TLB Side Channel Attacks are Practical

Aditya Rohan(160053), Mohd. Arsalaan Hameed(17111024)

ABSTRACT

Traditionally side channel attacks exploit the sharing of cache among different hardware threads. These cache-based side channel attacks are real and used [to leak data] on various attacks. However modern microarchitecture is designed to mitigate these attacks by cache partitioning and other techniques. There also exist microarchitectures other than cache which is shared and can be exploited to leak data. We provide a side channel attack that exploits TLB.

Translation look-aside buffers are an unusual choice for a side-channel attack. This could be because the information provided by this side-channel is too coarse-grained for any practical purpose or maybe because of the unknown addressing functions inside the TLB. The presented attack is a new cache side-channel attack that promises to go beyond these obstacles and successfully leak data. The coarse-grained data obtained from the attack can be made fine using machine learning techniques, and the addressing functions can be reverse engineered.

1. INTRODUCTION

Modern CPU's provide great performance, some of the major factors which boost performance are on-chip caches and hyperthreading. Generally, there are three levels of caches L1/L2 private cache per core, L3 cache (shared among all core). Hyperthreading allows two different threads to run on the same core while sharing L1/L2 cache and other hardware structures. While these advances in CPU provide great performance, they also threaten security by providing a way for an attacker to leak fine-grained sensitive data. An attacker can leak fine-grained data by classical attacks like PRIME + PROBE[1], FLUSH + RELOAD[2] and EVICT + RELOAD[3] using these caches. All these side-channel attacks exploit the sharing of cache among different execution entities. To improve security researcher's and CPU designers have provided defense techniques like Intel TSX [4], Intel CAT which mitigates these attacks by partitioning cache among threads/cores. While these defense implemented in CPU, there are still some hardware structures which are still shared can be used to leak data. We try to find out what are structures other than cache which can be exploited to leak sensitive data. As documented by micro architecture providers that there are multiple level of TLB cache used to cache virtual to physical address translation. Since each process access instruction & data by virtual address, but hardware need to converts virtual to physical address. CPU uses these TLB cache to speed up execution. We first try to verify what mentioned in the Intel manual. Some properties mentioned in manual are correct but for other undocumented properties like address maping function we perform reverse engineering. We have reverse engineered L1/L2 tlb on three micro architecture HASWELL, BROADWELL, KABYLAKE.

The report is organized in the following manner: First we discuss some background on cache side-channel attacks and their defences in Section II, we then discuss attack overview in Section III. In Section IV we discuss the reverse engineering process used.

2. BACKGROUND

2.1 Cache Side-Channel Attacks

There has been a steep increase in the number of micro-architectural side-channel attacks in a short period of the last twelve to fifteen months. The major class of side-channel attacks leaks information via the shared CPU data or instruction caches.

2.1.1 PRIME+PROBE

In a P+P attack the spy process, say A, operates in three states,

- PRIME: The spy fills all cache sets with its own code or data
- IDLE: The spy waits for for the victim to access someof it's data from the cache
- The spy then tries to access its own data from the cache and measure the latency, if the victim had accessed any of the lines that spy had primed then it will have higher latency

2.1.2 FLUSH+RELOAD

The F+R attack is a variation of the P+P attack that relies on the pages share between the victim and the spy. This allows the spy to ensure which memory line is evicted from the cache hierarchy and to monitor the accesses to memory line.

3. ATTACK OVERVIEW

To exploit TLB first we need to know TLB architecture. CPU manufactures provides information that there are TLB cache and MMU cache but these structures are not fully documented. As described in Intel manual there are two levels of TLB, four TLB's for level one namely data TLB (4KB pages), data TLB (Large Pages), instruction TLB (4KB pages), instruction TLB (Large pages) and a second level

