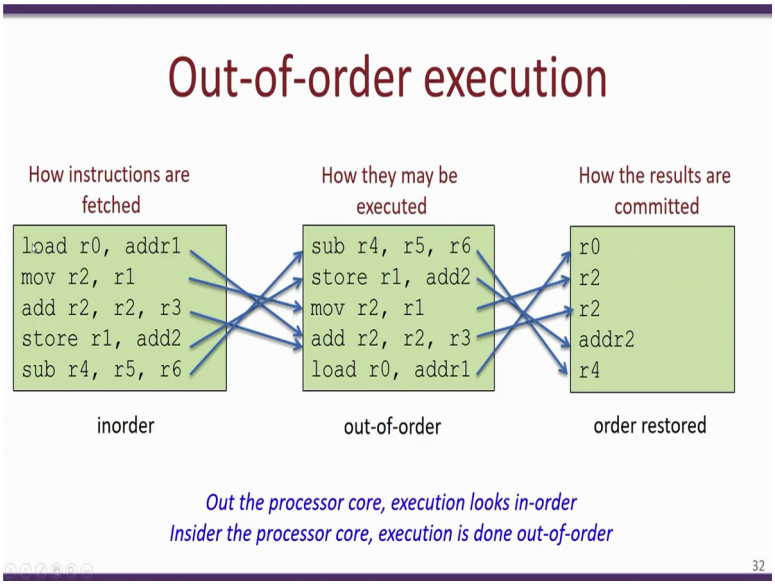
Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Meltdown

Hello and welcome to this lecture in the course for secure systems engineering. So in the previous video lectures we had looked at micro-architectural attacks, we had looked at flush and reload, the prime and probe and other such timing attach which uses the cache memory. In this video lecture we will look at something which is relatively new, so it is known as speculation attacks so these attacks are essentially targeted for Intel like platforms which have certain features, so the features which are exploited are the out of order execution and the speculative execution. Also a variance of these attacks also use the features of the branch predictor which is a common hardware in many of these modern day microprocessors.

So before we go into what these attacks are, so let us have a brief background into what speculation means and what these terms connected to speculation actually do.

(Refer Slide Time: 1:18)



So let us look at these set of instructions, so here we have like there are 5 instructions and this is what the compiler would have actually generated or if a programmer is writing code in assembly he would have written code in this particular way. So what you see here is 5 instructions they are the load, move, add, store and the subtract instruction and all of them work on different set of registers. So there are like the registers r0, r2, r1 and r4 which are actually the target registers for each instruction or in other words these are the registers where the result of that instruction is stored.

In order to actually run this these set of instructions in a correct manner or what is expected is that each of these instructions execute in precisely this order. So firstly load instruction executes, then move, add, store and subtract. However, quite often what has been noticed is that it would be beneficial from a performance perspective if these instructions are reordered. For example since we know that the load instruction actually would access a particular address in this case address 1 from the main memory or from say a cache memory this load instruction would take considerably longer than other instructions like the move and add which are just performed with registers stored within the processor.

