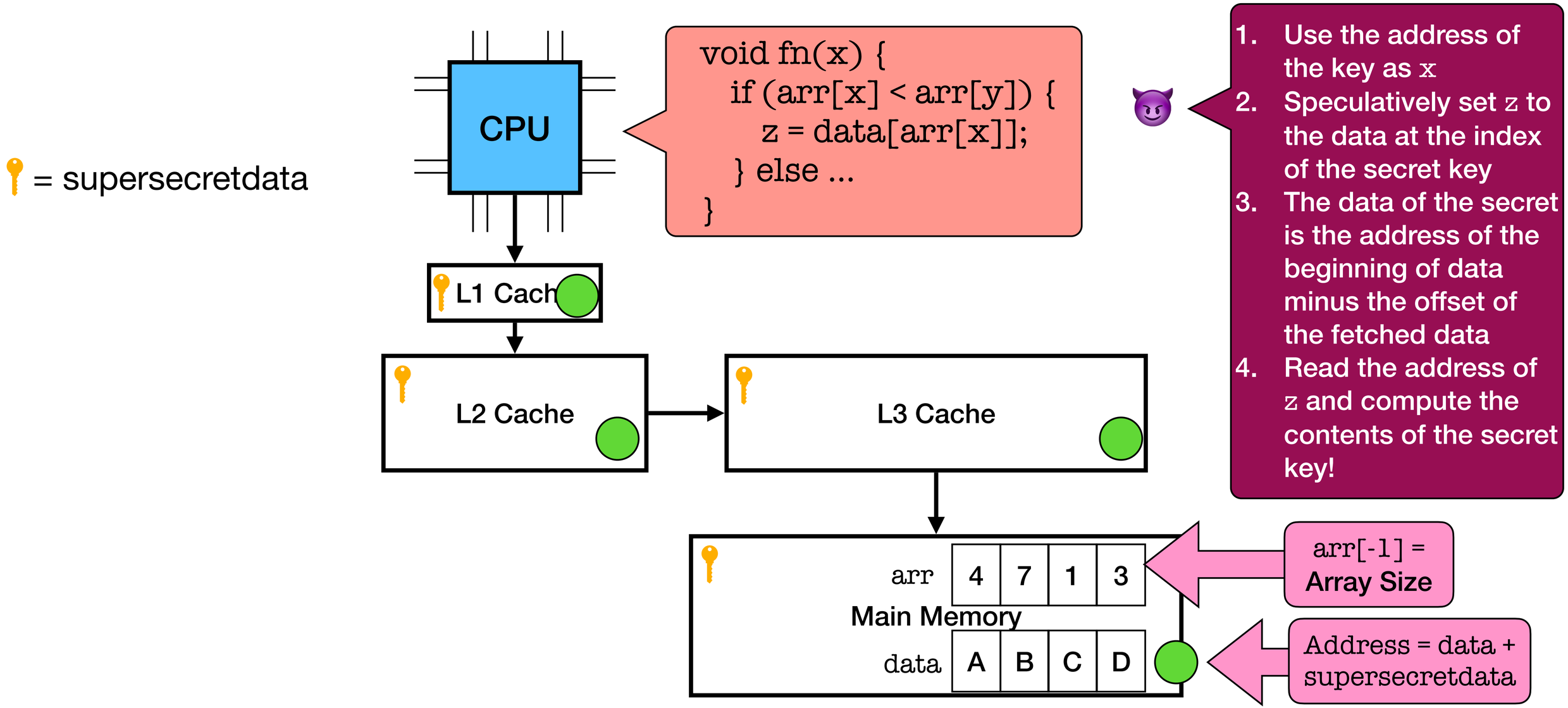
Advanced Instruction-Level Parallelism Methods

No class Friday;
Friday Colloquium on Zoom;
Check In 6 on Monday;
HW3 to be graded manually

(From Monday) Formalizing the Task of Spectre Adversary

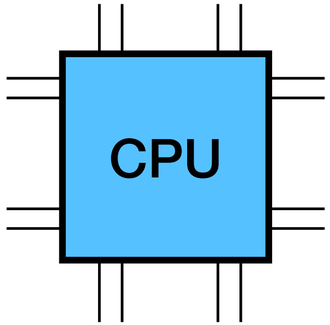
- Goal: leak sensitive information (e.g., secret keys) that may otherwise be protected by the software control-flow through the microarchitectural state
- Phase 1 (setup): find a Spectre gadget in some source around secret data to exploit; mistrain the branch predictor to allow unintended speculative 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)

(From Monday) The “Speculative” Memory Hierarchy



Tying Data Contents to Memory Location

Suppose we wanted to steal the contents of `arr[2]`...




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

L3 Cache

arr

data

Can we read the *contents* of the block? No! We can only read the address of the block...

