SCHOOL OF COMPUTER SCIENCE

THE UNIVERSITY of ADELAIDE

Automated Retrieval of Intel's Undisclosed Slice Mapping Function for Last-Level Caches

Bradley Morgan (A1716836)

Honours Thesis

COMP SCI 4015 Honours Project
Honours Degree of Bachelor of Computer Science

Supervisors: Dr Chitchanok Chuengsatiansup, Mr Nick Manser, Dr Yuval Yarom

Abstract

Vetting the security of micro-processors is of utmost importance due to their prevalence in modern society, and their critical usage in scenarios requiring high levels of trust for the integrity and safety of information. Unfortunately, research into this field of micro-architectural security is hindered by the lack of transparency from large semiconductor manufacturers, as they have little incentive to expose closely guarded proprietary trade secrets related to the design and implementation details of their processors. Behaviour such as this results in an endless game of catch-up for security researchers to determine the trustworthiness of hardware.

Modern processors incorporate large caches shared across cores, termed the 'Last-Level Cache'. This cache differs from the smaller, per-core caches. An undisclosed hash function distributes memory addresses among several distinct portions of this shared cache referred to as 'slices'. Without knowledge of this hash function, research into securing this shared cache stalls due to the locations of memory in each of the slices being unknown. This hinders the development and corresponding mitigation of stealthy cross-core side-channel attacks.

In response to these issues, we contribute an easy-to-use hardware performance counter interface to aid in the reverse engineering of specific components of a processor. This interface allows for gaining insight into low-level micro-architectural events through the use of in-built performance monitoring hardware featured in x86-64 processors. Using this interface, we describe the structure and function of an automated tool used to retrieve the undisclosed hash function inside of Intel processors.

Acknowledgments

Yuval Yarom and Chitchanok Chuengsatiansup from the School of Computer Science at the University of Adelaide provided invaluable support over the course of the past year. Previous discoveries by Yuval laid the path to the development and successful contributions of this project; his past work quite literally underpinned my own. Both his and Chitchanok's guidance over the course of this year steered the project towards success, I am grateful to have had them as my supervisors and mentors. Also appreciated were the weekly lunches they organised throughout the year, most definitely providing excellent 'research fuel' for getting this finished.

My joint supervisor Nick Manser from Defence Science and Technology Group of the Department of Defence was a wonderful resource to draw upon due to his past work on side-channel attacks, as well as his superb differential outlook on approaching the various hurdles the project came across throughout the duration of the year. His constant tutelage and support was, and is, consistently top-tier. Thanks Nick!

Dallas McNeil quickly became one of my constant peers from the beginning of my time at University. I respect him greatly as both a research partner and friend, where our persistent, incessant chatter throughout our simultaneous projects kept us both sane and in good spirits. Thanks Dallas, I could not have done it with out you.

Finally, a big thank-you to my university peers, work colleagues and family who allowed me to bounce ideas off them, contributing to the persistent learning environment we all share.

This project is supported by the Commonwealth of Australia as represented by the Defence Science and Technology Group of the Department of Defence.

Dedicated to my parents, whose decision to renovate half the house during this project brought me much anguish yet happiness…

In all honesty, they provided an excellent environment to allow me to work on this project. Thanks Mum, Dad.

Contents

List of Acronyms vii

List of Algorithms viii

List of Figures ix

List of Tables x

1 Introduction 1

2 Background 4
2.1 Processor Cache Hierarchy and Design . . . 4
2.1.1 Side-Channel Security . . . 5
2.2 Virtual and Physical Memory and Paging . . . 6
2.3 Ring Interconnect in Intel Processors . . . 6
2.4 Hardware Performance Counters . . . 7
2.4.1 Processor Core Counters . . . 8
2.4.2 Intel Uncore Counters . . . 8
2.5 Intel Last-Level Cache Address Slice Mapping . . . 8
2.5.1 Hash Function Structure . . . 9
2.5.2 Hash Function Retrieval on Intel Xeon Scalable Processors . . . 12

3 Performance Counter Interface 13
3.1 Current Interfaces . . . 13
3.2 Description . . . 14
3.3 Usage . . . 14
3.3.1 Defining Counters . . . 14
3.3.2 Initialisation of a Performance Monitoring Session . . . 15
3.3.3 Monitoring a Function . . . 15
3.3.4 Retrieval of Counter Values . . . 16
3.3.5 Clearing Session . . . 17

4 Automatic Tool for Hash Function Retrieval 18
4.1 Determining Slice Mapping for an Address . . . 19
4.1.1 Performance Counter Method . . . 20
4.1.2 Timing Memory Access . . . 21

4.2 System Initialisation . . . 21
4.2.1 Determining Sequence Length . . . 22
4.3 Searching for Adjacent Addresses . . . 23
4.4 Determining Adjacent Address Slice Mappings . . . 24
4.5 Computing XOR-Reduction Map . . . 24
4.5.1 Linear Functions . . . 25
4.5.2 Non-Linear Functions . . . 26
4.5.3 Correctness . . . 26
4.6 Master Sequence Retrieval for Non-Linear Functions . . . 27
4.6.1 Correctness . . . 27
4.7 Output . . . 27
4.7.1 Use of XOR Permutation Masks . . . 28

5 Results **30**
5.1 Retrieved Linear Functions . . . 30
5.1.1 2–Core Results . . . 30
5.1.2 4–Core Results . . . 31
5.1.3 8–Core Results . . . 31
5.2 Retrieved Non-Linear Functions . . . 32
5.2.1 6–Core Results . . . 32
5.2.2 10–Core Results . . . 32
5.3 Discussion and Future Work . . . 35
5.3.1 Common Ground between Linear and Non-Linear Functions . . . 35
5.3.2 Restricted Address Slice Mappings . . . 35
5.3.3 Further Improvements and Research . . . 36

6 Conclusion **38**

Appendix **39**
Sample Tool Output for i7-6700K Linear Function . . . 40
Sample Tool Output for i7-9850H Non-Linear Function . . . 43
Project Media . . . 46
Source Code . . . 48

Bibliography **49**

List of Acronyms

AMD	Advanced Micro Devices Inc.
ARB	Arbitration Agent
CBo	C-box
CHA	Cache/Home Agent
CPU	Central Processing Unit
HPC	Hardware Performance Counters
IMC	Integrated Memory Controller
LLC	Last-Level Cache
MDS	Micro-architectural Data Sampling
MSR	Model Specific Register
PMU	Performance Monitoring Unit
SMT	Simultaneous Multi-Threading
SoC	System On a Chip
XOR	Exclusive-Or

