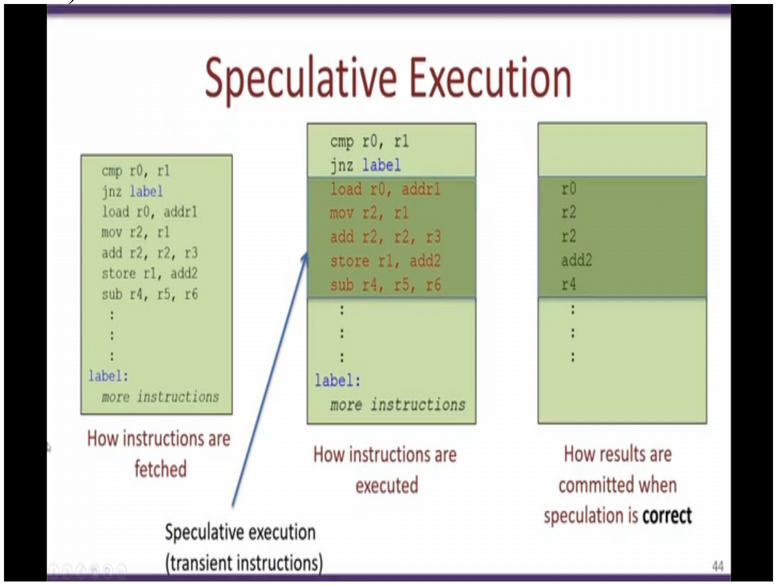
Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Spectre

Hello and welcome to this lecture in the Course for secular systems engineering in the previous video lecture we had actually started about speculative execution and we had look at this recent attack known as meltdown a in this video lecture we look at and others speculative attack known specter so this attack works basically in was this attack was discovered again in January 2018 and also makes use of the speculative execution of Intel processors.

(Refer Slide Time: 0:49)



Speculative Execution

```
cmp r0, r1
jnz label
load r0, addr1
mov r2, r1
add r2, r2, r3
store r1, add2
sub r4, r5, r6
  :
  :
  :
label:
  more instructions
```

How instructions are fetched

```
cmp r0, r1
jnz label
load r0, addr1
mov r2, r1
add r2, r2, r3
store r1, add2
sub r4, r5, r6
  :
  :
  :
label:
  more instructions
```

How instructions are executed

```
r0
r2
r2
add2
r4
 :
 :
 :
```

How results are committed when speculation is **correct**

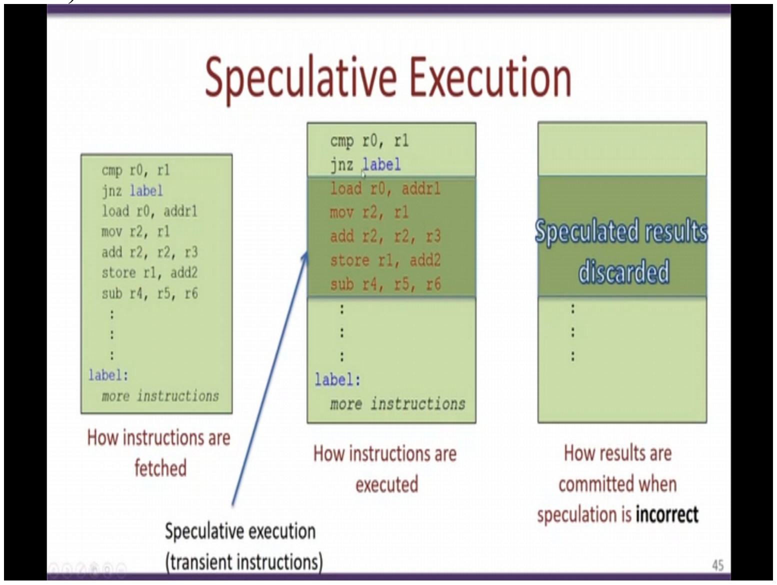
Speculative execution (transient instructions)

44

As we have seen in the previous video what speculative execution in a processor does is that certain Instructions can be speculatively executed even before prior instructions have actually being completed so for example over here this set of instructions can be speculatively executed and the result for these instructions will not be committed unless and until this previous result corresponding to the compare and the jump have completed execution only at the speculation is correct would these instructions be committed.

