Systematic Reverse Engineering of Cache Slice Selection in Intel Processors

Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar

Worcester Polytechnic Institute, Worcester, MA, USA
{girazoki,teisenbarth,sunar}@wpi.edu

Abstract. Dividing last level caches into slices is a popular method to prevent memory accesses from becoming a bottleneck on modern multicore processors. In order to assess and understand the benefits of cache slicing in detail, a precise knowledge of implementation details such as the slice selection algorithm are of high importance. However, slice selection methods are mostly unstudied, and processor manufacturers choose not to publish their designs, nor their design rationale. In this paper, we present a tool that allows to recover the slice selection algorithm for Intel processors. The tool uses cache access information to derive equations that allow the reconstruction of the applied slice selection algorithm. Thereby, the tool provides essential information for performing last level cache attacks and enables further exploration of the behavior of modern caches. The tool is successfully applied to a range of Intel CPUs with different slices and architectures. Results show that slice selection algorithms have become more complex over time by involving an increasing number of bits of the physical address. We also demonstrate that among the most recent processors, the slice selection algorithm depends on the number of CPU cores rather than the processor model.

Keywords: Cache slices, Intel, last level cache, Prime and Probe.

1 Introduction

The rapid increase in transistor densities over the past few decades brought numerous computing applications, previously thought impossible, into the realm of everyday computing. With dense integration and increased clock rates, heat dissipation in single core architectures has become a major challenge for processor manufacturers. The design of multicore architectures has been the method of choice to profit from further advances in integration, a solution that has shown to substantially improve the performance over single-core architectures.

Despite its many advantages, multicore architectures are susceptible to suffering under bandwidth bottlenecks especially when more and more cores are packed into a high-performance system. One clear example can be observed in Symmetric MultiProcessing (SMP) systems that use a shared Last Level Cache (LLC) to reduce off-chip memory requests. LLC contention can create a bandwidth bottleneck when more than one core attempts to access the LLC simultaneously. In the interest of mitigating LLC access latencies, modern multicore processors divide the LLC into pieces, usually referred to as banks or *slices*. Cache slices can be accessed in parallel, decreasing the probability of LLC access delays due to resource unavailability.

The slice that a memory block occupies is not chosen at random. Instead, a deterministic slice selection algorithm decides the slice that each memory block occupies based on its physical address. This algorithm, to the best of our knowledge, is not revealed by hardware designers such as Intel, making reverse engineering of the slice selection algorithm a requirement for a number of scientific problems:

Side channel attackers aiming to extract information in an efficient manner from LLC accesses through probing need to know how the physical address is mapped to a slice. For instance, *Prime+Probe* was used in [19] to recover sensitive data from different slices in the LLC. As opposed to earlier works which focused on upper level caches, see e.g., [13, 14], the LLC probing attacks provide much more accurate and therefore accelerated attacks [12, 32, 20, 19, 24]. In another recent work [18], cache timing was used to bypass address space layout randomization (ASLR) protections to discover kernel process addresses. ASLR has served as a

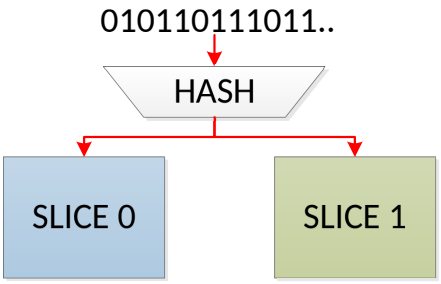

Fig. 1. A hash function based on the physical address decides whether the memory block belongs to slice 0 or 1.

strong countermeasure to prevent attackers from exploiting discovered vulnerabilities by rendering the target address unpredictable. Indeed, a significant contribution of the work in [18] is in reverse engineering the hash function used for slice selection on the Intel Sandy Bridge family of processors. For such attacks, i.e. probing the LLC for recovering cryptographic keys and other sensitive information and for bypassing protections such as ASLR, it becomes essential to figure out the slice selection algorithm.

The slice selection algorithm is also interesting for marketplace competitors that wish to implement a similar hash selection algorithm or perhaps make it more secure or unpredictable. The slice selection algorithms we have reverse engineered in this work for our selected platforms are all linear and therefore permit systematic recovery. If the slice selection algorithm is considered as a security feature, it needs to be made public and studied under the lens of security. Only then we may have any chance of constructing a practical slice selection function that can withstand reverse engineering.

Finally, knowing the details of the slice selection algorithm is crucial to improve our understanding of contemporary microarchitectural development. For instance, the slice selection algorithm can also be useful for building more accurate cache simulators, e.g. `Dinero` [2] and `SMPCache` [11]. These simulators emulate the behavior of the entire cache hierarchy in existing multicore systems. In order to more accurately simulate the LLC evictions and loads the slice selection algorithm is needed. Hidden functionalities prevent introspection and further optimizations from being discovered.

In this work we present a tool that reverse engineers the undocumented slice selection algorithm for Intel processors based on the *Prime+Probe* side channel technique. The applicability was tested by profiling a wide selection of different Intel architectures. In summary, in this work:

- We introduce a tool to recover both the necessary memory blocks to fill each slice in the LLC and the slice selection algorithm implemented by the processor. In contrast to the work in [18] we do not differentially probe individual bits to recover the hash functions, but rather profile the architecture first, and then *systematically* solve for the hash function equations. We eliminate the trial and error from this reverse engineering work. Thus, the automated tool we propose allows efficient recovery of the slice selection algorithm for arbitrary processors.
- We validate the proposed tool by profiling a wide selection of architectures and processor families. Specifically, in our validation work we were able to recover the slice selection algorithm for 6 platforms including Intel's Ivy Bridge, Nehalem and Haswell families with Intel Xeon, i5, and i7 processors. Previously, in [18] only Sandy Bridge's hash function was recovered.