List of Algorithms

1 Pseudocode for measuring *monitored function* with HPCs. 17

2 Pseudocode to retrieve sequence length for non-linear hash functions. . 23

List of Figures

2.1 Intel Coffee Lake System On a Chip (SoC) layout [Int20]. 7
2.2 Intel's undisclosed hash function sits between each core and the LLC slices. 9
2.3 Previously recovered 6-core initial stage *F* to calculate *ID(Address)* [YGL+15]. 11
2.4 Previously recovered 6-core second stage logical circuit [YGL+15]. . . . 11

4.1 Comparison between address bits used for linear versus non-linear functions. 20

5.1 Block visualisation of the 2–core processor hash function. 32
5.2 Block visualisation of the 4–core processor hash function. 32
5.3 Block visualisation of the 8–core processor hash function. 32
5.4 Block visualisation of the 6 and 10–core XOR-reduction. 33
5.5 Block visualisation comparison between 6–core and 10–core master sequences. 34

List of Tables

2.1 Intel Core i7-6700K cache specifications [Int15]. 5

4.1 Initialisation variables. 22
4.2 A 4-core Intel Core i7-6700K with a linear function gives the above mappings for each address. 24
4.3 Sequence mappings for a 6-core Intel Core i7-9850H with a non-linear function. 25
4.4 Example XOR-Reduction for bits 13, 14 and 33 using the same Intel Core i7-6700K as Table 4.2. 25
4.5 Example XOR-Reduction for bits 13, 14 and 33 using Sequence IDs, from the same Intel Core i7-9850H as Table 4.3. 26
4.6 Excerpt from the master sequence retrieved from an Intel Core i7-9580H. 27

5.1 2–Core permutation mask. 31
5.2 4–Core permutation masks. 31
5.3 8–Core permutation masks. 31
5.4 Permutation masks for various 6–core and 10–core processors. 33
5.5 Master sequence for various 6–core processors. 33
5.6 Master sequence for the 10–core hash function. 33
5.7 First 128 slice mappings from address 0 for all retrieved functions. . . 36

1 Introduction

Modern computing relies heavily on the use of complicated micro-processor technology, underpinning the secure processing of critical information and data. In order to validate and understand the trustworthiness of such machines, their inner workings must be understood. Significant effort expended by the computer security research community over the past several decades has lead to portions of micro-processor architecture being reverse engineered, with a heavy focus on AMD and Intel x86-x64 processors due to their popularity and performance. This revealed the existence of many side-channel and processor vulnerability exploits such as CPU cache timing attacks [Per05, YF14] and the first transient execution attacks [LSG+18, KHF+19] which have developed into Micro-architectural Data Sampling (MDS) attacks [vSM19, CGG+19]. The common underlying method for reverse engineering involves executing certain patterns of instructions and inferring processor design by observing the corresponding behaviour. Yet little of these community led efforts are allayed by major semiconductor manufacturers such as Intel and AMD, aside from the provision of software development manuals [Int21c, AMD20] and occasional confirmations of the existence of certain hardware features and flaws. Thus, reverse engineering efforts in this black-box scenario become limited to characterising a hardware component experimentally, and using the observed experimental behaviour to determine if any security issues exist.

In our work we specifically target and generically reverse engineer an undisclosed hash function in Intel's processors. This function coordinates the location mappings of memory into several distinct portions of the last-level CPU cache termed *slices*. Each Intel processor configuration implements an architectural–specific mapping, causing difficulty in exploring as well as mitigating cross-core micro-architectural side-channel leakage. Therefore, in order to enable such work, researchers require knowl-

edge of this mapping to finely control groups of memory addresses indexing into the same slice.

We implement a custom interface to interact with in-built performance monitoring hardware inside Intel's processors. The design of our interface allows for a user–defined function to be monitored in a fine-grained fashion using various performance counters, providing an accessible method for exposing low-level processor behaviour with ease. We use this interface to finely execute and measure the effect of certain instructions on the CPU's internal state, incorporating this into an automated tool to infer the structure and operation of the undisclosed hash function. This builds off previous work to reverse engineer the function on different processor configurations[MSN+15, YGL+15]. Our tool completely automates retrieval of the hash function from any Intel consumer processor. We discuss results gathered thus far from a variety of processors, noting the occurrence of alterations to the function since previous publications in 2015, and some commonalities between different functions.

Contributions

In summary, we make the following contributions:

- We instrument a low-level hardware performance counter interface to monitor user–defined sections of code for the exploration of micro-architectural effects. We contribute an accompanying custom kernel module to allow performance counters to be accessed from user space.

- We develop an automated tool to generically reverse engineer slice mappings across any recent Intel consumer processor. Utilising our performance counter interface results in our knowledge of the slice mappings for memory addresses being exact. The retrieved function is provided in a readily usable format to support further research. The tool also visualises the results for comparison between processors. We will make the tool open source.

Overview

Chapter 2 presents background information on processor micro-architecture to inform the reader on the context of the contributions for this thesis. We intersperse related works to side-channel attacks, and the reverse engineering efforts required to understand modern processors from a black-box perspective. We also detail the currently known structure of Intel's last-level cache slice mapping algorithm and previous efforts to retrieve this.

Chapter 3 describes the hardware performance counter interface and the reasoning behind our use of it over other tools. We describe its programmatic usage for monitoring of a user–specified function in order to gain insights into micro-architectural events using such fine-grained execution.

Chapter 4 details our approach to retrieving the Intel slice mapping function using our command-line tool to automatically reverse engineer it through the use of the aforementioned performance counter interface. Our tool discovers memory addresses differing on a single bit, recovering an XOR-reduction function designed to reduce physical address bits into a slice index on processors with 2^n slices. The tool retrieves a processor dependent second stage if the number of slices is not a power of two.

Chapter 5 analyses several retrieved hash functions across a variety of modern Intel processors across several generations. We note Intel's reuse of the hash function in newer processor models, while changing it in the most recent Rocket Lake generation. We discover the XOR-reduction used by processors with non-2^n slices to be linked by a common permuted XOR value.

2 Background

Each generation of micro-processor design introduces increasingly complex optimisations in the pursuit of better performance and information throughput. Caches reduce the average time to access memory by storing a small amount of recently used data in close proximity to the processor [Smi82], with most modern designs now incorporating a multi-level cache hierarchy. Processors execute instructions out of order[KHC94] to increase instruction throughput by minimising idle time, keeping the processor constantly busy. Speculative execution and branch prediction allow for the execution of instructions down one side of a branch. If this branch is not taken, then the processor rolls back its micro-architectural state. Simultaneous Multi-Threading (SMT) [Mar02] optimises the instruction pipeline further by allowing the execution of programs in parallel on the same physical hardware, by dividing the execution units of one physical core into two logical processors. The underlying design details of these optimisations are unavailable to the public, although processors allow access to some limited micro-architectural information through the use of the `CPUID` instruction [Int21c]. Due to these optimisations, reverse engineering Intel and AMD micro-architecture requires significant effort to explore the security of modern processors. In the context of side-channel attack research, such pursuits enable hidden hardware design flaws to be exposed, exploited and correspondingly mitigated.