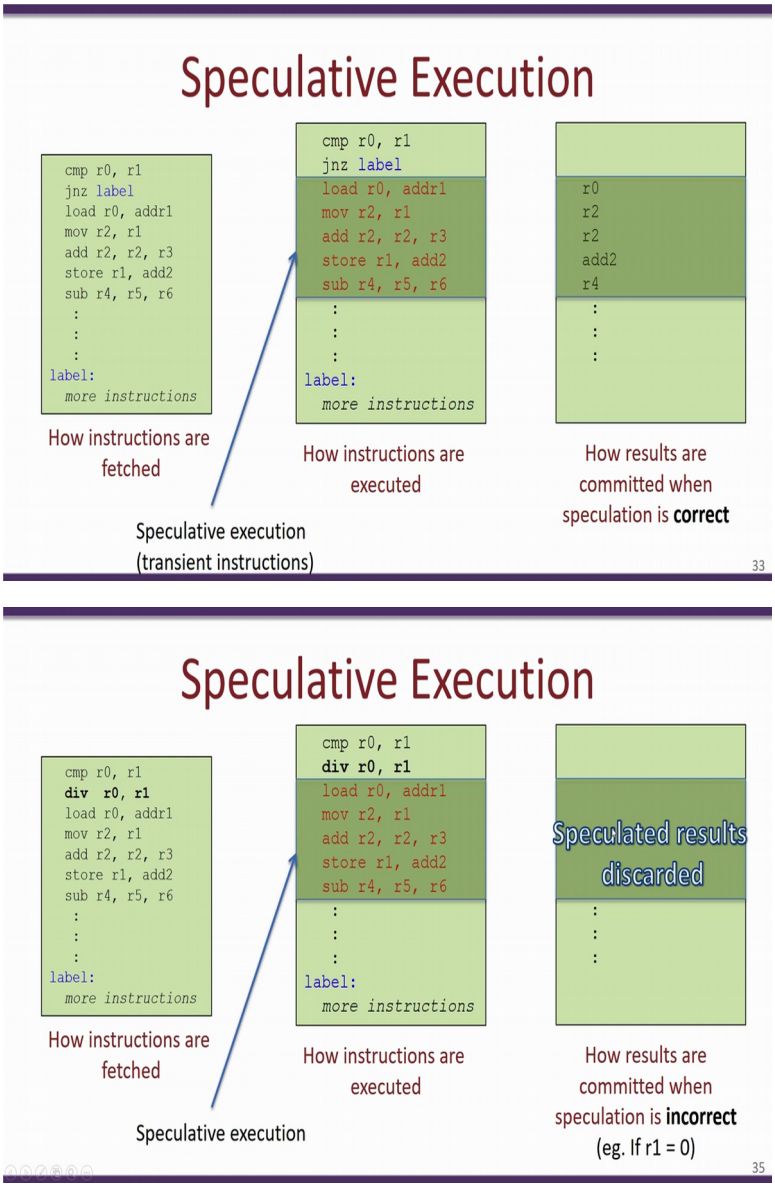
So as long as the arguments in the load instructions and those of these subsequent instructions are independent it would be possible for the processor to actually execute these instructions in an order which is quite different from what is specified in the program. In other words the processor would still be able to get the correct result with even executing these instructions in an out-of-order manner.

In the second figure what we show is what happens inside the processor. So you see that the instructions are actually executed out of order, the subtract instruction in fact is executed first, then we have the store instruction and so on. So what we notice over here is that eventually since all of these instructions are independent of each other the ordering of these instructions will not matter, what does matter eventually and what is done in the processor is at the end of execution the results are actually committed in order.

So you look at this set of results or the results of these instructions and what you notice is that the results correspond to the original ordering of these instructions. So for example r0 was the destination for this load that is the contents corresponding to this memory address was loaded into r0 and similarly r0 was the first result to be return back. So essentially what we see in this particular slide is that even though a program is return in a particular manner it would could be executed in any manner which we known as out of order, but eventually the order is restored by the processor so that the results or the registers return back would match the original expectation for the program.

So what we observe here is that even though the we have an out of order execution the result of the program will be exactly the same.

(Refer Slide Time: 5:09)



Another feature which is popular in many modern day microprocessors is speculative execution. So speculative execution is used essentially to improve the performance of the processor, so let us actually take a small example and we have a set of instructions as before and one important instruction that is important for this example is this this is a jump on no 0 to a label.

So what happens over here is that depending on this comparison instruction if r0 is equal to r1, then the 0 flag is set within the processor and as a result this jump on no 0 would not cause a jump and the program will continue to execute. On the other hand if r0 is not equal to r1 then the 0 flag is reset or in other words the 0 flag is equal to 0 and this particular check would cause the program counter to jump to this label and start to execute from here, or in

other words what we see over here is a value of 0 flag set to 1 would cause these instructions to be executed.

On the other hand if the 0 flag has a value of 0 then (there was a) there is a jump which takes place and the instructions following the label would execute. Now the problem with this thing comes from a performance perspective. So whenever there is a jump like this what would happen internally in a processor is that the entire pipeline would get flushed and new instructions corresponding to these instructions following the label would get fetched from the memory.