Outline: We summarize the relevant background material in Section 2. The reverse engineering algorithm and the tool is introduced in Section 3. The experiment setup and results are presented in Sections 4 and 5. Finally, the conclusions are drawn in Section 6.

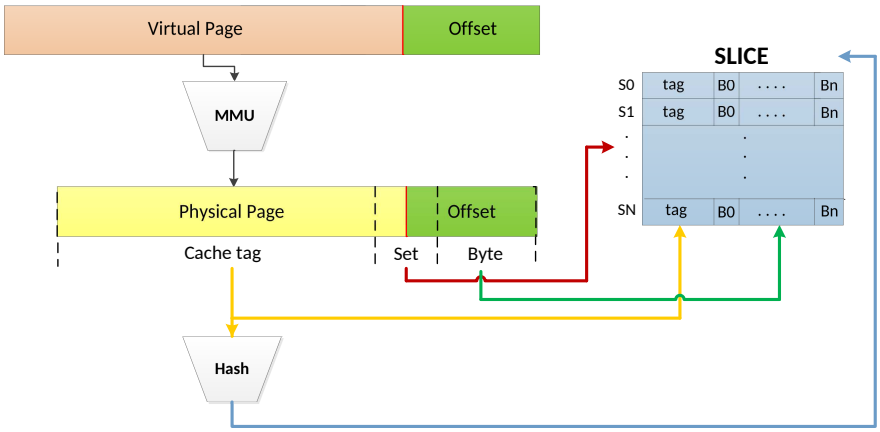

Fig. 2. Last level cache addressing methodology for Intel processors. Slices are selected by the tag, which is given as the MSBs for the physical address.

2 Background

In this section we give a brief overview of the background needed to understand the new tool presented in this work. After detailing on cache slices and LLC side channel attacks, a short explanation of cache mapping is provided.

2.1 LLC Cache Slices

Recent SMP microarchitecures divide the LLC into slices with the purpose of reducing the bandwidth bottlenecks when more than one core attempts to retrieve data from the LLC at the same time. The number of slices that the LLC is divided into usually matches the number of physical cores. For instance, a processor with s cores divides the LLC into s slices, decreasing the probability of resource conflict while accessing it. However, each core is still able to access the whole LLC, i.e., each core can access every slice. Since the data will be spread into s "smaller caches" it is less likely that two cores will try to access the same slice at the same time. In fact, if each slice can support one access per cycle, the LLC does not introduce a bottleneck on the data throughput with respect to the processors as long as each processor issues no more than one access per cycle. The slice that a memory block is going to occupy directly depends on its own physical address and a non-public hash function, as in Figure 1.

Performance optimization of sliced caches has received a lot of attention in the past few years. In 2006, Cho et al. [16] analyzed a distributed management approach for sliced L2 caches through OS page allocation. In 2007, Zhao et al. [34] described a design for LLC where part of the slice allocates core-private data. Cho et al [22] describe a two-dimensional page coloring method to improve access latencies and miss rates in sliced caches. Similarly, Tam et al. [30] also proposed an OS based method for partitioning the cache to avoid cache contention. In 2010 Hardavellas et al. [17] proposed an optimized cache block placement for caches divided into slices. Srikantaiah et al. [29] presented a new adaptive multilevel cache hierarchy utilizing cache slices for L2 and L3 caches. In 2013 Chen et al. [15] detail on the approach that Intel is planning to take for their next generation processors. The paper shows that the slices will be workload dependent and that some of them might be dynamically switched off for power saving. In 2014 Kurian et al. [23] proposed a data replication protocol in the LLC slices. Ye et al. [33] studied a cache partitioning system treating each slice as an independent smaller cache.

However, only very little effort has been put into analyzing the slice selection hash function used for selecting the LLC slice. A detailed analysis of the cache performance in Nehalem processors is described in [26] without an explanation of cache slices. The LLC slices and interconnections in a Sandy Bridge

microarchitecture are discussed in [10], but the slice selection algorithm is not provided. In [9], a cache analysis of the Intel Ivy Bridge architecture is presented and cache slices are mentioned. However, the hash function describing the slice selection algorithm is again not described, although it is mentioned that many bits of the physical address are involved. Hund et al. [18] were the only ones describing the slice selection algorithm for a specific processor, i.e, the i7-2600 processor. They recover the slice selection algorithm by comparing Hamming distances of different physical addresses. Finally, Irazoqui et al. [19] used a side channel technique involving LLC slices to recover an AES key. Their attack requires rudimentary understanding of the slice selection algorithm to succeed.

LLC Side Channel Attacks: One important advantage of knowing the behavior of the LLC is the ability to implement side channel attacks across cores, utilizing the LLC as a covert channel. Some techniques like *Flush+Reload* do not require the knowledge of the slice selection algorithm, as proved by Yarom et al. [32], Benger et al. [12] or Irazoqui et al. [20] who recover RSA, ECDSA and AES keys respectively. However, other techniques like *Prime+Probe* (also utilized in the tool presented in this work), require a certain knowledge about the slice selection method, as stated by Hund et al. [18], Irazoqui et al. [19] or Liu et al. [24]. These studies show that *Prime+Probe* attacks can bypass the ASLR or recover El Gamal and AES keys across cores. Therefore, the knowledge of the slice selection algorithm is advantageous when implementing *Prime+Probe* LLC side channel attacks.

