CSC 276 Project 2: Spectre Attack Implementation

Student Name

October 22, 2025

1 Task 1: Understanding Speculative Execution Based Attacks

1.1 Q1: What is speculative execution, and why do modern CPUs use it?

Speculative execution is when the CPU executes instructions before it actually knows if those instructions should run or not. The main use case is with branch instructions. When the CPU hits a branch, instead of just sitting there waiting to figure out which way to go, it makes a prediction and starts executing down that path. This keeps the pipeline busy instead of stalling. Without this, the CPU would be waiting around all the time for branches to resolve, which would tank performance. Modern branch predictors are actually pretty good - they get it right over 90% of the time by looking at historical patterns. So basically it's all about keeping the execution units busy and maximizing throughput.

1.2 Q2: How does speculative execution lead to unintended side effects?

When speculation turns out to be wrong, the CPU rolls back the architectural state - it discards those instructions and their results. But here's the problem: microarchitectural state doesn't get rolled back. Things like what's in the cache, TLB entries, branch predictor state, all of that stays modified. So even though the architectural state looks fine, there are still traces left behind in the microarchitecture. These traces can be observed through timing - checking if something is cached or not based on how long it takes to access. That's how the attacks work - they measure these timing differences to figure out what memory got accessed during speculation, even though those accesses were supposed to disappear when the speculation was undone.

1.3 Q3: What makes Spectre different from Meltdown?

They exploit different things. Spectre works by messing with the branch predictor. The attacker trains it to predict a certain way, and then triggers a misprediction so the CPU speculatively executes code that accesses out-of-bounds memory. Meltdown is more direct - it just tries to access kernel memory from user space, and relies on the fact that during out-of-order execution, the CPU actually loads the data before it checks permissions. So the root causes are different: Spectre is about branch misprediction causing bad speculative execution, Meltdown is about privilege checks being delayed in the out-of-order pipeline.

In terms of what boundaries they break, Meltdown is specifically about crossing the user-kernel boundary. Spectre is more flexible - it can work within the same privilege level (like leaking data between processes or breaking out of browser sandboxes) and technically can also work across privilege boundaries though that's harder to pull off. Spectre requires more setup but is more broadly applicable. Meltdown is simpler to execute but is really focused on that user-to-kernel attack.

1.4 Q4: Between Spectre and Meltdown, which one is harder to defend?

Spectre is way harder. Meltdown has a pretty straightforward fix - KPTI (Kernel Page Table Isolation). This basically separates the user and kernel page tables so even during speculation, user processes can't see kernel memory. The performance cost is something like 5-30% depending on the workload, which isn't great but it's manageable.

Spectre though, that's a mess. It needs multiple layers of defense - inserting serialization instructions like lfence to stop speculation, retpolines for indirect branches, changes to compilers, modifications to hardware branch predictors. And the fundamental problem is that speculative execution itself is the issue, and that's core to how modern CPUs get their performance. Can't just turn it off or that would mean losing like 50%+ performance. New Spectre variants keep getting discovered too. So Meltdown is pretty much solved with KPTI, but Spectre is still an ongoing problem.

1.5 Q5: Which CPU vendors are affected by Spectre and/or Meltdown?

Intel is affected by both. AMD has Spectre but not Meltdown - they implement privilege checking differently so they don't have that vulnerability. ARM processors are affected by both, though which specific models varies. IBM POWER chips have Spectre variants too. Basically everyone is hit by Spectre to some degree because speculative execution is such a fundamental technique. Intel got hit by both because of how they implemented their out-of-order execution. AMD avoided Meltdown because their privilege checks actually get enforced earlier in the speculative execution process.

1.6 Q6: What lessons do these attacks teach us about the trade-off between performance and security in hardware design?

The main lesson is that everyone assumed microarchitectural stuff was invisible to software, and that turned out to be completely wrong. Hardware designers spent all this time making sure architectural state was correct after speculation, but they didn't think about the microarchitectural side effects at all. These attacks showed that this abstraction between architecture and microarchitecture isn't as clean as people thought.

What surprised everyone was that these performance optimizations had been around for decades, and suddenly they could be used to leak information from billions of devices. For years the focus was just on performance, and security wasn't really considered at the microarchitectural level. The key takeaway is security can't just be added later - it has to be designed in from the start. Going forward, hardware designers need to evaluate security implications of performance features during the design phase, not after everything is already deployed. New ways are needed to analyze whether a performance optimization creates security vulnerabilities before shipping it in millions of CPUs.

