

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

**Lecture - 57**  
**Microarchitectural Attacks: Part 2 Branch Prediction Attacks**

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

(Refer Slide Time: 00:23)



So, in today's class we shall be talking about in particular branch prediction attacks, but in order to illustrate branch prediction attacks we shall be resulting to a tool which is called as Hardware Performance Counters or HPCs which essentially is a rich repository of several performance events inside our computing systems. So, we shall be seeing one of the tools which is very common and commonly used in Linux, its an tool that can be freely used. Its called as a perf tool and then we shall be defining our attack which is essentially the branch prediction attack and in particular we shall be seeing branch prediction attacks in the context of implementation of public key cryptographic algorithms.

(Refer Slide Time: 01:02)

## Hardware Performance Counters

- **Hardware Performance Counters (HPCs) are a set of special purpose registers, which are present in most of the modern microprocessor's Performance Monitoring Unit (PMU).**
- These registers store hardware and software events related to the execution of a program, such as cache misses, retired instructions, retired branch instructions, and so on.
- Every popular operating systems have HPC-based profilers, type and number of hardware events vary across different Instruction Set Architectures (ISA).
- There are various open-source tools which can measure these HPC values, such as, **perf tools, PAPI, OProfile, Valgrind** and many more.

```
Performance counter stats for 'system-wide':
20843.395552 task-clock (msec) # 2.000 CPUs utilized
7,473 context-switches # 0.353 K/sec
541 page-faults # 0.025 K/sec
7,114 cycles # 0.333 sec (99.85%)
3,998,457,744 instructions # 0.487 instr per cycle (75.25%)
181,795,724 branches # 47.514 K/sec (75.26%)
181,795,724 branch-misses # 0.76% of all branches (74.59%)

10.000000000 seconds time elapsed
```

values HPC

To start with here is a brief overview on HPCs or Hardware Performance Counters. So, hardware performance counters are a set of special purpose registers which are present in most of the modern microprocessors; essentially they are provided in the Performance Monitoring Unit or PMU of the microprocessors, in order to provide a rich set of or rich repository of several performance events. So, these registers store hardware and software events which are related to the execution of a program.

So, typically we execute a program and in that when we are executing right we would like to for example collect a or observe a bunch of instruction or events for example, the cache misses, the retired instructions and several others such instructions several such events. But always remember that when we are collecting these events we are although we are essentially accumulating this events in conjunction to executing a specific program, but the event is when for example, when we are collecting hardware events, then the hardware is shared by all the executables which are running on my system.

And therefore, right the hardware events for example, the branch misses right which we will see in this particular discussion is not only kind of you know like restricted to the program that I am executing, but pretty much on all the programs that are executing in the system. So, therefore, right we will be trying to conceive a scenario where the target or the victim executes an encryption or decryption algorithm and I would probably as a spy execute another program and I would be trying to observe the performance events or

the hardware performance events which are in particular you know like related to the branch misses. And when we do that we also kind of gather the branch misses which are generated from the target executable that is the target cipher.

So, that makes a potent or that gives an opportunity of an attack which you would like to elaborate it in our discussion. So, coming back to performance counters, every popular operating systems have got HPC based profilers and of course, like the specific cases or the specific way of using them might differ but the symmetrically they are kind of say.

So, there are various open source tools for example, which can be used to measure these HPC values like the perf tools we have very common, then in windows we have got PAPI which is very commonly used in window systems, then you have got OProfile, Valgrind and many more similar things. So, here is a snapshot of a perf stat. So, for example, you can see that there are some figures and there are some events which are being kind of highlighted. So, these events are essentially the HPC events. So, for example, the page faults, the cycles, the instructions, the branches, the branch misses and so, on along with it we also observe the corresponding values.

So, this is essentially accumulated when we execute a specific program ok. So, for example, like this could be a simple program as if like an LS or a even a PWD which is a very common Linux executable.

