CSE 410/510 Special Topics: Software Security

Instructor: Dr. Ziming Zhao

Location: Obrian 109

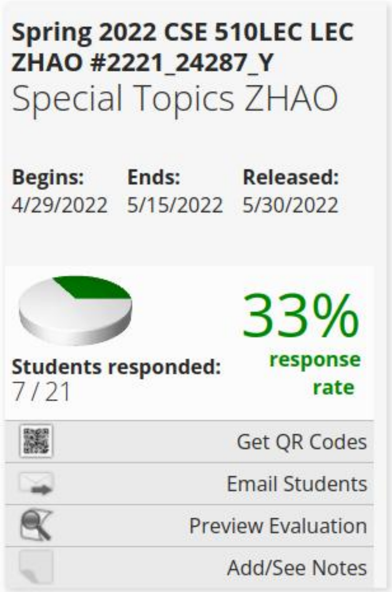
Time: Monday, Wednesday 5:00PM-6:20PM

Course Evaluation

Begins: 4/29/2022
Ends: 5/15/2022

If 90% of student submit the evaluation, all of the class will get **10** bonus points.

39 students. So 35 **evaluations**!!

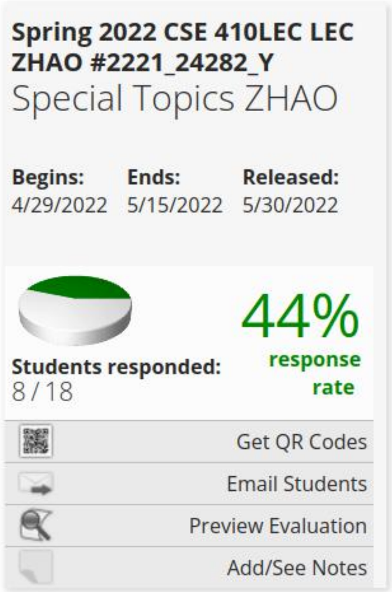


Spring 2022 CSE 510LEC LEC ZHAO #2221_24287_Y
Special Topics ZHAO

Begins:	Ends:	Released:
4/29/2022	5/15/2022	5/30/2022

Students responded: 7 / 21

33% response rate



Spring 2022 CSE 410LEC LEC ZHAO #2221_24282_Y
Special Topics ZHAO

Begins:	Ends:	Released:
4/29/2022	5/15/2022	5/30/2022

Students responded: 8 / 18

44% response rate

Final CTFs

5/16/2022. **Must be in-person**.
7:15PM - 10:15PM and 30 mins extra
3+0.5 hours in total.
200 points in total.
There will be no written final exam.

1 ROP challenge
1 Heap exploitation challenge
2 Format string challenges
1 Stack buffer overflow + ?

Meltdown and Spectre

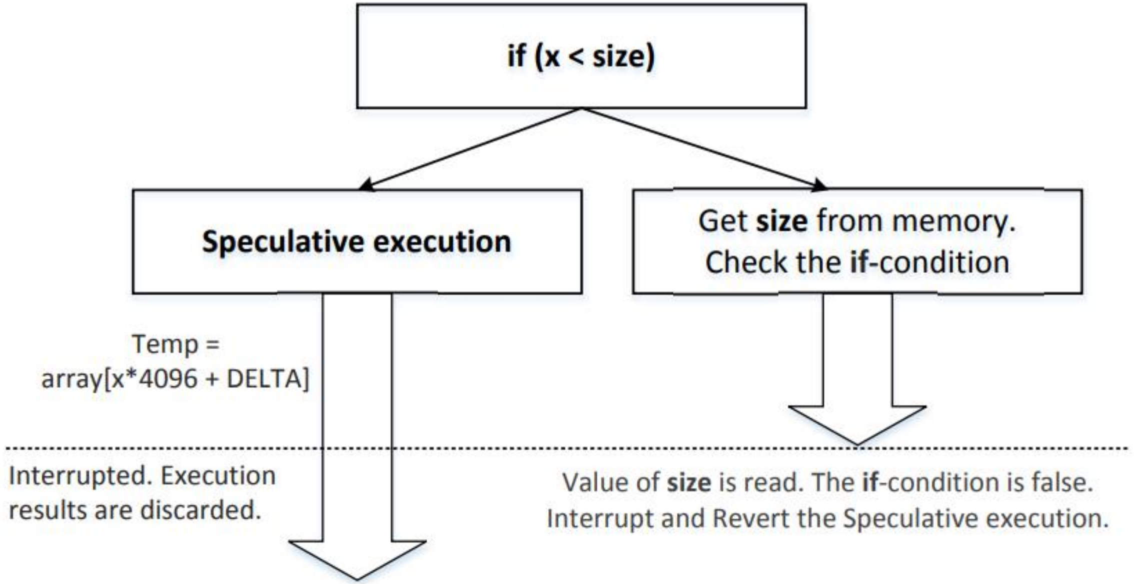
https://meltdownattack.com/

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754

Slides from SEED project and Jake Williams

More examples on Out-of-order execution

```
data = 0;
if (x < size)
        {
        data = data + 5;
        }
```


if (x < size)
Speculative execution
Get size from memory. Check the if-condition
Temp = array[x*4096 + DELTA]
Interrupted. Execution results are discarded.
Value of size is read. The if-condition is false. Interrupt and Revert the Speculative execution.