Now advanced or very popular recent microprocessors have a considerably large pipeline and as a result there will be several hundred or so instructions which may be present in the pipeline. So every time there is a branch like this to another memory location, this entire pipeline would get flushed and new instructions would need to be fetched from the target memory. So this could cause quite a considerable performance overhead. So in order to actually reduce these performance overheads what microprocessors do is they speculate about the result of a jump instruction.

For example the processor would speculate that the instructions following this jump would be the consecutive instructions and therefore it will start to fetch these instructions from the memory and start to execute them in a speculative manner, what this means is that the results of these instructions are not committed unless and until the result of this jump instruction is known.

So let us consider this particular case, so first of all within the processor the instructions that is following these jump on no 0 would get fetched from the memory and it will be speculatively executed. Now when we actually execute this instruction or compare r0, comma r1 and let us assume that r0 is equal to r1 as a result the 0 flag is set. Now in such a case jump on no 0 label so this particular instruction would fail and the instruction that follows needs to be executed.

But what happens within the processor is that these instructions are already speculatively executed and the only thing that is required to be done is that the results of these instructions should be committed. So what the processor does is that once this the result of this jump on no 0 is obtained in this case it means that the speculation is correct, the processor would only

commit the results and the results corresponding to the various registers r0, r2 address 2 and r4 would be actually committed.

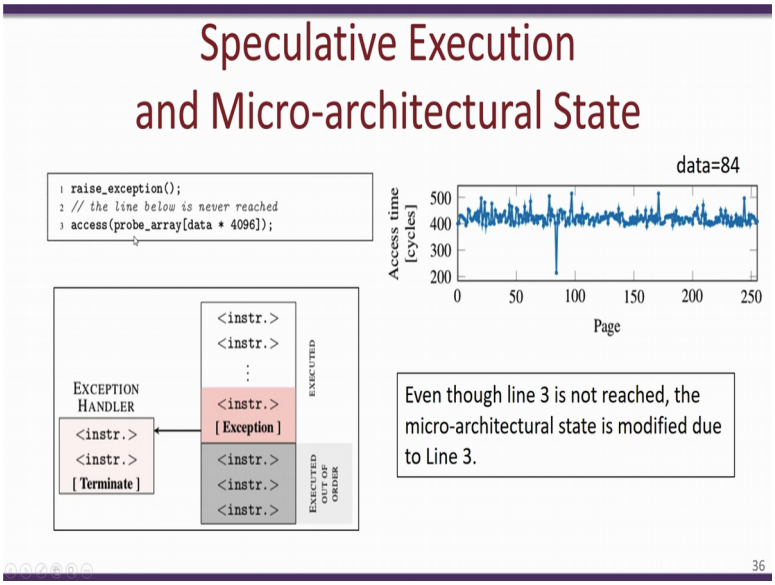
So what we actually achieve over here is that even before completing the execution of this compare and jump on no zero instructions the processor could have actually speculatively executed these set of instructions and then commit these instructions only when the result of compare and jump on no 0 is obtained. Now the thing which one would ask is this would work fine for the specific example when r0 is equal to r1, but what would happen when r0 is not equal to r1, well in such a case when r0 is not equal to r1 we know that the 0 flag would be set to zero and the jump on no 0 to label would result in the execution being transferred to these instruction that is following the label.

So in this condition 2 since the these instructions following have been speculatively executed the processor would realize that in fact this speculation was wrong and these instructions were actually not to be executed. As a result the speculated results are discarded and execution continues from the instructions following the label. So what we see is that the speculation could possibly improve the performance of the program. So when the processor as executed correctly in this case when r0 is equal to r1 then a performance is gained because the speculated instructions could actually be committed at a much more faster rate.

On the other hand when the speculation is wrong as in this case r0 not equal to r1 then all the speculated (instructions) result should be discarded and there would be a performance overhead, all modern processes try to speculate correctly in order to maximize their performance. Another case where speculation is also observed is in something like this way, here what we have done is that we have replaced this jump on no 0 label instruction with a divided r0 by r1.