So does we see that if there is a speculation and the speculation is correct then of all of these results can be immediately committed and the performance of the program would increase constantly.

(Refer Slide Time: 1:47)



Speculative Execution

```
cmp r0, r1
jnz label
load r0, addr1
mov r2, r1
add r2, r2, r3
store r1, add2
sub r4, r5, r6
  :
  :
  :
label:
  more instructions
```

How instructions are fetched

```
cmp r0, r1
jnz label
load r0, addr1
mov r2, r1
add r2, r2, r3
store r1, add2
sub r4, r5, r6
  :
  :
  :
label:
  more instructions
```

How instructions are executed

Speculated results discarded

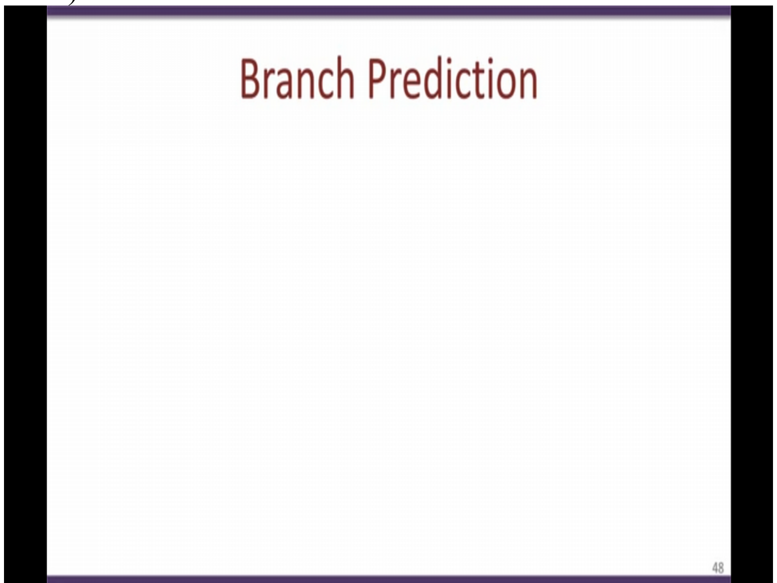
How results are committed when speculation is **incorrect**

Speculative execution (transient instructions)

45

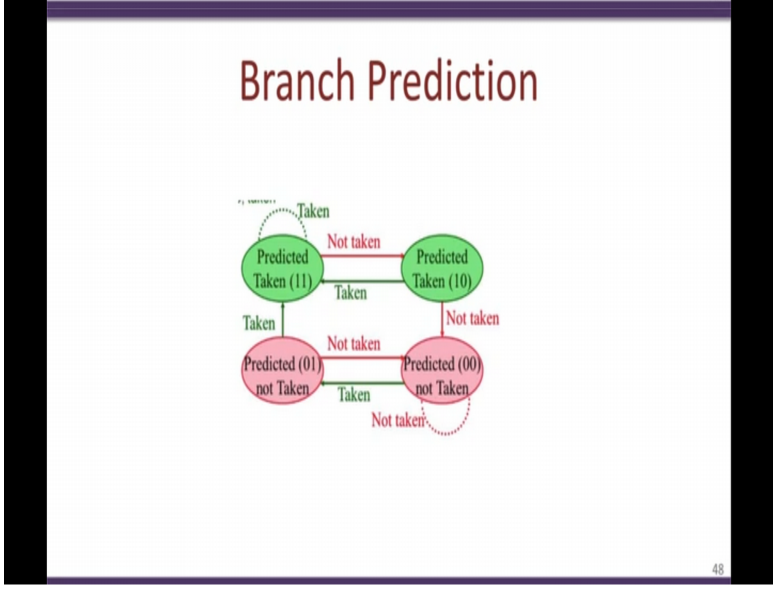
On the other hand if the speculation was wrong that is there was a branch that occurred over here too this particular address and the instructions that followed follows this label has to be executed then all the speculatively executed result needs to be discarded now here that it would be a performance overhead processors try their best therefore to speculatively execute correctly they would try to predict what would possibly be the result of this compare instruction and would try to speculatively execute either this code following the jump on no zero instruction or the code following the label depending on what they predicted would be the result of this jump on no zero instruction.

