Advanced Computer Architecture second coursework: Spectre PoC

Harvin Iriawan
hi116
01179324
hi116@ic.ac.uk

1. Background

In this exercise, we are exploring ways to optimise a program that defeats array bounds checking by means of side-channel attacks. Our aim is to study the performance of this program and find ways to make it exfiltrate secret data quickly with reliable accuracy. We want the program to run as fast as possible.

1.1. How the program works

The program `spectre-ACA.c` declares some arrays in memory:

```
unsigned int array1_size = 16;
uint8_t unused1[512];
uint8_t array1[16] = {
  1, ... ,16
};
uint8_t unused2[512];
uint8_t array2[256 * 512];
char * secret = "The Magic Words are Squeamish
    Ossifrage";
```

At no point in the program the char array `secret` is accessed directly, but the attacker can deduce what is in it, with the help of a victim function.

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

The `if (x < array1_size)` is referred to as bounds checking. From a software perspective, this safeguards access to memory that is larger than `array1`. However, most modern processors implement branch prediction, speculation and out-of-order execution. Therefore, the speculative memory access might be finished first before the bounds checking is complete. If the bounds checking fails, the CPU will restore its state to what it was before memory access is done.
However, that memory address and contents may have been put in the cache. This ”side-channel” can be used to extract the speculative memory access’ contents. The method flush and reload is employed. In essence the attack works by:

1. `array1` and `array2` are flushed from the cache before `victim_function` is run.
2. Train the branch predictor to assume that the bounds check always succeds by accessing `victim_function` with a valid `array1` address some number of time. The default training iteration is 5.
3. Access `victim_function` with an out-of-bounds address that is the location of a particular `char` in `secret`. Hopefully this will be stored in `array2`
4. Iterate multiple times to ensure that the secret obtained is accurate. Default iteration is 1000.
5. Check the access time of `array2` entries. The entry that contains the secret will have significantly low access cycle latency (measured by `rdtscp`) than other entries. Default cycle latency is 120.

1.2. Hardware and Software used in the test

Testing of the program was done on several hardware and software environments

Processor	**Clock Frequency**	**L1 cache size**	**L2 cache size**	**L3 cache size**	**Compiler**	**OS Kernel**
Intel i7-6500U	2.50GHz	64KiB D and I Cache	512KiB	4MiB	GCC 7.4.0	5.0.0-32-generic

Table 1. Hardware and Software under Testing

2. Experiments and Results

The baseline performance of the program is recorded.

Iteration	**Run Time**	**Exfiltration rate**
1	25.469432	100.000000
2	24.569828	100.000000
3	25.228421	100.000000
4	24.350611	100.000000
5	24.379639	100.000000

Table 2. Baseline program performance

Several changes are done to the program to see if there are improvements in the execution time and exfiltration rate.

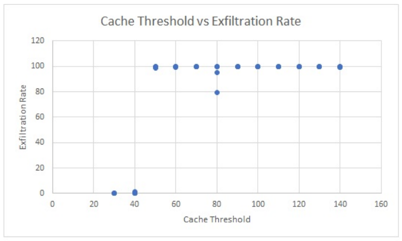
2.1. Varying Cache Threshold

Running the script *varycachethreshold.sh*, the program is run for different values of cache threshold. The cache threshold value should correspond to the Average Memory Access Time (AMAT) of a particular CPU (clock cycle latencies of memory access) when data is in the cache. We expect the latency for access of `array2[array1[x]*512]` to be significantly lower for `x` that is `malicious_x`. Hence, for all entries of array2, we can pick the

entry that corresponds to the secret.
For low threshold value, lower than in-cache AMAT, exfiltration rate will be bad and exfiltration rate is expected to improve with cache threshold increasing.
Similar trend should be observed in execution time as well. Below the in-cache AMAT, the program will run for a long time as it will run until `NTRIES` is exhausted. `BREAK` statement will not get invoked as secret cannot be extracted. Afterwards, program should run significantly faster.