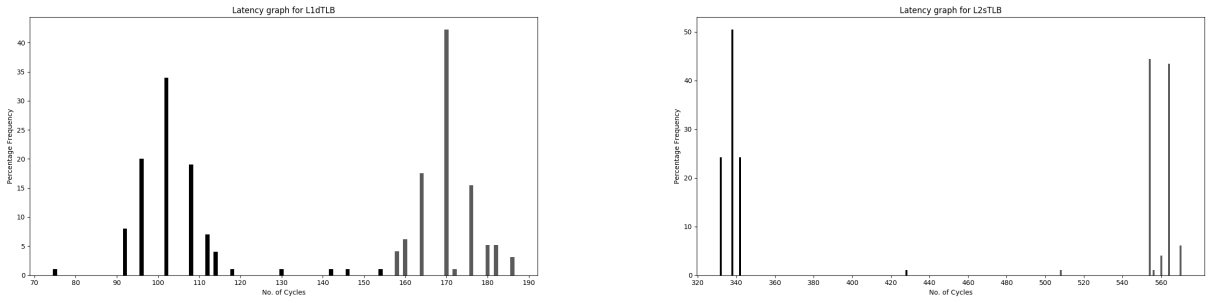

Figure 1: Latencies for L1dTLB and L2sTLBs

unified TLB (4KB pages). To perform side channel attack we need to know the characteristics of TLB, are TLB shared among threads?, are TLB partitioned?, How does the address maps to TLB cache? To know these characteristics we try to reverse engineer TLB information on Intel architectures. Section 4 describes our Reverse engineering process.

Reverse engineering requires to use performance counter which uses root privilege, But for an attack, we require to monitor change in TLB cache (hit or miss) by an unprivileged process. We have instrumented code for measuring hit/miss latency in TLB cache. Section 5 shows our effort to get latency from the unprivileged process and the figure shows the latency measured in unprivileged process.

Since the instruction TLB is not shared (the result of reverse engineering in section 4) and most of previous attack F+R, P+P, E+R exploits sharing of instruction cache, We can't attack on instruction TLB. As TLBLEED [5] has exploited temporal order of accessing in libgcrypt implementation, we have created a program that generates a similar pattern to libgcrypt. Section 6 shows our attack strategy. Section 7 we show our result on attack described in section 6.

THREAT MODEL: The attack requires an attacker to have unprivileged access to the victim machine. The attacker should be able to run an unprivileged process on the machine. Hyperthreading should be enabled and attacker can share the core with the victim process.

4. REVERSE ENGINEERING TLB ARCHITECTURE

TLB Architecture mentioned in Intel manual is given in table 1. Intel manual specifically mentions that the size and characteristics of these units are machine specific and may change in future. As Intel mentions that these characteristics are machine specific, we try to first verify these set associative for our machine. For L1 dtlb the characteristics were same but for other levels of TLB and instruction we have reverse engineered address mapping & TLB size. In previous work to reverse engineer cache mapping [6], Intel performance counter PMC are used. We also looked to Intel manual for PMC specific to TLB, among various counter these counters are useful for our purpose,

- **dtlb_load_misses.stlb_hit** [L1 TLB load misses that hit in L2 shared TLB],
- **dtlb_load_misses.misses_cause_a_walk** [L1, L2 TLB misses that cause the full page walk by MMU],
- **itlb_misses.stl_hit** [L1 itlb miss that causes a L2-stlb hit]

4.1 TLB CACHE MAPPING

Reverse engineering for different TLBs are as

L1 data TLB: To verify the sets and ways for TLB cache as mentioned in Intel manual (16 sets, 4-way associative) we allocated a large number of data pages. We created 3 scenarios, we ran a program that access Case 1: 4(w) addresses , Case 2: 5(w+1) address , Case 3: 3(w-1) addresses with stride of 16 pages each. We measure performance counter values [dtlb_load_misses.stlb_hit] for each case. For case 1 & case 3 the values are almost similar but there was a huge increase in PMC value for case 2. The addresses that we are accessing are mapping to same set and on accessing 5 addresses the miss in L1DLTB start increasing which confirm that the L1 DTLB is consist of 16 sets with 4-way associative. We ran this on Intel HASWELL, BROADWELL and KABYLAKE all showing the same result. **L1 Instruction TLB**: To verify number of sets and ways for L2 iTLB as mentioned in table 1. We follow similar approach to L1 DTLB. We allocated a huge number of code pages, we accessed 3, 4 and 5 address with stride of 16 pages. Here we get similar latency in all three cases, resulting that itlb sets & ways are not as same as described in 1. To examine TLB size we monitor performance counter for every stride between 4 to 16 pages & for each stride access address starting from 2 until we didn't get a significant increase in counter value. Figure **??** shows the eviction heatmap when accessing w address s pages apart. x axis shows stride in number of pages & y axis shows (w) number of address accessed, more lighter color shows more eviction in L1 iTLB. Lowest set number showing lightest color depict the set size & corresponding s value show number of sets. It is clear from the figure that the iTLB consist of 8 set each consist of 8 TLB entry. These results are same for intel HASWELL, BROADWELL and KABYLAKE.