(Refer Slide Time: 2:42)



Branch Prediction

48



Branch Prediction

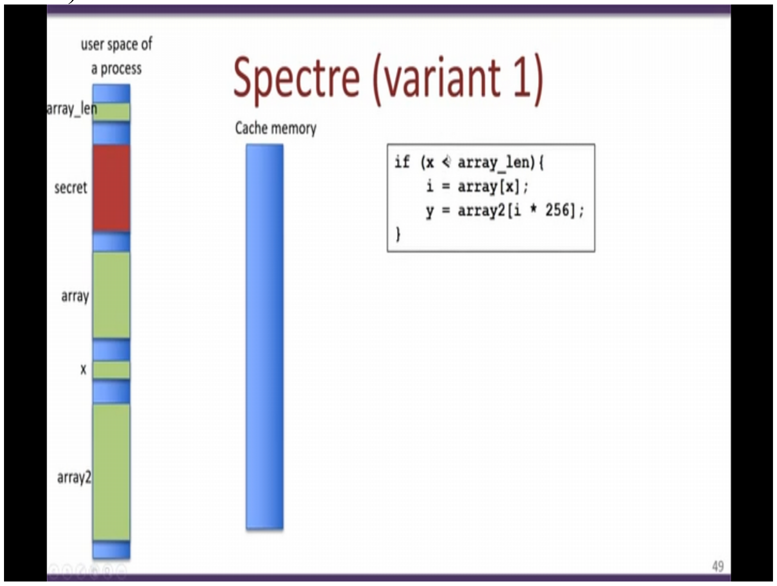
48

So in order to achieve this What is added to the processor is something known as a brunch prediction logic a branch prediction logic would keep track of all the branches that were taken at that particular instruction so for example in this figure it shows a 2 bit branch predictor and the branch prediction logic would either predict that the branches taken or not taken based on what had happened of previously when that instruction was executed so for example let us say that we start off with the prediction not taken and the first iteration of that 2 resulted in the branch being taken.

So then the state of the brunch predict comes from here to from 00201 another iteration of the loop the next iteration of the loop if again the branch in the branches is taken then a the state of this branch predictor moves to predicted taken and which has a value of 1 1 so now you see that the 3rd time when that a jump on no zero gets executed the branch predictor would automatically predict that the branch would be taken.

So does we see that the branch predictor is learning based on prior branches and trying to predict the result offered particular branches whether the branch would be taken all the branch would not be taken the speculative execution that follows would hopefully have a better accuracy and therefore would boost the performance so what was shown in the specter attack was that these branch prediction plus the speculation could result in a wonder ability by which secret data present in the program can be read.

(Refer Slide Time: 4:37)