2.1.1 Experimental Data

The data collected from the 1st experiment is presented below:



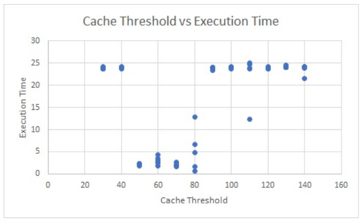


Figure 1. Varying cache threshold against exfiltration rate and program runtime

2.1.2 Discussion

The results from Figure 1 are interesting. While the variation of cache threshold against exfiltration rate agrees with the hypothesis, the variation of cache threshold against program execution time does not.
It is observed that between cache threshold value of 40 and 50, exfiltration rate increases dramatically. Further varying of cache threshold between 40 and 60 indicate a good cache threshold for this program is 56, where exfiltration rate of 95% and above is consistently achieved.
Interestingly, the program runtime behaves not as as expected. From cache threshold of 40 to 90, the runtime gets significantly faster, but execution time increases again, equalling execution time when cache threshold is below the in-cache AMAT. This means that when cache threshold is varied from 90 to 140, the program is having a hard time to determine the secret character as there are other `array2` entries that is cached as well.
A plausible explanation for this is that some extent of stride prediction is actually successful and the some `array2` entries are brought into the cache, but not into the L1 cache (possibly L2 or L3). Although `array2` cannot fit into the L1 cache, it fits into the L2 or L3 cache, as seen from 1. Only the secret character is actually brought to the L1 cache as it is accessed speculatively very often. When the cache threshold value is high, it makes no distinction of accesses from low-level or high-level cache.
However, exfiltration rate does not suffer, as hardware prefetching does not always prefetch the same `array2` entries at every `NTRIES`. More often than not, the secret character is able to be extracted, just that it takes more `NTRIES` for every character to do it, making program execution time much longer. This is proven as most of the time `second best` guess has significantly lower scores than the 1^{st} guess.

2.2. Reduce Delay Time

Another idea to be tested is the effect of reducing delay time. Delay time is done before calling `victim_function`.

```
for (volatile int z = 0; z < 100; z++) {}
```

The idea is that this will give the processor to completely flush `array2[i*512] for i = 0 to 255` and `array1_size`, hence improving accuracy of the attack as only `array2` entry that corresponds to `secret` will remain in the cache. This is done when the cache threshold is fixed while the upper bound for `z` is varied. For this experiment, it is fixed to 55.

2.2.1 Experimental Data

Experimental data is presented in tables below:

Iteration	Run Time	Exfiltration rate
1	3.640011	100.000000
2	4.060715	99.315068
3	3.403614	99.543379
4	4.224926	99.771689
5	3.911183	100.000000
6	4.372423	100.000000
7	5.286335	91.780822
8	4.079683	100.000000
9	3.016507	100.000000
10	6.115482	80.821918

Table 3. z = 300 for cache threshold = 55

Iteration	Run Time	Exfiltration rate
1	3.489226	97.716895
2	1.273060	99.315068
3	3.365827	97.488584
4	3.860283	97.488584
5	3.801041	97.031963
6	3.658824	97.260274
7	0.961276	99.543379
8	2.865577	97.260274
9	3.297679	97.945205
10	2.740439	95.890411

Table 4. z = 100 for cache threshold = 55

Iteration	Run Time	Exfiltration rate
1	4.403434	93.835616
2	5.006961	93.378995
3	4.080401	94.748858
4	4.863879	94.748858
5	4.216993	93.378995
6	3.731856	94.520548
7	4.270535	91.552511
8	4.182670	83.561644
9	3.975527	93.378995
10	3.787568	90.639269

Table 5. z = 50 for cache threshold = 55

Iteration	Run Time	Exfiltration rate
1	4.526055	89.726027
2	4.931401	91.324201
3	4.756682	87.671233
4	4.298997	94.063927
5	3.742548	94.748858
6	5.243935	92.009132
7	4.649918	93.150685
8	4.439118	93.835616
9	4.421565	91.324201
10	4.809178	82.191781

Table 6. z = 20 for cache threshold = 55

2.2.2 Discussion

The data indicates that exfiltration rate increases as delay time increases. This observation concurs with the hypothesis made, that giving more delay means ensuring `array2` to be flushed from all levels of the cache as much as possible before speculative access is done. Although, no reduction of program runtime is observed by changing this.

2.3. Increase the number of training iteration

Number of training iteration can be varied to ensure that the branch predictor is primed to take the branch although bounds condition is violated. If the branch predictor is complex enough it might predict that the program tries to access the secret every 6^{th} try. In theory, increasing training time will reduce the possibility of this happening. Keeping cache threshold at the same value, `NLOOPS` and training iteration are varied. To make the testing fair, whenever training iteration is changed, `NLOOPS` must follow accordingly so that in every try, we are accessing `malicious_x` 5 times.