Cache Simulators: With more and more cores being added to modern multicore processors, cache simulators become an important tool to identify bottlenecks and potential optimizations. In this sense, many cache simulators that analyze cache misses have been proposed. For instance, Pieper et al. [27] presented a cache simulator for heterogeneous multiprocessors. Later, Tao et al. [31] proposed a cache simulator aiming at detecting bottlenecks. In 2008, Jaleel et al. [21] improved over the previous works and proposed a cache hit and miss simulator when many concurrent applications are running in multilevel caches. A few years later, Mittal et al. [25] implemented a multicore cache simulator taking into account page sharing features. More recently, Razaghi et al [28] proposed a simulator to profile the cache performance caused by host compilation processes. Two other popular simulators for cache behavior on modern CPUs are the `Sniper` Multicore Simulator and `Dinero`. Both tools allow to predict the behavior of applications in terms of timing and/or memory performance.

To the best of our knowledge, none of these simulators consider the effect of the cache slice selection algorithm in their measurements, probably because these have not been made public. We believe this is crucial information when identifying cache hits or misses, since those are directly affected by the slice in which the data resides. In consequence, the tool presented in this work can provide valuable information that allows to make existing simulators more accurate.

2.2 Virtual Address Translation and Cache Addressing

Memory accesses require a translation from virtual addresses to physical addresses. During the translation, the virtual address is divided into two fields: the *offset* field and the *page* field. The offset field is the same for both the physical and virtual addresses, whereas the page field differs and requires translation. The length of the offset field directly depends on the page size. If the page size is p bytes, the offset corresponds to the lower $\log_2(p)$ bits of the virtual address. Regular pages usually have a size of 4KB whereas huge size pages usually have a size of 2MB. Thus, the offset field has a length of 12 and 21 bits, respectively.

Cache Addressing: The set that a memory block occupies in the cache is decided by its physical address. Small caches like the L1 cache can usually be covered with just the regular page offset field. In contrast, bigger caches usually use portions of the page field to decide the position that each data block occupies in the cache. In general, a m-way (and k sets) set associative cache with c byte cache lines:

- Uses the least significant $\log_2(c)$ bits of the physical address to address a byte or word within a cache line.
- Uses the following $\log_2(k)$ bits of the physical address to select the set in the cache
- Utilizes the remaining bits as a tag for comparison purposes when looking for data in the cache

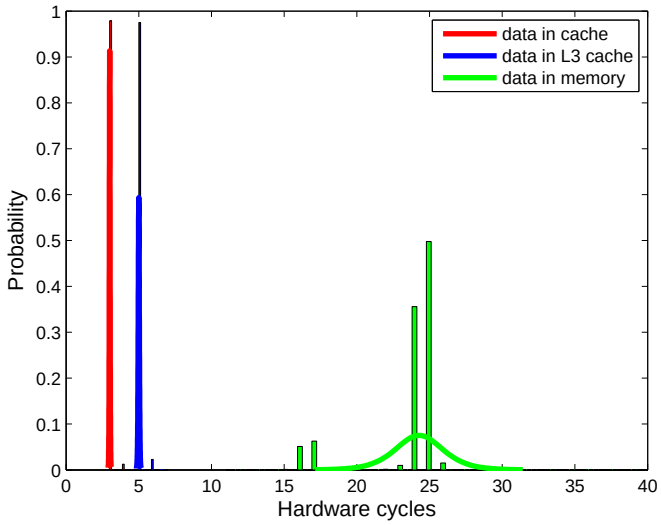

Fig. 3. Reload timing distribution when a memory block is loaded from the: L1 cache, L3 cache and memory.The experiments were performed in an Intel i5-3320M.

If $\log_2(c)+\log_2(k)$ exceed the offset field, the user does not have control over the set in which his data is going to reside. For the LLC, regular page offsets are not enough to control the locations, therefore we utilize huge size pages to increase the offset length to 21 bits. As we will see in Section 5, typical values for c and k are 64 and 2048 in last level caches in modern Intel processors. Even though the last level cache usually has more than 2048 sets, it is also divided into more than one slice, with typically 2048 sets in each slice. Usually the lower $c + k$ bits do not play any role in the slice selection process.

3 Reverse Engineering the Slice Selection Algorithm

This section describes the methodology applied to reverse engineer the slice selection algorithm for specific Intel processors. Note that the method can be used to reverse engineer slice selection algorithms for other Intel processors that have not been analyzed in this work. To the best of our knowledge, this slice selection hash function is not public. This work solves the issue by:

- Generating data blocks at slice-colliding addresses to fill a specific set. Access timings are used to determine which data is stored in the same slice.
- Using the addresses of data blocks identified to be co-residing in slices to generate a system of equations. The resulting equation systems can then be solved to identify the slice selection algorithm implemented by the processor.
- Building a scalable tool, i.e, proving its applicability for a wide range of different architectures.