2.1 Processor Cache Hierarchy and Design

To understand the terminology used in this work, the structure of caches must be understood, as well as the way in which they behave. A CPU cache stores contiguous sections of main memory when any data is used by a program to reduce potential

Cache	Size	Associativity	Sets	Set Bits (incl.)	Cache Line
L1D	32KB	8-way	64	6-11	64 bytes
L2	256KB	4-way	1024	6-15	64 bytes
L3	4×2MB (sliced)	16-way	2048	6-16	64 bytes

Table 2.1: Intel Core i7-6700K cache specifications [Int15].

access time in the future. When a cache is queried but does not hold requested data, this results in a *cache miss*, and when found, a *cache hit*. In terms of the cache hierarchy, each core in an Intel CPU contains its own Level 1 (L1) caches for data and instructions. This provides fast access to these caches, albeit at the cost of their size. A larger Level 2 (L2) cache accompanies these, storing both data and instructions. These caches are private to each CPU core. An L3 or Last-Level Cache (LLC) is the largest cache present in the CPU, shared across all cores in a processor. It resides in a section of the processor which Intel calls the *uncore* [Int21c]. The uncore encompasses all portions of the processor not contained with an individual processor core.

Accessing memory brings the surrounding 64 byte chunk into a cache as a *cache line*, this is the lowest granularity one can consider cache behaviour with. The L1 cache is the first cache to hold a copy of memory, where eventual eviction occurs as determined by the cache's replacement policy into the L2 cache and so forth. Older Intel processors feature an *inclusive cache*, holding copies of L1 and L2 data inside the LLC. Modern Intel Xeon processors now use a *non–inclusive* cache [Int21a], where none of the upper level cached data copies reside in the LLC.

Both data and instruction caches in Intel processors use an n-way set-associative structure. A cache is constructed from m sets, with each set dividing into n cache lines of a specific size, commonly 64 bytes. This number of n divisions refers to the cache's associativity, or ways. Specific bits of a memory address determine its indexing into a certain cache set. This is true for L1 and L2 caches, however it is different in the LLC due to its use of slices, as we discuss in Section 2.5. Table 2.1 shows the cache specifications for an Intel Core i7-6700K and the bits used for each cache level. Therefore, the dimensions of a cache determine its size as such:

$$Total\ Cache\ Size = Cache\ Line\ Size \times Associativity \times Cache\ Sets$$

2.1.1 Side-Channel Security

Attacks can be mounted using these caches by inferring information from observed differences for cache access times, while taking into account replacement policies and other micro-architectural behaviour such as speculative and out-of-order execution. These form the basis of micro-architectural side-channel attacks.

Tromer et al. [TOS10] introduce two methods for interacting and leaking from victim process accesses to private caches, EVICT+TIME and PRIME+PROBE. These

attacks centre around investigating and understanding cache structure, through the analysis of miss and hit behaviour based on certain access patterns. The authors take into account several popular Intel processor models and discuss the cache parameters for each. They use these techniques in combination with cryptanalysis based on the victim's memory accesses to reveal AES private key information.

Yarom and Falkner [YF14] contribute a method for measuring shared access times to the LLC, expanding on the FLUSH+RELOAD technique presented in previous work by Gullasch et al. [GBK11]. By utilising shared pages between processes, the authors use the `CLFLUSH` instruction to flush the victim's contents from the cache, followed by a 'reload' occurring when the victim accesses the data again. This signals to the attacker whether certain portions of code were used or not, leaking information about the execution state of victim. To counteract measurement noise generated by speculative execution and out of order execution, the authors describe a technique which serialises the instruction pipeline by using the `CPUID` and `MFENCE` instructions.

2.2 Virtual and Physical Memory and Paging

Operating systems enforce a virtual address space in order to separate the execution of processes and allow programs to exceed the amount of availably memory. These virtual address spaces translate into the physical address space in a manner defined by the amount of total RAM installed in the machine. The separation of addresses also ensures the security boundary between processes such that they cannot interfere with each other's data and execution.

Contiguous portions of physical memory form virtually addressed pages of a specific size determined by the operating system, most commonly 4 KB, however processors can support larger huge pages at 2MB, with increased sizes up to 1GB. A replacement algorithm swaps these pages in and out of main memory depending on factors such as the amount of use and time spent in memory. These are stored in page tables located in the main memory subsystem, with lookups to this cached in an architectural component called the Translation Lookaside Buffer (TLB). Several of these exist in modern processors. Previous work by Gras et al. [GRBG18] uses specially designed access patterns to reverse engineer several TLBs in the caches of various Intel processors. Their results detail the existence of both linear and complex addressed TLBs for L1 and L2 data and instruction caches, and they use this knowledge used to implement a timing side-channel attack.

2.3 Ring Interconnect in Intel Processors

Since the release of Intel's Sandy Bridge micro-architecture, the company's consumer multi-core processors [DKL+17] utilise a ring topology within the uncore. This ring interconnect (shown within Figure 2.1 for a more recent Coffee Lake octa-core processor) allows information to be sent back and forth between cores, with data transferring in both clockwise and anti-clockwise directions within the CPU die. This facili-

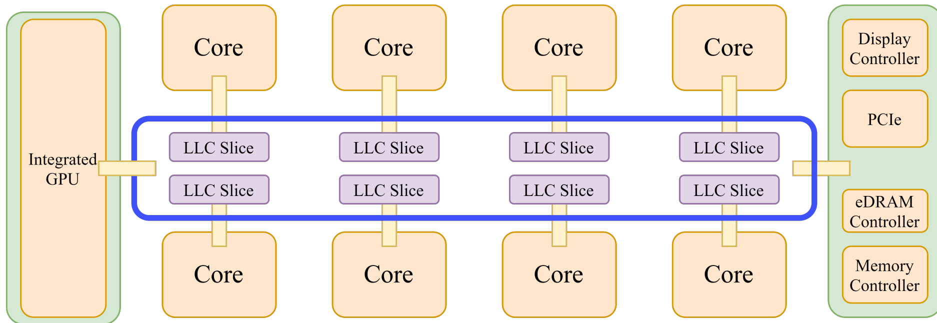

Figure 2.1: Intel Coffee Lake System On a Chip (SoC) layout [Int20].

tates a data pipeline connection between each slice in the LLC, as well as the processor graphics, integrated memory controller and other components [Int21b]. Each core features a LLC slice in close proximity, with the other slices accessible via pathways on the ring interconnect. This introduces latency within the interconnect as cores requesting data from a slice on the other side of the processor take longer to respond. Paccagnella et al. [PLF21] take advantage of this latency to reverse engineer the structure of the interconnect and create a covert channel. The authors document several observations which occur based on observed timing differences, and also explore various side-channel attacks based on data leakage through interconnect contention. Intel no longer uses this topology in their enterprise Xeon processors due to latency issues with higher core counts [Kum17]. Instead, a mesh topology connects the cores in a grid layout. The release of Skylake-SP architecture [Int17] standardised this across Intel's Xeon processors.

