch misses; that means, when the exactly when the entire description happened I get a total count of the decryption I get the total count of the number of branch misses. But I do not have a very fine grained control; that means, I do not have a control like what happens in every bit of the input for example. So, therefore, right, but we can use our model to do that partial simulation ok.

So, now the question is right how we how can we employ this to perform and complete an n to n attack.

(Refer Slide Time: 22:49)

Working of the Attack: Offline Phase

Figure: Partitioning randomly generated Ciphertexts set based on simulated Branch miss Modelling

$t_{m,i}$ is simulated as either taken or not-taken branch depending on whether the conditional reduction branch statement at the i th squaring operation is being executed.

So, therefore, right what we do can be illustrated here in this diagram. So, this is the offline phase. So, we take or we collect several inputs. So, these are my plain text and we basically divide my n plain text by using an offline phase or in the offline phase. So, what I do is here shown here is that I basically guess this i 'th bit. So, this i 'th bit can be either 0 or can be either 1 and I know so, this is already known. So, this is my prior knowledge. So, I already know say the previous key bits.

I may a guess here. So, this guess can be 0 or this guess can be 1 and note that I also know the trace of these taken not taken, because since I know this $d_0 d_1$ till d_{i-1} , I can calculate right this m to the power of d_0 till say d_{i-1} and I observe the corresponding squaring operation of the Montgomery reduction ok. I mean I observe the corresponding I mean I observe the squaring of the you know like I basically observe the squaring of the you know like the square and multiply algorithm for example, and in particular what we observe is the Montgomery reduction ok.

So, because the squaring right will also use or call the Montgomery reduction underneath and we basically observe the Montgomery reduction and note that in the Montgomery reduction we had this line right which is something like $t_i > N$ or $t_i \geq N$ I basically do a $t \text{ equal to } t \text{ minus } N$ operation. So, this is the 1 this is the operation which I target. And in particular I target the outcome of this branch whether the branch is taken or whether the branch is not taken.

But in order to understand that I need to simulate this value of t and this essentially I can get what is the function of my previous $i-1$ bits, and also on the value of the message based upon that I can simulate this possible you know like sequences. So, now, right I mean what I do is, I basically take my simple to be predictor and I feed my inputs t_1 to t_i for example, and that gives me you know like a guess of the next out come. So, if the next outcome and also right based upon the value of $d_i \text{ equal to } 0$ or $d_i \text{ equal to } 1$, I can extend this stimulation to know what is the actual value of the next outcome of this of this branch.

If the idea is that if my 2 bit predictor correctly gives me the next sequence; that means, if there is a match when I am giving t_1 to t_i and if indeed matches with your $t_i + 1$. So, $t_i + 1$ means you know like essentially is this outcome for example, then I put m to the basket M_1 . So, again I made 10, 2 baskets here call it as M_1 and call it as M_2 the

idea is that if I get the this is these 2 are indeed equal then I put this message m into the bucket M_1 or I put the put m into the bucket M_2 ok.

On the other hand if this is equal to $T_i + 10$. So, that is essentially the other possibility; that means, if d_i equal to 0, then write the outcome here; that means, whether these outcome was 0 I mean whether the this branch was taken or not or not taken is captured by say $t_i + 10$ ok. So, if my model; that means, if the model T essentially correctly essentially matches with $t_i + 10$ ok, then I basically take again or define 2 more buckets call it as M_3 and call it as M_4 and I put the message.

So, if it matches then I put it into m or if it does not match then I put this m into the bucket M_4 ok. So, therefore, right I basically kind of partition my input m into these buckets M_1 M_2 M_3 and M_4 and I basically. So, you see that there are 2 alternatives this m can go into either M_1 or it can go into M_2 or this message m can also go into M_3 and M_4 . So, there can be potentially a message m which goes into say M_1 or it also goes to say may be M_3 that is also possible.

So, we basically kind of in order to kind of improve the accuracy of the attack, what we do is basically we basically do a further step and we basically ensure that there is no common cipher texts in the sets M_1 M_3 and M_2 M_4 ok.