L2 Unified TLB: For L2 stlb we first verify number of set and set size as given in table 1 using the same approach as used for L1 dtlb. But unlike dtlb, for stlb miss we use `dtlb_load_misses.misses_cause_a_walk` perf event to identify miss in stlb. The size was not equal to what mentioned in table 1. We tried brute force with w ways & s set and result wasn't conclusive. Previous work on reverse

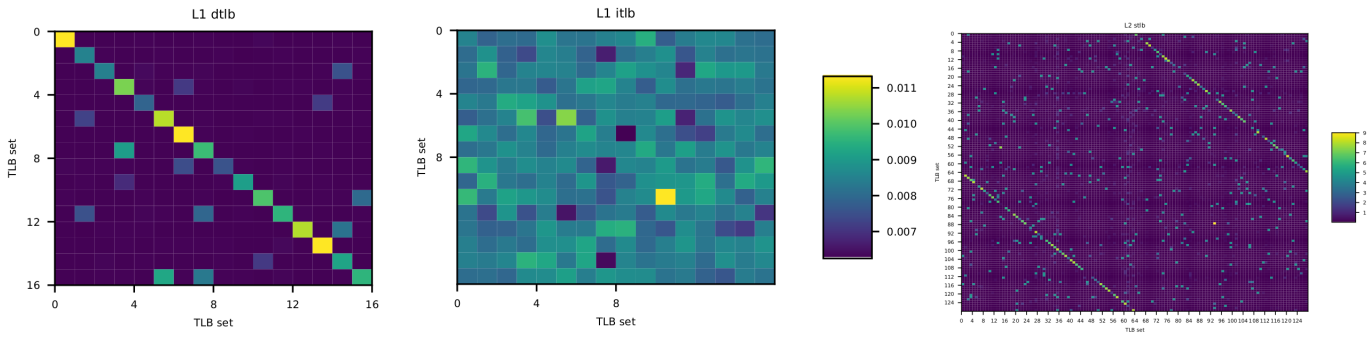

Figure 2: TLB interaction for Haswell,

engineering cache address mapping shows that intel use XOR function for address mapping in last level cache to optimize cache. On using last 7 bits [0-6] & 7 bits [7-13] of page frame number and using XOR of these 7 bits as index to stlb we got this result.

4.2 TLB INTERACTION

After finding address mapping function in itlb, dtlb & stlb, we need to know how these tlb interact with each other i.e. are they inclusive or non-inclusive? are they shared between threads? **TLB SHARING** To test sharing of TLB between hyperthreads, we need to run two threads on same core and access same set of TLB in both thread. We pin two processes on same core and access sets from 0 to max set number in one thread, while other thread also accessing some set. Figure 4.1 shows the TLB interaction between sets in two different threads. x-axis show set number accessed in thread one, y-axis shows set number accessed in second thread. Lighter color shows more eviction, depicting that both thread accessing same set. L1 dtlb is shared between threads, while itlb is not shared between the threads. For L2 stlb cache intel also uses thread id with virtual page frame number for mapping.

4.2.1 CHALLENGES

There are various challenges we faced while performing the above experiments. While verifying set and way associativity we were accessing w pages with s pages strides and measuring the performance counter event value. We weren't getting any difference as each time we ran the values different for each run for the same access. Since there was another thread running & there might be context switches happening the counts were getting varied each time. Accessing w pages just one time wasn't making any difference, upon accessing w pages thousands of time and accessing w+1 pages thousands of time then there was some difference.

5. UNPRIVILEGED MONITORING

To exploit TLB sharing and perform an attack like prime + probe [1] an unprivileged process should be able to identify between a hit or miss in a particular TLB set. For monitoring TLB miss & hit latencies we have used previous strategy like pointer chasing strategy to measure time. We have use code give in figure 1 to measure the TLB latency. We have created two evictions set for each level of tlb. For each level tlb eviction set1 contains ways + 1 address, all these address maps to same set but two of address are same and all other addresses are different making all address accessed will always get hit in that particular set. In eviction set2 also contains ways + 1 address, but all these address are different causing one access to always miss tlb. So in eviction set1 we always get all address hit in tlb but in set2 one address will always miss tlb, hence creating hit/miss difference. In code in figure one we access these eviction sets by pointer chasing strategy making these accesses serialized. The lfence instruction at the start makes all previous loads to be serialized. After accessing eviction set we put lfence instruction followed by rdtscp making sure to measure time only after all the instruction are finished and memory loads are also finished. rdtscp instruction loads current cpu cycle in rax register and also serialize instruction, so we don't need to execute cpuid after each time we access eviction set. figure 3 shows the latency graph on haswell machine. Figure 3 part a shows the latency difference on l1 dtlb part b shows the latency difference in L2 dtlb. from the latency graph, we can easily distinguish between a tlb hit or miss.