Now as we know there are a couple of things which are going on, one is the out of order and also the speculation and what happens is that even before this divide gets executed it may be likely that the instructions that follow this divide may be speculatively executed. So there would be a problem that would occur if r1 happens to be equal to 0, in such a case we know that a divide by zero exception would be thrown and as a result the processor would need to discard the speculated execution. So as a result if r1 is equal to 0 all the speculatively executed instructions would need to be discarded.

(Refer Slide Time: 12:34)



In January 2018, what was shown was that this out of order and speculative execution can be actually used to gain secret information out of the processor. So let us consider this small snippet of code there are in fact 3 instructions which are present, one is a raise exception instruction so this for example could be a divide r0, comma r1 and the case where r1 is equal to 0 then we have a memory access to an array at a location data times 4096.

If we evaluate these two instructions in a normal operation what would happen is that due to the raise exception this memory access to the probe array would never occur. However, due to speculation and if we consider what happens within the processor due to speculation out of order execution, the probe array at the location data into 4096 maybe speculatively executed and when the risk (exception) instruction is actually executed the results corresponding to probe array data into 4096 would be actually discarded.

Now what these new attacks actually showed was that even though the processor discussed the result due to speculation in such a case the speculative execution has an effect on the state of the processor, essentially due to this speculative execution of this memory axis or the state of the processor may be in the cache memories or the branch predictors or so on may be modified corresponding to the data which gets accessed.

So this particular figure shows how this program executes so we have a set of instructions that gets executed and then there is an exception and what happens when these actually gets executed in the processor is that many of these instructions will actually be executed

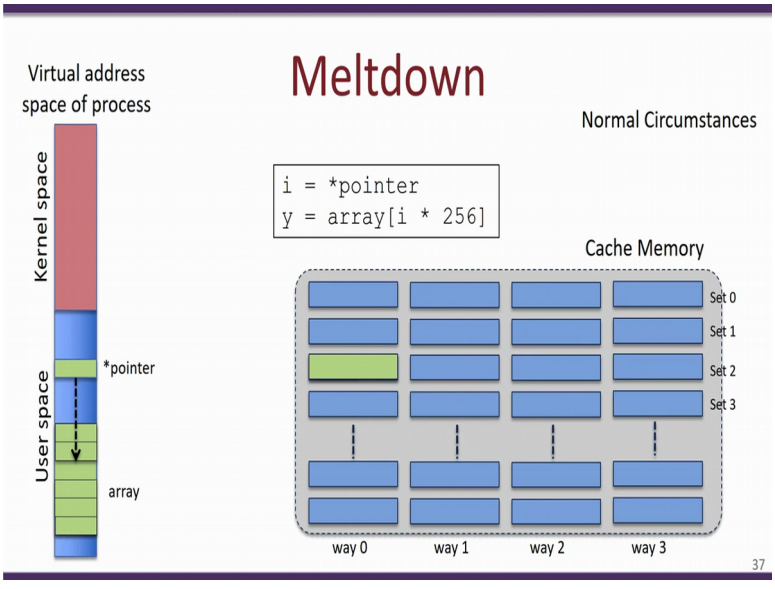
speculatively and out of order. So it may happen that these instructions without these instructions following the exception may be speculatively executed.

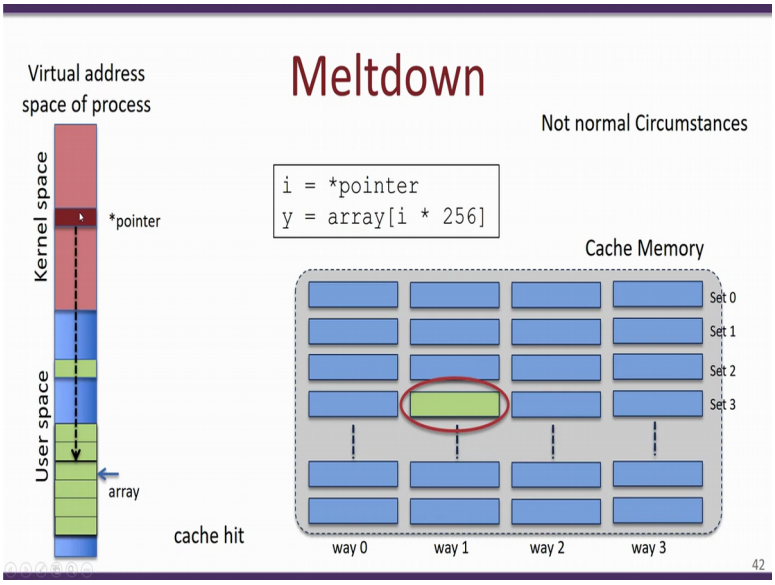
However, due to the exception that is present over here the results of the speculatively executed instructions would be discarded. However, what is seen is that these speculatively executed instructions may have a signature inside the memory axis. So for example what would happen here is that there would be your memory axis which is done speculatively and data corresponding to probe array data into 4096 would be loaded into the cache memory from the lower size of the memory.