3.1 Probing the Last Level Cache

As stated in Section 2, the shared last level caches in SMP multicore architectures are usually divided into slices, with an unknown hash function that determines the slice. In order to be able to reverse engineer this hash function, we need to recover addresses of data blocks co-residing in a set of a specific slice. The set where a data block is placed can be controlled, even in the presence of virtual addressing, if huge size pages are used. Recall that by using 2MB huge size pages we gain control over the least significant 21 bits of the

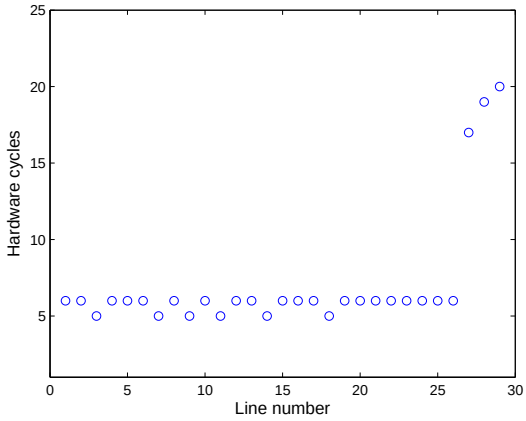

Fig. 4. Generating additional memory blocks until a high reload value is observed, i.e., the monitored block is evicted from the LLC. The experiments were performed in an Intel i5-3320M.

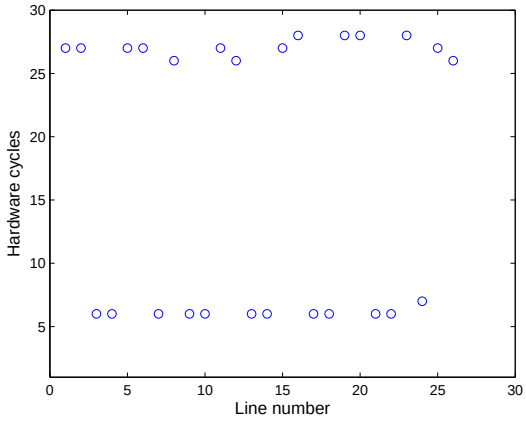

Fig. 5. Subtracting memory blocks to identify the m blocks mapping to one slice in an Intel i5 3320-M. Low reload values indicate that the line occupies the same slice as the monitored data.

physical address, thereby controlling the set in which our blocks of data will reside. Once we have control over the set a data block is placed in, we can try to detect data blocks co-residing in the same slice. Co-residency can be inferred by distinguishing LLC accesses from memory accesses. In order to detect this distinction we will apply a method based on the *Prime+Probe* side channel attack. However, we perform additional steps to generate the set colliding memory blocks and the equations. In a nutshell, the *Prime+Probe* side channel attack consists of two steps:

- **Priming the LLC Cache:** The first step is to generate one (or more) memory block(s) that will reside in a *known* set in the last level cache. Given the number of slices s present in the computer being monitored, the last level cache is addressed as if it had a size equal to $\mathsf{size}(LLC)/s$. Assume, as in Section 2, that each slice takes 2MB of the last level cache. This is a common choice, as we will see in Section 5. Since Intel processors have 64 byte cache lines (i.e, the first 6 bits are used to decide a byte in the cache line), the following 10 bits $(6-16)$ of the physical address decide the set that the memory block will occupy in the LLC (see Figure 2).
- **Reloading the Data:** The second step is to reload the data generated in the first step. If the data still resides in the LLC (and therefore has not been evicted), we see a low reload time. If the data has to be fetched from the memory (and therefore has been evicted), we will see a higher reload time. The timing difference between L1, L3, and memory accesses for an Intel-3350M can be observed in Figure 3, where L1 cache accesses take around 3 cycles, L3 cache accesses take around 6 cycles, and memory accesses take from 15 to 25 cycles.

3.2 Identifying m Data Blocks Co-Residing in a Slice

After identifying accesses from the LLC, we need to identify the m memory blocks that fill each one of the slices for a specific set. Note that we are *not* doing a pure prime and probe protocol, since we still do not know the memory blocks that collide in a specific set. In order to achieve this goal we perform the following steps:

- **Step 1:** Prime one memory block b_0 in a set in the LLC

- **Step 2:** Access several additional memory blocks $b_1, b_2, \ldots, b_n$ that reside in the same set, but may reside in a different slice, in order to fill the slice where b_0 resides.
- **Step 3:** Reload the memory block b_0 to check whether it still resides in the last level cache or in the memory. A high reload time indicates that the memory block b_0 has been evicted from the slice. Therefore we know that the required m memory blocks are part of the accessed additional memory blocks $b_1, b_2, \ldots, b_n$.
- **Step 4:** Subtract one of the accessed additional memory blocks b_i and repeat the prime and probe protocol. If b_0 still resides in memory, b_i does not reside in the same slice. If b_0 resides in the cache, it can be concluded that b_i resides in the same cache slice as b_0.

Steps 2 and 3 can be graphically seen in Figure 4, where additional memory blocks are generated until a high reload time is observed, indicating that the monitored block b_0 was evicted from the LLC cache. Step 4 is also graphically presented in Figure 5 where, for each additional block, it is checked whether it affects the reload time observed in Figure 4. If the reload time remains high when one of the blocks b_i is no longer accessed, b_i does not reside in the same slice as the monitored block b_0.

3.3 Generating Equations Mapping the Slices

Once m memory blocks that fill one of the cache slices have been identified, we generate additional blocks that reside in the same slice to be able to generate more equations. The approach is similar to the previous one:

- Prime the m memory blocks that fill one slice in a set in the LLC
- Access, one at a time, additional memory blocks that reside in the same set, but may reside in a different slice
- Reload the m memory blocks $b_1, b_2, \ldots, b_m$ to check whether they still reside in the LLC or in memory. Again, a high reload time indicates that one of the memory blocks has been evicted from the slice. Hence, the additional memory block also resides in the same cache slice.
- Once a sufficiently large group of memory blocks that occupy the same LLC slice has been identified, we get their physical addresses to construct a matrix P_i of equations, where each row is one of the physical addresses mapping to the monitored slice.