2.4 Hardware Performance Counters

Hardware Performance Counters (HPCs) are a specific feature of CPUs and allow for programmers to expose a variety of processor events to be monitored and counted. This allows for accurate benchmarking of programs. This in-built hardware exists on major CPU architectures including x86-64 and ARM. Intel's software development manual [Int21c, Chapter 19] describes the usage of this feature as well as descriptions of events provided for x86-64 processors. Model-specific registers (MSR) control certain processor components related to the performance monitoring process, and are accessible via the `RDMSR` and `WRMSR` instructions for reading and writing. These instructions only execute when a program operates in in kernel space or the processor is configured to use real-address mode. Programs read performance counters using `RDPMC`, however this is by default a privileged instruction, and is accessible in user mode by setting bit 8 of control register `CR4` [Int21c].

Available counters vary between generations of processors; newly implemented counters do not function on older generation of processor. Two distinct types of Per-

formance Monitoring Units (PMU) exist within the CPU. There are those specific to a processor core, counting general execution events such as retired instructions and L1, and L2 cache hits and misses. The second relate to performance monitoring in the uncore and are specific to Intel CPUs. These allow insight into events such as reads, writes and accesses for each LLC slice.

2.4.1 Processor Core Counters

The PMU inside a core features three fixed function counters and eight general purpose counters (four if hyperthreading is enabled). The fixed counters measure the number of retired instructions, un-halted CPU cycles, and also cycles at a time stamp counter adjusted rate, unaffected by core frequency changes. General purpose counters can measure a variety of architectural–specific events such as the number of L1 cache misses or the number of offcore requests. Selecting an event to count is managed through the `IA32_PERFEVTSELx` MSR. This register is programmed by the user to contain the information describing a specific event to count. There are also a variety of flags controlling how each event is counted, allowing further methods to fine tune results and infer processor behaviour. For example, the `EN` flag enables the specific counter, and the `INV` flag counts the specified event when it does not occur.

2.4.2 Intel Uncore Counters

A separate set of counter hardware exists for measuring uncore events. For Intel CPUs, there are several uncore units which expose performance monitoring interfaces for their specific context [Int16]. Specifically, several PMUs exist for each LLC slice. For Intel Core CPU's these PMUs reside in a 'coherency engine' referred to as a C-Box or CBo. Each slice interfaces with a processor core via their associated CBo. In Xeon processors, the equivalent of a CBo refers to a cache/home agent or CHA. Similar to the previous core counters, each CBo expose MSRs which allow certain uncore events to be measured. Similar flags exist to control the method of counting such as those in processor core counters.

2.5 Intel Last-Level Cache Address Slice Mapping

Intel incorporates an undisclosed hash function to facilitate the division of the shared LLC into several *slices* as shown in Figure 2.2. By splitting the LLC in such a manner, memory accesses distribute across each slice evenly, increasing the total memory bandwidth to the processor uncore. This function operates on physical address bits and calculates a slice index within the range of available uncore slices. Due to the cache line granularity of 64 bytes, bits 5:0 of an address have no bearing on the hash function's slice index calculation. When a cache line is copied into one of these LLC slices its address bits index into a certain cache set, as previously explained for L1 and L2 caches.

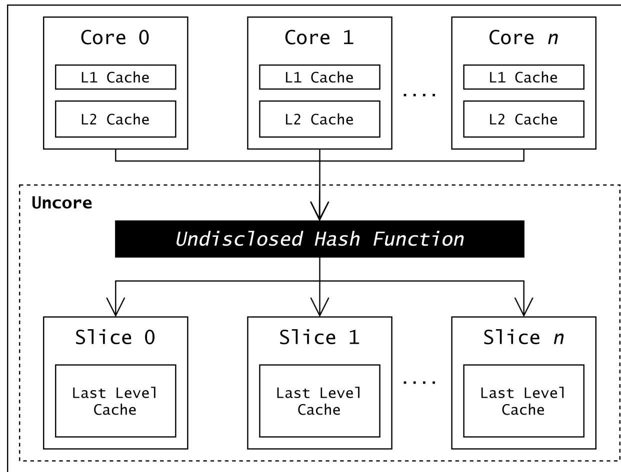

Figure 2.2: Intel's undisclosed hash function sits between each core and the LLC slices.

2.5.1 Hash Function Structure

Knowledge on the structure of this hash function originates from past reverse engineering efforts. No official documentation exists on the format of the function, its intended use, or variance among processors, although the splitting of the LLC into slices is mentioned for the purposes of uncore performance monitoring.

Linear Functions

Early works by Hund et al. [HWH13] and Irazoqui et al. [IES15] were successful in partially retrieving the hash function. As they rely on conflicting addresses in the same LLC sets, this masks any information gained from the bits to determine the set index. To determine differences in the function between addresses, Hund et al. utilise addresses differing on a single bit. Irazoqui et al. formulate equations based on the address bits and solve for these to identify the slice operation. Both sets of authors observe Intel's design of an *XOR-reduction* on the upper address bits to compute slice indices. For processors with 2^n slices, a series of XORs determine the slice index using combination of physical memory address bits. They determined the slice index for a given physical address a as follows:

$$Slice\{0,1\}^0 = a_{31} \oplus a_{30} \oplus a_{29} \oplus a_{27} \oplus a_{25} \oplus a_{23} \oplus a_{21} \oplus a_{19} \oplus a_{18}$$
$$Slice\{0,1\}^1 = a_{31} \oplus a_{29} \oplus a_{28} \oplus a_{26} \oplus a_{24} \oplus a_{23} \oplus a_{22} \oplus a_{21} \oplus a_{20} \oplus a_{19} \oplus a_{17}$$

When the number of slices in a processor is a power of two (e.g. 2^n) the described hash function is *linear*, as the function consistently maps the address bits into distinct slices. Maurice et al. [MSN+15] present the first method for successfully retrieving the entire slice mapping algorithm. With the use of hardware performance counters,

they monitor each slice's CBo for the `LLC_LOOKUP` event. Utilising the `CLFLUSH` instruction, which forcefully evicts an address from all cache levels, they determine which slice the address is placed in, regardless of its presence in the cache. This has the added benefit of providing exactly which slice the address indexes into, whereas the previous methods use indirect measurements and cannot distinguish which literal slice they access. To recover the mapping, the authors use pairs of physical addresses (as done by Hund et al.) differing by one bit, on every addressable bit of memory, and record their slices. Their results cover several generations of processors ranging in size from 2 to 8 cores.