(Refer Slide Time: 04:13)

**Hardware Performance Counters**

- **Extra processor logic inserted to count specific events**
- Updated at every cycle
- **Strengths**
  - Non-intrusive
  - Very accurate
  - Low overhead
- **Weaknesses**
  - Provides only hard counts
  - Specific for each processor
  - Access is not appropriate for the end user nor well documented
  - Lack of standard on what is counted

The slide features a yellow background with a dark blue curved shape on the right side. At the bottom, there is a blue banner with logos for Swamyam (Free Online Education) and a circular logo on the right. A small video inset in the bottom right corner shows a man with glasses speaking.

So, therefore, right I mean the to look into more details thus. So, therefore, right the HPCs are basically an extra processor logic which is inserted to count specific events, which are updated at every cycle. The strengths of HPCs are that it is non intrusive it is very accurate, it has got low overhead, but the weaknesses are also can also be observed like it provides only hard counts its very specific for each processor and the access is not appropriate for the end user and its not very well documented and there is a kind of lack of standard on what is exactly being counted.

At the same time for example, there are lot of improvements in HPCs which makes it kind of easy to operate like for example, there is something which is called as easy perf which is really kind of you know like nice to use and also right although I can specify here that it provides hard counts their modifications of this utility, which also provides us opportunities in subsequent versions on obtaining dynamic information on the performance counters.

But exact or elaborate discussions of them are kind of beyond the scope of this course. So, at this point we will just be using the hard counts and we will be showing that how even them or even they can be used as a technique for attacking ciphers and also for evaluating implementations.

(Refer Slide Time: 05:39)

**The Perf tool for Linux**

- **Perf is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command-line interface.**
- The perf tool offers a rich set of commands to collect and analyze performance and trace data.
- perf stat
  - Run a command and gather performance counter statistics
- Perf measures several types of events.
- In particular they measure hardware events generated from the PMU (Performance Monitoring Unit) of the processor.

**Example of PMU Events Measured:**

- cpu-cycles OR cycles
- instructions
- cache-references
- cache-misses
- branch-instructions OR branches
- branch-misses
- bus-cycles

**Usage:** perf stat -e cycles,instructions,cache-misses [executable]

swayam

So, the perf tool for Linux is a very good or very useful tool particularly for doing experimentations ah. So, perf is a profiler tool for Linux 2.6 plus based systems, that

abstracts away CPU hardware differences in Linux performance measurement units and presents a very simple interface. So, if you want to use perf stat, essentially you can run it in the common line and you can gather performance counter statistics and it will it basically as I said that it provides a repository to several types of events. For example, you can mention you can measure the CPU cycles or cycles you can measure the instructions, the cache references the cache misses, branch instructions or branches the branch misses bus cycles and so on.

So, here is a possible usage of that. So, as you can see like we write here perf stat minus e. So, the cycles instructions, cache misses or and follow that is followed by the executables. So, these executables would be something like a simple hello world executable or as well. So, essentially its pretty much.

So, when we observe the hardware events for example, the cache misses, it is not only because of the executable that I am running, but also because of any other executable that is running in conjunction with this executable and that implies that we can develop a technique, where an observer observes pretty much what is happening and what is kind of affecting the hardware events of your system.

(Refer Slide Time: 07:07)

**Prediction in Computers**

**Simple Branch Instruction:**  
 BEQ: R1, R2, Imm  
 R1!=R2, PC++  
 R1=R2, PC+4+Imm

	Fetch	Read	ALU	MEM	WRITE BACK
BEQ					
??	BEQ				
??	??	BEQ			
✓	x	x	BEQ		

2 cycles for mis-prediction!  
 Branch prediction is based on the address only!

swayam

So, therefore, right with this background let us try to look or try to understand how a prediction logic works in our computers? So, this is simple you know like slide which explains the idea of prediction in computers. So, I will, so, just very briefly these are

very simple branch instruction that we consider here, like suppose you consider the instruction like `branch equal R 1 comma R 2` that is followed by immediate flag; that means, if R 1 and R 2 are equal, then you essentially the idea is that as a elaborant here if R 1 and R 2 are equal, then you basically increment the program counter by and go to the next program counter location which essentially could be PC plus 4 plus the immediate value. Or if R 1 R 2 are not equal then what you do is you basically jump to a location which is the next location that is PC plus plus.

