Mapping the Intel Last-Level Cache

Yuval Yarom[1], Qian Ge[2], Fangfei Liu[3], Ruby B. Lee[3], Gernot Heiser[2]

[1] NICTA and The University of Adelaide
yval@cs.adelaide.edu.au
[2] NICTA and UNSW
Sydney, Australia
{qian.ge,gernot.heiser}@nicta.com.au
[3] Princeton University
{fangfeil,rblee}@princeton.edu

Abstract. Modern Intel processors use an undisclosed hash function to map memory lines into last-level cache slices. In this work we develop a technique for reverse-engineering the hash function. We apply the technique to a 6-core Intel processor and demonstrate that knowledge of this hash function can facilitate cache-based side channel attacks, reducing the amount of work required for profiling the cache by three orders of magnitude. We also show how using the hash function we can double the number of colours used for page-colouring techniques.

1 Introduction

Cache-based side channel attacks have been known for more than a decade. In such attacks, the attacker forces contention on cache sets the victim uses, in order to identify victim activity in these cache sets. From this activity, the attacker can reconstruct the victim operation including recovery of encryption keys [Acıiçmez, 2007; Brumley and Hakala, 2009; Osvik et al., 2005; Percival, 2005; Zhang et al., 2012]

Until recently, these attacks have targeted the L1 cache level and were, therefore, limited to a single core. This situation, however, changed with the publication of several attacks on the last-level cache (LLC) [Irazoqui et al., 2015a; Liu et al., 2015; Oren et al., 2015]. These attacks can be applied between cores sharing the LLC.

The possibility of such attacks has been suspected for a while [Percival, 2005] and low-bandwidth channels based on the LLC have been demonstrated in the past [Ristenpart et al., 2009; Wu et al., 2012; Xu et al., 2011], promoting research into mitigation techniques. One of the commonly suggested mitigation techniques is page colouring [Bershad et al., 1994; Braun et al., 2015; Kim et al., 2012; Liedtke et al., 1997; Shi et al., 2011], which partitions the memory of the computer such that processes from different security domains never compete for the same parts of the cache. More specifically, virtual memory pages are partitioned into colours, such that memory addresses from differently coloured pages map to disjoint cache sets. By allocating page frames to different domains from disjoint colours, cache accesses from different domains cannot conflict with each other in the cache.

To mount the attack or to provide the defence, the attacker or defender needs to know the mapping of memory addresses to cache sets. In a typical set-associative cache design, the *cache-set index bits* of the physical address specify the index of the cache-set used for caching the contents of the address. The LLC design of recent Intel processors deviates from this typical design, making it harder to determine the mapping. Specifically, the Intel LLC is divided into multiple *slices*, each operating as a typical set-associative cache. Intel uses an undisclosed hash function to map memory locations to cache slices [Intel 64 & IA-32 AORM]. Without the knowledge of the hash function, attackers and defenders need to probe the memory to find which memory locations map to each cache set [Irazoqui et al., 2015a; Liu et al., 2015; Oren et al., 2015]. To facilitate this probe, such work relies on Hund et al. [2013], which recovers the hash function for a specific processor, and demonstrates that the address bits that specify the cache-set index of a memory location are not considered as inputs to the hash function. As a consequence, recovering the mapping for a single cache-set index also provides the mapping for all other cache set indices. Thus, instead of probing each of the 2,048 cache-set indices of the Intel LLC, we only need to probe one, achieving a reduction of over three orders of magnitude in the effort required for mapping the cache.

The mapping also helps page colouring, where the system divides the memory into *colours* that do not map to the same cache sets. Without knowing the hash function, page colouring can only rely on the cache-set index bits,

IACR Cryptology ePrint Archive, Report 2015/905, September 2015.

and because each virtual-memory page spans over 64 cache lines, relying only on the cache-set index bits limits page colouring to only 32 different colours. However, if we know that all of the memory locations within a memory page share the same inputs to the hash function, pages that map to different cache slices are guaranteed not to conflict on cache sets and can be used for different colours. This allows using different colours for pages that share the same cache-set index address bits, increasing the number of supported colours.

Liu et al. [2015] and Irazoqui et al. [2015b] build on Hund et al. [2013] and claim that the above property holds for processors with 2, 4 and 8 cores. However, for processors with 6 or 10 cores, the property no longer holds. As a result, applying the LLC attack on these processors requires finding the mapping for each cache-set index. Furthermore, because different locations within the same page map to different cache slices, some of the memory locations in pages that share the cache-set index bits map to the same slice, whereas others map to different slices. While lines mapped to different slices can obviously not conflict, and as such have different colour, without knowledge of the slice mapping function, we cannot determine the colour, thus reducing the number of colours we have available for partitioning cache use.

In this paper we demonstrate that measuring the memory access time from the different cores enables matching memory locations to cache slices. We use this technique to completely recover the hash function of the Xeon E5-2430 processor and show how the knowledge of the function helps both attackers and defenders. In summary, we make the following contributions:

- We present a technique for finding the cache slice used for each cache set (Section 3).
- We use the technique to reverse-engineer the hash function used in the Xeon E5-2430 processor (Section 4).
- We show how to use the information to double the number of colours usable for page colouring (Section 5.1).
- We show how the knowledge of the hash function helps reduce the effort of mapping the cache by three orders of magnitude (Section 5.2).