(Refer Slide Time: 27:11)

Separation of Random Inputs in the Offline Phase

- 1 $M_1 = \{m|m \text{ does not cause a miss during MM of } (i+1)^{th} \text{ squaring if } d_i = 1\}$
- 2 $M_2 = \{m|m \text{ causes a misprediction during MM of } (i+1)^{th} \text{ squaring if } d_i = 1\}$
- 3 $M_3 = \{m|m \text{ does not cause a miss during MM of } (i+1)^{th} \text{ squaring if } d_i = 0\}$
- 4 $M_4 = \{m|m \text{ causes a misprediction during MM of } (i+1)^{th} \text{ squaring if } d_i = 0\}$

We ensure that there must be no common ciphertexts in sets (M_1, M_3) and (M_2, M_4) and the sets should be disjoint.

So, note that it may happen that a sequence matches here and therefore, right it goes into M_1 ok. So, now, it may also happen that the sequence. So, the idea is that if it goes to M_3 , then it is kind of contradiction right because essentially in one case right I am predicting that it matches with t_{i+1} and in the other case right it is as I am saying that it matches with $t_{i+1} - 1$.

So, in order to remove the contradiction I ensure that there is no common ciphertext which goes into M_1 M_3 and M_2 M_4 and the sets are ideally disjoint and note that what I do is. So, therefore, right I have got now four buckets like M_1 M_2 M_3 and M_4 the idea is there M_1 carries those messages which does not cause a mis or mis prediction during the Montgomery multiplication of the $i+1$ th squaring if d_i is equal to 1 and M_2 are those messages which causes a mis prediction during this particular operation ok.

As you can observe that if it goes into M_1 then; that means, that there is no mis prediction and therefore, right it does not cause any mis prediction on if d_i is equal to 1 ok. On the other hand if it is not equal to this, then it will causes the mis prediction and since right your comparing with t_{i+1} ; that means, you are basically doing it for the case or the assumption that d_i is equal to 1 ok. Likewise right M_3 essentially contains those messages which does not cause a miss during the Montgomery multiplication of the $i+1$ th squaring if d_i is equal to 0 and it is an M_4 consists of those messages which consists of the mis prediction during the Montgomery multiplication of the $i+1$ th squaring if d_i is equal to 0.

(Refer Slide Time: 29:03)

Online Phase

The probable next bit is decided by the following:

- ▶ If $(avg(\mathcal{M}_{M_2}) > avg(\mathcal{M}_{M_1}))$ and $(avg(\mathcal{M}_{M_4}) < avg(\mathcal{M}_{M_3}))$, then the next bit $(nb_i) = 1$
- ▶ Otherwise, if $(avg(\mathcal{M}_{M_4}) > avg(\mathcal{M}_{M_3}))$ and $(avg(\mathcal{M}_{M_2}) < avg(\mathcal{M}_{M_1}))$ then, next bit $(nb_i) = 0$

During the online phase, branch misses from the HPCs are monitored for the execution of the cipher over the entire key.

So, therefore now, the attack is very straight forward. So, basically kind of have a online phase where the probable next bit is decided by the following observations. So, basically kind of measure the average branch misses. So, this is observed by the actual hardware performance counter events and the idea is that if the average of the bucket of M 2 that means, of all the messages which has gone into the M 2 bucket is more than you know like the average of those branch misses which has gone into the M 1 bucket, then you find that what you are basically saying is that the average of M 2 is more than the average of M 1 ok.

So, you see that M 2 actually causes a mis prediction, and M 1 it cause it does not cause a mis prediction so; that means, right this is kind of incoherence with what we expect and therefore, right it is most likely more likely that d_i is indeed equal to 1 ok. On the contrary if you also if I would like to confirm your test you can also compared with M 3 and M 4 and you see the here the average of M 4 is less than the average of M 3 ok. So, again if you come to M 3 and M 4 you see that here the average is of M 4 essentially this is a mis prediction and this is the no mis prediction.

So, if you get the you know like the average of M 3 more than the average of M 4, then for example, what you observe here then; that means, that it is not incoherence with what should happen if d_i was 0 and therefore, it confirms that d_i is not 0 ok.

So, therefore, this test confirm that d_i is equal to 1 and this test confirms the d_i is not 0 and therefore, it kind of improves my confidence in the fact that the next bit is probably

1 and likewise for the you know that the opposite will happen when the next bit when I would predict the next bit to be 0.