So, the idea is that if R 1 and R 2 are equal, then we jump to the location which is essentially a kind of described in the immediate flag or the immediate value, if R 1 and R 2 are not equal then you go to the next subsequent instructions. So, that you just increment the program counter. So, now, you can so, here is a you know like the idea is that, why is branch prediction so, important in our computing systems.

So, in order to understand that let us consider a simple 5 stage pipeline and consider that we have this instruction `branch equal` which is initially fetched for execution. So, when it is fetched for execution, then remember that this point right this instruction is not even decoded. So, we just kind of fetch the instruction and we have got this instruction at hand. So, now what we do is that we allow it therefore, we even do not know whether its a branch instruction.

So, therefore, right you see that at this point right when we even I have not encoded or I mean not decoded this instruction, we have to basically make a prediction of what we will fetch next. But as the logic here shows that if this is the branch instruction, then there are 2 possibilities we can either increment the program counter or the program counter can be updated by the immediate value; that means, it can go or jump to a specific location. So, therefore, right in a way here itself I have to make a prediction, and I have to basically ensure that the prediction works only on the address of the instruction.

Because at this point I even do not know whether its a branch instruction. So, therefore, right I, so, when I allow the branch instruction to go to the next stage, which is essentially the read registers and also decode at this point I probably know that its a branch instruction because I can decode the up code, but then by that time already I have kind of dragged in the next instruction I have fetched in the next instruction. So, then what happens is that this instruction goes to the ALU and only then I know the outcome

of the branch instruction. Remember at this point I do not know the outcome I do not I have got I have not yet read the values of R 1 and R 2 and also I do not know whether they are equal or they are unequal.

So, therefore, right at this point when there is in the third stage of the pipeline and this could be further deep when your pipeline is even more deep now, which essentially happens in typical systems that we use or where you work nowadays. So, that would imply that collectively we have you know like already fetched in few instructions and at this point right if you believe that this instruction is essentially wrongly fetched; that means, the branch for example, resulted in an R 1 equal to R 2, then we have to basically kind of throw away these instructions and we have to bring in the correct instruction now.

So; that means, right there are 2 things that we have already brought into our pipeline, which we have to throw away from the pipeline. So, this essentially makes a essentially is not very nice and this essentially is often a as in the computer architecture paradigm called as control dependency. So, we need to kind of have a mechanism, and branch prediction ways provides as a mechanism of kind of you know like reducing the effect or bad effect of a branch misprediction.

(Refer Slide Time: 11:02)

**1-bit Predictor**

not taken                      taken

0                      1

predict "do not take branch" state      predict "take branch" state

T T T T N T T T T

✓ ✓ ✓ ✓ x x ✓ ✓ ✓

Each anomaly there are two mispredictions

**Predicts well:**

- Always Taken
- Always Not Taken
- Taken>>>Not Taken
- Not Taken>>>Always Taken

**Does not Predict so well:**

- Taken>Not Taken
- Not Taken>Always Taken
- Short Loops (loops with an exit condition, one which stays in the loop, and goes back)
- **Work bad for cases where Taken  $\approx$  Not Taken**

swayam

So, there are different ways of doing or developing predictors and there is a rich history in that. So, I will just try to kind of illustrate few of the important ones that we would

probably need in our discussion. So, the most common form of a predictor would be a 1-bit predictor, where we essentially have a 2 state finite state machine. So, there are 2 states like there is a state is in 0 or the state is in 1. The moment I basically see for example, that the input or the branch input is taken.

So, in that case right I kind of jump from a 0 to 1 state and therefore, for the next instance where I have to do a prediction I predict taken. So, therefore, the one occurrence of it taken or not taken kind of changes my decision. And you will immediately understand that this is not a very nice prediction logic although it is very simple and similarly you know like the other things are also defined. For example, 0 to 1 transition as I said and 1 to 0 transition will happen in case of a not taken likewise in a state will remain in 0 for example, if there is not taken and likewise if a state will stay in 1 if the taken branches keeps on occurring again and again.

