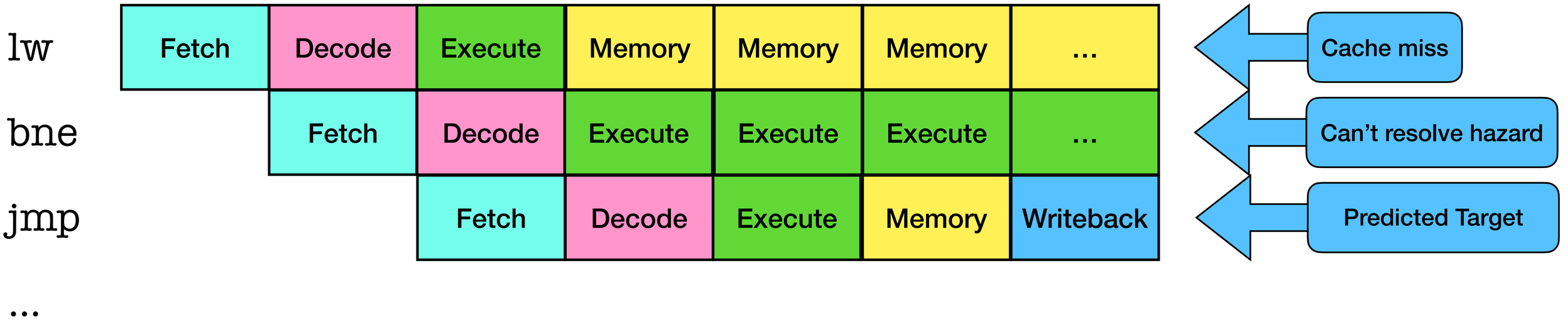
Speculative Execution Attacks (Part 2)

No class Friday;
Friday Colloquium on Zoom;
Check In 6 next Monday

Outline

- Revisiting the Spectre Attack setup
- Improving the Spectre Attack
- Mitigations

(From Friday) Slow to Resolve Branches



Problem! We don't want to writeback the state of an instruction if we don't know that this instruction was the right one to execute!

Speculative processors have a temporary buffer where data is written back to called the "Reorder Buffer" (ROB), which decouples instruction *writeback* from *commit*

(From Friday) Unintended Behavior from “Safe” Code

Basically a buffer overflow vulnerability!

```
void fn(x) {
   if (x < size) {
      y = array[x];
   } else ...
}
```

If the branch predictor will guess that the branch is *not taken* then the “`if`” condition will execute speculatively

An adversary can pass an input value `x` where `x > size` which will lead to a `load` from an arbitrary location in memory… bad!

Is this dangerous? From our cache side channels, all that the adversary can leak is that an address exists in the cache, not its data…

Components of a Speculative Execution Attack

- Processors predict the next instructions to execute, and incorrect predictions need to be rolled back
- Branch predictors track the behavior of programs dynamically at runtime, and can therefore be adversarially trained for what “expected” input behaviors could look like
- If a branch instruction depends on data from memory and the memory system is slow, then lots of execution will be performed speculatively while waiting for the branch to be resolved
- Once the unintended code has executed, a side channel needs to exist for the adversary to leak out the state of the data

Chat with your neighbor(s)!