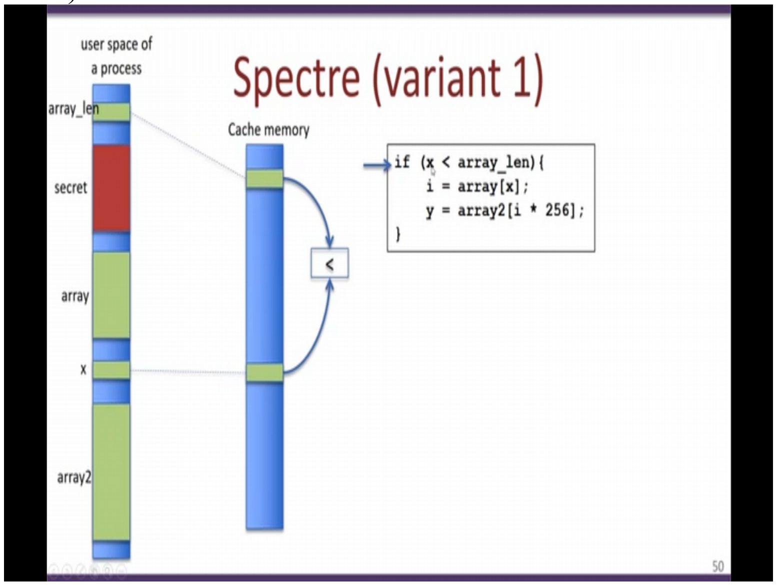
Spectre (variant 1)

```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

In order to understand the specter attacks let us take the small a snippet of code this code comprises of an if statement read where the value of X is checked with array len and if the value of X is less than array len then we are permeating access to this array at the index X and the value of take array is taken into I And then they are also accessing a second array array 2 at the location I into 256 and the result is stored in our this variable called viol so what we see with here is that the array 2 is accessed at a location which is specified by array of X.

Now what we look at next is the use of space part of the process now we will assume that there is a secret information that is present here and the attackers wants to read some parts of this a secret data the other components of this small snippet of code see the X array and array 2 are also shown in this user space part of the process do you have array over here X array 2 and also array len for simplicity we have actually ignore the value of I and we assume that I is also present somewhere in this a users space part of the process or you can also assume that I is going to be part of that I and Y are just going to be stored in registers.

(Refer Slide Time: 6:28)



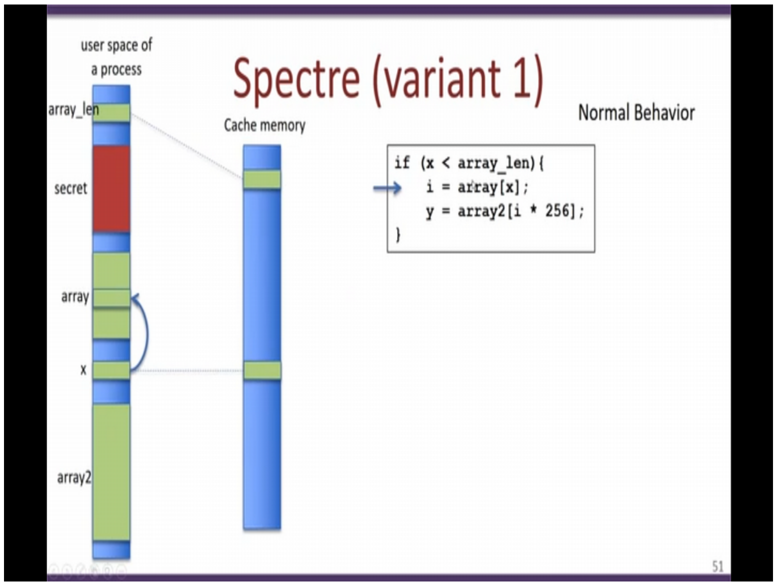
Spectre (variant 1)

```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

So let us see how execution off this small snippet of code takes place in normal circumstances so the first thing that is done is that when this line gets executed we have to load instructions one is the load for this variable X and the load for this variable array_len so these 2 loads would cause X and array len to be loaded into the registers and during that particular process it would also get load loaded into the cache memory.

Now what the showed here is that there is a comparison between X and the array len now if X is indeed lesser than the array len which we assume is true right now then if condition is entered and these two instructions are executed and now let us assume this is the case right now.

(Refer Slide Time: 7:18)



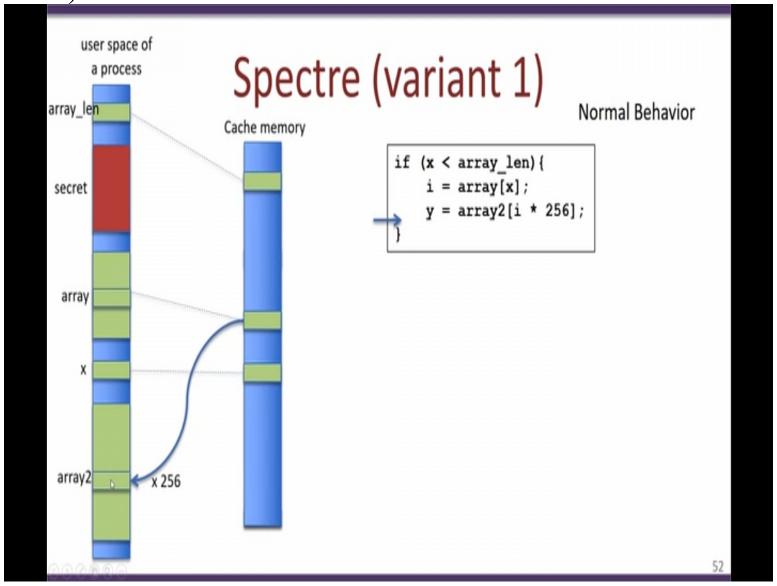
Spectre (variant 1)

