Home Assignment 1

Multi-Core Architecture and Systems (0368-4183)

Submitters	Yuval Birman	Hod Badihi
Ids	211719919	315314922
Emails	yuvalbriman@mail.tau.ac.il	hodbadihi@mail.tau.ac.il

TEL AVIV UNIVERSITY אוניברסיטת תל אביב

Our work involved conducting 4 experiments, which are partially detailed in this report, and in greater detail in the git repository [4]. The full attack can be found under "Experiment 3" or in the submitted C file. Through two primary sources, namely Intel's manual and academic research([2] , [3]). We believe that Herbert's hypothesis is probably incorrect. We also believe that the failure of our full attack (experiment 3) provides additional evidence that supports our conclusion.

Our (possible) Explanation:

According to the Intel optimization manual [1], the use of 4k aliasing may result in a load latency penalty of 5 cycles. Herbert suggests that the slowdown is due to speculative load/store forwarding, and if there is a mismatch in the physical address, the load is canceled and reissued.

“...When a load is checked against previous stores, not all of its address bits are compared to the store addresses. This can cause a load to be blocked because its address is similar (LD_BLOCKS.4K_ALIAS) to a pending store, even though technically the load does not need to be blocked)”

After examining the "Table B-1. Performance Monitoring Taxonomy" in the optimization manual, we discovered that a load is blocked when a 4k alias case occurs. The term "blocked" implies that the load is put on hold and awaits a specific condition before proceeding. We therefore sought to determine what that condition might be.

Apparently the L1D cache on Intel processors is virtually indexed and physically tagged using some of the offset bits for indexing and the PA tag for selecting the cache line. Using the VA allows an overlapping access between the TLB & L1D Cache as can be seen here:

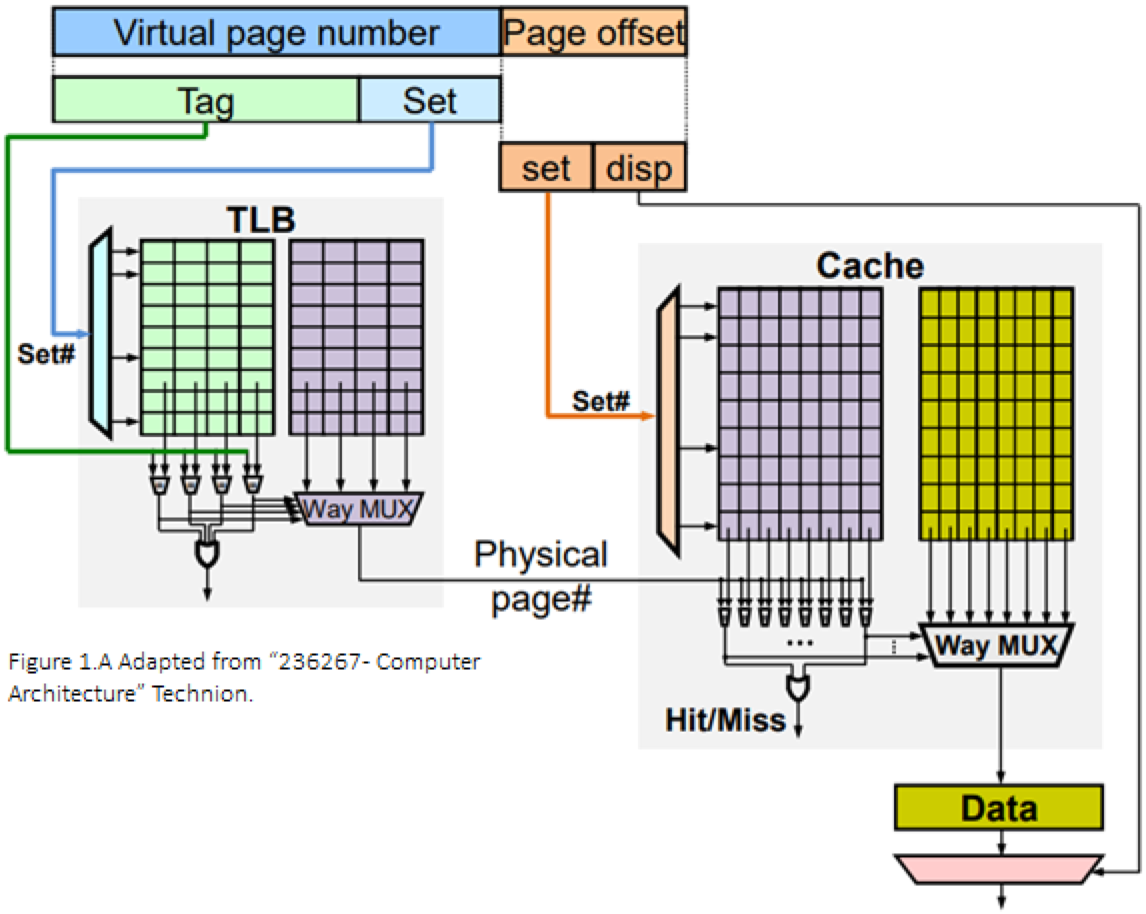
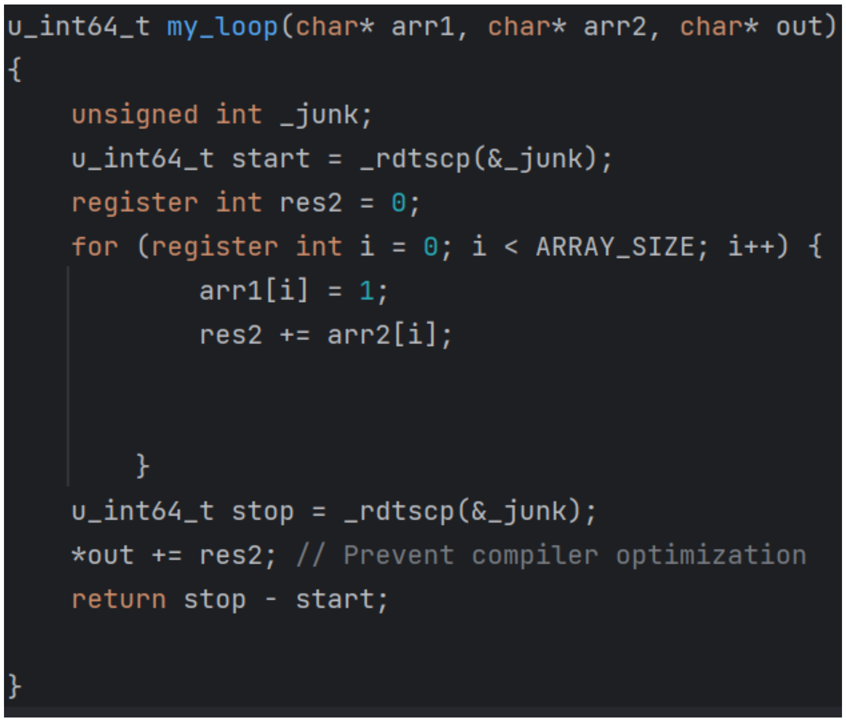