So, therefore, right this is not a very accurate or very nice way of doing prediction for example, you can easily understand that if you have a sequence like taken, taken, taken and so on followed by taken not taken and so on again taken; then the moment there is a taken I basically change my decision. So, therefore, I make I start predicting taken, but then suppose I get a not taken. So, that would imply that I change my decision and therefore, I make a misprediction again when I make a when I get a not taken right I again make another misprediction and therefore, it pretty much implies that I make for every anomaly there are two mispredictions which occur.

So, this typically predicts well when there is always taken for example, like the sequence would have been having all takens and it worked well; if it was always not taken then also it would have worked well. This would have also worked if the number of takens is significantly you know like more than the number of non-taken and so, all for example, if the not taken is significantly more than always taken.

But it does not work well in particular when the taken and the not taken are kind of equivalent in their numbers and the taken is slightly greater than for example, not taken or the not taken is slightly greater than always taken or for short loops, that is loops with an exit condition, where one which stays in the loop and goes back. So, these are typically loops which are often used in our programs and essentially this implies these



are loops like whether it is a non exit condition and there is one which stays in the loop and one and then goes back.

So, essentially you can consider a simple for loop to illustrate this kind of loops. It also works bad for cases where taken is equal to not taken for so, this essentially typically works bad in those instances.

(Refer Slide Time: 13:51)

**2-bit Predictor**

Prediction bit	Hysteresis (conviction) bit
0	0
0	1
1	0
1	1

Strongly not taken, Weakly not taken, Weakly taken, Strongly taken

T T T T N T T T T

✓✓✓✓x✓✓✓✓

Likewise, we can move to 4-bit predictors, but the cost increases exponentially. Further, the chance of such sequences in real programs is limited. So, in real life we have either 2-bit or 3-bit predictors.

Having a 3-bit predictor can be good when there are 2 NT in between.

So, therefore right people of are architects are developed a better predictor which is called as a 2-bit predictor, then the idea with 2-bit predictor can be illustrated by a 4 state machine. It is something like a counter and therefore, it is often called as a 2-bit counter; as you can see that the initial state is strongly not taken state. So, now, here right I mean as opposed to the previous one, we basically bring in another extra bit which is not only the prediction bit which I already have in the 1-bit predictor, but I also bring another extra bit which is called as a hysteresis or the conviction bit.

So, the idea here is that the state machine can be now can described by 4 states. So, this is a strongly not taken state, there is a weekly not taken state there is a weekly taken state and a strongly taken state. So, the idea is that if you get or start getting sequences which are not taken, then you remain in your strongly not taken state. And then right if you start getting taken sequences, then you basically migrate from the strongly taken strongly not taken to the weakly not taken, but you do not change your opinion. That means, a single

taken input right does not change your opinion and you still keep on predicting not taken in the weakly not taken state.

But if you get one more taken input then you basically go to the weakly taken state and you start predicting taken as an outcome. Likewise when you again get a take input then now you basically go into the strongly taken state and you again start predicting taken, and if you keep on getting taken then you remain in the strongly taken state. Likewise your transition from strongly not taken to weakly taken and again you know from weakly taken to the weakly not taken and from weakly not taken to the strongly not taken will happen on a sequence of not takens.

So, idea is that, you will basically change your decision if you get 2 different kinds of inputs; like if you get a taken the you go from strongly not taken to a weakly taken state and start predicting taken and likewise right if you have for example either in the strongly taken state or in the weakly taken state, if you just you know like get 2 not taken sequences.

So, for example, if you get if you are in the strongly taken state and if you get a not taken sequence, but you get two of those kind of not taken states. Then you basically go from the strongly taken state to the weakly not taken state and you start predicting not taken. So, here if you again you know like give the previous inputs. So, you see that you have got a sequence of taken, taken, taken so on followed by one anomaly. So, now, note that this predictor is better than 1-bit predictor, because now because of this one not taken state and if you are in the strongly taken state you do not change your decision.

You do not and therefore, right if you start getting taken you again start predicting taken because you have not changed your decision. So, therefore, if you are in the strongly taken state and if you get a not taken as input, then this not taken will bring it to the weakly taken state and you will still predict taken as an outcome and therefore, when the next taken occurs you there is no misprediction. So, therefore, you just get one misprediction per anomaly, which is kind of 2 times better than what you got previously. So, likewise right you can have a 3-bit predictor which can be even good for example, if there are 2 not takens in between.

