Homework 4 (Spectre) Report

Introduction

This report presents an overview on a spectre attack performed as a part of the Advanced Computer Architecture course (CS-470) at EPFL. A cache attack was performed on a speculatively executed load, for the following victim function, to read the secret "The Magic Words are Squeamish Ossifrage."

```
void victim_function(size_t x) {
  if (x < array1_size) {
    temp ^= array2[array1[x] * 512];
  }
}
```

Forcing mis-speculation

First, we need to force the victim to incorrectly predict the branch so that the array load instruction is speculatively executed. In order to do so, we train the global and local branch predictors. For training, we use a legitimate x, generated as *iteration_number % array1_size*, which is within the range. The training call is done 5 times before 1 attack. This is repeated 5 times, thus giving a total of 30 victim function calls. In order to perform this, we need to set the *call_x* as the legitimate value 5 times out of 6, and as the malicious value 1 time out of 6.

It can be achieved easily by using modulo function and an if-else statement, but we need to avoid polluting the branch predictor. Hence, we avoid branches and use bit-level manipulation as follows:

```
// Set call_x=FFF.FF0000 if j%6==0, else call_x=0
call_x = ((j % 6) - 1) & ~0xFFFF;
// Set call_x=-1 if j%6=0, else call_x=0
call_x = (call_x | (call_x >> 16));
// Set call_x=training_x if j%6!=0 or malicious_x if j%6==0
call_x = training_x ^ (call_x & (malicious_x ^ training_x));
```

Delaying branch resolution

Since we need to read a few bytes of data, we need to speculatively execute as many load instructions as we can. Hence, we need to delay the branch resolution, so that the victim function can put the complete secret into the cache. This is achieved by slowing down the comparison: *x < array1_size.* The *array1_size* variable is removed from the cache before every victim function call, thus slowing the condition evaluation and delaying the branch resolution.

After flushing the *array1_size* from the cache, we wait for 100 cycles (and use a memory fence) for the flush to commit. This loop also serves as a deterministic long loop right before the function call aiding in our aim of training the branch predictors.

Performing the cache attack

Finally, we need to retrieve the data written into the cache by the victim function. This can be done using a prime + probe attack on the cache. The *array2* is first flushed from the cache, followed by training the branch predictors and the victim function call (as discussed above). The function call puts the secret into the cache at some index. The index can be deduced by accessing all the elements of *array2* and reading every entry while measuring the execution time. The execution time will be significantly less for a cache hit, which is the index we are interested in.

The cache hit is identified using a threshold optimized for my own machine experimentally. The *array2* read is not done sequentially in order to trick the data prefetcher. The access order is shuffled by using a modified index as follows: *shuff_i = ((i * 177) + 15) & 255*. It is simply a random linear mapping optimized for best performance experimentally.

Reporting the results

The above-described attack is repeated 999 times (not 1000 for the sake of readability) in order to improve the accuracy. The number of cache hits are stored on every attack for statistics. Finally, the top two indices with the highest hit counts are chosen as the results, and the hit counts are reported as the scores. This is repeated for every character in the secret string to gain the information.

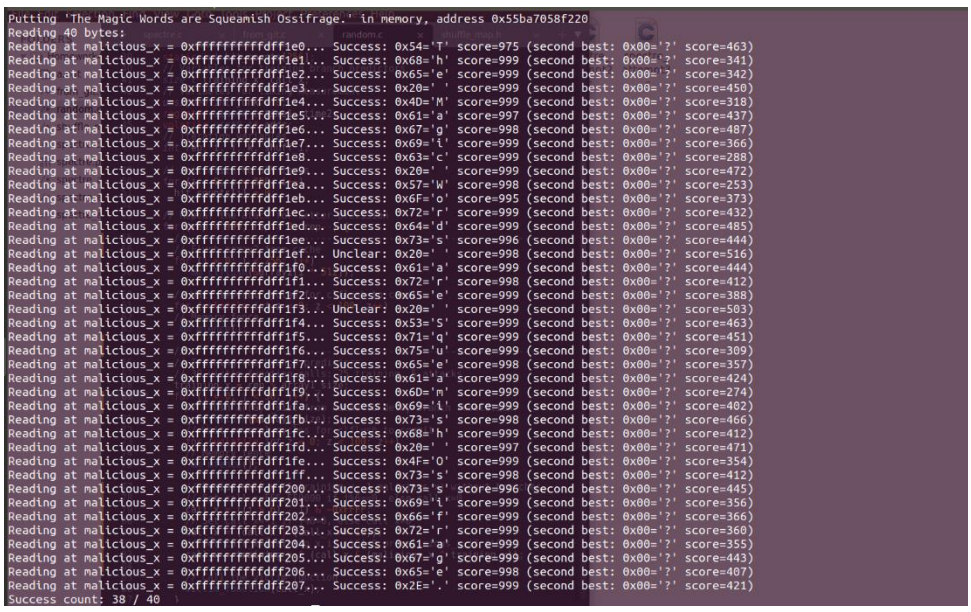
Troubles faced

For this assignment, I simply followed the steps provided in the homework handout, the cache attack lab and the moodle forum. I had to incorporate all the extra steps for a successful attack on my computer as the basic one without any optimizations did not work. The extra steps included:

1. Allowing extra cycles and memory fence for *clflush* to commit.
2. Multiple legitimate calls for training the branch predictors.
3. Using bit-level predication instead of branches.
4. Shuffling the access order using a custom linear mapping.
5. Using the junk variable somewhere to avoid optimization.
6. Tweaking parameters for improving the performance.

Conclusion

The attack was performed successfully, and some parameters such as the number of iterations, threshold, shuffle order etc were tuned according to my computer in order to optimize the attack. The attack results are fairly stable, correctly reading the secret with consistently competitive accuracy. The following screenshot shows one of the best achieved results:



Submitted by: Kushagra Shah (316002)

Dated: 14/06/2021