Figure 1.A Adapted from "236267- Computer Architecture" Technion.

The cache uses the 12 least significant bits of the VA to detect potential dependencies between multiple reads and writes, preventing simultaneous reading and writing of 4K-aliasing addresses. If a dependence is detected, even if it's a false one as our RaW, we believe the read will be blocked and reissued and that's the real cause of the 5 cycles penalty.

Experiment1

Our goal was to confirm that 4k aliasing occurs in our configuration and that the latency gaps are feasible.



```
u_int64_t my_loop(char* arr1, char* arr2, char* out)
{
    unsigned int _junk;
    u_int64_t start = _rdtscp(&_junk);
    register int res2 = 0;
    for (register int i = 0; i < ARRAY_SIZE; i++) {
            arr1[i] = 1;
            res2 += arr2[i];


        }
    u_int64_t stop = _rdtscp(&_junk);
    *out += res2; // Prevent compiler optimization
    return stop - start;

}
```

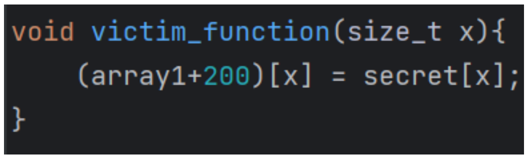
In our experiment, `my_loop` looped through two arrays, `arr1` and `arr2`. `arr1` initialized a store instruction, while `arr2` initialized a load instruction, resulting in a desired read-after-write scenario. We conducted the experiment using arrays that were 4k-aliased and non-4k-aliased. We performed the experiment on arrays of size 16k and obtained the subsequent results:

Run	arr1-offset	arr2-offset	Clock cycles	ld_blocks_partial_address_alias
#1	0	0	44,290	18,070
#2	0	160	39,084	2,439

The difference in latency is acceptable, and we observed a linear relationship between the loop size and the performance counter difference, which makes sense.