The equation generation stage can be observed in Figure 6, where 4000 additional memory blocks occupying the same set were generated. Knowing the m blocks that fill one slice, accessing an additional memory block will output a higher reload value if it resides in the same slice as b_0 (since it evicts b_0 from the cache).

Handling Noise: We choose a detection threshold in such a way that we most likely *only* deal with false negatives, which do not affect correctness of the solutions of the equation system. As it can be observed in Figure 6, there are still a few values that are not clearly identified (i.e, those with reload values of 10-11 cycles). By simply not considering these, false positives are avoided and the resulting equation system remains correct.

3.4 Recovering the Hash Function

The mapping of a memory block to a specific slice in LLC cache is based on its physical address. A hash function $H(p)$ takes the physical address p as an input and returns the slice the address is mapped to. We know that H maps all possible p to s outputs, where s is the number of slices for the processor.

$$H : \{0,1\}^{\lceil \log_2 p \rceil} \xrightarrow{h} \{0,1\}^{\lceil \log_2 s \rceil}$$

The labeling of these outputs is arbitrary. However, each output should occur with a roughly equal likelihood, so that accesses are balanced over the slices. We model H as a function on the address bits. In fact, as we will see, the observed hash functions are linear in the address bits p_i. In such a case we can model H as a

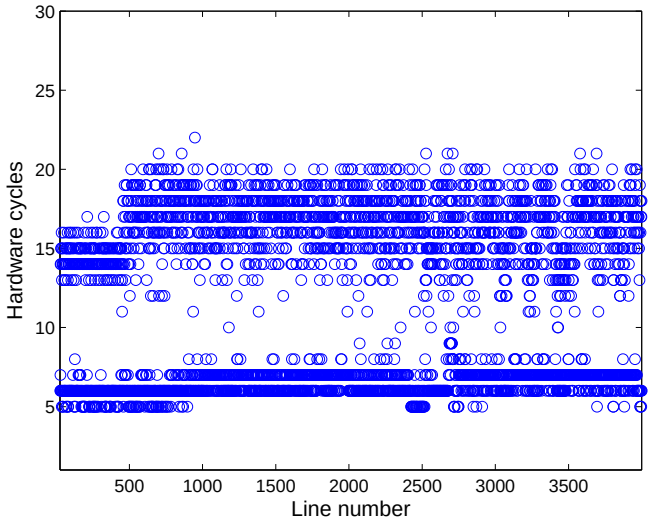

Fig. 6. Generating equations mapping one slice for 4000 memory blocks in an Intel i5 3320-M. High reload values indicate that the line occupies the same slice as the monitored data.

concatenation of linear Boolean functions $H(p) = H_0(p)||\ldots||H_{\lceil \log_2 s \rceil - 1}(p)$, where $||$ is concatenation. Then, H_i is given as

$$H_i(p) = h_{i,0}p_0 + h_{i,1}p_1 + \ldots h_{i,l}p_l = \sum_{0}^{l} h_{i,j}p_j.$$

Here, $h_{i,j} \in \{0,1\}$ is a coefficient and p_j is a physical address bit. The steps in the previous subsections provide addresses p mapped to a specific slice, which are combined in a matrix P_i, where each row is a physical address p. The goal is to recover the functions H_i, given as the coefficients $h_{i,j}$. In general, for linear systems, the H_i can be determined by solving the equations

$$P_i \cdot \hat{H}_i = \hat{0}, \tag{1}$$

$$P_i \cdot \hat{H}_i = \hat{1} \tag{2}$$

where $\hat{H}_i = (h_{i,0}, h_{i,1}, \ldots, h_{i,l})^T$ is a vector containing all coefficients of H_i. The right hand side is the ith bit of the representation of the respective slice, where $\hat{0}$ and $\hat{1}$ are the all zeros and all ones vectors, respectively. Note that finding a solution to Equation (1) is equivalent to finding the kernel (null space) of the matrix P_i. Also note that any linear combination of the vectors in the kernel is also a solution to Equation (1), whereas any linear combination of a particular solution to Equation (2) and any vector in the kernel is also a particular solution to Equation (2). In general:

$$\hat{h} \in \ker P_i \iff P_i \cdot \hat{h} = \hat{0}$$
$$\forall \hat{h_1}, \hat{h_2} \in \ker P_i \quad \hat{h_1} + \hat{h_2} \in \ker P_i$$
$$\forall \hat{h_1}, \hat{h_2} \mid P_i \cdot \hat{h_1} = \hat{1}, \hat{h_2} \in \ker P_i : \quad P_i \cdot (\hat{h_1} + \hat{h_2}) = \hat{1}$$

Recall that each equation system should map to $x = \lceil \log_2(s) \rceil$ bit selection functions H_i. Also note that we cannot infer the labeling of the slices, although the equation system mapping to slice 0 will never output a solution to Equation (2). This means that there is more than one possible solution, all of them valid, if the number of slices is greater than 2. In this case, $\binom{2^x-1}{x}$ solutions will satisfy Equation (1). However, any combination of x solutions is valid, differing only in the label referring to each slice.

