Homework 4: Spectre

In this homework, you will learn to perform a cache attack on a speculatively executed load.

Victim

Assume that the victim runs the following C code:

```c
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16};
uint8_t unused2[64];
uint8_t array2[256 * 512];

uint8_t temp = 0;

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

Appropriately, `victim_function()` checks the index boundary before accessing `array1`. However, modern processors use speculative execution to improve performance, and a **malicious load** anywhere in memory could be speculatively executed. For instance, if the malicious value of `x` is `(p0) - array1`, the index becomes `p0`, which could point to a variable you have no permission to access. After that, the value read from the location pointed by `p0` will be used as an index to access `array2`, bringing the related block into cache. Using the techniques introduced in the Cache Attack Lab, it is possible to figure out the index used in accessing `array2` and therefore discover the secret returned by the speculatively executed load.

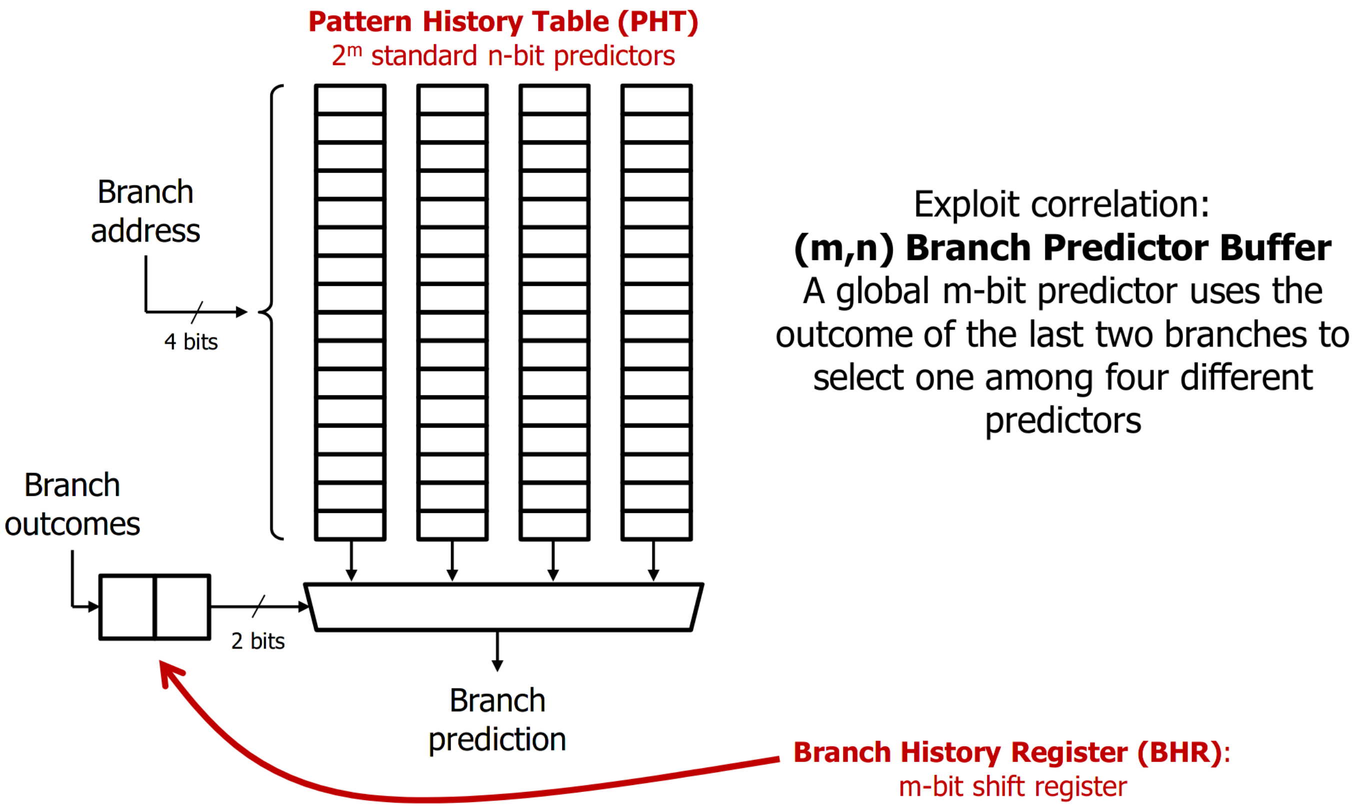
Insight

To succeed in the attack, you have to do the following:

- **Train the branch predictors** so that the load could be speculative executed.
- **Extend the side channel** (latency to resolve the wrongly predicted branch) so that enough code is executed speculatively to bring the desired block into the cache.

Training the Global Branch Predictor

As suggested in an earlier lecture, most modern processors use correlation predictions to predict branches. This means that the Pattern History Table (PHT) is accessed using both the branch address (in the PC) and the global branch history (Branch History Register, BHR).


Pattern History Table (PHT)
2^m standard n-bit predictors
Branch address
4 bits
Branch outcomes
2 bits
Branch prediction
Branch History Register (BHR):
m-bit shift register

Exploit correlation:
(m,n) Branch Predictor Buffer
A global m-bit predictor uses the outcome of the last two branches to select one among four different predictors

To make sure that you are correctly training the good entry in the PHT that will be used during the attack, two techniques may help:

- Create a standard global branch history before calling the `victim_function()` by introducing a deterministic long loop. When the CPU executes this loop, it will generate several *taken* entries (i.e., continue loop) in the history followed by one *not-taken* (i.e., exit loop) entry. The actual content does not matter, provided it is long enough to fill the BHR and it is the same during training and attack.