<u>Experiment 2</u>

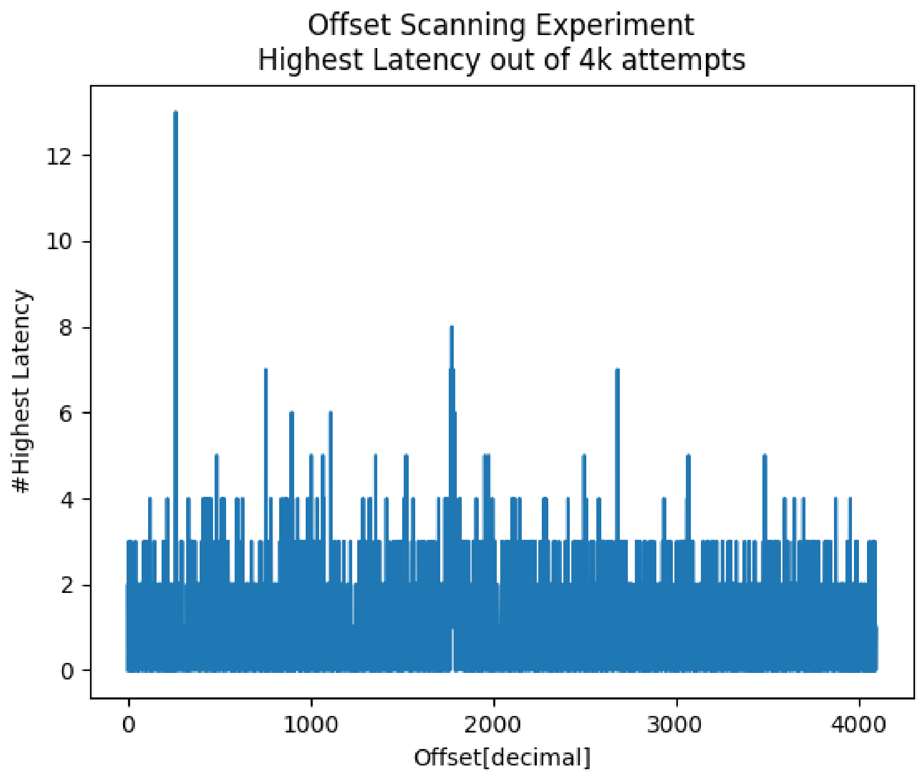
The objective of this experiment is to confirm that a time-channel attack is possible to get a victim`s secret offset.



```
void victim_function(size_t x){
    (array1+200)[x] = secret[x];
}
```

Where our target array(`array1`) offset is $0x40 + (200)_{10} = 0x108 = (264)_{10}$

We conducted 4000 attempts to scan all 4096 possible offsets, where the attacker executes multiple loops and calls the `victim_function`. In each attempt, we recorded the offset with the highest latency.

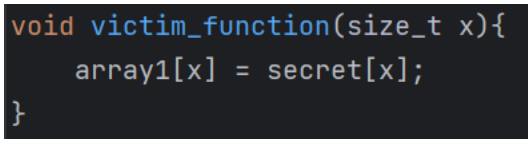


As expected, we have observed a peak with 16 hits at offset $(264)_{10}$

Experiment 3

Having confirmed that the 4k-aliasing latency is achievable in our setup and using the time-side channel for offset scanning, we were able to proceed with the next attack to test Herbert's hypothesis.

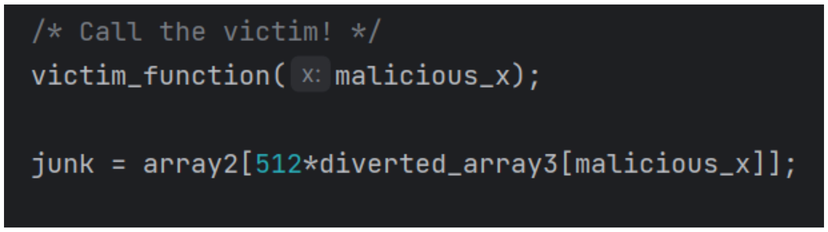
Victim`s function:



```
void victim_function(size_t x){
    array1[x] = secret[x];
}
```

Assume array1 address is unknown.

Attacker`s store to load speculative forwarding:



```
/* Call the victim! */
victim_function( x: malicious_x);

junk = array2[512*diverted_array3[malicious_x]];
```

Where `divereted_array3` is aligned to `array1` (see attack steps below)

We planned the following attack:

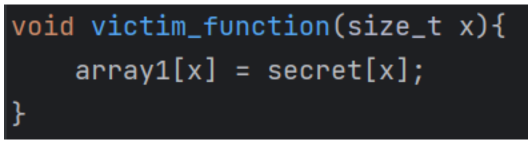
1.Run an offset scan as in experiment 2 to determine the starting offset of `array1` and align `array3` accordingly.

2.Utilize store-to-load speculative forwarding to read from `array2` and proceed with the attack by leveraging the cache side channel, as demonstrated in the original `spectre.c` implementation.

The experiment was unsuccessful, we attempted to reduce the complexity of the experiment in the next iteration.

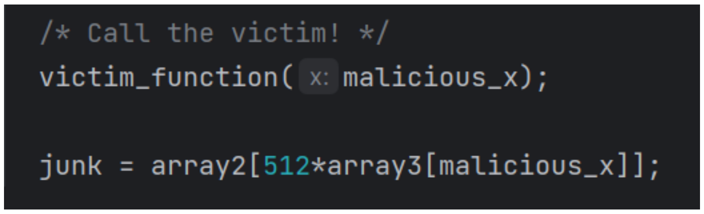
Experiment 4

In this experiment we used the same victim`s function:



```
void victim_function(size_t x){
    array1[x] = secret[x];
}
```

But used an aligned `array1` and `array3` to eliminate the usage of the offset scanner method.



```
  /* Call the victim! */
  victim_function( x: malicious_x);

  junk = array2[512*array3[malicious_x]];
```

The experiment failed as well.

References:

[1] Intel® 64 and IA-32 Architectures Optimization Reference Manual

[2] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. International Journal of Parallel Programming, 47(4):538–570, 2019

[3] Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker Zirui Neil Zhao Adam Morrison Christopher W. Fletcher

[4] https://github.com/yuvalbir10/herbert_attack_exercise