But definitely right a 3-bit predictor will be more costly compared to a 2-bit predictor. So, therefore, right in real life we will either have a 2-bit or we will probably have either

a combination of 2-bit predictors or 3-bit predictors and probably it does not really its probably it is not so, advantageous to go from a 3-bit predictor to more bit predictors.

(Refer Slide Time: 17:17)

How do we Improve?: History vs Majority

NT T NT T NT T NT T ... **100 % predictable**  
NT NT T NT NT T NT NT... **But not with n-bit predictors.**

With 2-bit predictors you will have significant number of mis-predictions!

History: NT T  
Prediction: T NT

From history we can learn the pattern!  
Then this sequence is 100% predictable!

History: NT,NT NT,T T,NT  
Prediction: T NT NT

From history we can learn the pattern!  
In this case, we need 2-bits of history.

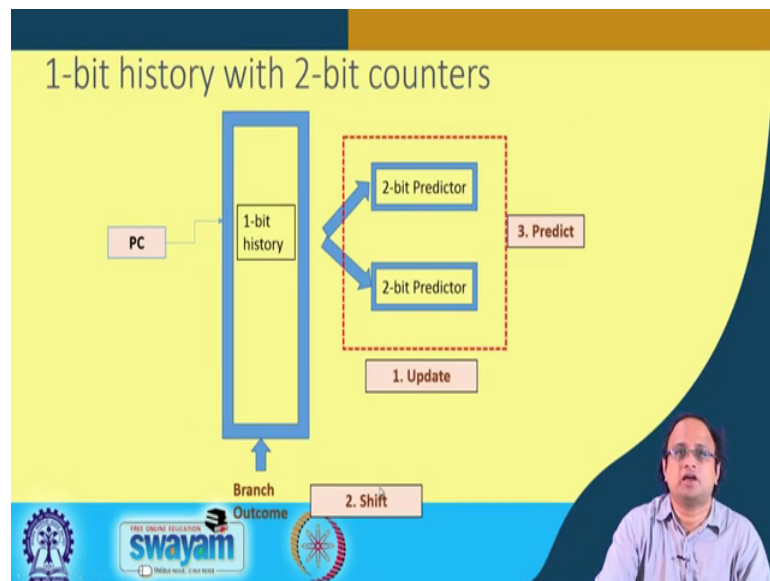
So, rather what we need to improve is essentially, but we still need to improve? For example, like these are certain instances which are 100 percent predictable, but they are not so, predictable with n-bit predictors like with 2-bit predictors or with 3-bit predictors. For example, if I give a sequence like not taken, not taken, taken, not taken, taken then as a human you immediately know that the next sequence would be taken ok. Likewise, if I give the sequence like not taken, not taken, taken, not taken, not taken, taken, not taken, not taken you immediately know that the next sequence is expected to be taken.

So, there are a kind of 100 percent predictable, but they are not predictable with a 2-bit predictor. The reason why it is not predictable because in a 2-bit predictor or the predictors that we have seen in the context of n-bit predictors largely performs its prediction based upon what is called as majority of the inputs. So, we basically try to see which is the majority among taken or not taken.

But here what is important is the history of the of the input. So, for example, if you observe the history, then you see that immediately write that observe that the first one is predictable because you know that if the input is not taken or if the history is not taken then I should predict taken, if the history is taken then I should predict not taken.

So, I just should invert my history bit. Likewise right if you see for the second sequence you observe that if the input is not taken not taken, then I should predict taken likewise if the input is not taken, if I observe 2 bits of history then I should predict not taken. If the input is taken not taken then I should predict not taken ok. So, therefore, from history we can learn the pattern and then this all this both the sequences are 100 percent predictable. So, therefore, right we can indeed you know like make it improved and therefore, right we basically can improve in both the cases.

(Refer Slide Time: 19:06)




So, therefore, right we basically have got developments in branch predictions, which has happened by combining 2-bit counters with history bits. So, here is a very simple this elaboration of that. So, therefore, you either you take the program counter and then there is a 1-bit history. So, now, this 1-bit history is essentially combined or is used to kind of address two 2-bit predictors.