Non-Linear Functions

Yarom et al. [YGL+15] and İnci et al. [IGI+15] concurrently reverse engineered the hash functions of 6 and 10 core Xeon processors respectively. These processors employ hash functions described as *non–linear* due to the number of slices not being a power of two. Such hash functions require multiple stages to map addresses into the slices, no longer creating an even distribution among slices such as in previously described linear functions.

Yarom et al. reverse engineer the hash function on a 6-core Xeon processor by considering it in two separate stages. They observe the slice mappings for a series of sequential cache lines within a buffer of memory. To get the slice mappings, they measure each memory address's access time from each available core of the CPU and find the minimum. As a core and its associated LLC slice reside in close proximity to each other inside the CPU die, this minimum access time provides the slice which a memory address maps to. They find the sequential series of slice mappings to be grouped into $sequences$, 128 cache lines in size. The authors explain how each sequence relates to one another via an XOR permutation of the offsets for each slice in the sequence. They term the XOR value defining the permutation as the sequence ID. For their 6-core processor, the values of ID ranges from 0-127. They demonstrate for two arbitrary sequences, A and B, and a sequence slice offset o, that the following holds for all slices in both sequences:

$$A_o = B_{o \oplus ID(B)}$$

We directly utilise this finding in our work and implement it for the recovery of non-linear slice functions. The authors structure the first stage of the hash function as a method to take any memory address and use it to find its corresponding sequence ID through an XOR-reduction, similar to linear functions. To recover this first stage mapping, the authors search for pairs of addresses a and a', differing on bit k as in [HWH13, MSN+15] and construct their corresponding sequences X and Y of 128 sequential slice mappings. By relating these two sequences back to a known arbitrary 'root' sequence, the IDs for $ID(X)$ and $ID(Y)$ are computed. Finding the XOR of the two resulting ID values yields the XOR permutation for the differing bit k:

$$xor_k = ID(X) \oplus ID(Y)$$

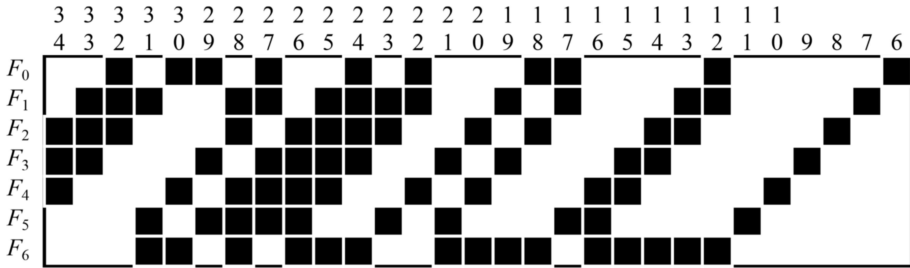

Figure 2.3: Previously recovered 6-core initial stage F to calculate *ID(Address)* [YGL+15].

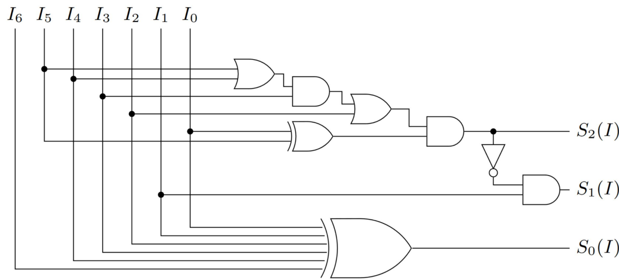

Figure 2.4: Previously recovered 6-core second stage logical circuit [YGL+15].

Repeating this for each possible value of k results in the first stage, and allows calculation of an address's ID only from its physical bits. To visualise this, the authors construct a block visualisation for each address bit with the corresponding applied XOR permutation, shown in Figure 2.3. F here is the value of ID between 0-127 represented as a 6-bit binary number. To retrieve the second stage S of the hash function, the authors note each possible slice ID from the first stage will always map to the same physical slice. Therefore, a logical circuit reduction from the 6 bit ID value into slice values from 0-5 was constructed, with the authors representing this as Figure 2.4. This now completes retrieval of the hash function, represented as H using the two described stages. The offset of the address into its sequence XOR with its retrieved ID correspond to the input for the second stage logical circuit.

$$F(Address) = ID(Address) \oplus (Address_6 ... Address_{12})$$
$$H(Address) = S(F(Address))$$

The format of the results contributed by İnci et al. differs significantly from Yarom et al. making direct comparisons difficult. However, their reconstruction of the function does show 13/128 addresses distributing into slices 0–7 evenly, falling off for slices 8, 9 at 12/128. Their recovered function corresponds to the interpretation by

Yarom et al. with a linear function creating 128 values followed by a non-linear second function to map into slice values 0–9.

2.5.2 Hash Function Retrieval on Intel Xeon Scalable Processors

In concurrent work, McCalpin [McC21] applies these methods to several past and current Intel Xeon Scalable processors featuring much higher core counters than previous work, with such processors containing between 14 and 38 LLC slices. This selection of processors includes both linear and non-linear slice functions. To structure the results, McCalpin presents the initial stage of the hash function as an *XOR-reduction*, and then the second stage as a *base-sequence*, representing the second stage's logical circuit outputs from the XOR-reduction. Experimental results show the XOR-reduction to no longer generate values between 0–128, instead ranging from 0–4096 for processors with large core counts (e.g. 22, 28, 38). The author uses performance counters, adapted to Xeon processor uncore monitoring to measure the `LLC_LOOKUPS` through the use of the `CLFLUSH` instruction on each address. The author provides an analysis of the reconstructed functions, with the observation of patterns between some of the XOR-reductions, base sequences, and also the overall distribution of addresses across each slice. However, the full XOR-reduction is not retrieved for processors with linear functions using this method due to the inclusion of base sequences, as the second stage here is not required in such cases. The author notes some recovered XOR-reductions could be structured differently, but still result in the same slice mapping, highlighting the ambiguity resulting from the reverse engineering process.

3 Performance Counter Interface

This chapter details the custom hardware performance counter interface designed for micro-architectural reverse engineering. This interface provides both intra-core and uncore monitoring functionality for Intel Core processors.

3.1 Current Interfaces

Several performance counter interfaces exist currently and expose methods to interact with in-built hardware event counters as described in manuals provided by Intel [Int21c, Int16]. However to finely execute specific portions of code, we find current interfaces either have insufficient precision to finely measure code (e.g. only an entire execution of a program could be monitored), or introduce too much noise during measurement.

perf [Var09] provides a Linux command line utility for performance analysis of programs. It supports event counting in both kernel and user execution contexts through the use of a system call to start the counting process. This reports the counter data into a mapped memory region using a file descriptor. A programmatic interface allows for monitoring of a specific portion of a program using the same system call method. Due to context switches caused by the usage of these system calls, this utility pollutes the processor greatly, resulting in large amounts of noise in the measured counter values even for fine grained execution.