Concurrent work. Concurrent with our own work, İnci et al. [2015] reported recovering the hash function of a 10-core Intel processor. We have been in communication with them, but as of this writing neither team has seen the technical details of the other's work.

2 Background

2.1 Memory hierarchy

While per-core processing speed has levelled out in recent years, the growing core count keeps increasing the demands on the memory subsystem. Although technological advances improve the memory access speed, this speed has not kept pace with the demand. Processor designs, therefore, include caches that bridge the speed gap between the fast processor and the slower memory.

The cache is a small memory that stores the contents of recently used memory locations, as well as of locations the processor predicts might be required. The combination of the small cache size, its fast connection to the processor and the memory technology used allows caches to be much faster than the main memory. Thus, the cache exploits the spatial and temporal locality of memory access to enable fast access to frequently or recently used data.

A typical processor includes a hierarchy of cache levels. Caches at the top of the hierarchy, typically known as Level 1 or L1 caches, are the smallest and fastest. In modern Intel processors[4] the size of the L1 cache is 64 KiB (split into 32 KiB for data and 32 KiB for instructions), and the access time to cached data is 4 CPU cycles. The Level 2 (L2) cache has a size of 256 KiB and a latency of 7 cycles. In a multicore processor, each of the execution cores has dedicated L1 and L2 caches. Table 1 shows typical parameters for modern Intel processors.

The third level of cache is the L3 or last-level cache (LLC). The size of the LLC varies between specific processor models and ranges from 3 MiB to over 20 MiB, with latency of 26–31 CPU cycles [Intel 64 & IA-32 AORM]. The two causes for the variations in cache size are the different associativity of various processor models and the number of cache sets, which depends on the number of cache slices used. The varying access time to different cache slices causes the variations in LLC latency. (See Section 2.3.) Unlike higher-level caches, which are core-private, the LLC is shared between the cores of the processor.

[4] In this paper we only discuss the cache architecture as found in Intel Sandy Bridge and newer micro-architectures.

Level	size	Latency	Index Bits	Sets	Associativity	Line Size
L1 Data	32 KiB	4	6	64	8	64 B
L1 Instructions	32 KiB	4	6	64	8	64 B
L2	256 KiB	7	9	512	8	64 B
LLC	≥3 MiB	26–31	11	≥4,096	12–20	64 B

Table 1. Typical cache parameters in Intel processors.

2.2 How caches work

The cache stores fixed-size memory units called *lines*. Each line maps to a *cache set*. The *associativity* of a cache is the number of lines that can be stored concurrently in each set. When the associativity is n, the cache is called an n*-way set-associative cache*. Typical parameters for cache associativity are shown in Table 1

When the processor needs access to a certain memory address, it checks whether the line containing the address is currently stored in one of the ways of the corresponding set in the L1 cache. If the line is there, the access request is served from the L1 cache, a situation we refer to as an L1 cache *hit*. Otherwise, in an L1 cache *miss*, the process repeats for the next level in the memory hierarchy, until the line is found. The processor, then, uses a *cache replacement algorithm* to chose an already-cached memory line from the same cache set and evicts it from the cache. The newly-retrieved line replaces the evicted line in the cache.

In the classical cache design, the cache set used for each memory line is determined by a sequence of bits in the memory address, which we refer to as the *cache-set index*. In recent Intel processors, the L1 cache-set index is 6 bits wide, supporting $2^6 = 64$ cache sets.

2.3 The Sandy Bridge last-level cache design

Starting with the Sandy Bridge microarchitecture, Intel introduced a new design for the last-level cache. In this new design, the cache is split into multiple *slices*, one for each core. Each of the slices operates as a standard cache, indexed by 11 bits of the physical address. The cores and the cache slices are interconnected by a bi-directional ring bus.

To choose the slice a memory address maps to, the processor uses an unpublished hash function, which supposedly distributes the physical addresses uniformly among the cores [Intel 64 & IA-32 AORM]. Hund et al. [2013] describe the hash function used in the Intel i7-2600 processor. The function is a linear combination of bits 17 to 31 of the physical address. Due to memory size limitations, Hund et al. [2013] do not provide information on bits 32 and above. Liu et al. [2015] claim that when the number of cores is a power of two, the mapping of memory addresses to slices does not depend on the cache-set index bits, confirming the finding of Hund et al. [2013], but for other core counts the cache-set index bits are also used for computing the slice number.

In a recent study, Irazoqui et al. [2015b] investigate the hash function of several Intel processors, with 2, 4 and 8 cores. These findings confirm the results of earlier work, and show that the function depends only on the number of cores and remains the same between different models with the same number of cores. Irazoqui et al. [2015b] also suggest a method for recovering the non-linear function for other core counts, but do not demonstrate that the method works.

All of the past attempts to reverse the hash map rely on identifying cache conflicts between memory lines in order to find the partitions. Because memory lines with different cache-set index bits never conflict, using conflicts alone is insufficient to determine whether these memory lines map to the same slice. As such, the techniques used in past mapping attempts cannot state categorically that the cache-set index bits are not used for determining the slice that memory lines map to.

2.4 Cache-based side-channel attacks

A shared cache can be used to leak information between processes. An *attacker* process can set the cache to a known state, e.g. by filling a cache set with its own data, allow a *victim* process to execute and then measure the time to access the previously cached data to identify cache sets that have been used by the victim. This technique has been used in the past to leak potentially sensitive information including cryptographic keys [Acıiçmez, 2007; Brumley and Hakala, 2009; Osvik et al., 2005; Percival, 2005; Ristenpart et al., 2009; Zhang et al., 2012].