Table 1. Comparison of the profiled architectures

Processor	Architecture	LLC size	Associativity	Slices	Sets/slice
Intel i5-650 [5]	Nehalem	4MB	16	2	2048
Intel i5-3320M [3]	Ivy Bridge	3MB	12	2	2048
Intel i5-4200M [4]	Haswell	3MB	12	2	2048
Intel i7-4702M [8]	Haswell	6MB	12	4	2048
Intel Xeon E5-2609v2 [6]	Ivy bridge	10MB	20	4	2048
Intel Xeon E5-2640v3 [7]	Haswell	20MB	20	8	2048

We have only considered linear systems in our explanation. In the case of having a nonlinear system (i.e., the number of slices is not a power of two), the system becomes non-linear and needs to be re-linearized. This can be done by expanding the above matrix P_i with the non-linear terms $P_{i_{linear}}|P_{i_{nonlinear}}$ and solve Equations (1) and (2). Note that the higher the degree of the non-linear terms, the bigger number of equations that are required to solve the system. Therefore, even non-linear systems are susceptible to be recovered with this technique, as long as the number of equations required for solving it can be obtained.

This tool can also be useful in cases where the user can not determine the slice selection algorithm, e.g., due to a too low number of derived equations. Indeed, the first step that this tool implements is generating the memory blocks that co-reside in each of the slices. This information can already be used to mount a side channel attack.

3.5 Applicability in other processors

The applicability of the presented method is based on the inclusiveness of the LLC with the upper level caches. In fact, *Prime+Probe* like techniques would be difficult to implement on exclusive last level caches, since we will need to fill the upper level caches before reaching the LLC. That is the reason why this methodology can not be directly applied to AMD processors. Therefore the method can be only applied in processors that divide an inclusive LLC in slices and which have the ability of using huge size pages.

4 Experiment Setup

In this section we describe our experiment setup. In order to test the applicability of our tool, we implemented our slice selection algorithm recovery method in a wide range of different computer architectures. The different architectures on which our tool was tested are listed in Table 1, together with the relevant parameters. Our experiments cover a wide range of slice sizes as well as different architectures.

All architectures except the Intel Xeon e5-2640 v3, were running Ubuntu 12.04 LTS as the operating system, whereas the last one used Ubuntu 14.04. Ubuntu, in root mode, allows the usage of huge size pages. The huge page size in all the processors is 2MB [1]. We also use a tool to obtain the physical addresses of the variables used in our code by looking at the `/proc/PID/pagemap` file. In order to obtain the slice selection algorithm, we profiled a single set, i.e, set 0. However, in all the architectures profiled in this paper, we verified that the set did not affect the slice selection algorithm. This might not be true for all Intel processors.

The experiments cover wide selection of architectures, ranging from Nehalem (released in 2008) to Haswell (released in 2013) architectures. The processors include laptop CPUs (i5-3320, i5-4200M and i7-4200M), desktop CPUs (i5-650) and server CPUs (Xeon E5-2609v2, Xeon E5-2640v3), demonstrating the viability of our tool in a wide range of scenarios.

As it can be seen, in the entire set of processors that we analyzed, each slice gets 2048 sets in the L3 cache. Apparently, Intel designs all of its processors in such a way that every slice gets all 2048 sets of the LLC. Indeed, this is not surprising, since it allows to use cache addressing mechanisms independent from the size or the number of cores present in the cache. This also means that 17 bits are required from the physical

Table 2. Slice selection hash function for the profiled architectures

Processor	Architecture	Solutions	Slice selection algorithm
Intel i7-2600 [18]	Sandy Bridge		$p_{18} \oplus p_{19} \oplus p_{21} \oplus p_{23} \oplus p_{25} \oplus p_{27} \oplus p_{29} \oplus p_{30} \oplus p_{31}$ $p_{17} \oplus p_{19} \oplus p_{20} \oplus p_{21} \oplus p_{22} \oplus p_{23} \oplus p_{24} \oplus p_{26} \oplus p_{28} \oplus p_{29} \oplus p_{31}$
Intel i650	Nehalem	1	p_{17}
Intel i5-3320M	Ivy Bridge	1	$p_{17} \oplus p_{18} \oplus p_{20} \oplus p_{22} \oplus p_{24} \oplus p_{25} \oplus p_{26} \oplus p_{27} \oplus p_{28} \oplus p_{30} \oplus p_{32}$
Intel i5-4200M	Haswell	1	$p_{17} \oplus p_{18} \oplus p_{20} \oplus p_{22} \oplus p_{24} \oplus p_{25} \oplus p_{26} \oplus p_{27} \oplus p_{28} \oplus p_{30} \oplus p_{32}$
Intel i7-4702M	Haswell	3	$p_{17} \oplus p_{18} \oplus p_{20} \oplus p_{22} \oplus p_{24} \oplus p_{25} \oplus p_{26} \oplus p_{27} \oplus p_{28} \oplus p_{30} \oplus p_{32}$ $p_{18} \oplus p_{19} \oplus p_{21} \oplus p_{23} \oplus p_{25} \oplus p_{27} \oplus p_{29} \oplus p_{30} \oplus p_{31} \oplus p_{32}$
Intel Xeon E5-2609 v2	Ivy Bridge	3	$p_{17} \oplus p_{18} \oplus p_{20} \oplus p_{22} \oplus p_{24} \oplus p_{25} \oplus p_{26} \oplus p_{27} \oplus p_{28} \oplus p_{30} \oplus p_{32}$ $p_{18} \oplus p_{19} \oplus p_{21} \oplus p_{23} \oplus p_{25} \oplus p_{27} \oplus p_{29} \oplus p_{30} \oplus p_{31} \oplus p_{32}$
Intel Xeon E5-2640 v2	Haswell	35	$p_{17} \oplus p_{18} \oplus p_{20} \oplus p_{22} \oplus p_{24} \oplus p_{25} \oplus p_{26} \oplus p_{27} \oplus p_{28} \oplus p_{30} \oplus p_{32}$ $p_{19} \oplus p_{22} \oplus p_{23} \oplus p_{26} \oplus p_{27} \oplus p_{30} \oplus p_{31}$ $p_{17} \oplus p_{20} \oplus p_{21} \oplus p_{24} \oplus p_{27} \oplus p_{28} \oplus p_{29} \oplus p_{30}$