(Refer Slide Time: 30:43)

Offline Phase

Input: $(d_0, d_1, \dots, d_{j-1}), M$
Output: Probable next bit nb_j

```

begin
  Offline Phase;
  for  $\forall m \in M$  do
    Generate taken/ not-taken trace for input  $m$  as  $t_{m,1}, t_{m,2}, \dots, t_{m,j}$ ;
    Assume  $d_j = 0$  and 1, generate  $t_{m,i+1}^0, t_{m,i+1}^1$  respectively;
     $p_{m,i+1} = T(t_{m,1}, t_{m,2}, \dots, t_{m,i})$ ;
    if  $p_{m,i+1} = t_{m,i+1}^1$  then
      Add  $m$  to  $M_1$ ;
    end
  else
    Add  $m$  to  $M_2$ ;
  end
  if  $p_{m,i+1} = t_{m,i+1}^0$  then
    Add  $m$  to  $M_3$ ;
  end
  else
    Add  $m$  to  $M_4$ ;
  end
end
Remove Duplicate Ciphertexts in the sets  $M_1, M_3$  and  $M_2, M_4$ ;

```

So, with this basically attack in attack in attack setup right we basically have got this online phase or the offline phase where basically take all the messages and I use my simulated model to basically partition, the inputs into the buckets M_1, M_2, M_3 and M_4 and there is an online phase where I basically do this comparison ok.

(Refer Slide Time: 30:59)

Online Phase

```

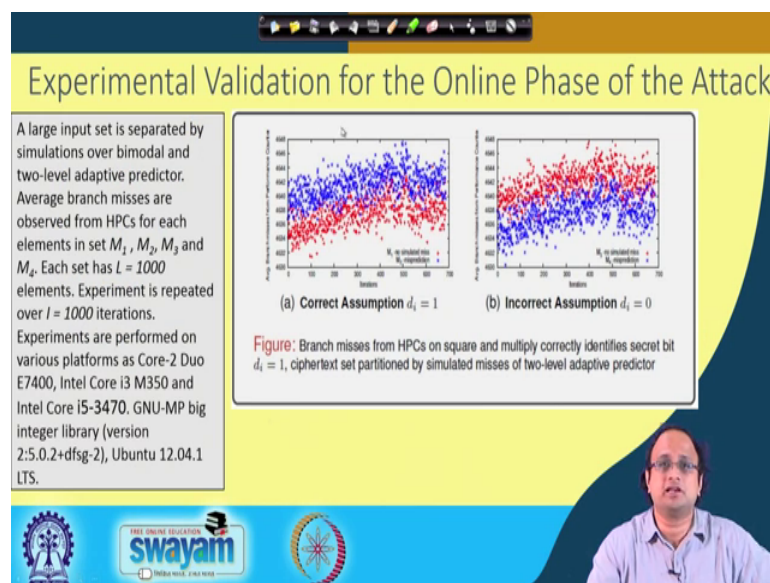
Online Phase;
Observe distribution of branch misses from performance counters as  $\mathcal{M}_{M_1}, \mathcal{M}_{M_2}, \mathcal{M}_{M_3}, \mathcal{M}_{M_4}$ ;
if  $(avg(\mathcal{M}_{M_2}) > avg(\mathcal{M}_{M_1}))$  and  $(avg(\mathcal{M}_{M_4}) < avg(\mathcal{M}_{M_3}))$  then
   $nb_j = 1$ ;
end
if  $(avg(\mathcal{M}_{M_4}) > avg(\mathcal{M}_{M_3}))$  and  $(avg(\mathcal{M}_{M_2}) < avg(\mathcal{M}_{M_1}))$  then
   $nb_j = 0$ ;
end
return  $nb_j$ ;

```

Remember that when you are doing online phase you just have a bunch of hardware performance counters for the corresponding messages. So, you basically kind of decrypt using M 1 decrypt using M 2 decrypt using M 3 and so on and you basically get a bunch of hardware performance counter events ok.

So, then you basically calculate this averages and then you make a decision about the next bit ok. So, you basically kind of do this attack in an iterative fashion. Here is an experimental validation for the attack.

(Refer Slide Time: 31:25)