Because the last-level cache is shared between all the cores in the processor, it presents a higher risk than the L1 or L2 caches. However, to attack the last-level cache the attacker needs to overcome *addressing uncertainty*, which hides cache-set information.

Virtual memory is the main cause of addressing uncertainty. The virtual address space, within which processes execute, is divided into *pages*, each consisting of a contiguous sequence of addresses. Each page is mapped to an arbitrary *frame* in the physical memory.

The typical page size in the Intel architecture is 2^{12} bytes. During virtual to physical address translation, bits 12 and above, which encode the page number in the virtual address, are replaced with the physical page frame number. Bits 0–11, which encode the page offset, are preserved. In L1 caches, the cache-set index is encoded in bits 6–11. Thus virtual to physical address translation preserves the L1 cache-set index bits, allowing the attacker to know the cache-set index used for each virtual address. (See Figure 1.)

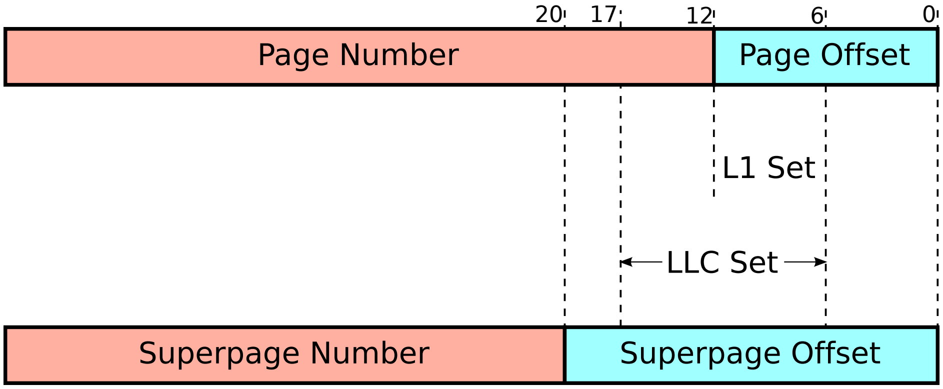

Fig. 1. Virtual memory and caches

In the LLC, bits 6–16 encode the cache-set index, hence the virtual to physical translation masks 5 of these bits. Not knowing the physical frame number, the attacker cannot determine the cache set that a memory address maps to.

To resolve this uncertainty, Liu et al. [2015] and Irazoqui et al. [2015a] suggest using *superpages*. Superpages are virtual memory pages that are larger than the typical 4 KiB pages. The Intel architecture supports several sizes of superpages, with the common ones being 2 MiB and 1 GiB. As Figure 1 demonstrates, the page offset in 2 MiB pages is large enough to cover all of the LLC set index bits. Hence, when using superpages, the LLC set index bits are preserved during virtual to physical address translation, allowing the attacker to determine the cache-set index from the superpage offset.

Another cause of addressing uncertainty is the hash function used to map memory into cache slices. Intel does not disclose the hash function and, while recent works describe the function used for some processor models, these works only cover a limited number of core counts and there is no guarantee that the same function applies to other processor models or even to other processors of the same model. Furthermore the hash function depends on all of the most significant bits of the physical address and some of these bits are hidden from the attacker even when using superpages. Hence, knowledge of the function is not sufficient for mapping memory addresses to cache slices—the attacker also needs to find out the physical address of the memory.

It is worth remembering that the attacker does not need to know the slice number a memory address maps to. All the attacker requires is to find enough memory lines that map to the same slice. To achieve this, Liu et al. [2015] perform a probing process that identifies lines mapping to the same slice by exploiting the timing difference between reading from memory and reading from the LLC. When the number of memory lines of the same slice exceeds the associativity of the cache, accessing these lines would result in a cache miss and a subsequent memory access. By measuring access times, Liu et al. [2015] identify the cache miss and uses the information to partition the memory lines of a single cache set such that each partition corresponds to a different cache slice.

The procedure described above only partitions a single cache-set index. As discussed in Section 2.3, when the number of cores in the processor is a power of two, the hash function does not depend on the cache-set index bits in the address. Consequently, partitioning one cache-set index provides the information required for partitioning all of the other indices.

For processors where the number of cores is not a power of two, this property does not hold. Without knowing the hash function, the attacker might need to partition each and every cache-set index. In the following sections we show how we recover the hash function for a 6-core processor.

3 Mapping the memory

The partitioning technique of Liu et al. [2015] does not identify the cache slice each partition is stored in. Without the identification of the slice we cannot decide whether two partitions with different set indices map to the same slice or not.

We use access timing variations to identify the core associated with each partition. Recall that in a multicore processor, each core is associated with one slice of the cache and all cores are interconnected via a bi-directional ring [Intel 64 & IA-32 AORM]. A consequence of this design is that access to data stored in the slice associated with the accessing core is faster than access to data stored in other slices.

We measure the access time to LLC slices on a server machine, featuring an Intel Xeon E5-2430 with 32 GiB of memory, running CentOS 6.6. We allocate a 16 GiB buffer consisting of 1 GiB superpages and use the procedure of Liu et al. [2015] to partition lines of the same cache-set index into slices. We then use the Linux /proc/pid/pagemap interface[5] to recover the mapping of the process virtual addresses to physical addresses. This allows us to find the physical memory address corresponding to each memory line in the buffer.