5.1 CHALLENGES

5.1.1 POINTER CHASING STRATEGY

Previous work shows a strategy to measure time in units of cpu cycles. Since today CPU executes out of order they can parallelize the order of execution of instruction if the instruction are not dependent on each other. We also have faced similar issue while accessing eviction set. We have stored all the address in eviction set in an array and then we were accessing those address from an array. This strategy leads to issue of accessing addresses in parallel and we were unable to get the difference between tlb hit & miss. Pointer chasing strategy is cited in various papers, on closely looking we notices this execution behaviour. In pointer chasing strategy we allocate eviction set address in linked list. mov (%1) (%1) will load the next address in linked list to register 1. Now the next instruction will became dependent on this one & processor have to wait for this instruction to finish before start executing another instruction. After implementing this strategy there was observable difference in eviction set1 & set2.

TLB	Size with set associativity
Instruction TLB (4-KByte Pages)	64-entries per thread, 4-way set associative
Instruction TLB (4-KByte Pages)	7-entries per thread, fully associative
Data TLB (4-KByte Pages)	64-entries, 4-way set associative
Data TLB (Large Pages)	32-entries, 4-way set associative
Second-level Unified TLB (4-KByte Pages)	512-entries, 4-way set associative

Table 1: Diffrent TLB and their size for intel i7, i5, i3 processors documented in intel manual

5.1.2 RDTSC

RDTSC instruction in x86 doesn't provide execution time of instruction at 4-5 cpu cycles granularity. If we access RDTSC instruction two times consecutively it always gives 16-20 cycles difference in both executions, it shows that RDTSC has it's on latency and it depends on cpu operating frequency. When we access only one instruction we can't determine the accurate cpu cycle latency, so we have to access multiple addresses resulting into some approximation of latency and RDTSC latency will hide by the different address access.

5.1.3 NOISE

Since many processing running on the system there was much noise while measuring latencies. Since cache is inclusive, the processes executing on other core were interfering with our process. There might be some process which thrashes a set in LLC and our allocated pages in eviction set may also map to the same set, causing the eviction of data block in L1 cache hence the noise. We have done some optimization to remove this noise. While allocating the address in eviction set we make sure the all the virtual address maps to same physical address, now these virtual address will map to just one cache line in LLC and the latency difference will mostly depend on just tlb cache. Also before measuring latency we are evicting all L1 cache by accessing a very large array. Also each time we measure latency we might get different latency chart as plotted in figure 1, since it depends on workload but we can always distinguish between hit or miss.

6. ATTACK

Previous work exploit sharing of cache, in traditional prime + probe attack attacker fills the cache with its own cache block and if victim accessed that set then attacker can know that victim has accessed this set. Since here itlb is not shared between the threads we can't attack based on instruction execution. TLBLEED [5] exploits the temporal order of access of a particular set in libgcrypt implementation. This libgcrypt implementation access same set at a different time based on encryption key. So if we can measure the accessing pattern of the victim to a particular set we can identify the key. We have instrumented our own victim binary, which generates same access pattern as generated by ligcrypt mention in TLBLEED. Victim process access function mul if key value is one and it will always access dup function. We have used pin tool to generate access pattern of the victim binary, it is similar to libgcrypt. Figure 3 shows the pattern of victim binary, this graph shows the time at which a particular set is accessed. The blue background indicates the region for add function, wherein we access the target address two times at very close