So what has been shown in this particular figure is if you look at page numbers and on the x-axis and the memory access time essentially make this memory access to probe underscore array what we see is that the memory access time due to the speculative execution may be a bit lower corresponding to the value of data. So over here for example the data has a value of 84 and therefore at the 84th page what is observed is that there is much lesser memory access time.

The takeaway from this particular slide is the fact that even though this line 3 in this particular program has never been executed due to this exception that is raised in line 1, nevertheless the micro-architectural state of the processor is modified by this third line by this memory access.

(Refer Slide Time: 16:35)





So let us take a look at meltdown, so this was an attack which was discovered in January 2018 and it showed how we could actually read secret data from say parts of the kernel space. So before we go into how the attack actually works, we would have to look at the virtual address space of a process. So the virtual address space as we have seen in the previous lectures comprises of two components, so it comprises of a user space where your typical code stack heap and other data segments are present and above a particular memory location you have the kernel space.

So the various protection mechanisms like the ring 0, ring 1, ring 2 and ring 3 guarantee that when you are running in user mode a program (could) can only observe the user space part of the process. On the other hand when a system call is invoked or you are running in the kernel space or in the operating system the entire virtual address space including that of the kernel space plus that of the user space is observable, typically in a 32-bit Linux kernel this boundary is present at a location 0XC0000000, any memory address above this particular location corresponds to a kernel space and any memory address below (that) this location corresponds to that of a user space.

Now what meltdown showed is that with a simple attack exploiting that features of what speculation actually provides a user space program would be able to read contents of the kernel space. In other words a user space would be able to read code and data present in the kernel space. So the attack as such is quite simple the attacker we assume is running in the user space and has these two lines written in the program, the first line i equal to star pointer corresponds to something like this we have a pointer variable over here and let us assume that this point of variable is pointing to a location say this.

Now in the second statement an array which is present over here is accessed at a location i into 256. Therefore, the end result of these two instructions is that corresponding to this pointer one element of the array is loaded into this variable called i, so due to this particular load what we know due to the processor architecture is that the contents of the array corresponding to i into 256 would be loaded from the main memory into the various caches and would also get loaded into one of the general purpose registers present in the processor.

So it would look something like this, when the second statement gets executed a particular block corresponding to array of i into 256 would be copied from the DRAM into the cache memory and also be copied into the corresponding general purpose register, now this is what happens during the normal operation of the program. Now let us say that we craft a pointer which is pointing to the kernel space.