Normal Behavior

```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

The next we see is that the if statement is entered an X is used to index into the array and cause one block of data Corresponding to array X to be loaded into cache so this data you can assume is loaded into cache and into a register which we denote as I.

(Refer Slide Time: 7:44)



Spectre (variant 1)

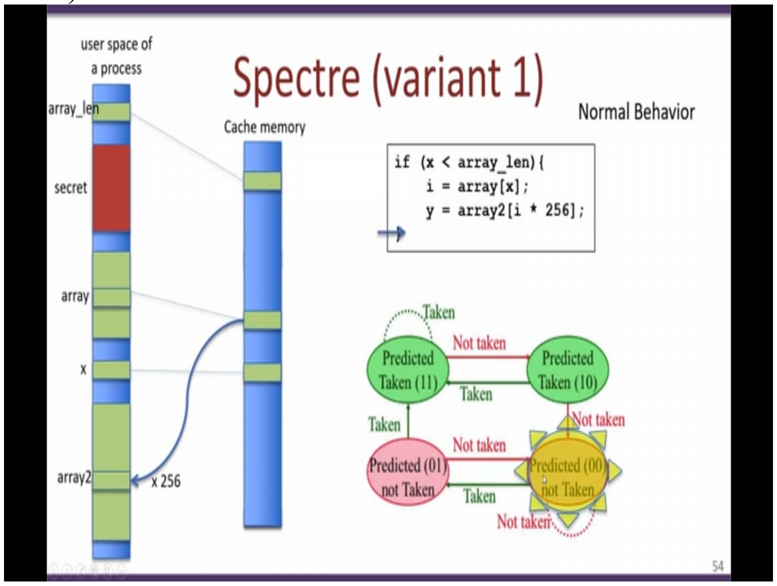
Normal Behavior

```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

Next what we see is that this value I is then used to indexed in to array 2 in other words we have this value I which is present in the cache or in the register is used in to used index in to array 2 and cause one block of data present in array 2 to be loaded into the cache memory typically if X is less than array len then what we achieved at the end of this execution is that we have the array len present in the cache memory similarly we have X present in the cache memory and 2 blocks of data one corresponding too I in the cache and the other corresponding to Y so all of these data would be present in the cache.

Now this is under the normal behavior when X is indeed less than array len so know that these kind of checks is quite common in lot of programs to ensure that an array is always indexed at the location which is within it's bounce now consider the case that these small snippet of code is in a loop and we are calling the small snippet of code multiple times so as a result we know that if over here and therefore there's a brunch instruction this repeated invocation of these 3 statements would actually tune the branch predictor to predict that the branch is not taken.

(Refer Slide Time: 9:15)



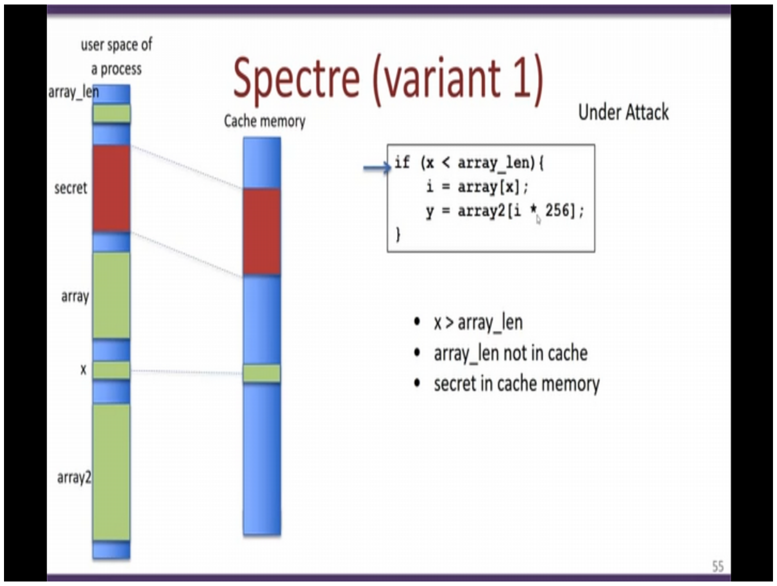
Spectre (variant 1)