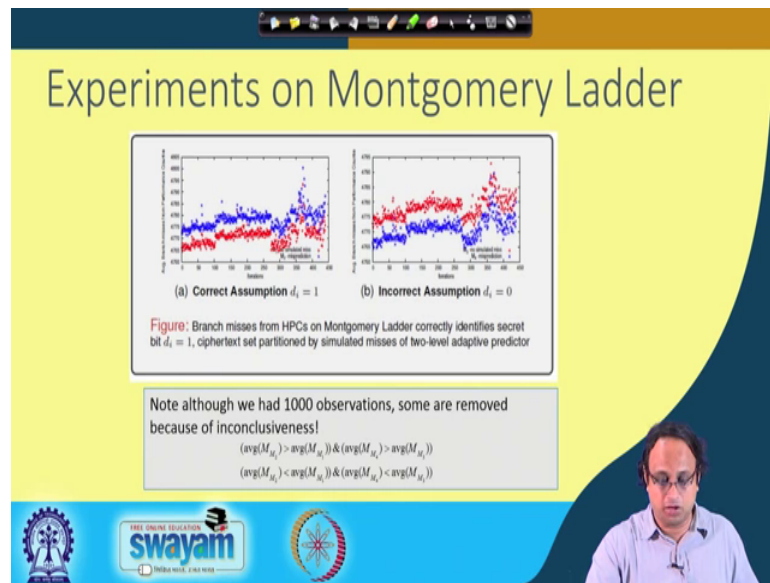
So, you see that we show here in particular an online phase of the attack where the target is a square and multiply algorithm, which has been performed with thousand iterations. So, in particular this has been done on a on an Intel Core 2 duo a platform and you can observe here that the first one shows that the correct assumption is d_i is equal to 1. So, it kind of shows that the average time for example, for the shown in the blue lines is more than that for the red ones and the blue ones correspond or the blue dots correspond to the fact where for M_2 and the we what we see is that the average for M_2 is more than the average for M_1 .

And if you go to the algorithm right you see that the average for M_2 being more than average for M_1 , the algorithm predicts that it is equal to 1 ok. The next bit is 1 which is indeed the correct assumption likewise if you partition with M_3 and M_4 , we observe that the average for M_4 is less than the average for M_3 . So, the average for M_3 is more

and here also you observe that if the average for M 3 is more then you basically predict the next bit to be 1 and therefore, indeed right the incorrect assumption would be d i equal to 0 and this also confirms that the next bit is 1 ok. So, therefore, this kind of accurately estimates from specific i'th bit.

So, likewise we can perform this attack for other corresponding bits and retrieve the attack 1 by 1.

(Refer Slide Time: 32:44)



So, we can also perform the attack. So, here result on how the attack works on the Montgomery ladder. Again you know like the similar kind of experimentation that we do and you can observe that here the separation also can be observed and also kind of correctly estimates the fact that d i is equal to 1.

(Refer Slide Time: 33:00)

Effect of Inconclusiveness and a Wrong Bit on Subsequent Bits

Bit Position (i)	Keystream	Inconclusiveness	i^{th} Bit Decision on the Incorrect $(i-1)^{\text{th}}$ Bit	Inconclusiveness (%)
36	...011100010...	22.7%	Incorrect	48.8%
142	...010000001...	29.86%	Incorrect	66%
180	...100010000...	43.42%	Incorrect	52.4%

Effect of Wrong Bit Guess on an Inconclusive Decision for Each Bit

The attack being an iterative attack, is important to have an inbuilt error correction. In this case, it is reflected by the inconclusiveness, which increases in case of a wrong guess in the previous iteration.

swamyam
FREE ONLINE EDUCATION
INDIA WISE, TIME WISE

So, this attack right can also help one thing we should be kept in mind that this is an iterative attack. So, if you do a mistake in one of the bits then the attack in accuracy would percolate. One way of understanding that we have made a mistake is that once you make a mistake and if you would like to kind of discover the next subsequent bits, then you will find that the inconclusiveness of the confidence is pretty low. For example, like in the previous cases if you are made the correctly guess the guess the previous bits correctly, then you would probably have something like a 90 versus ten ratio. Whereas, here probably you have something like 60 versus 40 or something like which is essentially kind of inconclusive.

So, this probably will kind of there is an alert, that there is a mistake that you have done in the previous iteration. So, you can there is an inherent amount of you know like error correction involved in this direction mechanism.

(Refer Slide Time: 33:48)