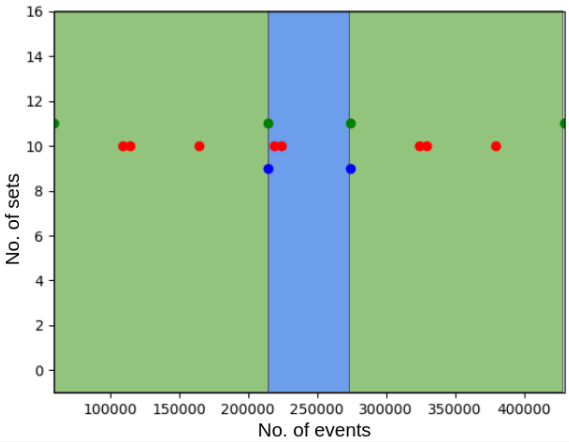

Figure 3: Results of the Pintool tracing

intervals. In contrast, the green background represents the multiply function of our crypto-library. The target address was accessed thrice within the multiply function, twice at very close intervals and then at after a gap of larger interval. This pattern is repetitive and can be used to identify the activity of the victim.

```
void encryptKey(char *key){
    for(int i=0; i<16; i++){
        int curr = 1;
        for(int j=0; j<8; j++){
            mul(key[i]);
            if((testBit[j]){
                add(key[i]);
            }
            curr *= 2;
        }
    }
    return;
}
```

To perform attack we first calibrate the latency for hit/miss in the system and generate a threshold. We pin the victim & spy process to the same physical core and executes them concurrently. On victim we are accessing an address that maps to particular tlb set(s) at some time interval. Spy process first fills that particular tlb set(s) by accessing a set of address, then it accesses same addresses again, if the victim has accessed an address that maps to same set then we can identify it by latency difference. We also turned off ASLR so that the set number we are accessing doesn't change on each execution.

ISSUES Since a victim can get a context switch and other

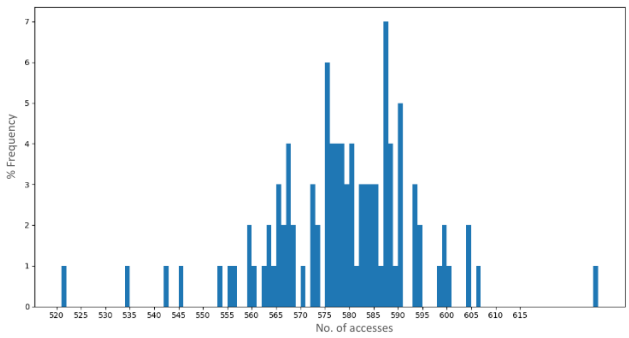

Figure 4: Attack on L2TLB

processes may get scheduled which will cause noise. The newly scheduled process may access the same set as a victim was accessing, which results into false positives.

7. EVALUATION

In our victim process, we have a key of 16 characters. dup function in code figure [victim code] will always be executed, but based on bit value mul function will be executed. Our key generates access to dup function that accesses a set around 580 times. We run both victim & spy process multiple time. Figure 4 shows the frequency that we get for the number of times a particular number of accesses were made. Our spy process wasn't always able to recognize the hit/miss due to noise from other processes. We have to calibrate again and again while performing an attack to get more accurate results. For our evaluation, we have set threshold manually and we are able to identify 65% of the time with accuracy with 10 access out of 580 access by victim process. By calculating threshold dynamically spy process was able to identify 50% of the time with accuracy within 10 access out of 580 access by the victim process. If we use a temporal access pattern we would be able to get more accurate results, we left this for future work.

8. FUTURE WORK

We have done reverse engineering of the L1 dtlb, L1 itlb & L2 stlb for page size 4-KB, but there are a separate structure with LARGE pages as documented in table 1. There are also separate structure PDE_CACHE, PML4_CACHE, PDPTE_CACHE which stores the partial translation, however for haswell performance counter is there only for PDE_CACHE. We have done some experiment on PDE_CACHE using this counter, we are in the middle of reverse engineering for LARGE page tlb & PDE_CACHE. As large pages are much less frequent use and we are exploiting the temporal pattern, we can do more accurate attack by using LARGE pages, but to verify it first we have to reverse engineer its characteristics. We left attacking using LARGE page to the future work.

In our attack code spy process was just measuring how many time a victim has accessed a particular set, but to get the original key we have to use temporal pattern. Recovering key using temporal access will be done in future work.

9. REFERENCES

[1] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *CryptographersâĂŹ Track at the RSA Conference*, pp. 1–20, Springer, 2006.

[2] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack.," in *USENIX Security Symposium*, vol. 1, pp. 22–25, 2014.

[3] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 605–622, IEEE, 2015.

[4] R. Rajwar and M. Dixon, "Intel transactional synchronization extensions," in *Intel Developer Forum San Francisco*, vol. 2012, 2012.

[5] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 955–972, 2018.

[6] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering intel last-level cache complex addressing using performance counters," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 48–65, Springer, 2015.