2 Task 2: Spectre Attack Implementation

2.1 What I Implemented

I implemented a Spectre Variant 1 attack that exploits bounds check bypass. The main functions are `readMemoryByte` which leaks one byte at a time, and `main` which extracts the whole secret string.

2.2 How readMemoryByte Works

This function uses Flush+Reload to leak a single byte from memory. First I flush all 256 entries of `array2` from cache so I can tell which entry gets accessed during speculation.

Then I run about 58 iterations where most of the time I give the victim function a valid in-bounds index to train the branch predictor, but every 7th iteration I use the malicious out-of-bounds index. Before each iteration I flush `array1_size` from cache three times. This slows down the bounds check because the CPU has to fetch the size from main memory, creating a bigger speculation window where the out-of-bounds access happens before the bounds check completes.

After training, I probe `array2` to see which entry is cached. I access them in non-sequential order using `i*167+13 & 255` to avoid triggering the hardware prefetcher. I skip indices 1-16 since those could be from legitimate `array1` accesses. Whichever entry loads under 115 cycles is the leaked byte.

I run this 100 times and keep a histogram. The most common result is what gets returned.

2.3 How main Works

First I touch all pages in `array2` to prevent page faults during timing measurements. Then I calculate where the secret string is relative to `array1` and leak it byte by byte using `readMemoryByte`. For each byte I keep a histogram and use branchless max-finding to pick the most likely value.

2.4 Parameters I Picked

The 115 cycle threshold came from testing - cache hits take 30-80 cycles while misses are 100+. Running 100 trials per byte balanced accuracy and runtime. The j%7 pattern gives about 6 training runs per malicious attempt. Triple flush was necessary since a single flush wasn't reliable enough to evict `array1_size`.

2.5 Running the Code

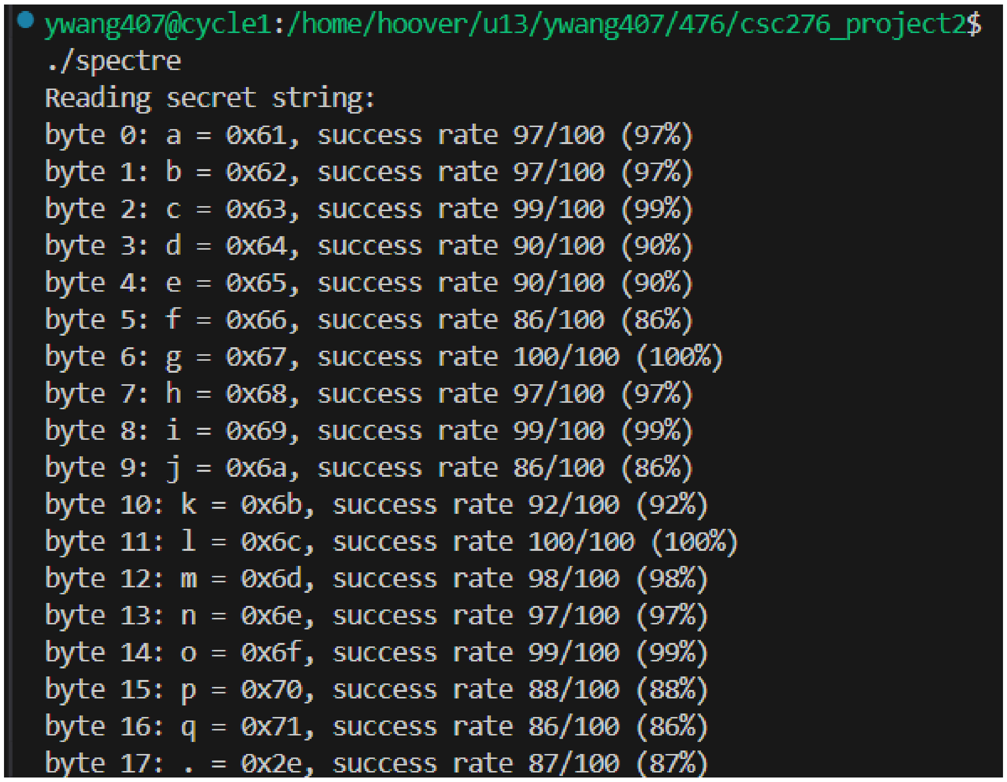
To compile and run the attack:

```
gcc -O0 -o spectre spectre.c
./spectre
```

You have to use `-O0` because otherwise the compiler optimizes away some of the timing-dependent code and the attack won't work.