So, there are two 2-bit predictors and you use the history bit to kind of tag each of them. So, therefore, there is. So, there you have to develop an architecture for doing so and the architecture typically can be remembered by understanding the sequence of what is called as update, shift and predict ok.

(Refer Slide Time: 19:47)

## Update-Shift-Predict

State	Pred	Outcome	Correct
(0,SN,SN)	N	T	x
(1,WN,SN)	N	N	✓
(0,WN,SN)	N	T	x
(1,WT,SN)	N	N	✓
(0,WT,SN)	T	T	✓
(1,ST,SN)	N	N	✓
(0,ST,SN)	T	T	✓



So, therefore what it means here can be elaborated or observed here. So, therefore, this is the sequence that we just had like the sequence was taken, not taken, taken, not taken, taken, not taken and we show now how it is 100 percent predictable. So, now, what we do is that we take the history bit for example, the history bit is initialized to 0 and the 2 counters or the two 2-bit counters are initialized to 2 say strongly not taken.

So, now, what you do is that you use this 0 bit to basically predict. So, now, therefore, so, the as I said that the two 2-bit counters are addressed the first ones address is 0 the next ones address is 1. So, therefore, seeing 0 you see its predictions. So, you see that is strongly not taken. So, you predict not taken.

So, therefore, you come here you see it is not taken, the moment you see it is not taken you see that the actual outcome is taken. So, that is a misprediction. So, you get a misprediction in the first instance. So, now, as I said the sequence is update, shift and predict. So, what we will do is that we will take the outcome bit and we will shift it ok. So, there is only 1-bit here. So, therefore, the shifted bit becomes. So, 0 becomes 1 here because of this outcome in case of more history bits you can do a shift in you can do a circular shift in for example.

So, therefore, this taken, we will make it one and therefore, now you have done the shift and now you will predict. So, therefore, now you will use this address 1 to predict. So, observe that this is this address is 0 this address is 1. So, you see strongly not taken and therefore, you predict not taken. So, therefore, right now what you do is that you come

you basically predict here and therefore, the prediction is correct and that is shown here, that the prediction is indeed matching with your actual outcome. So, now, what you will do is, basically you will basically update the counters. So, you are basically you have to you have a sequence of update, shift and predict.

So, you have to update the counters which means like your flag is now 1 and your input is not taken. So, therefore, you basically you have to update this strongly taken and not taken and therefore, it remains in the strongly not taken and therefore, again you what you will do is you will shift. So, again you have got a not taken. So, you basically shift the not taken here it becomes 0 and now we will predict. So, observe that here since this is 0, you will use the first 2-bit counter and that predicts not taken. So, therefore, you will be predicting not taken because it is still in the weakly not taken state.

So, you basically predict not taken and that results in a misprediction. So, now, what will happen is that you have already predicted. So, now, you will basically update and since 0 is your counters and weak not taken is the state and the outcome was taken. So, you go from the weak not taken to the weak taken state ok. So, the 0th counter gets updated and then you basically shift. So, this taken becomes or comes here shifts in, and this becomes one. So, it is 1 weak taken and strongly not taken. So, now, what you do is that you basically predict.

So, now, when you are predicting you basically use the address of the second counter and the address of the second counter is not taken. So, you basically predict not taken and pretty much after that right you get all of them as correctly predicted ok. So, your prediction starts matching. And therefore, right if you just ignore the few initial hiccups you can see that this particular predictor is able to learn this entire prediction or this entire input sequence. So, therefore, people have developed further and further to improve the you know like the inaccuracy of this kind of branch mispredictions.