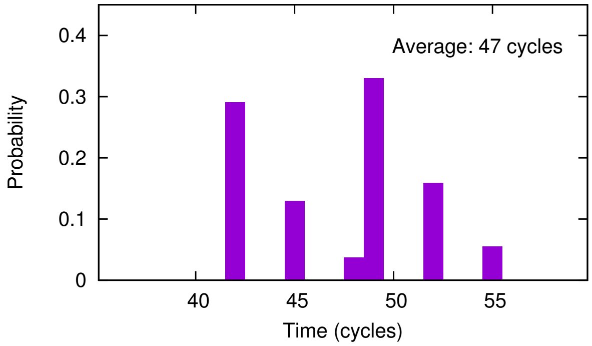

Fig. 2. The distribution and the average time to access a cached memory line.

We set the processor affinity of the measurement code to one of the cores and measure the time to access a memory line cached in the LLC. To ensure we measure LLC access times rather than L1 or L2 times, we evict the memory line we use for measuring from the L1 and the L2 caches. To evict, we repeatedly access 10 other memory lines of the same LLC set. Because these 10 memory lines share the same cache-set index bits in the LLC and because the cache-set index bits of both L1 and L2 are a subset of the cache-set index bits of the LLC, we know that these 10 memory line conflict with the memory line we measure on L1 and L2. The associativity of both L1 and L2 is 8, hence accessing 10 memory lines of the same cache set in each of these caches ensures that other lines that map to these cache sets are evicted from the L1 and the L2 caches.

[5] See https://www.kernel.org/doc/Documentation/vm/pagemap.txt

Repeating the measurements for 100,000 times gives us a distribution of the measurement results. One such distribution is shown in Figure 2, which also shows that the average access time to that specific memory line is 47 cycles. As we can see, the timing is not fixed. Repeated measurements from the same core with memory lines from the same cache set consistently show similar distributions. However, changing the memory line or the core we access from results in a different distribuion.

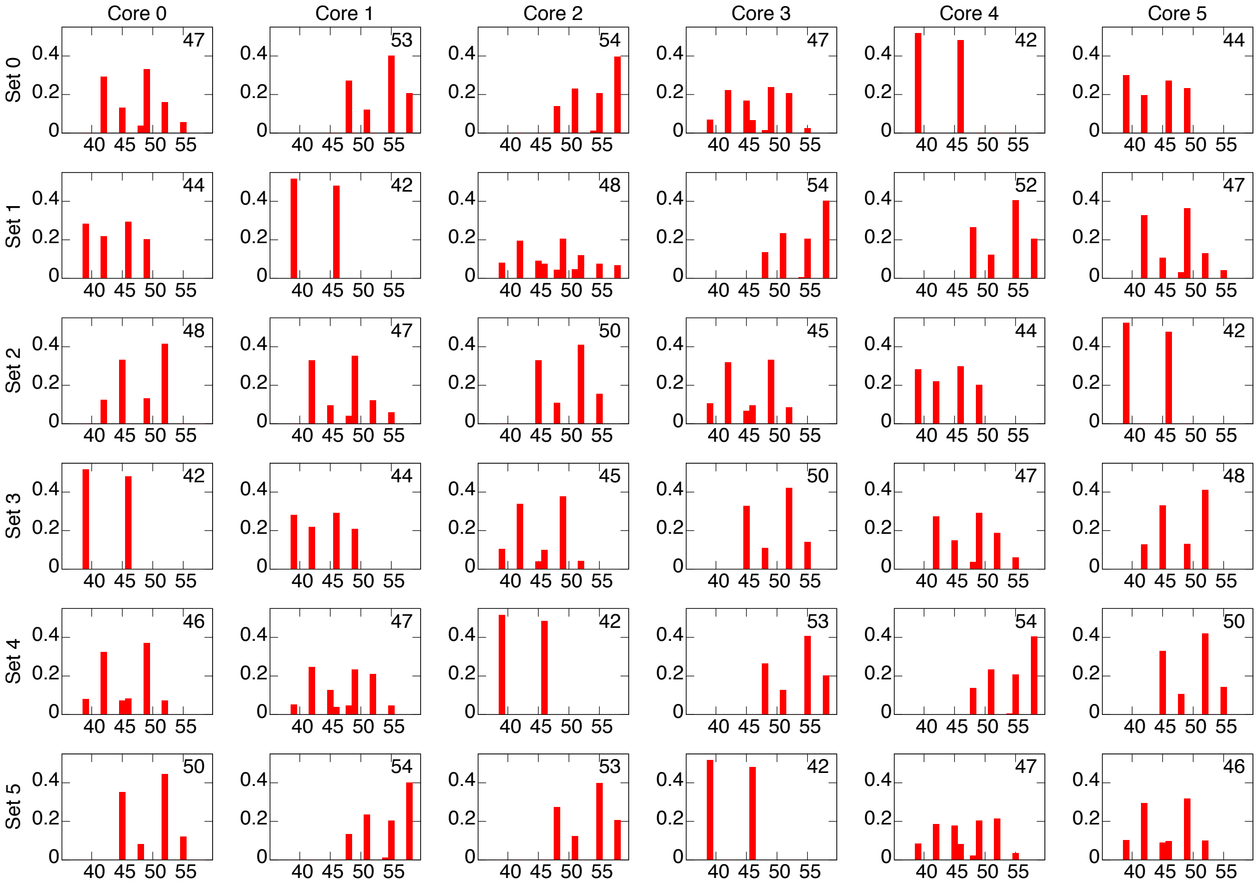

Fig. 3. The distribution and the average time to access LLC sets from the CPU cores.