PAPI [TJYD10] provides a programmatic interface for accessing performance monitoring hardware, with both high and low-level counter interfaces. However we observe its measurement functions contain other local library function calls, therefore introducing noise into its reported results. Although this library provides an

in-depth and flexible performance counter interface, such issues limit its use in micro-architectural reverse engineering.

nanoBench [AR19] allows users to define and monitor 'microbenchmarks' in an attempt to provide insights into micro-architectural events. The tool attempts to reduce noise from compiler optimisations, and only execute the groups of instructions specified by the user. It supports a variety of options for measurement, featuring loop unrolling and a reduced memory footprint counter sampling method. However, this tool does not allow for interaction with the specified code before and after execution. This leads to less flexibility, for example in situations where a microbenchmark requires some instructions and context executed prior to measurement.

3.2 Description

We implement our own custom C library for initialising and reading performance counters, exposing both x86-64 and uncore micro-architectural events to the user. We divide the interface into two sections, the first providing access to counters within a processor core, and the second for event monitoring in the uncore, stemming from low-level incompatibility between their structure. Our interface allows for the user to prepare the context of a program prior to running a defined *monitored function*. This function specifies the portion of code to be measured, and can be user defined, as long as a function pointer is provided to the interface.

As performance counters require elevated privileges to access, we use a custom kernel module [Mor21], updated from prior work [BT16]. This kernel module must be loaded prior to using the interface and allows for reading from processor core performance counters as superuser with the `RDPMC` instruction. We use `RDMSR` to initialise all counterss via Intel's `msr-tools` [Int13] to expose access to all MSRs available on the processor. This package allows our interface to initialise `IA32_PERFEVTSELx` MSRs for core monitoring and also `MSR_UNC_PERFEVTSELx` MSRs for uncore monitoring and counter reading.

3.3 Usage

In this section we describe the steps required to use this interface in a C program to measure a user specified function using hardware performance counters. Our library requires minimal steps to initialise a session and monitor code, providing an easily accessible method for interacting with these low level event counters. The prefixes `pmu` and `uncore` prepend the interface functions used to access processor core and uncore monitoring, respectively. We denote this using `*`.

3.3.1 Defining Counters

To begin, the events to count must be defined by specifying the `event` variable, unit mask (`umask`), counter mask (`cmask`), and optionally the `name` for the counter, used

```
typedef struct counter {
    uint8_t event;
    uint8_t umask;
    uint16_t cmask;
    char name[128];
} COUNTER_T;

typedef struct counter_info {
    COUNTER_T counter;
    uint64_t flags;
} COUNTER_INFO_T;

typedef struct cbo_counter_info {
    COUNTER_T counter;
    uint8_t cbo;
    uint64_t flags;
} CBO_COUNTER_INFO_T;
```

Listing 3.1: C structs to hold counter initialisation information.

for printing header titles for the results. Listing 3.1 defines C structs to hold this information. As shown, the CBo for uncore counters must be specified. As each CBo has slots for two separate event counters, the interface accepts the first two defined counters and errors if any more are specified for the specific CBo.

3.3.2 Initialisation of a Performance Monitoring Session

The performance monitoring session must be initialised using `*_perfmon_init()`. The user must specify the previously defined counters, the total number of counters, the affinity of the monitored function's execution as well as the number of execution samples as arguments to these functions. They initialise a variable of type `*_perfmon_t`, providing access to the counter values. From the provided counter information, the interface writes to the in-built hardware MSRs, enabling counters globally on the processor and setting either `IA32_PERFEVTSELx` or `MSR_UNC_PERFEVTSELx` registers to the specified counter and flag values.

3.3.3 Monitoring a Function

Once initialisation of the session is complete, the counters begin to increment based off the specified micro-architectural events, enabling the monitoring of a specified section of code. Our interface exposes this capability through the `*_perfmon_monitor()` function. The user must provide the `*_perfmon_t` variable, the function to monitor as well as any inputs.

```
1  inline void rdpmc(uint32_t counter, uint64_t *result)
2  {
3      uint32_t lo, hi;
4      __asm volatile
5      (
6          "mfence\n"
7          "rdpmc\n"
8          : "=a" (lo), "=d" (hi)  // outputs
9          : "c" (counter)         // inputs
10     );
11     *result = ((uint64_t)hi << 32) | (uint64_t)lo;
12 }
```

Listing 3.2: C code to read from RDPMC.

Reading the Performance Counters

Reading from the hardware counters requires fine handling by the calling program to minimise any changes to the CPU state. Our approach involves serialising all memory accesses using MFENCE and immediately reading the counter information as quickly as possible, using registers to hold the data. This prevents any alterations to the state of the processor, ensuring counter accuracy. Two arrays hold the results from before and after execution, which are then subtracted to get the final counter values, shown in Algorithm 1. The interface queries RDPMC as described in Listing 3.2. We note the function for uncore monitoring differs through the use of RDMSR the via `msr-tools` package. Due to the occurrence of system events such as uncontrolled context switches, a higher sample count can be specified to remove noise by averaging. These monitoring functions require a warm-up period to properly measure the counters, as immediate measurement leads to a significant amount of noise, and therefore inaccurate results. This noise eventually stabilises after 1000 executions of the *monitored function*, which we find experimentally to be suitable without significant performance costs.

3.3.4 Retrieval of Counter Values

After the execution of the function, the interface stores the counter values in the `*_perfmon_init.results` field in the order the counters were specified at the start of the session. Dividing by the specified sample count provides the average counter value over each of the executions of the monitored function.

Printing Results

To format the results nicely, use of `*_perfmon_print_headers_csv()` prints the human -readable names of each of the counters, if they are defined. Otherwise, a combination of the event and unit masks print instead. `*_perfmon_print_results_csv()` prints the average value for each of the counters using the specified sample count. The `*_perfmon_t` type must be passed to both as the only argument.

Algorithm 1: Pseudocode for measuring *monitored function* with HPCs.

```
/* Provided are the targeted function, arguments,
   execution samples and counter information                    */
Inputs: monitored_function(), n_ctrs, input1, input2, samples
Result: Read performance counter values
ctr_results[n_ctrs], ctr_before[n_ctrs], ctr_after[n_ctrs];
s = 0;
/* Warm-up                                                      */
while s < 1000 do
    monitored_function(input1, input2);
    s++;
end
/* Serialise and read each counter                              */
cpuid();
for c ← 0 to n_ctrs by 1 do
    rdpmc(ctr_before[c]);
end
/* Begin execution of monitored function                        */
s = 0;
while s < samples do
    monitored_function(input1, input2);
end
/* Serialise and read each counter                              */
cpuid();
for c ← 0 to n_ctrs by 1 do
    rdpmc(ctr_after[c]);
end
/* Totalling rdpmc() results                                    */
for c ← 0 to n_ctrs by 1 do
    ctr_results[c] = ctr_after[c] - ctr_before[c];
end
```

3.3.5 Clearing Session

The user deletes a performance monitoring session via the `*_perfmon_destroy()` functions by providing the `*_perfmon_t`. This frees memory used to store the variable number of counters across different sessions.