Normal Behavior

```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
```

So if we consider what the branch predictor would do is that repeated invocation of these if statements and with the condition that X is less than array len would eventually move the branch predictor to this particular state where the branch is considered to be not taken therefore from a speculative execution what can happen is that these instructions I equal to array of X and Y equal to array 2 at the index of I into 256 so these two statements can be speculatively executed by the processor.

(Refer Slide Time: 9:50)



Spectre (variant 1)

```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

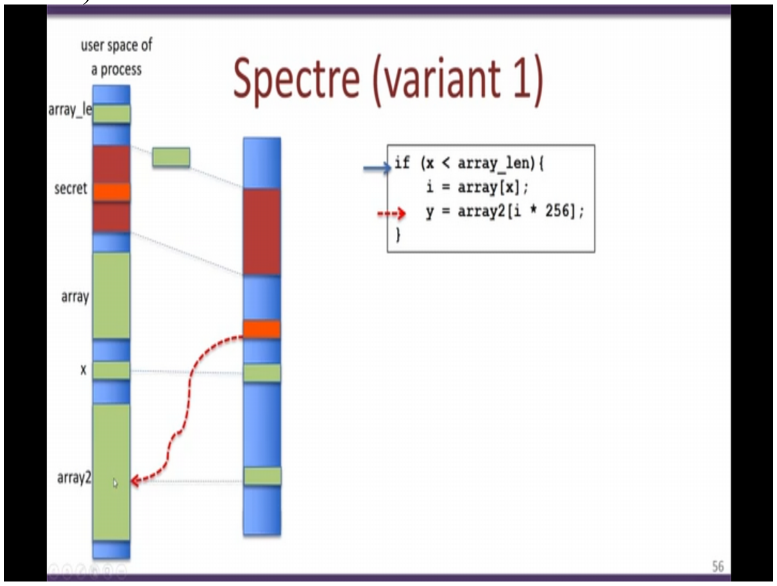
- x > array_len
- array_len not in cache
- secret in cache memory

55

Now consider the this particular case we would assume the case where X has already been loaded into the cache and we have the secret data already present in the cache but the array len is not present in the cache now consider again the execution of these statements but consider the case that we have X is greater than array len so typically in what should happen over here is that since X is greater than or equal to array len these statements inside the if should not be executed so typically since X is greater than array len the statements inside this if condition should not be executed.

okay so let us see when a such a scenario occurs how this program behaves now since we have a assuming that array len is not present in the cache we also are assuming that the branch predictor it has learned and is in the state my prediction is not taken so there for a what would happen is that even though X is greater than array len these instructions within the if would be speculatively executed.

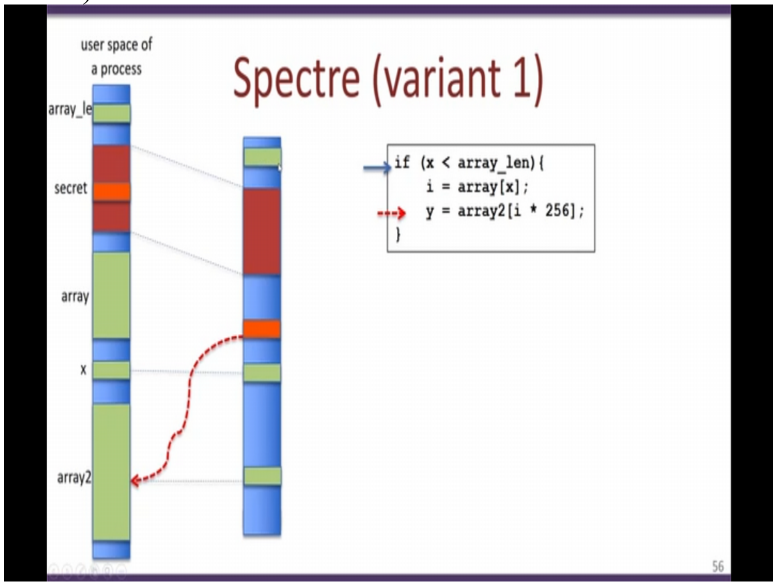
(Refer Slide Time: 11:09)


Spectre (variant 1)

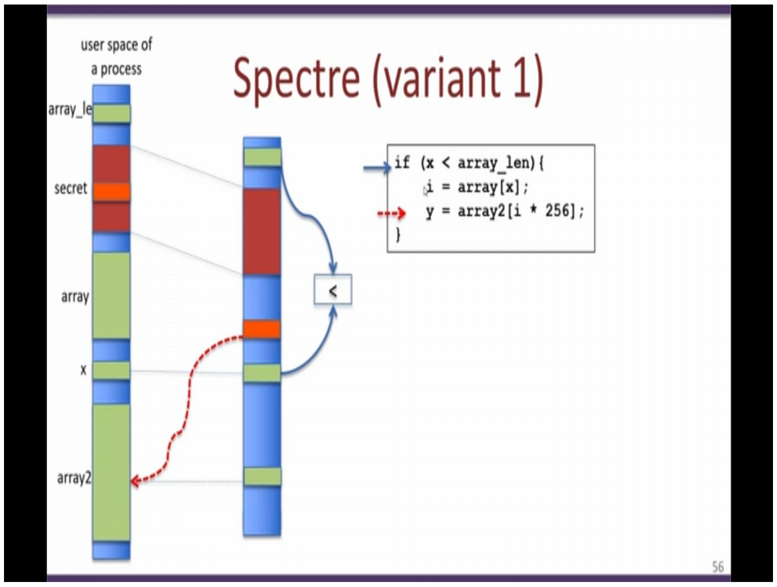
```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