The slide is titled "56 ECC (secp256r1)". It features two scatter plots side-by-side. Plot (a) is labeled "(a) Correct Assumption $d_i = 0$ " and plot (b) is labeled "(b) Incorrect Assumption $d_i = 1$ ". Both plots show "Avg. Branch Misses Per Cycle" on the y-axis (ranging from 0 to 1400) and "Iterations" on the x-axis (ranging from 0 to 600). The legend for both plots indicates that blue dots represent " V_i , vs. smaller register" and red dots represent " W_i , register". Plot (a) shows a clear separation between the two data series, while plot (b) shows significant overlap. Below the plots, a text box explains: "Branch misses from HPCs on the ECC double and add scalar multiplication implementation correctly identify secret bit $d_i = 0$ on an Intel Core i5-3470 since $atcy(M_{M_0}) > atcy(M_{M_1})$ and $atcy(M_{M_0}) < atcy(M_{M_1})$, where ciphertext set is partitioned by simulated misses from a bimodal predictor." To the right of the plots is a video feed of a man speaking. Below the plots is a list of operations under the heading "2P":
 $T_1 \leftarrow Z_1^2$
 $T_2 \leftarrow X_1 + T_1$
 $T_1 \leftarrow X_1 + T_1$
 $T_2 \leftarrow T_2 + T_1$
 $T_3 \leftarrow 3T_2$
 $Y_5 \leftarrow 2Y_1$
 $Z_1 \leftarrow Y_5 + Z_1$
A text box next to the operations states: "Compared to square & multiply, here there are many multiplications. We target one of them in the 8 multiplications in the double operation." The slide also includes a logo for "swayam" and "INDIA WISE, LEAD WISE" at the bottom left.

So, here is an example of how the, you know like the attack works on a set to 56 curve. Again this is an elliptic curve crypto system and in particular right I am not going into the details, but what you can do here is that is attack is based on say double and add algorithms.

So, what you can do is that you can target the double operation like in the previous case I was targeting the squaring operation. So, here I will be targeting the double operation; and in the double operation right there again subsequent several operations and you can choose any one of them in particular right since there are more multiplications and all of them are doing the Montgomery reduction operation. So, you basically expect that the separation would be probably even more clear and that is again observed here by this observation.

(Refer Slide Time: 34:31)

Targeting NIST P-256 ECC (*secp256r1*)

(a) Correct Assumption $d_i = 0$

(b) Incorrect Assumption $d_i = 1$

Branch misses from HPCs on the ECC double and add scalar multiplication implementation correctly identify secret bit $d_i = 0$ on an Intel Core i5-3470 since $(\text{avg}(M_{d_i}) > \text{avg}(M_{1-d_i}))$ and $(\text{avg}(M_{d_i}) < \text{avg}(M_{1-d_i}))$, where ciphertext set is partitioned by simulated misses from a bimodal predictor.

```

2P
T1 ← Z1^2
T2 ← X1 - T1
T1 ← X1 + T1
T2 ← T2 · T1
T2 ← 3T2
Y1 ← 2T1
Z1 ← Y1 · Z1

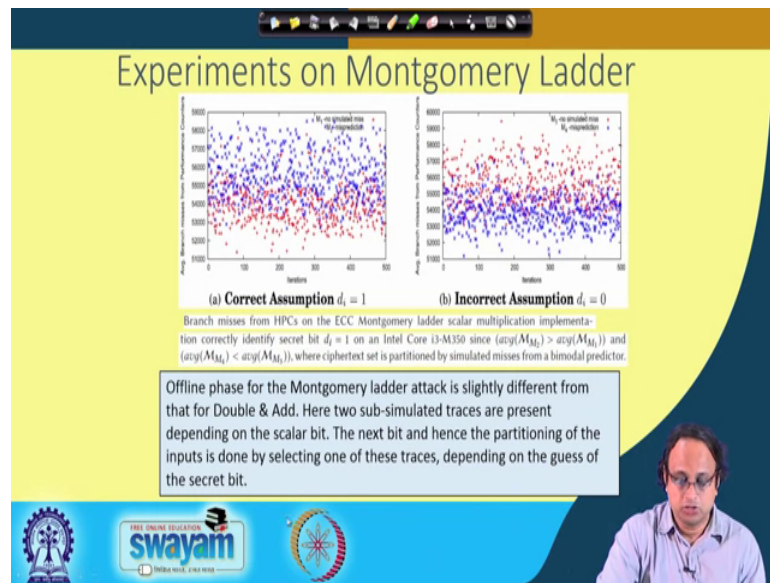
```

Compared to square & multiply, here there are many multiplications. We target one of them in the 8 multiplications in the double operation.

Here is an illustration of the attack performing or targeting an actual elliptic curve implementation of the NIST curves. So, so this is the NIST 256 or NIST P 2 256 in particular the curve is called as sec P 256 R 1. So, here you can again observe that an a similar attack has been demonstrated here, which essentially uses the technique that we have just now discussed a just to mention you know like in the previous cases we were targeting the squaring operations. So, here we are basically targeting the doubling operation.