2.3.1 Experimental Data

Data is collected into the table below:

2.3.2 Discussion

Referring to 7 and 8. there seems to be no significant improvement in neither program runtime nor exfiltration rate. Therefore, 5 training iteration is enough to ”fool” the branch predictor into taking the branch despite being fed `malicious_x`. It is possible that this is because most branch predictors use 2-bit saturating counter and thus training the branch to be taken 4 consecutive time is sufficient.

Iteration	Run Time	Exfiltration rate
1	3.397401	98.401826
2	3.053442	98.401826
3	2.854912	100.000000
4	0.874920	99.543379
5	3.192955	99.543379
6	2.236603	97.488584
7	3.462253	99.086758
8	3.048740	98.858447
9	2.958109	98.173516
10	3.582881	98.173516

Table 7. NLOOPS=39, 7 training + 1 testing

Iteration	Run Time	Exfiltration rate
1	4.832587	86.301370
2	1.458536	99.315068
3	3.324521	97.716895
4	3.652151	99.315068
5	1.887016	99.315068
6	5.625747	97.031963
7	2.943824	98.630137
8	2.389800	97.945205
9	3.089554	97.031963
10	3.533626	97.260274

Table 8. NLOOPS=49, 9 training + 1 testing

2.4. Reduce Number of Tries

Reducing `NTRIES` undoubtedly will improve the performance of the code. Every character in `secret` can take at most `NTRIES` iterations. The default at the moment is 999. At the moment, the program only exits the loop less than `NTRIES` when either the secret character is accessed twice with no other `array2` access below cache threshold value or when the highest result more than twice the second-highest result plus 5 :

```
if (results[j] >= (2 * results[k] + 5) ||
    (results[j] == 2 && results[k] == 0))
  break;
```

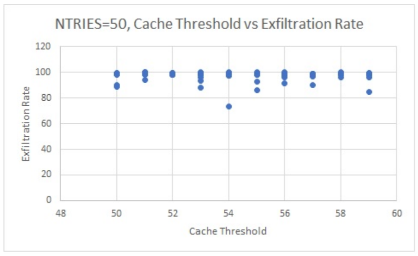
Therefore, if in a particular try, the hardware prefetcher loads another index of `array2[i*512]`, it can take the whole `NTRIES` for a particular character. By experimentation, this does not happen often, as when cache threshold value is appropriate, `Reading at malicious_x...` returns success with `score=2` frequently.
Therefore, we can either reduce `NTRIES` or changing the first part of the `break` condtion: `if(results[j] >= (2*results[k]+5)`.

2.4.1 Experimental Data

2 experiments were conducted, at NTRIES = 400 and NTRIES = 50. Below is the results obtained:

2.4.2 Discussion

As hypothesised, reducing NTRIES make the program complete in significantly shorter time with no impact in exfiltration rate, seen



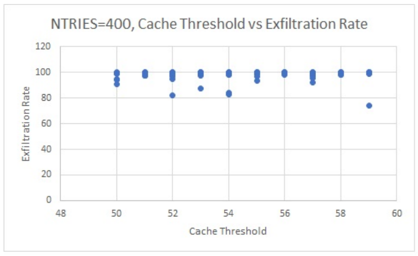
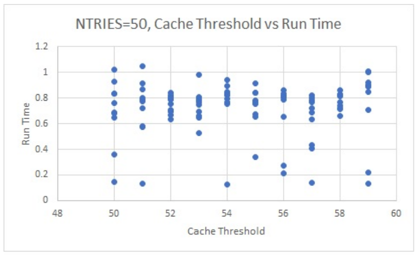


Figure 2. Different NTRIES effect on exfiltration rate



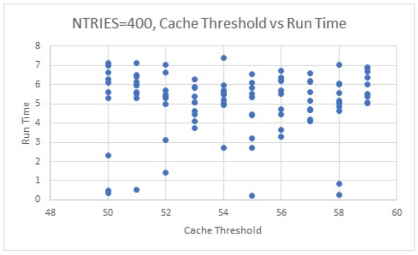


Figure 3. Different NTRIES effect on program runtime

in 2 and 3. Further experiments show that `NTRIES` can be even reduced to 5 with an acceptable decrease in exfiltration rate.