  > **Hint**: Since we do not know the exact details of the branch predictor, trials are necessary to find the correct way to fill the global branch history. For instance, you may need to repeat the training loop several times, change the number of iterations for each loop, and maybe even use nested loops.

- Avoid using branches during the attack and, when necessary, use bit-level predication instead. For instance, when selecting the training or malicious payload to call the victim function, rather than an `if` statement or the `?:` expression, try to achieve the same functionality with bitwise logic operations (e.g., `&` for *and*, `|` for *or*, `^` for *xor*, and `~` for complement). A couple of examples of this style of programming can be found in the lectures *Exploiting ILP Statically* and *From Processor Customization to HLS*.

Depending on your specific situation (processor, compiler, etc.), you may have to use one, the other, or both of these techniques.

Training the Local Branch Predictor

All branch predictions are based on the history (as the PHT in the figure above), so before calling the function with a malicious `x`, you should use any legitimate `x` to call it so that the `then` clause is executed. Since you know little of the exact structure of the PHT in your processor, you can train it with several calls to a legitimate `x` before the attack. And maybe you want to repeat the process several times.

Extending the Side Channel

The longer is the time between branch prediction and branch resolution, the more instructions will be executed speculatively and the more time will be available for the speculated instructions to complete (in particular, the more chances the access to `array2` will have to affect the cache). In order to delay the branch resolution, you need to delay the result of the comparison it depends on. For instance, if the operands needed in the comparison are not in the cache, extra time would be needed to fetch them from DRAM before the comparison will be performed.

Task

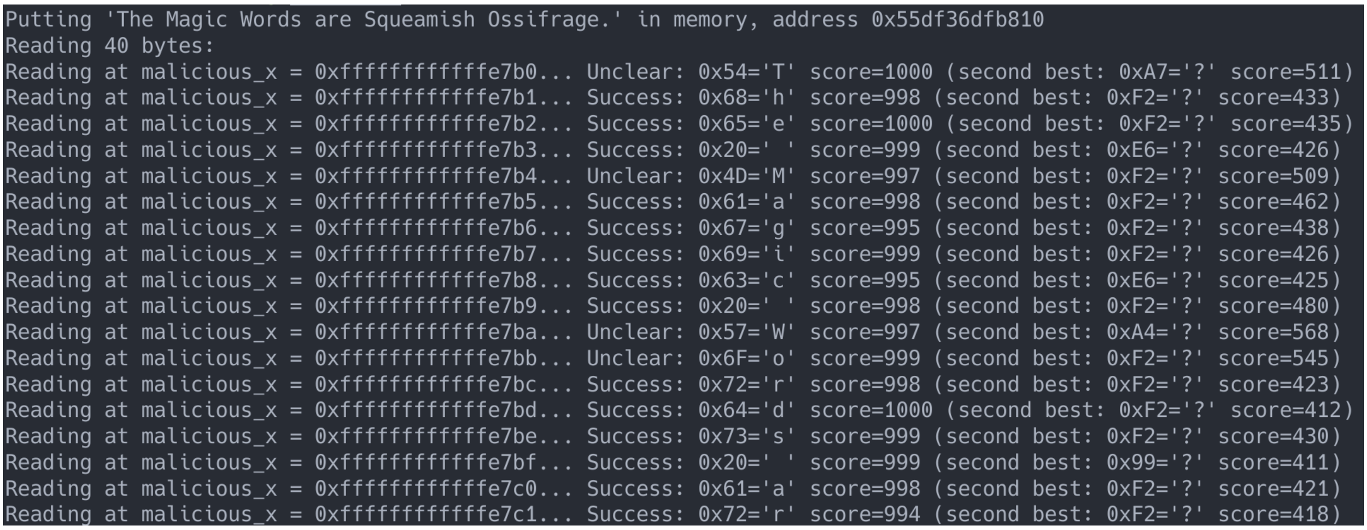
A template is provided for you in file `spectre.c`. You need to write the attack function `attack()`. As with the Cache Attack Lab, the attack may need to be repeated multiple times to statistically improve its accuracy.