(Refer Slide Time: 23:08)

2-bit history with 2-bit counters

2 bits C0 C1 C2 C3 Total=2+2.2<sup>2</sup>=10 bits

(NNT)\*: The previous predictor fails to learn

N	N	T	N	N	T
		00	01	10	00
		C0↑	C1↓	C2↓	C0↑

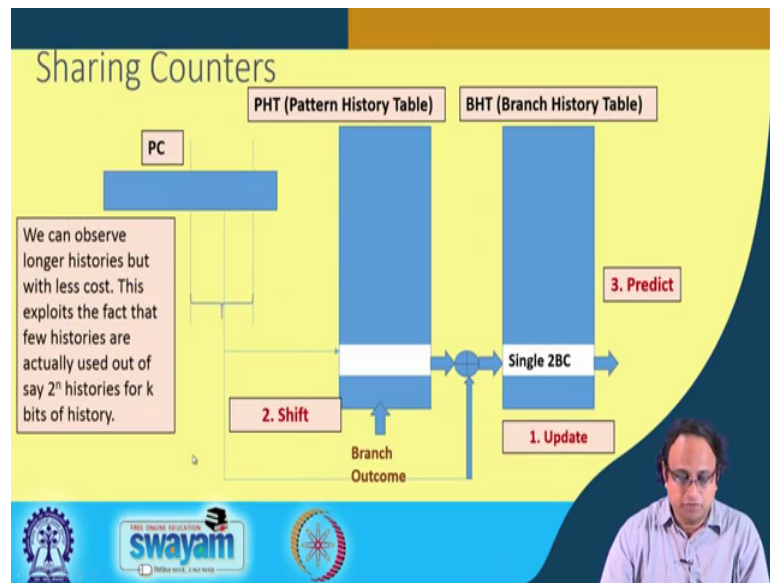
2 bits ST SN SN ← Not used

And here is an example of how to show that even the other sequence like followed by NNT and so, on like NNT star NNT star can be completely learnt with a 2-bit history and with 2-bit counters. For example, if I have got 2-bit counters then I can pretty much have 4 counters like C0 C1 C2 and C3 and if my input is NNT and without going to the details we can understand that it can be completely learnt, because if you have the sequence like NNT star then that is like NNT NNT and so on.

Then if you have got the 0th counter with basically up counts at the input taken then this basically up counts and this that is the second counter basically down counts when the input is not taken and the third counter again down counts when the input is not taken and the fourth counter essentially up counts when the input is taken. Then you can see that after some inputs then this will basically be go to the strongly taken state, this will be coming to the strongly not taken state and this will be coming to the strongly not taken state and this is actually not useful.

And therefore, right this will be predicting as taken which basically matches this will be predicting as not taken which also matches and this will also be predicting as not taken which also matches 100 percent time. That means like it is always correct.

(Refer Slide Time: 24:23)



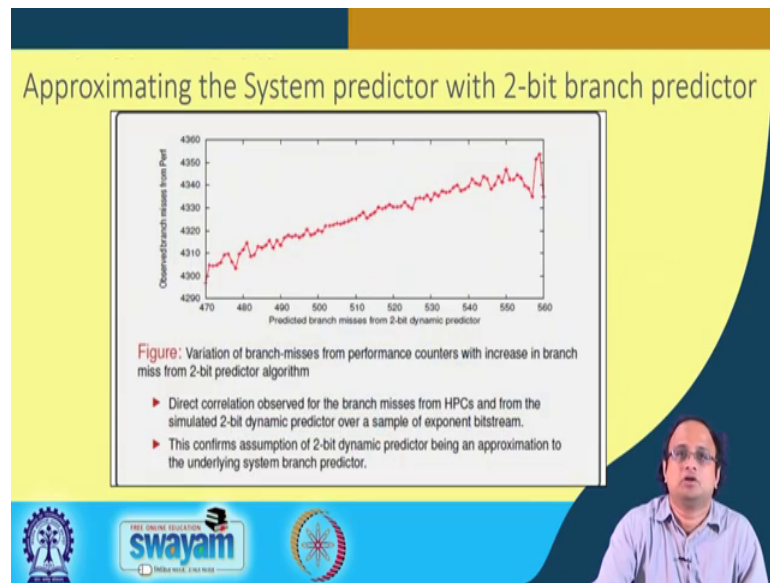
So, therefore, right I mean people have further developed this kind of prediction logics and that brings us to something which is called as essentially the pattern history table, which basically kind of basically the objective is now to observe longer histories, but with less cost. Because cost is an important aspect when you are designing high performance computers and this exploits the fact that few histories are actually used out of say 2 to the power of n histories. For example, if you are using say n-bit counters and you are observing say 2 to the power of n histories maybe all of them are not useful.