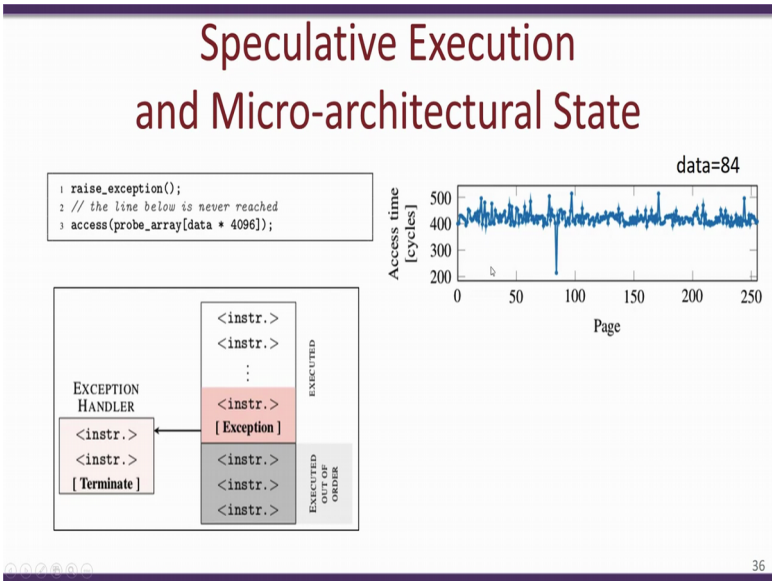
So now what we are trying to do in the first statement is load some contents (of) from the kernel space into this variable called i, so as we know that the kernel space is not accessible from a user level program, now this would result in an exception to be thrown and the program would terminate. So however due to the out of order and speculative execution of the processor the second statement would be speculatively executed. As a result we would have this statement reading the value of this memory location into i and speculatively accessing array of i into 256.

Thus, the array would be accessed at a location which is dependent on the data which is present in this kernel space memory location. Thus, as we seen before one block of data present in array would be loaded into the cache. Now important for us to observer over here is the set which gets accessed. Now depending on the value of the memories present in this kernel space memory location since this memory location is used to index into the array the indexed location may access one of these multiple sets.

So what typically would happen is that if an attacker is able to determine which of these sets has been accessed during the second operation the attacker would then be able to gain some information about the value of i. So in order to do this what the attacker would do is it would use something like the prime and probe what the attacker would do in order to find out which cache set was actually used previously, the attacker would start to access every element in the array notice. Notice that the first element in the table is not present in the cache and as a result the memory access time would be large due to the cache misses.

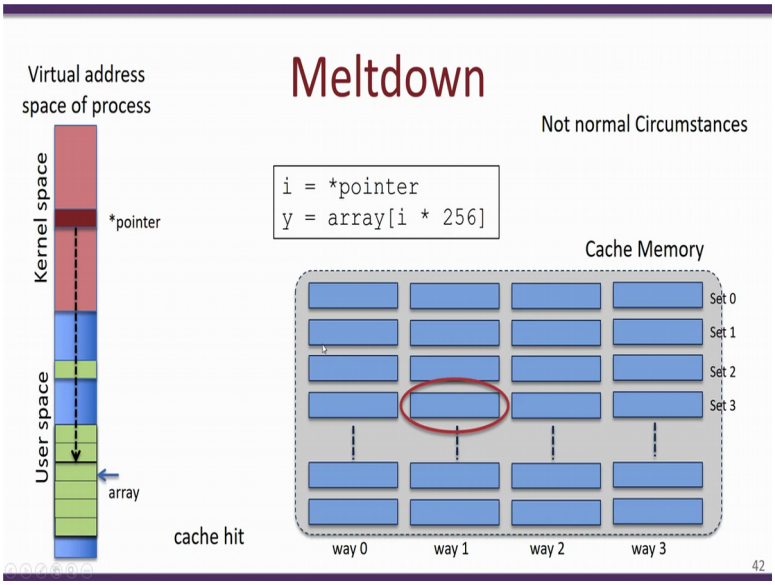
Similarly if the attacker accesses the second element it would also result in a cache miss because it is not available in the cache. On the other hand when the attacker finally accesses this specific memory location it would obtain a cache hit due to the fact that this particular element is present in the cache. Now the attacker would use this lower execution time for memory access of this particular location, identify some information about the value of i and therefore be able to determine some information about what was present in this specific memory location in the kernel space.

(Refer Slide Time: 22:56)



So if you go back and look at this particular slide what we see is that when i has a value of 84 it shows a execution time which is considerably lower compared to all other memory accesses to that array. Thus, the attacker would know that the value of i in other words the contents of that kernel space memory has a value of 84.

(Refer Slide Time: 23:20)



So this simple example showed how one attacker could use the features of an Intel processor to be able to read kernel space code or data just by manipulating these two statements and forcing the processor to speculatively execute and then try to identify what was actually speculatively executed by evaluating the memory access time, thank you.