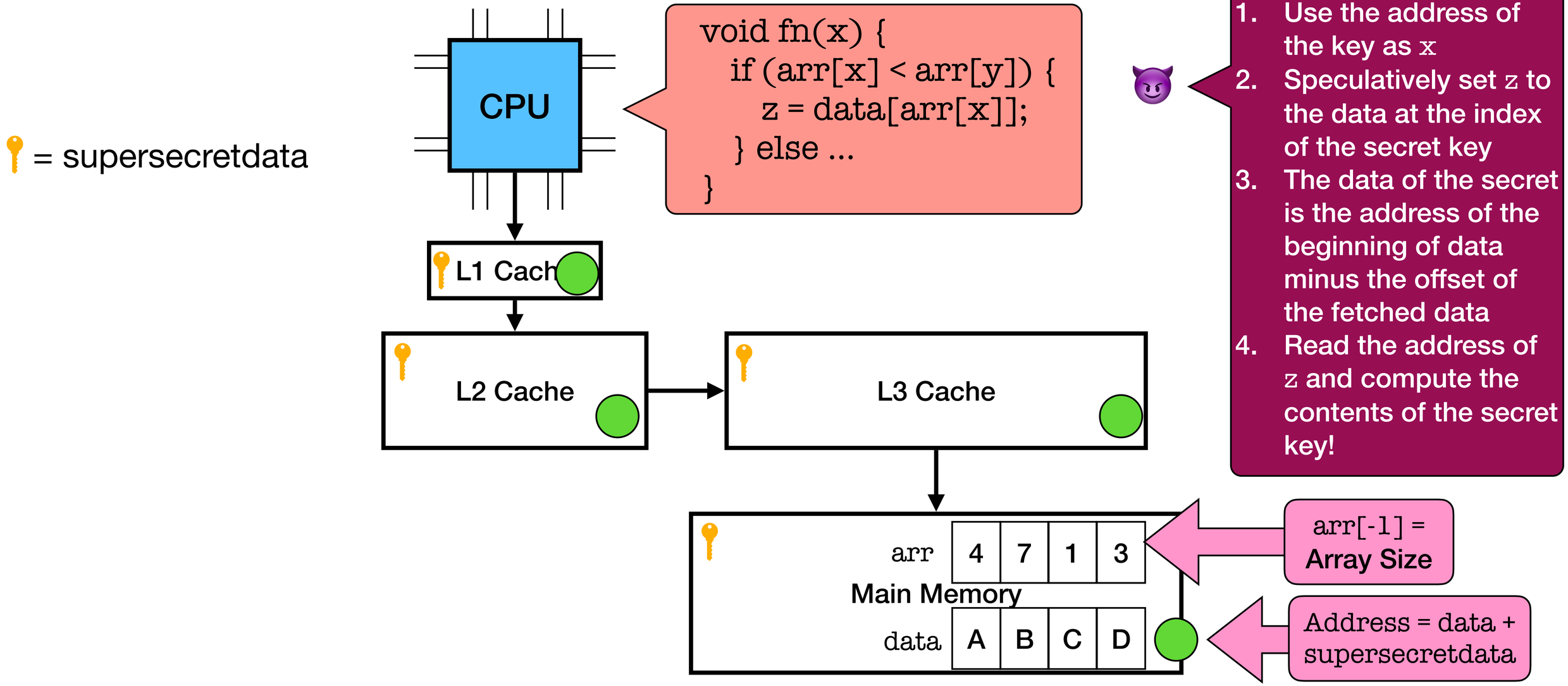
We have been using the red code block as an example of a function susceptible to speculative execution attacks. What could we do to transform this code into the ideal “Spectre Gadget”?

```
void fn(x) {
  if (x < size) {
    y = array[x];
  } else ...
}
```

```
void fn(x) {
  if (arr[x] < arr[y]) {
    z = data[arr[x]];
  } else ...
}
```

Insight: We can only leak addresses via side channels (not data itself). If our data *is* an address then we can leak it!

The “Speculative” Memory Hierarchy



Formalizing the Task of Spectre Adversary

- Goal: leak sensitive information (e.g., secret keys) that may otherwise be protected by the software control-flow through the microarchitectural state
- Phase 1 (setup): find a Spectre gadget in some source around secret data to exploit; mistrain the branch predictor to allow unintended speculative execution during the attack; prepare the covert channel (e.g., the cache side channel) to leak the sensitive information at the end of the attack
- Phase 2 (trigger attack): force the victim to execute the unintended code in the Spectre gadget speculatively such that the sensitive data is loaded into the shared state to be leaked from the covert channel
- Phase 3 (leaking the secret): sensitive data is recovered via the covert channel (e.g., perform a flush+reload attack or prime+probe attack on a shared cache)

Chat with your neighbor(s)!

Think about the attack holistically. How scared of this attack are you? If you were working at a chip fabrication company, how may this change (or not change) your perspective?

(Opinion) Scariness is a Function of Sensitivity

Finding vulnerable code is easier when you know which gadgets you are looking for…


```
1  void looped(uint8_t *plaintext, uint8_t *ciphertext, AES_KEY *key)
2  {
3      __m128i state = _mm_loadu_si128((__m128i *)plaintext);
4      __m128i *rd_key = (__m128i *)key->rd_key;
5      state = _mm_xor_si128(state, *(rd_key++));
6      for (size_t i = 1; i < key->rounds; i++) {
7          state = _mm_aesenc_si128(state, *(rd_key++));
8      }
9      state = _mm_aesenclast_si128(state, *rd_key);
10     _mm_storeu_si128((__m128i *)ciphertext, state);
11 }
```

Listing 1. Pseudo Code for Looped Implementation of AES ECB Encryption Using AES-NI.

Takeaway: there is so much code out there… if there exists a vulnerability in hardware to exploit, then there is probably sensitive code that depends on it

Image Credit: https://dl.acm.org/doi/pdf/10.1145/3620666.3651382

Mitigations

- Preventing speculative execution: if the attack is predicated on speculatively executing unintended code, can we turn off processor speculation?
  - ➡️ speculation has become a fundamental component of processor execution (embedded into designs since the late 1970s), so its performance benefits are expected in program performance
  - ➡️ can we *conditionally* turn off speculation around sensitive data or gadgets? Academically, yes. In practice, unclear!
- Preventing access to secret data: can we enforce bounds checking *in hardware* to avoid illegal secret accesses?
  - ➡️ software is insufficient if hardware can execute unintended code
  - ➡️ the proposal with the most headway is called *capabilities* which requires a complete end-to-end redesign of the processor and memory system

Takeaways

- Adversaries can exploit hardware vulnerabilities to work around safe programming practices in software ➡️ they are more complex, so you should still use safe programming practices in software!!
- The mitigation strategies for speculative execution attacks are *difficult to deploy* and addressing these attacks is very much an open problem!
- Without speculation, our processors could be hugely penalized by control hazards and these attacks are a byproduct of attempting to increase instruction level parallelism despite potential delays!