SPECTRE ATTACK LAB

SEED Lab #3

Identification

- **Group nº3**
  - **José Rodrigues** : 201708806
  - **Paulo Ribeiro** : 201806505
  - **Pedro Ferreira** : 201806506

Task 1:

- After compiling and running the code 10 times, we averaged the results we received, and displayed them below:

Array element	Time taken
[0*4096]	189
[1*4096]	333
[2*4096]	256
[3*4096]	108
[4*4096]	302
[5*4096]	234
[6*4096]	234
[7*4096]	83
[8*4096]	248
[9*4096]	330

- As we can see, the access to array[3*4096] and array[7*4096] were faster than the access to the other elements.
- We defined a threshold of 130 CPU cycles to distinguish between cache and main memory access.

Task 2:

- After adjusting the `CACHE_HIT_THRESHOLD` variable to the value previously calculated (130) and running the program 20 times we noted that the secret was correctly labelled 18 times out of 20 (90%).

Task 3:

- In our first experiment, we first trained our program to always assume the **IF** statement will be true (`if (x < size)`), so that the CPU will execute line ② (`temp = array[x * 4096 + DELTA]`) through *out-*

of-order execution after predicting the outcome of the branch to be true, thus saving `array[97 * 4096 + DELTA]` in the CPU cache. We could then observe that when executing the program the key array was indeed in the cache and we could detect it.

- After this, we commented the line with the star (`_mm_clflush(&size);`), which flushes the variable `size` from the memory, so getting its value from the memory may take a while. We noticed that the program has no output, meaning that line ② is not executed when 97 is fed into the `victim()`. This happens because when the program needs to get the value of the `size` variable, it reads it from the cache (it was not flushed), making its access a lot faster. This way, the CPU does not have the chance to make a prediction and start speculative execution.

- When replacing Line ④ (`victim(i);`) with `victim(i + 20)`, we trained the CPU to assume the **IF** statement will be false (`if (x < size)`), which means that the CPU will not execute line ② (`temp = array[x * 4096 + DELTA]`). Therefore, `array[97 * 4096 + DELTA]` is not stored in the CPU cache.

Task 4:

- Running this Spectre attack, we managed to steal the first byte of the secret (the character 'S', corresponding to the ASCII code 83), but along with it we also received 0 as being part of the secret. This happens because, in the function `restrictedAccess()`, the value 0 is returned when the **IF** condition is false, being therefore also stored in the cache due to speculative execution.

```
[03/27/22]seed@VM:~/.../Labsetup$ ./SpectreAttack
secret: 0x55e724436008
buffer: 0x55e724438018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
```

Task 5:

- After running the improved attack many times, we verified that most of the time the value 0 is returned as the secret due to having the highest score. This happens because, as explained in **Task 4**, the function `restrictedAccess()` returns 0 when the **IF** condition is false, being therefore also stored in the cache due to speculative execution. We fixed it by excluding this value as shown below:

```
int max = 1;
for (i = 1; i < 256; i++) {
    if(scores[max] < scores[i]) max = i;
}
```

- We run the program without the line ① (`printf("*****\n")`) and indeed the program stops working, either by not counting the hits or by not getting any hits at all. Unfortunately, just like the seed lab instructors, we could not figure out the reason for this behaviour.

```
[03/27/22]seed@VM:~/.../Labsetup$ ./SpectreAttackImproved
Reading secret value at index -8208
The secret value is 1()
The number of hits is 0
```

- We experimented with some different values for the `usleep()` call, such as 1, 10, 100 and 1000 microseconds. With all these values, the program worked as expected, achieving the first byte of the secret every time.
- Although, after removing the sleep function (sleep of 0, basically) we managed to get a rare failure in the program in about forty attempts, we did not manage to replicate it more than once.

Task 6:

- In order to reveal the entire secret string, we just looped through all the values of the secret, assuming that either we already know the length of the secret, or we've brute forced our way with several attempts to figure it out.

```
int main() {
  FILE *fp;
  fp = fopen("tmp.txt", "w+");

  int i;
  uint8_t s;
  size_t index_beyond = (size_t)(secret - (char*)buffer);

  for(int m = 0; m < strlen(secret); m++) {
    flushSideChannel();
    for(i=0;i<256; i++) scores[i]=0;

    for (i = 0; i < 1000; i++) {
      printf("*****\n");  // This seemly "useless" line is necessary for the
attack to succeed
      spectreAttack(index_beyond + m);
      usleep(10);
      reloadSideChannelImproved();
    }

    int max = 1;
    for (i = 1; i < 256; i++){
      if(scores[max] < scores[i]) max = i;
    }

    //fprintf(fp, "Reading secret value at index %ld\n", index_beyond);
    fprintf(fp, "The secret value is %d(%c)\n", max, max);
    //fprintf(fp, "The number of hits is %d\n", scores[max]);
  }
```

```
    return (0);
}
```

- Notice that, for readability purposes, we redirected the values of the secret to a file called `tmp.txt`, because the console would be flooded with the prints from the line ① (`printf("*****\n")`), which is necessary for the attack to succeed in Ubuntu 20.04.

- The secret is revealed in the resulting file `tmp.txt`:

```
The secret value is 83(S)
The secret value is 111(o)
The secret value is 109(m)
The secret value is 101(e)
The secret value is 32( )
The secret value is 83(S)
The secret value is 101(e)
The secret value is 99(c)
The secret value is 114(r)
The secret value is 101(e)
The secret value is 116(t)
The secret value is 32( )
The secret value is 86(V)
The secret value is 97(a)
The secret value is 108(l)
The secret value is 117(u)
The secret value is 101(e)
```