4
Automatic Tool for Hash Function Retrieval

In this chapter we describe a tool we have created for automatically retrieving the last-level cache slice hashing function used in Intel processors. As stated, this function is not publicly disclosed nor described by Intel, negatively impacting the development of processor cache side-channel research. To develop this tool, we take several liberties with the systems it is run on, mainly by requiring high privilege levels to expose low level processor tools and interfaces. As a brief overview, our tool works by finding pairs of physical addresses which differ on one bit, which we define as *adjacent addresses*. These addresses tell us information about the slice mapping for their differing bit k. Thus, by finding pairs of these adjacent addresses, the slice mapping can be recovered for every addressable bit of memory. This is followed by further intricacies which depend on whether the function is linear (2^n slices), or non-linear (non-2^n slices). Our tool is split into several stages to determine the slice mapping used on a processor:

1. System initialisation (Section 4.2).
   (a) Non-linear functions: Find sequence length (Section 4.2.1).
2. Search for adjacent addresses (Section 4.3).
3. Retrieve slice mappings for adjacent address pairs (Section 4.3).
4. Determine XOR-reduction function (Section 4.5).
   (a) Non-linear functions: Compute second stage master sequence (Section 4.6).

5. Output of retrieved function (Section 4.7).

In Figure 4.1 we visualise our tool's interpretation of the hash function's relationship to physical memory address bits. This diagram displays the difference between the two types of functions. Both feature what is described as an *XOR-reduction* stage to reduce a subset of the physical address bits into a smaller value. For linear functions, this directly results in the slice for the given address. For non-linear functions, the sequence ID is found for the address, in combination with the lower bits determining the offset into a known *master sequence*, to derive the corresponding cache slice index.

Assumptions

Our main assumption for running the tool is that it executes on at least a 2nd generation Intel Core model processor. Older models require testing for compatibility, but may work if the designated MSRs to configure and enable performance counters have not been altered by Intel.

Although the tool operates automatically, our approach requires other assumptions when designing for generic execution on a variety of machines.

- UEFI secure boot must be turned off in the machine's BIOS menu as this restricts the proper use of `msr-tools`.
- Root access must be available, to both download and install packages (on Ubuntu Linux), as well as install the custom kernel module for our performance counter interface described in Chapter 3.
- The system must be 'quiet' with minimal CPU usage and memory accesses; a busy processor and memory pipeline will introduce noise when determining an address's slice, therefore disrupting the collection of results.

Offline Retrieval

The tool is designed to run on Ubuntu-derived operating systems. However, offline retrieval of the hash function is possible using a live persistent USB with Xubuntu installed for the operating system. Using this method bypasses the aforementioned limitations, and hastens function retrieval while leaving the machine unmodified. From our tests, this process takes less than two minutes in most cases, depending on the system's memory performance and the number of cores. It is not required to have internet access when using this method, apart from initially installing the tool on the live USB.

4.1 Determining Slice Mapping for an Address

Throughout the tool's execution, it must determine the slice mapping for any address at various stages. To do this we implement the performance counter method

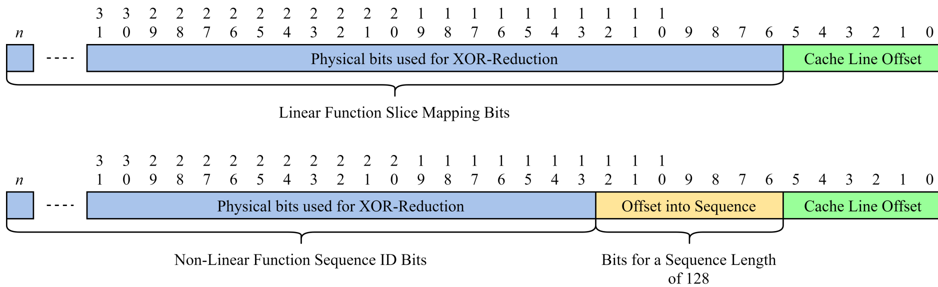

Figure 4.1: Comparison between address bits used for linear versus non-linear functions.

as in [MSN+15]. However if this fails or is not available, the tool automatically falls back to LLC slice access timing. We find using performance counters is faster than the access time method and features better accuracy.

4.1.1 Performance Counter Method

When executing CLFLUSH with a given memory address, this causes a lookup to be generated in the coresponding slice determined by the hash function. Therefore, by utilising the uncore performance monitoring of the interface described in Chapter 3, we initialise each CBo PMU to increment based on the UNC_CBO_CACHE_LOOKUP.ANY_MESI performance counter. However, further post-processing is required on the resulting data to accurately determine which slice an address maps to. We use the following steps to prune the results:

1. Access the address to map it into memory.
2. CLFLUSH the address and monitor this using the performance counter interface taking 10,000 samples.
3. Retrieve CBo counter results for each slice and divide by 10,000 to get the average amount of cache look ups across each slice, which should be between 0 and 2.0, varying with noise.
   (a) If any counter reports a value greater than 2.0 or less than 0.0, repeat Step 2.
   (b) If there is more than one counter which has a value greater than 1.0, repeat Step 2.
   (c) If none of the CBos report a lookup, use memory access timing to determine the slice.

4. To further remove any possible errors, we calculate the z-scores for the pruned counter values. From experimentation, only one will be further than +1 standard deviations from the mean. The counter index this belongs to is the LLC slice for that address.

4.1.2 Timing Memory Access

In rare cases the above method could fail. For example, on some processors such as the 10-core Intel i9-10900K, we find the MSRs required to initialise and monitor CBos 7–9 to not be exposed incorrectly by Intel. The required MSRs instead provide unrelated functionality to performance counters, leaving an error in the implementation of uncore performance monitoring. To counteract this, we fall back to the method described by Yarom et al. [YGL+15] used to determine memory address slice mappings through observed access times. Recall how each core and associated cache slice reside in close proximity to each other on the CPU die. Through the use of cache set conflicts our tool forces the eviction of memory into the LLC using knowledge of the L1 and L2 cache associativity. By finding the minimum access time for the memory from each processor core, the index this corresponds to indicates the LLC slice accessed. We describe our implementation of this method as follows:

1. Access the address to map it into memory.
2. For each processor p, including hyperthreading:
   (a) Set the current process to execute on processor p
   (b) Find the average access time over 100,000 samples:
      i. Get the minimum of the following routine over 10 samples.
      ii. `CLFLUSH` the targeted address from all caches.
      iii. Access several other addresses which map to the same L1 and L2 set as the targeted address, evicting the address into L1, L2 and then into the LLC. We use a combination of both the L1 and L2 caches' associativity to determine the number of conflicting addresses required.
      iv. Get the access time to the address, shown in Listing 4.1.
3. Find the minimum over each result. If the processor currently uses hyperthreading, modulo the result by the number of physical cores to get the correct slice.