Figure 3 shows the 36 distributions that we find when measuring the access times from each core to each cache set. Each graph also displays the average access time. As we can see, for each core, there is one set where the average access time is 42 cycles which is smaller than the time to access any of the other sets. Thus, by measuring the access times from each core to a cache set we can determine the core associated with the cache slice of the set.

Pipeline effects and clock granularity preclude measuring the cache latency separately from the measurement overhead. Hence the access times in Figure 3 do not match the cache latency as published by Intel (Table 1). We note, however, that the gap of 12 cycles between the fastest (42 cycles) and slowest (54 cycles) access times is significantly larger than the 5 cycles suggested in Intel 64 & IA-32 AORM.

4 The Intel Hash Function

The previous section shows how to collect the slice numbers used for a significant part of the physical memory. From this information we can reconstruct the hash function.

More formally, given an address $A = (A_0, A_1, \ldots, A_{n-1})$ we want to find a hash function $H : \{0,1\}^n \rightarrow \{0,1\}^3$ that matches the data we collected in Section 3.

Table 2 shows a sample of the data collected containing the mapping of the first 512 memory lines in the buffer. This buffer starts at physical address 0x240000000. Observing the data in the table we can see that the memory is divided into sequences of 128 memory lines (two rows in the table). Each such sequence has 21 lines of each of slices 0, 1, 2 and 3, and 22 lines of slices 4 and 5. The order of the assignment to slices varies between these sequences. Examining the rest of the collected data confirms that this pattern continues throughout the memory.

	Memory Line Offset															
	0000	0000	0000	0000	1111	1111	1111	1111	2222	2222	2222	2222	3333	3333	3333	3333
Address	0123	4567	89ab	cdef	0123	4567	89ab	cdef	0123	4567	89ab	cdef	0123	4567	89ab	cdef
0x240000000	5241	4350	2310	5241	4350	5241	3201	4350	3405	2514	2310	3405	2310	3405	3201	2514
0x240001000	0534	1425	1023	0534	1425	0534	0132	1425	4152	5043	1023	4152	1023	4152	0132	5043
0x240002000	4152	5043	5043	0132	5043	4152	4152	1023	0534	1425	1425	0132	1425	0132	0534	1023
0x240003000	3405	2514	2514	3201	2514	3405	3405	2310	5241	4350	4350	3201	4350	3201	5241	2310
0x240004000	4350	3201	5241	4350	5241	2310	4350	5241	2514	3201	3405	2514	3405	2310	2514	3201
0x240005000	1425	0132	0534	1425	0534	1023	1425	0534	5043	0132	4152	5043	4152	1023	5043	0132
0x240006000	1023	4152	4152	5043	0132	5043	5043	4152	1023	0534	0534	1425	0132	1425	1023	0534
0x240007000	2310	3405	3405	2514	3201	2514	2514	3405	2310	5241	5241	4350	3201	4350	2310	5241

Table 2. Mapping of memory lines to cache slices.

In Table 2 we see four different ways to order these 128 assignments into cache slices. Overall, we observe 128 different orderings. We can now name these sequences, e.g. by giving them identifiers from 0 to 127 (or equivalently 0x00 to 0x7f), and check how they are distributed through the memory. It turns out that there is an internal structure to these sequences that facilitates the naming.

To highlight this structure we need to focus on some of the sequences. Table 3 shows the first 64 memory lines of a sample of the sequences. It also shows the ID we assign to each of these sequences. We now describe how we assign the ID.

	Memory Line Offset																
Starting	0000	0000	0000	0000	1111	1111	1111	1111	2222	2222	2222	2222	3333	3333	3333	3333	
Address	0123	4567	89ab	cdef	0123	4567	89ab	cdef	0123	4567	89ab	cdef	0123	4567	89ab	cdef	**ID**
0x240000000	5241	4350	2310	5241	4350	5241	3201	4350	3405	2514	2310	3405	2310	3405	3201	2514	0x00
0x24003e000	2514	3405	3201	2514	3405	2514	2310	3405	4350	5241	3201	4350	3201	4350	2310	5241	0x01
0x240066000	1023	0534	0534	1425	0132	1425	1023	0534	1023	4152	4152	5043	0132	5043	5043	4152	0x2a
0x24007c000	4152	5043	1023	4152	5043	4152	0132	5043	0534	1425	1023	0534	1023	0534	0132	1425	0x02
0x240088000	5043	4152	0132	5043	4152	5043	1023	4152	1023	0534	0132	1425	0534	1425	1023	0534	0x12
0x2400c6000	3405	2514	2514	3201	2514	3405	3405	2310	5241	4350	4350	3201	4350	3201	5241	2310	0x05
0x2400f8000	4350	5241	5241	2310	5241	4350	4350	3201	2514	3405	3405	2310	3405	2310	2514	3201	0x04

Table 3. Assigning ID to sequences of memory lines.

If we look at the sequences we can notice that the sequence at address 0x24003e000 can be generated from the sequence at address 0x240000000 by simply swapping each pair of assignments. The same relationship holds for the sequences starting at 0x2400c6000 and at 0x2400f8000. Another way of expressing this relationship is saying that the memory line at offset d in one of these sequences maps to the same slice as the memory line at offset $d \oplus 1$ in the other.