So, in the doubling operation now like in the squaring operation there was only one multiplication, but in the doubling operation there are several multiplications that you can target in particular in this attack we target the $T_1 Z$ to Z^1 square as the you know like underlying operation ok. So, there are eight multiplications and you know you can target any one of them. So, we target get 1 of them in the eight multiplications in the double operation ok. So, the also right and since there are many multiplications apparently what we observe is that the separation or the performance of the attack is even better on elliptic curve based implementations.

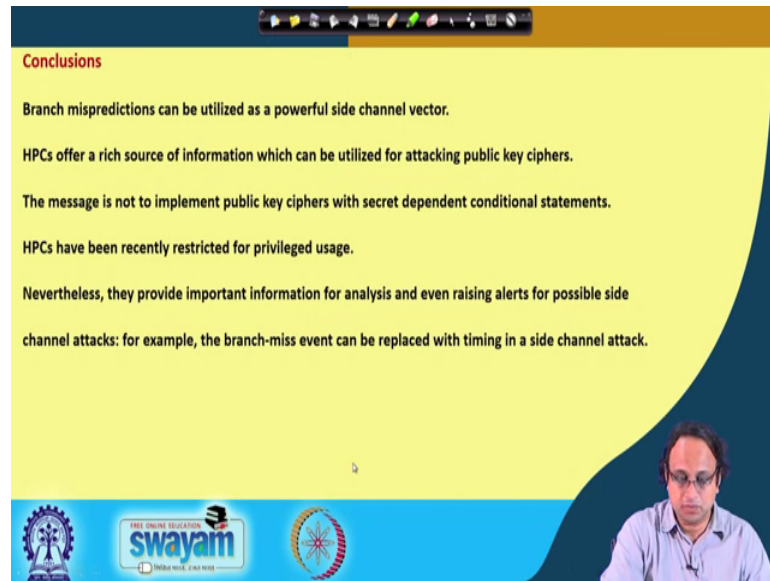
(Refer Slide Time: 35:41)



So, likewise you can also perform the attack on Montgomery ladder which we essentially know is a more balanced structure. So, here is an example to show that again how the separate how again we can perform the attack on a similar platform for example, this attack has been demonstrated on an Intel Core I 3 machine. And the offline phase for the Montgomery attack algorithm is slightly different from that of the double and add. Here 2 sub simulated traces are present depending on the scalar bits like in the previous case there was only one sub simulation which were doing.

But in the Montgomery ladder there will be 2 sub simulations which you would which you have to basically take care because of the fact that because you know that in one of them you are you are you are probably doing a doubling of you know you are doing it over there are two registers and you are basically operating on both of them. So, here two sub simulated traces are present depending on the scalar bit the next bit and hence the partitioning of the inputs is done by selecting one of these traces depending on the guess of the secret bit.

(Refer Slide Time: 36:36)



Conclusions

- Branch mispredictions can be utilized as a powerful side channel vector.
- HPCs offer a rich source of information which can be utilized for attacking public key ciphers.
- The message is not to implement public key ciphers with secret dependent conditional statements.
- HPCs have been recently restricted for privileged usage.
- Nevertheless, they provide important information for analysis and even raising alerts for possible side channel attacks: for example, the branch-miss event can be replaced with timing in a side channel attack.

swayam
INDIA WISE, LEAD WISE

So, to conclude branch mis predictions can be utilized. So, the powerful side channel vector. HPCs offer a rich source of information which can be utilized for attacking public key ciphers. The message is not to implement public key ciphers with secret dependent conditional statements, which we see are vulnerable to side channels and micro architectural attacks.

HPCs have been recently restricted for privileged uses and therefore right probably it is not easy to mount or it is not so, easy to mount able to mount attacks directly using HPC values. Nevertheless they provide important information for analysis and even raising alerts for possible side channel attacks. For example, right once you develop this attack you can probably try to do the redo this attack by just replacing the branch miss event with timing as a side channel vector ok.

(Refer Slide Time: 37:28)

References:

Discusses various timing attack algorithms in detail allowing readers to reconstruct the attack.

Presents the application of timing attacks on remote systems and cloud environments.

Examines information leakage models that would help quantify leakage in a covert timing channel!

Timing Channels in Cryptography
A Micro-Architectural Perspective
Oscar Ibarra, Gadiel Srivastava, Suresh Bhatnagar
Springer

swayam
MHRD

So, with this we would like to conclude and the reference that we have used for this book is essentially shown here is timing channels in cryptography published by springer.

So, thank you for your attention.