so first we would see that since array len is not in the cache it would cause some cache miss which would take constable amount of time and of course array len to be loaded from the main memory and to the cache memory.

(Refer Slide Time: 11:27)


Spectre (variant 1)

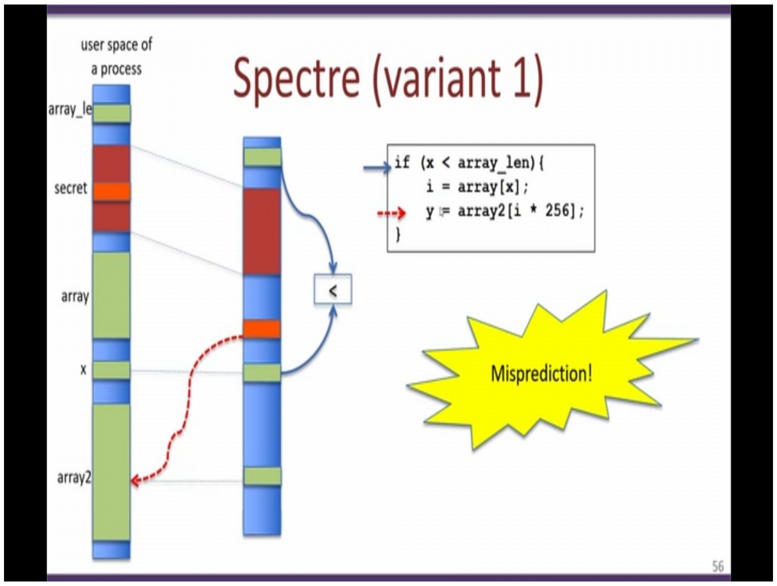
```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```



```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

On the other hand the other components are X is present in the cache and we fill the value of X with some location comprising of secret so since the secret is present in the cache the index array of X I equal to array of X can be very quickly evaluated pick and similarly we would what we would have is that some location in the cache would contain the value of array of X next what we are doing is we are using this particular value which is essentially a secret value to index into of this array 2 and all the contents of the array 2 into the cache memory now while this process of process is going on the this to mean that array len has after while has finally reached the cache memory and now since X an array len have find the arrived this check that is pointing to this particular statement can finally be invoked but the processer would have would now observe that X is greater than array len and there for all the speculative execution that has been done should be discarded and therefore the process so would discarded this speculatively executed instructions.