Similarly, comparing the sequence at 0x240000000 and at 0x2400f8000 we see that we can generate one from the other by swapping groups of four assignments. That is, the memory line at offset d in one maps to the memory line at offset $d \oplus 4$ in the other. Another example is the sequence at offset 0x240088000, which can be generated from the sequence at offset 0x24007c000 by swapping groups of 16 assignments, i.e. offest d in one maps to the same slice as offset $d \oplus 16$ in the other.

This kind of relationship holds for any pair of sequences. For each two sequences we can always find a number I such that the memory line at offset d in one of the sequences maps to the same cache set as the memory line at offset $d \oplus I$ in the other. By choosing an arbitrary sequence, e.g. the sequence starting at address 0x240000000, as sequence 0, we can use I as an identifier for each of the sequences.

We can now define a function $ID(A)$, which returns the identifier of the sequence that contains the address A. The collected data indicates that ID is a linear function. More specifically, given any two addresses A and A' and an offset o, such that our data includes the sequences that contain the addresses A, A', $A \oplus o$ and $A' \oplus o$, we always have $ID(A) \oplus ID(A \oplus o) = ID(A') \oplus ID(A' \oplus o)$.

Using the linearity of ID we can now recover the function. For each $n \geq 13$ we find a pair of addresses A and $A \oplus 2^n$ in our data, and calculate $ID(2^n) = ID(A) \oplus ID(A \oplus 2^n)$. (We start with bit 13 because each 128 line sequence spans 8,192 $= 2^{13}$ bytes.) From the linearity of ID we have

$$ID(A) = \bigoplus_{n \geq 13} ID(A_n \cdot 2^n) \ .$$

Bits $6 \ldots 12$ of the address specify the position of the memory line containing the address within the 128-lines sequence. We can now represent the hash function $H(A)$ as a two stage function. That is, $H(A) = S(F(A))$ for $F : \{0,1\}^n \rightarrow \{0,1\}^7$ and $S : \{0,1\}^7 \rightarrow \{0,1\}^3$.

For the first stage, we use $F(A) = ID(A) \oplus (A_6 \ldots A_{12})$. We note that because ID is linear, F is also a linear function. For the second stage we use the mapping defined by sequence with ID 0. Due to the construction of ID, we know that A maps to the same slice as the memory line at index $F(A)$ in the sequnce with ID 0. Hence, we get $H(A) = S(F(A))$.

The choice of which sequence to use as sequence 0 is arbitrary. Different choices of the sequence result in different, albeit equivalent, functions. In fact, we are not limited to choosing one of the 128-lines sequences in the data. Any invertible linear combination of one of the sequences could be used. One such combination of F, which is presented in Figure 4, is:

$$\begin{aligned}
F_0(A) &= A_6 \oplus A_{12} \oplus A_{17} \oplus A_{18} \oplus A_{22} \oplus A_{24} \oplus A_{27} \oplus A_{29} \oplus A_{30} \oplus A_{32} \ , \\
F_1(A) &= A_7 \oplus A_{12} \oplus A_{13} \oplus A_{17} \oplus A_{19} \oplus A_{22} \oplus A_{23} \oplus A_{24} \oplus A_{25} \oplus \ , \\
&\quad\ A_{27} \oplus A_{28} \oplus A_{31} \oplus A_{32} \oplus A_{33} \ , \\
F_2(A) &= A_8 \oplus A_{13} \oplus A_{14} \oplus A_{18} \oplus A_{20} \oplus A_{23} \oplus A_{24} \oplus A_{25} \oplus A_{26} \oplus \ , \\
&\quad\ A_{28} \oplus A_{32} \oplus A_{33} \oplus A_{34} \ , \\
F_3(A) &= A_9 \oplus A_{14} \oplus A_{15} \oplus A_{19} \oplus A_{21} \oplus A_{24} \oplus A_{25} \oplus A_{26} \oplus A_{27} \oplus \ , \\
&\quad\ A_{29} \oplus A_{33} \oplus A_{34} \ , \\
F_4(A) &= A_{10} \oplus A_{15} \oplus A_{16} \oplus A_{20} \oplus A_{22} \oplus A_{25} \oplus A_{26} \oplus A_{27} \oplus A_{28} \oplus \ , \\
&\quad\ A_{30} \oplus A_{34} \ , \\
F_5(A) &= A_{11} \oplus A_{16} \oplus A_{17} \oplus A_{21} \oplus A_{23} \oplus A_{26} \oplus A_{27} \oplus A_{28} \oplus A_{29} \oplus A_{31} \ , \\
F_6(A) &= A_{12} \oplus A_{13} \oplus A_{14} \oplus A_{15} \oplus A_{16} \oplus A_{18} \oplus A_{19} \oplus A_{20} \oplus A_{21} \oplus \ , \\
&\quad\ A_{24} \oplus A_{25} \oplus A_{26} \oplus A_{28} \oplus A_{30} \oplus A_{31} \ .
\end{aligned}$$

The corresponding second stage S is presented below. A logical circuit that calculates an equivalent function is displayed in Figure 5.