address to select the set in the last level cache. This is far from the 21 bit freedom that we obtain with the huge size pages.

5 Results

Table 2 summarizes the slice selection algorithm for all the processors analyzed in this work.

The Intel i650 processor is the oldest one that was analyzed. Indeed, the slice selection algorithm that it implements is much simpler than the rest of them, involving only one single bit to decide between the two slices. This bit is the 17th bit, i.e., the bit consecutive to the last one selecting the set. This means that an attacker using *Prime+Probe* techniques can fully control both slices, since all the bits are under his control.

It can be seen that the Intel i5-3320M and Intel i5-4200M processors implement a much more complicated slice selection algorithm than the previous one. Since both processors have 2 slices, our method outputs one single solution, i.e, a single vector in the kernel of the system of equations mapping to the zero slice. Note the substantial change between these processors and the previous one, which evaluate many bits in the hash function. Note that, even though both processors have different microarchitectures and are from different generations, they implement the same slice selection algorithm.

We next focus on the 4 slice processors analyzed, i.e., the Intel i7-4702MQ and the Intel Xeon E5-2609. Again, many upper bits are used by the hash function to select the slice. We obtain 3 kernel vectors for the system of equations mapping to the zero slice (two vectors and the linear combination of them). From the three solutions, any combination of two solutions (one for h0 and one for h1) is a valid solution, i.e, the 4 different slices are represented. However, the labeling of the slices is not known. Therefore, choosing a different solution combination will only affect the labeling of the non-zero slices, which is not important for the scope of the tool. It can also be observed that, even when we compare high end servers (Xeon-2609) and laptops (i7-4702MQ) with different architectures, the slice selection algorithm implemented by both of them is the same. Further note that one of the functions is equal to the one discussed for the two core architectures. We can therefore say that the hash function that selects the slice for the newest architectures only depends on the number of slices.

Finally, we focus on the Intel Xeon E5-2640v3, which divides the last level cache in 8 slices. Note that this is a new high end server, which might be commonly found in public cloud services. In this case, since 8 slices have to be addressed, we need 3 hash functions to map them. The procedure is the same as for the previous processor: we first identify the set of equations that maps to slice 0 (remember, this will not output any possible solution to 1) by finding its kernel. The kernel gives us 3 possible vectors plus all their linear

combinations. As before, any solution that takes a set of 3 vectors will be a valid solution for the equation system, differing only in the labeling of the slices. Note also that some of the solutions have only up to 6 bits of the physical address, making them suitable for side channel attacks.

In summary, we were able to obtain all the slice selection hash functions for all cases. Our results show that the slice selection algorithm was simpler in Nehalem architectures, while newer architectures like Ivy Bridge or Haswell use several bits of the physical address to select the slice. We also observed that the slice selection algorithm mostly depends on the number of slices present, regardless of the type of the analyzed CPU (laptop or high end servers). In consequence, one can use this information to perform LLC prime probing attacks, as demonstrated by Hund et al., Irazoqui et al. and Lei et al. [18, 24, 19], or for improving cache profiling tools.

6 Conclusion

This work presents a tool to recover the LLC slice selection function on Intel processors. The slice selection algorithm can be used for constructive purposes, such as building a more realistic multicore cache simulator, or for malicious purposes, as a prerequisite for side channel attacks exploiting LLC accesses. The proposed tool uses the *Prime+Probe* technique in the LLC together with huge size pages to identify the memory blocks that reside in a specific slice. Based on the physical address of these memory blocks, the tool generates equations that are then solved using an algebraic solver. The tool is successfully tested on a wide range of processors, from desktop computer and laptop CPUs to high end server CPUs. Due to the high number of processors reverse engineered, the scalability of the tool is also successfully proven. Our results show that the used hash function is as simple as a single address bit in old processors, whereas the newer ones utilize several of the physical address bits to choose the slice where the data will reside.

An interesting research question would be how difficult it is to reverse engineer non-linear slice selection algorithms. From a theoretical point of view it should be possible to reverse engineer even non-linear slice selection functions due to the limited number of address bits (e.g. $\leq$ 64-bits), which gives a tractable space. However, other more practical reasons, such as the lack of access to the full set of physical address bits or the presence of measurement noise, may prevent reverse engineering from succeeding.

Acknowledgment This material is based upon work supported by the National Science Foundation under Grant CNS-1318919.