04	07	01	03	00	00	00	0A	0B	0C	0D	00	00	supersecretdata	
0xff00	0xff01	0xff02	0xff03	0xff04	0xff05	0xff06	0xff07	0xff08	0xff09	0xff0a	0xff0b	0xff0c	0xff0d	

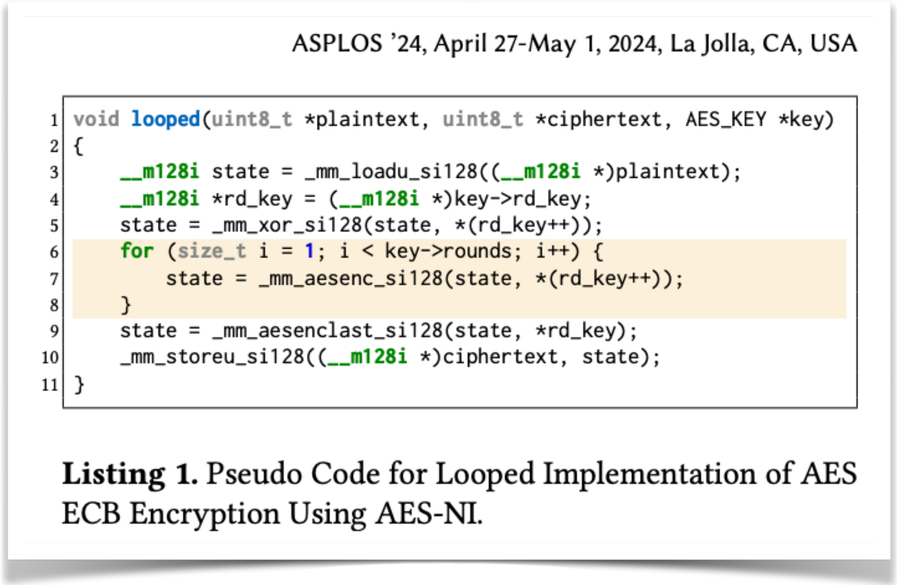
What was the data at `arr[2]`? The contents are address of read block - address of the start of `data`

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…



ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

```
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

For fun: https://leaky.page/

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

Advanced Instruction-Level Parallelism

- The parallelism described by filling our processor pipeline is referred to as “instruction-level parallelism” (or ILP)
  - ➡️ the more we can fill the pipeline with instructions per cycle, the better parallelism we can achieve
- We can exploit ILP via two primary means: dynamically at runtime in hardware *or* statically at compile time in software

Maximizing ILP via Software

- In theory, loops that operate on distinct elements of the array can be performed completely in parallel!
- In practice, compiling this code down to assembly directly will have control hazards ➡️ limits the amount of ILP in the processor!
- The practice of *loop unrolling* can be implemented by the *compiler* to reduce the number of control instructions (and control hazards as a result)
- The drawback of loop unrolling is that your binary size increases as a result of more instructions! Longer loops mean more instructions to explicitly define in the binary itself

```
for (int i = 0; i < 4; i++) {
    x[i] = x[i] + y[i];
}
```

```
x[0] = x[0] + y[0];
x[1] = x[1] + y[1];
x[2] = x[2] + y[2];
x[3] = x[3] + y[3];
```

Maximizing ILP via Software

- There are things that we can write in our software to give the compiler hints of optimization strategies to reduce control hazards in hardware
- The **inline** keyword in C/C++ allows you to declare a function that gets “stamped” into the body of any calling function in the raw assembly ➡️ no unconditional jumps to the function itself!
- Similar drawbacks to loop unrolling, multiple copies of the same instructions throughout the binary means that the binary size increases as a result

```
inline void fn() {   };
```

Chat with your neighbor(s)!

To maximize ILP via software, we ask the compiler to increase the *binary size* to explicitly do things sequentially that would be difficult to predict in hardware. To what extent should you care about the size of your executable binaries? Are there contexts in which you may care more or less?

Maximizing ILP with Hardware

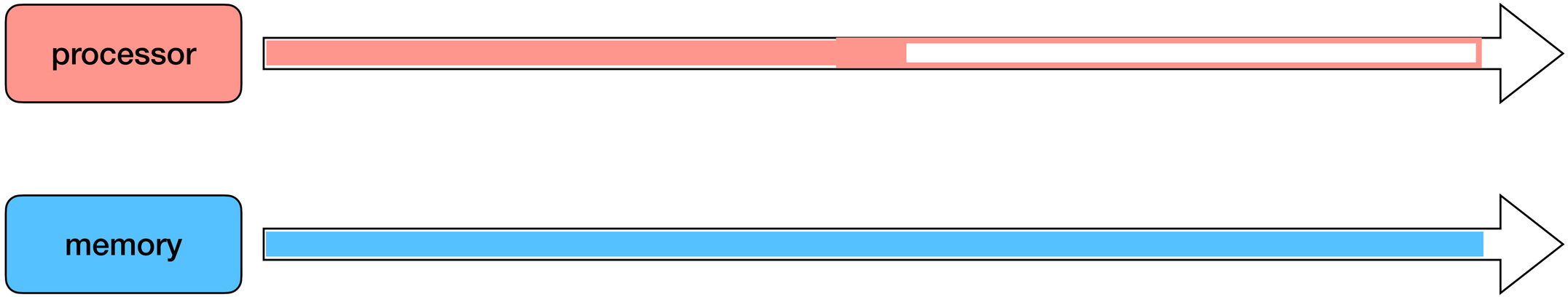
Problem

- Data Hazards
- Control Hazards
- Unrealized CPI
- Mitigate dependencies

Solution

- Forwarding (for many cases)
- Software ILP and Speculation
- Issue multiple instructions per cycle
- Dynamic instruction reordering

Takeaways



What's next? Scaling computation beyond a single core