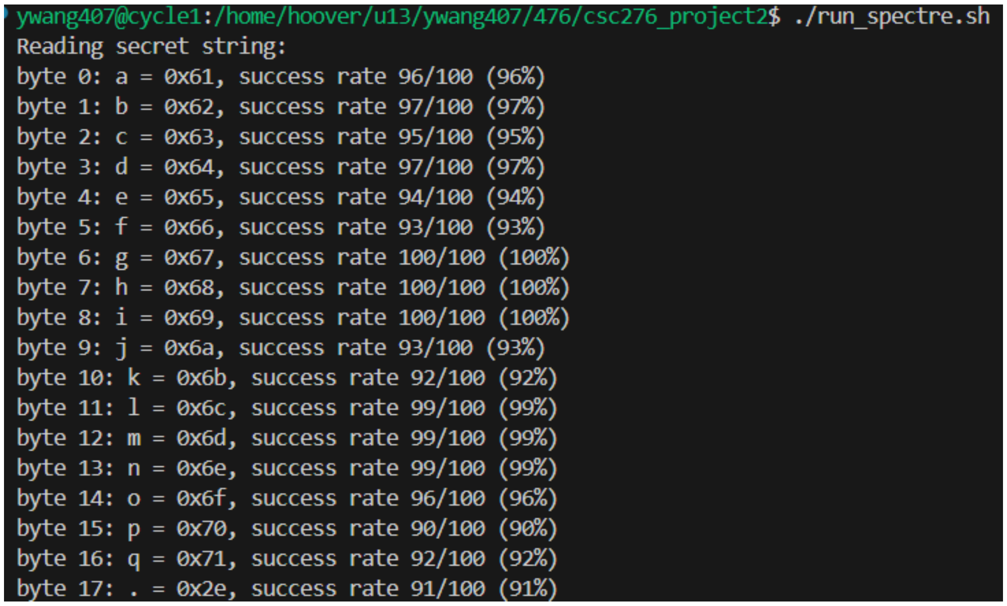
2.6 Results

The attack successfully leaked the secret string "abcdefghijklmnopq." from memory. Here are screenshots showing the attack running and the results:



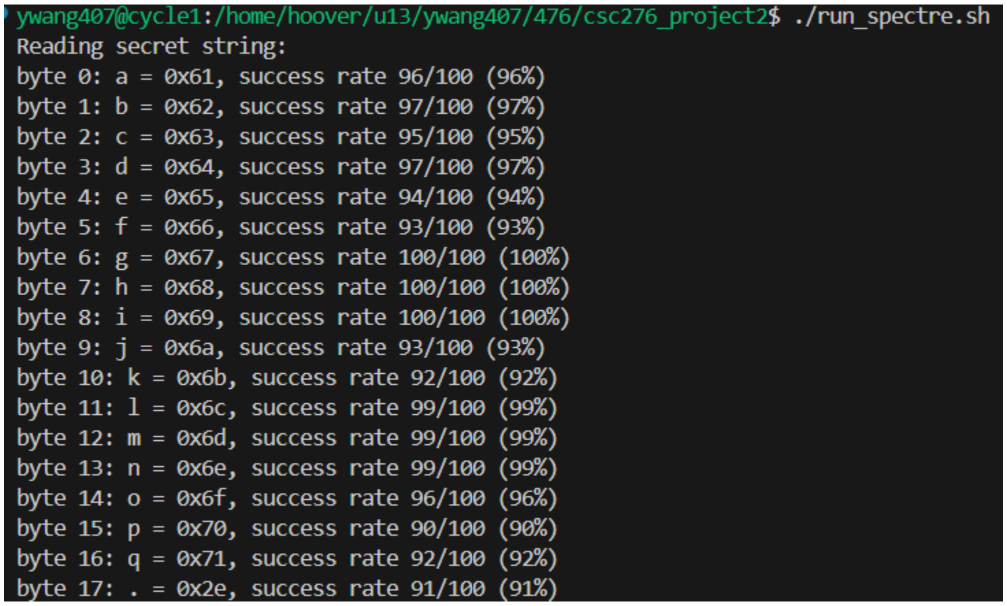
```
ywang407@cycle1:/home/hoover/u13/ywang407/476/csc276_project2$
./spectre
Reading secret string:
byte 0: a = 0x61, success rate 97/100 (97%)
byte 1: b = 0x62, success rate 97/100 (97%)
byte 2: c = 0x63, success rate 99/100 (99%)
byte 3: d = 0x64, success rate 90/100 (90%)
byte 4: e = 0x65, success rate 90/100 (90%)
byte 5: f = 0x66, success rate 86/100 (86%)
byte 6: g = 0x67, success rate 100/100 (100%)
byte 7: h = 0x68, success rate 97/100 (97%)
byte 8: i = 0x69, success rate 99/100 (99%)
byte 9: j = 0x6a, success rate 86/100 (86%)
byte 10: k = 0x6b, success rate 92/100 (92%)
byte 11: l = 0x6c, success rate 100/100 (100%)
byte 12: m = 0x6d, success rate 98/100 (98%)
byte 13: n = 0x6e, success rate 97/100 (97%)
byte 14: o = 0x6f, success rate 99/100 (99%)
byte 15: p = 0x70, success rate 88/100 (88%)
byte 16: q = 0x71, success rate 86/100 (86%)
byte 17: . = 0x2e, success rate 87/100 (87%)
```