References

1. Cross-Referenced Linux and Device Driver Code.
2. Dinero IV trace-driven uniprocessor cache simulator. http://pages.cs.wisc.edu/~markhill/DineroIV/.
3. Intel Core i5-3320M Processor . http://ark.intel.com/products/64896/Intel-Core-i5-3320M-Processor-3M-Cache-up-to-3_30-GHz.
4. Intel Core i5-4200M Processor. http://ark.intel.com/es/products/75459/Intel-Core-i5-4200U-Processor-3M-Cache-up-to-2_60-GHz.
5. Intel Core i5-650 Processor . http://ark.intel.com/es/products/43546/Intel-Core-i5-650-Processor-4M-Cache-3_20-GHz.
6. Intel Core Xeon E5-2609 Processor . http://ark.intel.com/products/64588/Intel-Xeon-Processor-E5-2609-10M-Cache-2_40-GHz-6_40-GTs-Intel-QPI.
7. Intel Core Xeon E5-2640 Processor . http://ark.intel.com/es/products/83359/Intel-Xeon-Processor-E5-2640-v3-20M-Cache-2_60-GHz.
8. Intel i7-4702M Processor . http://ark.intel.com/products/75119/Intel-Core-i7-4702MQ-Processor-6M-Cache-up-to-3_20-GHz.
9. Intel ivy bridge cache replacement policy. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/.
10. Intels sandy bridge microarchitecture. http://www.realworldtech.com/sandy-bridge/.
11. SMPCache:simulator for cache memory systems on symmetric multiprocessors. http://arco.unex.es/smpcache/.

12. Benger, N., van de Pol, J., Smart, N. P., and Yarom, Y. "Ooh Aah... Just a Little Bit": A Small Amount of Side Channel Can Go a Long Way. In *CHES* (2014), pp. 75–92.
13. Bernstein, D. J. Cache-timing attacks on AES, 2004. URL: `http://cr.yp.to/papers.html#cachetiming`.
14. Bonneau, J., and Mironov, I. Cache-Collision Timing Attacks against AES. In *Cryptographic Hardware and Embedded Systems—CHES 2006* (2006), vol. 4249 of *Springer LNCS*, Springer, pp. 201–215.
15. Chen, W., Chen, S.-L., Chiu, S., Ganesan, R., Lukka, V., Mar, W., and Rusu, S. A 22nm 2.5mb slice on-die L3 cache for the next generation xeon processor. In *VLSI Circuits (VLSIC), 2013 Symposium on* (June 2013), pp. C132–C133.
16. Cho, S., and Jin, L. Managing Distributed, Shared L2 Caches Through OS-Level Page Allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2006), MICRO 39, IEEE Computer Society, pp. 455–468.
17. Hardavellas, N., Ferdman, M., Falsafi, B., and Ailamaki, A. Near-optimal cache block placement with reactive nonuniform cache architectures. *IEEE Micro's Top Picks 30*, 1 (2010), 29.
18. Hund, R., Willems, C., and Holz, T. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 191–205.
19. Irazoqui, G., Eisenbarth, T., and Sunar, B. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxingand its Application to AES. In *36th IEEE Symposium on Security and Privacy (S&P 2015)* (San Jose, CA, 2015).
20. Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID* (2014), pp. 299–319.
21. Jaleel, A., Cohn, R. S., Luk, C.-K., and Jacob, B. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *MoSB* (2008).
22. Jin, L., and Cho, S. Better than the two: Exceeding private and shared caches via two-dimensional page coloring. In *Proc. Intl Workshop Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI* (2007).
23. Kurian, G., Devadas, S., and Khan, O. Locality-aware data replication in the last-level cache. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (Feb 2014), pp. 1–12.
24. Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last level cache side channel attacks are practical. In *36th IEEE Symposium on Security and Privacy (S&P 2015)* (San Jose, CA, 2015).
25. Mittal, S., and Nitin. Memory map: A multiprocessor cache simulator. *J. Electrical and Computer Engineering 2012* (2012).
26. Molka, D., Hackenberg, D., Schone, R., and Muller, M. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on* (Sept 2009), pp. 261–270.
27. Pieper, J. J., Mellan, A., Paul, J. M., Thomas, D. E., and Karim, F. High level cache simulation for heterogeneous multiprocessors. In *Proceedings of the 41st Annual Design Automation Conference* (New York, NY, USA, 2004), DAC '04, ACM, pp. 287–292.
28. Razaghi, P., and Gerstlauer, A. Multi-core cache hierarchy modeling for host-compiled performance simulation. In *Electronic System Level Synthesis Conference (ESLsyn), 2013* (May 2013), pp. 1–6.
29. Srikantaiah, S., Kultursay, E., Zhang, T., Kandemir, M. T., Irwin, M. J., and Xie, Y. MorphCache: A reconfigurable adaptive multi-level cache hierarchy. In *17th International Conference on High-Performance Computer Architecture (HPCA-17 2011), February 12-16 2011, San Antonio, Texas, USA* (2011), pp. 231–242.
30. Tam, D., Azimi, R., Soares, L., and Stumm, M. Managing shared L2 caches on multicore systems in software. In *In Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA* (2007).
31. Tao, J., and Karl, W. Detailed cache simulation for detecting bottleneck, miss reason and optimization potentialities. In *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools* (New York, NY, USA, 2006), valuetools '06, ACM.
32. Yarom, Y., and Falkner, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 719–732.
33. Ye, Y., West, R., Cheng, Z., and Li, Y. Coloris: A dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (New York, NY, USA, 2014), PACT '14, ACM, pp. 381–392.
34. Zhao, L., Iyer, R., Upton, M., and Newell, D. Towards Hybrid Last Level Caches for Chip-multiprocessors. *SIGARCH Comput. Archit. News 36*, 2 (May 2008), 56–63.