$$\begin{aligned}
S_2(I) &= (I_0 \oplus I_5) \cdot (I_2 + I_3 \cdot (I_4 + I_5)) \ , \\
S_1(I) &= I_1 \cdot \overline{S_2(I)} \ , \\
S_0(I) &= I_0 \oplus I_1 \oplus I_2 \oplus I_3 \oplus I_4 \oplus I_6 \ .
\end{aligned}$$

	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1				
	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6
F_0			■		■	■		■			■		■				■	■					■						■
F_1		■	■	■			■	■		■	■	■	■			■		■				■	■					■	
F_2	■	■	■				■		■	■	■	■			■		■				■	■					■		
F_3	■	■				■		■	■	■	■			■		■				■	■					■			
F_4	■				■		■	■	■	■			■		■				■	■					■				
F_5				■		■	■	■	■			■		■				■	■					■					
F_6				■	■		■		■	■	■			■	■	■	■		■	■	■	■	■						

Fig. 4. The first stage (F) of the hash function, reducing the physical address bits to a 7-bit number. Shaded boxes show address bits that are XORed to compute each hash bit.

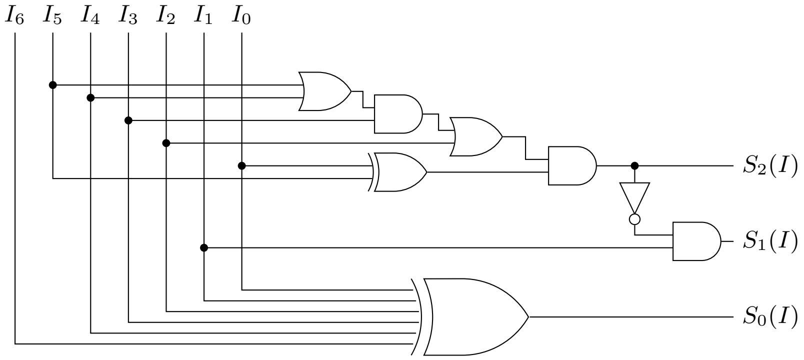

Fig. 5. Logical diagram of the Intel hash function

5 Using the slice mapping

We now turn to see how we can use the knowledge of the mapping of cache lines to cache slices. For a defender, knowing the hash function increases the number of colours that can be used for page colouring, as explained below. For an attacker, knowledge of the hash function significantly reduces the costs of mapping the cache.

5.1 Page colouring

As discussed above, page colouring is a side-channel mitigation scheme that partitions the cache between protection domains by ensuring that memory allocated to one protection domain does not map to the same cache sets as memory allocated to other domains.

The system allocates memory in page size chunks. Using small pages, of 4,096 bytes, allows the finest granularity of memory allocation. Each such page contains 64 cache lines.

Memory addresses that differ in the cache-set index bits are guaranteed to map to different cache sets. The Intel LLC uses 11 bits for the cache-set index bits, giving 2,048 possible values. With 64 cache lines per page, these 2,048 values can be divided into 32 *page colours*.

The number of colours limits the number of possible protection domains available. Furthermore, because the physical memory is divided equally between the colours, the number of colours limits the granularity of dividing memory between the various protection domains. Consequently, we would like to have as many colours as we can.

To achieve more than 32 colours we need to partition pages that share the same cache-set index, which is possible if they map to different slices. This is enabled by the linearity of S_0: Suppose we have two pages with base addresses A and A', such that A is stored in an even slice (i.e. one of slices 0, 2 and 4) whereas A' is stored in an odd slice (1, 3, or 5). Due to the linearity of S_0, we are guaranteed that for any offset o in these pages, the least significant bits of the slices $A + o$ and $A' + o$ map to will be different. Hence, we are guaranteed that addresses within these pages never conflict on cache sets.

We can, therefore, further partition pages that share the high order bits of the cache-set index into two groups based on the parity of the cache slice used to store the first address in the pages. This allows us to use 64 colours, or twice as much as were possible without knowing the hash function.

5.2 Mapping the cache

To mount the attack of Liu et al. [2015], we need to find *eviction sets* for cache sets. That is, we need to find enough memory lines that map to the same cache set to be able to evict all of the victim's data from the cache set. To find an eviction sets for every cache set, Liu et al. [2015] allocate a buffer and partitions the memory lines of the buffer to the cache sets. To implemet this partitioning we need to find both the cache-set index bits of each cache line in the buffer and the cache slice used for storing the cache line. To find the cache-set index bits we rely on using superpages for allocating the buffer. The superpage offset bits (bits 0–20 for 2 MiB superpages) include all of the cache set index bits (i.e. bits 6–16 of the address), and because the superpage offset bits are preserved during virtual to physical address translation, we can use the cache-set index bits directly from the virtual address.

To find the mapping of cache lines to cache slices, Liu et al. [2015] use a probing technique which partitions the cache lines of the same cache-set index into sets such that each set maps to a different cache slice. For processors where the number of cores is a power of two, a partition of one cache-set index also works for all other cache indices. Consequently, Liu et al. [2015] only need to use the probing step on a single cache-set index.

On processors whose number of cores is not a power of two, such as the Intel Xeon E5-2430 we investigate here, different cache set indices have different partitionings. Without knowing the mapping of memory lines to cache slices, we have to probe each cache-set index and partition memory lines in that index. With 2,048 different cache set indices, this is a lengthy process. We now show how we can use the knowledge of the hash function to significantly reduce the number of cache set indices we need to probe.