```
void attack(size_t malicious_x, uint8_t value[2], int score[2]);
```

The parameters are as follows:

- `malicious_x`: the malicious `x` used to call `victim_function()`.
- `value`: The two most likely guesses returned by your attack.
- `scores`: The score (larger is better) of the two most likely guesses.

The `main` function provided by the framework will try to attack a string and show the result. Here is the result of a sample run:



```
Putting 'The Magic Words are Squeamish Ossifrage.' in memory, address 0x55df36dfb810
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffffe7b0... Unclear: 0x54='T' score=1000 (second best: 0xA7='?' score=511)
Reading at malicious_x = 0xffffffffffffe7b1... Success: 0x68='h' score=998 (second best: 0xF2='?' score=433)
Reading at malicious_x = 0xffffffffffffe7b2... Success: 0x65='e' score=1000 (second best: 0xF2='?' score=435)
Reading at malicious_x = 0xffffffffffffe7b3... Success: 0x20=' ' score=999 (second best: 0xE6='?' score=426)
Reading at malicious_x = 0xffffffffffffe7b4... Unclear: 0x4D='M' score=997 (second best: 0xF2='?' score=509)
Reading at malicious_x = 0xffffffffffffe7b5... Success: 0x61='a' score=998 (second best: 0xF2='?' score=462)
Reading at malicious_x = 0xffffffffffffe7b6... Success: 0x67='g' score=995 (second best: 0xF2='?' score=438)
Reading at malicious_x = 0xffffffffffffe7b7... Success: 0x69='i' score=999 (second best: 0xF2='?' score=426)
Reading at malicious_x = 0xffffffffffffe7b8... Success: 0x63='c' score=995 (second best: 0xE6='?' score=425)
Reading at malicious_x = 0xffffffffffffe7b9... Success: 0x20=' ' score=998 (second best: 0xF2='?' score=480)
Reading at malicious_x = 0xffffffffffffe7ba... Unclear: 0x57='W' score=997 (second best: 0xA4='?' score=568)
Reading at malicious_x = 0xffffffffffffe7bb... Unclear: 0x6F='o' score=999 (second best: 0xF2='?' score=545)
Reading at malicious_x = 0xffffffffffffe7bc... Success: 0x72='r' score=998 (second best: 0xF2='?' score=423)
Reading at malicious_x = 0xffffffffffffe7bd... Success: 0x64='d' score=1000 (second best: 0xF2='?' score=412)
Reading at malicious_x = 0xffffffffffffe7be... Success: 0x73='s' score=999 (second best: 0xF2='?' score=430)
Reading at malicious_x = 0xffffffffffffe7bf... Success: 0x20=' ' score=999 (second best: 0x99='?' score=411)
Reading at malicious_x = 0xffffffffffffe7c0... Success: 0x61='a' score=998 (second best: 0xF2='?' score=421)
Reading at malicious_x = 0xffffffffffffe7c1... Success: 0x72='r' score=994 (second best: 0xF2='?' score=418)
```

The purpose is, firstly, to succed with the attack, of course, and then, secondly, to maximize its robustness by increasing the difference between the best score and the second best.

Submission

Besides the source code, you also have to submit a very brief report to explain (1) how did you train the global and local branch predictors, (2) how you extended the side channel, (3) all the techniques you used to improve the attack accuracy, and (4) how you solved any additional problems you encountered.

Acknowledgements

Inspired by an assignment by Marten van Dijk, CWI, The Netherlands.