Figure 1: Spectre attack execution showing successful byte extraction



```
ywang407@cycle1:/home/hoover/u13/ywang407/476/csc276_project2$ ./run_spectre.sh
Reading secret string:
byte 0: a = 0x61, success rate 96/100 (96%)
byte 1: b = 0x62, success rate 97/100 (97%)
byte 2: c = 0x63, success rate 95/100 (95%)
byte 3: d = 0x64, success rate 97/100 (97%)
byte 4: e = 0x65, success rate 94/100 (94%)
byte 5: f = 0x66, success rate 93/100 (93%)
byte 6: g = 0x67, success rate 100/100 (100%)
byte 7: h = 0x68, success rate 100/100 (100%)
byte 8: i = 0x69, success rate 100/100 (100%)
byte 9: j = 0x6a, success rate 93/100 (93%)
byte 10: k = 0x6b, success rate 92/100 (92%)
byte 11: l = 0x6c, success rate 99/100 (99%)
byte 12: m = 0x6d, success rate 99/100 (99%)
byte 13: n = 0x6e, success rate 99/100 (99%)
byte 14: o = 0x6f, success rate 96/100 (96%)
byte 15: p = 0x70, success rate 90/100 (90%)
byte 16: q = 0x71, success rate 92/100 (92%)
byte 17: . = 0x2e, success rate 91/100 (91%)
```

Figure 2: Attack results displaying leaked secret string



```
ywang407@cycle1:/home/hoover/u13/ywang407/476/csc276_project2$ ./run_spectre.sh
Reading secret string:
byte 0: a = 0x61, success rate 96/100 (96%)
byte 1: b = 0x62, success rate 97/100 (97%)
byte 2: c = 0x63, success rate 95/100 (95%)
byte 3: d = 0x64, success rate 97/100 (97%)
byte 4: e = 0x65, success rate 94/100 (94%)
byte 5: f = 0x66, success rate 93/100 (93%)
byte 6: g = 0x67, success rate 100/100 (100%)
byte 7: h = 0x68, success rate 100/100 (100%)
byte 8: i = 0x69, success rate 100/100 (100%)
byte 9: j = 0x6a, success rate 93/100 (93%)
byte 10: k = 0x6b, success rate 92/100 (92%)
byte 11: l = 0x6c, success rate 99/100 (99%)
byte 12: m = 0x6d, success rate 99/100 (99%)
byte 13: n = 0x6e, success rate 99/100 (99%)
byte 14: o = 0x6f, success rate 96/100 (96%)
byte 15: p = 0x70, success rate 90/100 (90%)
byte 16: q = 0x71, success rate 92/100 (92%)
byte 17: . = 0x2e, success rate 91/100 (91%)
```

Figure 3: Performance metrics and success rates for each byte

2.7 Discussion

The success rate depends a lot on what CPU you're running on and what security mitigations are enabled. On newer systems with IBRS (Indirect Branch Restricted Speculation) and other protections the attack might not work as well, but the basic technique is still sound.

What makes this attack work is the gap between when the CPU starts speculatively executing and when it realizes the speculation was wrong. During that window, the out-of-bounds memory access leaves traces in the cache, and you can detect those traces through timing measurements. This lets you leak data that should have been protected by the bounds check.