Let A be the physical address of a superpage. We know that the 21 least significant bits of A are all 0. Because we use virtual memory, we do not know any of the other bits of A. Consequently, although we know the hash function, we cannot calculate the mapping of addresses to the slices. However, if we can determine $F(A)$, we can use the linearity of F to calculate $F(A + o)$ for every offset $0 \leq o < 2^{21}$ in the superpage and from that we can find the cache slice used for every memory line in the superpage.

Given a memory buffer consisting of 2 MiB superpages, we use the probing procedure to partition the cache lines with cache-set index 0 into their respective cache sets. We then use the technique we developed in Section 3, i.e. we measure the time to access each of these cache lines from each core, to find the cache slice in which each of these lines is cached. This recovers the values of $H(A + i \cdot 2^{17})$ for $0 \leq i < 16$.

If A and A' are addresses of two superpages such that $F(A) = F(A')$, the linearity of F guarantees that $H(A + i \cdot 2^{17}) = H(A' + i \cdot 2^{17})$ for all $0 \leq i < 16$. Therefore, if we know $F(A)$ we can determine $(H(A), H(A + 2^{17}), \ldots, H(A + 15 \cdot 2^{17}))$ even without knowing the physical address A.

However, we know the sequence $(H(A), H(A + 2^{17}), \ldots, H(A + 15 \cdot 2^{17}))$ and want to find $F(A)$ from it. To be able to find $F(A)$ from the sequence, we need the have a unique value of $F(A)$ for each sequence, but this is not the case. As it turns out, for half of the superpages there is a unique sequence of mapping to slices, allowing us to determine $F(A)$ only in half the cases. For the other half, we repeat the process of probing, partitioning and mapping on memory lines with cache-set index 1, i.e. on addresses of the form $A + 64 + i \cdot 2^{17}$ The combined data uniquely determines $F(A)$.

The procedure described above assumes that we can map memory lines to cache slices. In virtualised environments this assumption is unrealistic—the processors are virtualised so we cannot determine the core ID a virtual processor executes on and, usually, do not have access to all of the cores. Without knowledge of the physical core ID and without access to all of the physical cores, we cannot use the technique of Section 3 to find the slices that memory lines map to.

If we cannot determine the slice number, the only information we have is the partitioning of the memory lines at addresses $A, A + 2^{17}, \ldots, A + 15 \cdot 2^{17}$ into slices. This means we can determine which of these addresses conflict in the same cache set, but not the slice number they map to. This partitioning does not uniquely determines $F(A)$. Instead, there are four different values of $F(A)$ that match each partitioning of the memory lines with cache set indices 0 and 1.

It turns out that when two superpages have the same partitioning of memory lines with cache set indices 0 and 1, these superpages have the same partitioning of all the other cache sets. In other words, if we determine the conflicting memory lines with cache-set indices 0 and 1 within a superpage, we can deduce the conflicting memory lines in all of the other 2,046 cache-set indices within the same superpage. Hence, even if we cannot map memory lines to cache slices, partitioning memory lines with cache set indices 0 and 1 is sufficient for recovering the partitioning of the whole superpage, which is what we require in order to create the eviction sets.

6 Conclusions

In this paper we demonstrate how to recover the hash function used in an Intel Sandy Bridge processor for mapping memory into last-level cache slices. We further analyse the function, demonstrating that knowledge of the function could benefit both side-channel attacks and their mitigation.

As we can see, familiarity with the details of micro-architectural designs can, sometimes, be useful for users. However, many of these details are often hidden from the users. More transparency on these details may offer benefits to some users. In particular, more transparency would have obviated the need for this work.

The work presented in this paper is limited to a specific processor model. Without knowledge of the internals of the processor, we cannot know how much the results of this work extend to other processors. Further experimentation is required for reversing the hash function used in processors with a different number of cores and to confirm the findings of Irazoqui et al. [2015b] that the function does not change between different processor models.

Acknowledgements

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

Bibliography

Onur Acıiçmez. Yet another microarchitectural attack: exploiting I-cache. In *ACM Computer Security Architecture Workshop (CSAW)*, Fairfax, VA, US, 2007.

Brian N. Bershad, D. Lee, Theodore H. Romer, and J. Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, October 1994.

Benjamin A. Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189*, 2015.

Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *Proceedings of the 15th Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 667–684, Tokyo, JP, December 2009.

Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy*, pages 191–205, San Francisco, CA, May 2013.

Mehmet Sinan İnci, Berk Gülmezoğlu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA key recovery in a public cloud. *IACR Cryptology ePrint Archive*, September 2015.

Intel 64 & IA-32 AORM. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, April 2012.

Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *IEEE Symposium on Security and Privacy*, San Jose, CA, US, May 2015a.

Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in Intel processors. IACR Cryptology ePrint Archive, Report 2015/690, July 2015b.

Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium*, pages 189–204, Bellevue, WA, US, August 2012.

Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS)*, Montreal, CA, June 1997.

Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, San Jose, CA, US, May 2015.

Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox–practical cache attacks in Javascript. *arXiv preprint arXiv:1502.07373*, 2015.

Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. http://www.cs.tau.ac.il/~tromer/papers/cache.pdf, November 2005.

Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, CA, 2005.

Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 199–212, Chicago, IL, US, 2009.

Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 194–199, HK, June 2011.

Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium*, Bellevue, WA, US, 2012.

Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *ACM Workshop on Cloud Computing Security*, pages 29–40, 2011.

Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 305–316, Raleigh, NC, US, 2012.