From out-of-order execution to speculative execution

The ability to issue instructions past branches that are yet to resolve is known as speculative execution.

The processor can preserve its current register state, make a prediction as to the path that the program will follow, and speculatively execute instructions along the path.

If the prediction turns out to be correct, the results of the speculative execution are committed (i.e., saved), yielding a performance advantage over idling during the wait.

Otherwise, when the processor determines that it followed the wrong path, it abandons the work it performed speculatively by reverting its register state and resuming along the correct path.

Speculative Execution

Speculative execution on modern CPUs can run several hundred instructions ahead.

Speculative execution is an optimization technique where a computer system performs some task that may not be needed.

Work is done before it is known whether it is actually needed, so as to prevent a delay that would have to be incurred by doing the work after it is known that it is needed.

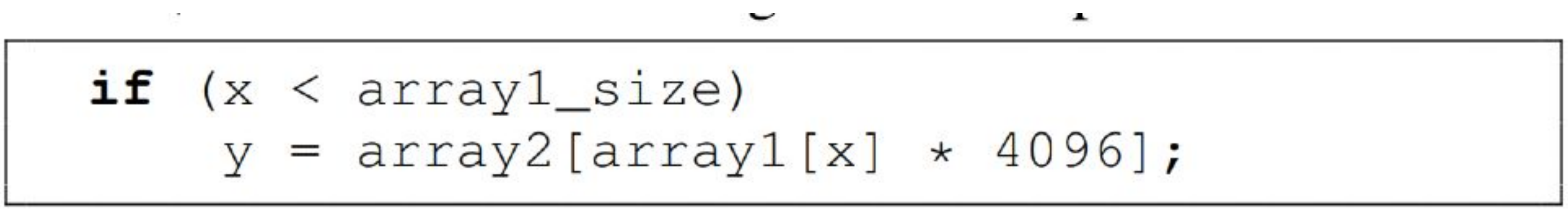
Branch Prediction

During speculative execution, the processor makes guesses as to the likely outcome of branch instructions.

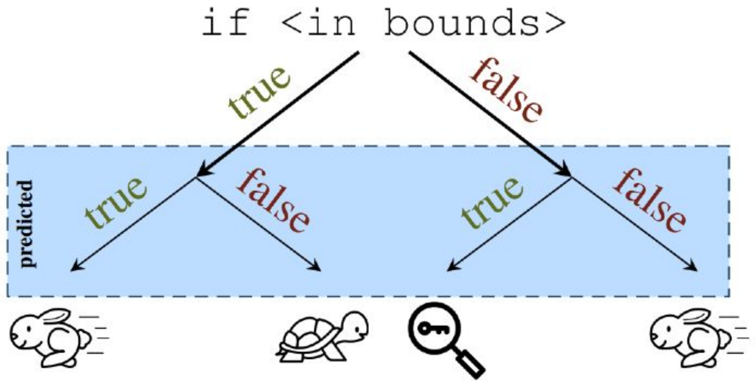
The branch predictors of modern Intel processors, e.g., Haswell Xeon processors, have multiple prediction mechanisms for direct and indirect branches.

Spectre V1

Conditional branch misprediction



```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```



Spectre V2

Indirect branches can be poisoned by an attacker and the resulting misprediction of indirect branches can be exploited to read arbitrary memory from another context.

Spectre vs. Meltdown

Meltdown does not use branch prediction. Instead, it relies on the observation that when an instruction causes a trap, following instructions are executed out-of-order before being terminated.

Second, Meltdown exploits a vulnerability specific to many Intel and some ARM processors which allows certain speculatively executed instructions to bypass memory protection.

Meltdown accesses kernel memory from user space. This access causes a trap, but before the trap is issued, the instructions that follow the access leak the contents of the accessed memory through a cache covert channel.

A design flaw leads to Spectre

Even though registers and memory will be reverted back to the original state if the speculative execution is discarded, the cache will not be reverted.

Listing 3: `SpectreExperiment.c`

```
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

int size = 10;
uint8_t array[256*4096];
uint8_t temp = 0;

void victim(size_t x)
{
  if (x < size) {                                ①
     temp = array[x * 4096 + DELTA];             ②
  }
}

int main()
{
  int i;

  // FLUSH the probing array
  flushSideChannel();

  // Train the CPU to take the true branch inside victim()
  for (i = 0; i < 10; i++) {                     ③
      victim(i);                                 ④
  }

  // Exploit the out-of-order execution
  _mm_clflush(&size);                            ☆
  for (i = 0; i < 256; i++)
      _mm_clflush(&array[i*4096 + DELTA]);
  victim(97);                                    ⑤

  // RELOAD the probing array
  reloadSideChannel();
  return (0);
}
```