As we already saw that in the previous case this counter was not really useful. So, therefore, right what we try to do is rather try to bring in a second data structure which is called as a pattern history table and the pattern history table basically kind of addresses few counters. And essentially its again a combination of the same update shift and prediction, but the only difference is that now we would like to share the counters and try to kind of reduce the cost of the misprediction or the branch predictor.

So, the idea which I am trying to kind of imply here is that, the branch predictions are much more complicated than say the simple model like a 2-bit predictor. But at the same time we also do not know what is probably you know like really a predictor which is used in actual systems like with a design by say Intel or AMD or so on.



(Refer Slide Time: 25:55)



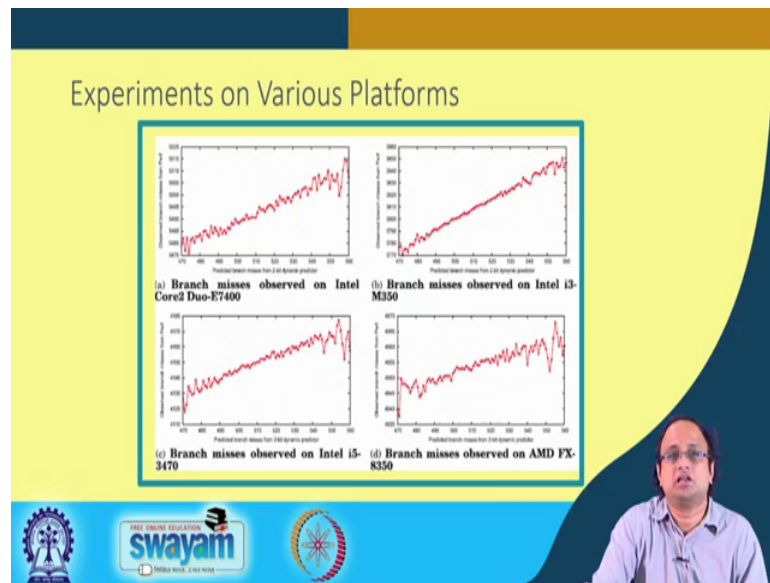
So, therefore, right we would. So, therefore, in our attack we show or in this attack right we basically show that we can observe very interesting phenomena. The phenomenon is that we can actually approximate the system predictor with the simple 2-bit branch predictor.

So, what we do is as follows, we basically see the actual branch misses which are gathered from a 2-bit dynamic predictor. So, this is a simple 2-bit predictor that we just saw and this is the branch misses which are observed from the perf statistics; that means, from the hardware performance counters of your actual system, when you are running say different kind of applications.

So, yes we are not saying that your you know like the prediction logic is as simple as a 2-bit predictor, but what we try to do is, we basically illustrate here or observe here that there is a very strong correlation between the branch misses which you basically gather from your simple 2-bit prediction logic and what you actually gather from your hardware performance counters.

So, therefore, the question is right the this essentially the question that we would try to address later on after this is that, whether this correlation can be exploited to illustrate an attack.

(Refer Slide Time: 27:07)



So, this essentially we also you can be verified on several platforms for example, these are branch misses which are observed on Intel Core2 Duo systems these are observed on Intel i3 systems these are observed on Intel i5 systems and this is on an AMD system. So, you can see that in all cases there is a strong correlation between the 2-bit predictor and what is actually happening inside your computer.

(Refer Slide Time: 27:30)

The slide, titled "Utilizing Performance Counters for Compromising Public Key Ciphers", contains the following text:

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.

The slide also features the Swamyam logo and a speaker in the bottom right corner.

So, therefore, now we will basically try to answer this question that whether we can use this approximate model of the predator for attacking public key encryptions. In particular

we shall be targeting a 1024 bit RSA and a 256 elliptic curve implementation in the first case we will that be using a simple plain square double and multiply variant whereas, in the second case we will try to illustrate the attack on a Montgomery ladder which is protected against timing channels, but we will discuss this in next class.

So thank you for your attention.