(Refer Slide Time: 12:42)

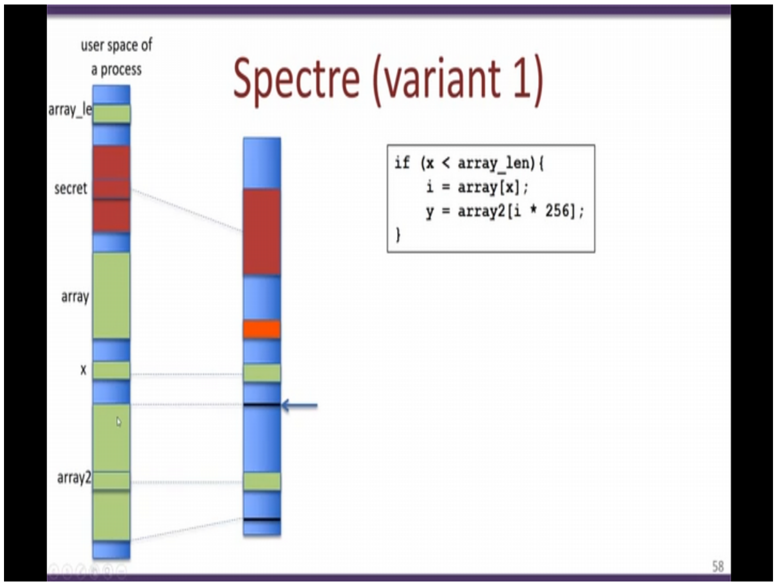


Spectre (variant 1)

```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

Due to the miss prediction that had occurred however what we observe is that there is some component of array 2 that is present in the cache now we know note that this component depends on the value of array of X and what we have seen is that we have given a sufficiently large value of array X so that it was actually causing a buffer overflow and appointing inside the secret in other words what has been loaded into the cache at this location is essentially a function off the secret location.

(Refer Slide Time: 11:44)

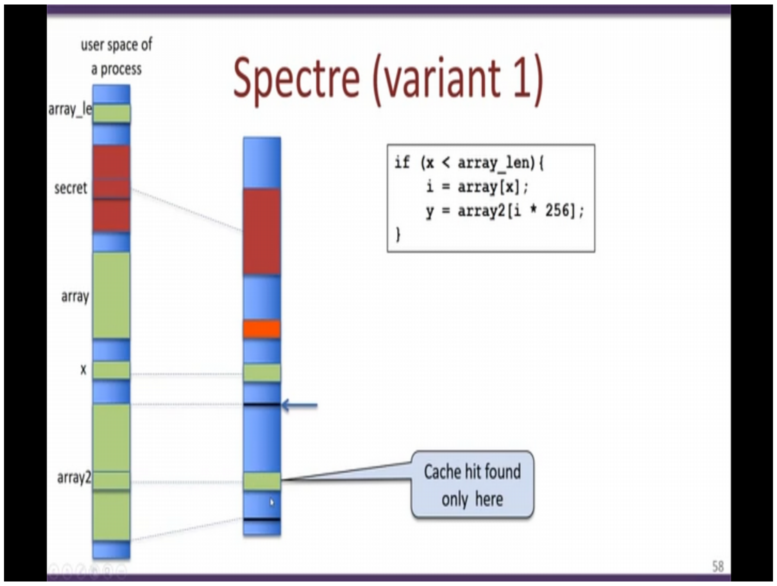


Spectre (variant 1)

```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

The next thing to do for the attacker is to identify which block in array 2 has actually been loaded into the cache what he does for this is that he starts to access every main element in array 2 so he excesses the first element and he figures out that this axis is going to take a long time because the first element is not present in the cache then he would try the second element again he would get a cache miss and therefore along a longer good execution time and this continues.

(Refer Slide Time: 13:59)



Spectre (variant 1)

```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

Over and over again until he finds out that one particular element is taking a shorter time so the shorter time is due to the fact that this element is present in the cache and noticed that this element is in the cache due to the secret of information which has invoked it speculatively and does the attacker would get some information about the contents of the secret, thank you.