From these two methods, we can now determine an address's slice accurately in any processor.

4.2 System Initialisation

To begin, our tool must acquaint itself with the hardware it executes on. It queries the machine for various pieces of information to enable proper retrieval of the hash function. Table 4.1 details several of these variables and their uses.

```
1  static inline uint32_t memaccesstime(void *address) {
2    uint32_t time;
3    asm volatile (
4        "mfence\n"
5        "lfence\n"
6        "rdtscp\n"
7        "mov %%eax, %%esi\n"
8        "mov (%1), %%eax\n"
9        "rdtscp\n"
10       "sub %%esi, %%eax\n"
11       : "=&a" (time): "r" (address): "ecx", "edx", "esi");
12   return time;
13 }
```

Listing 4.1: C code to find the access time for a memory address, from [Yar16].

Variables	Comment
CORES	Number of processors available to search for adjacent addresses. Includes hyperthreads.
HT	Hyperthreading, used to calculate the number of physical cores.
RAM	Total amount of RAM to determine buffer size.
ADDR_BITS	The number of possible physical address bits depending on RAM.
PAGE_BITS	Gets the number of bits for the current huge page size.
L1D, L2	Size of the L1 data and L2 caches.
[L1|L2]_ASSOCIATIVITY	Used to calculate the number of set collisions needed to evict address into LLC for determining the slice via timing access.
[L1|L2|LLC]_CACHELINE	Used to calculate access offsets for the memory address.
SEQ_LEN	Length of sequences. If the processor uses a linear function, then this is by default 1. If non-linear, determined by brute force.

Table 4.1: Initialisation variables.

4.2.1 Determining Sequence Length

For a non-linear slice hash function the tool must recover the first stage XOR-reduction. However, to reconstruct this requires slice sequences of a certain length as discussed in Chapter 2 from Yarom et al. [YGL+15]. In their work, they experimentally found this to be 128 for a 6–core processor. Our method works for any processor. We expect the sequence length to be 2^n in size as the first stage of the hash function must be linear, resulting in each address mapping to a specific sequence ID. Conducting a brute force search to find the closest power of two retrieves the maximum sequence ID for each increasing 2^n sequence lengths in an area of contiguous memory. Algorithm 2 describes this routine. We utilise the methods from Section 4.1 to retrieve slice mappings for each sequence.

Algorithm 2: Pseudocode to retrieve sequence length for non-linear hash functions.

```
/* The buffer of memory is split into sequences of cache
   lines to have their slice mapping determined        */
Inputs: buffer, num_seq, max_seq_len
Result: Sequence Length
slice_map[max_seq_len*2], seq_map[num_seq];
seq_len = 1, max_id = 0;
/* Get slice mappings for two max sequence lengths of
   memory                                              */
get_slice_values(buffer, max_seq_len*2, slice_map);
/* Try each sequence length increasing by powers of 2  */
while seq_len < max_seq_len do
    /* Find sequence IDs using seq_len and num_seq      */
    seq_data = fill_seq_data(slice_map, seq_len, num_seq);
    /* Find the maximum sequence ID from the sequences  */
    max_id = find_max_seq_id(seq_data);
    /* Increment sequence length to the next power of 2 */
    if seq_len < max_id then
        seq_len = seq_len ≪ 1;
    else
        return seq_len;
    end
end
```

4.3 Searching for Adjacent Addresses

As in previous work, we find several pairs of physical addresses which differ on one bit. Formally, this can be represented as a pair of addresses a and $a \oplus 2^k$ (which we refer to as a'). To retrieve as much of the slice mapping as possible, our method finds pairs of addresses up to the maximum addressable bit determined by the amount of memory in the machine. From the initialisation section, `ADDR_BITS` defines the number of adjacent address pairs required. We consider the machine's memory as the search space for finding adjacent addresses and use huge pages to get larger 2MB contiguous blocks of memory to speed up the search process.

Our tool creates a buffer 75% the size of total available memory using `mmap` to facilitate this search. Through the use of `/proc/pid/pagemap` we translate virtual addresses into physical addresses. Our tool starts searching from the beginning of the buffer for a physical address with bit k set. We skip lower bits related to the cache line index and sequence offset as per the following:

$$k = MSB(sequence\ bits \times cache\ line\ bits)$$
$$\therefore k = MSB(log_2(\texttt{SEQ_LEN}) \times log_2(\texttt{LLC_CACHELINE}))$$

When the search finds a physical address bit k set, separate threads on each available core search through equally divided portions of the buffer for adjacent addresses. To further hasten this process, the `PAGE_BITS` variable jumps within a contiguous huge page to find adjacent addresses at known offsets. When k is larger than the bits revealed by huge pages, we check the physical address of the start of a huge page to check if this could be an adjacent address. When the search finds enough adjacent addresses on bit k, the search resumes in the buffer for an address with bit $k+1$ set. The search continues until $k = $ `ADDR_BITS`.

4.4 Determining Adjacent Address Slice Mappings

When the tool finds sufficient adjacent address pairs, their slice mappings must be found. The performance counter and timing methods described in Section 4.1 allow this. For a linear function, we require mapping for each adjacent address a and a'. For non-linear functions, we require the mappings for the sequences starting with a and a', resulting in sequences A and A'. Each of the slice mappings building a sequence represent individual 64-byte cache lines. The previously found sequence length provides the correct number of these. Table 4.2 and Table 4.3 display two example sets of slice mapping data for both linear and non-linear hash functions.

Bit k	Adjacent Addresses	Slice Mapping
6	0x268600000	0
	0x268600040	1
7	0x268600040	1
	0x2686000c0	3
	...	
33	0x5ce800000	1
	0x7ce800000	2

Table 4.2: A 4-core Intel Core i7-6700K with a linear function gives the above mappings for each address.

Note, as we use sequences here, the tool starts retrieving slice information from bit 13 for the following data. Refer to Figure 4.1 for a visualisation of this.

4.5 Computing XOR-Reduction Map

Finding the XOR-reduction map allows for either an address's slice to be determined in the case of a linear function, or for an address's sequence ID to be calculated for non-linear functions. This map represents a series of XOR operations to compute the slice index, which occur depending on if a physical address's bit is set or not, reducing a larger set of address bits into a smaller value. Later in Chapter 5, we explain how

Bit k	Adjacent Addresses	Sequence Mapping
13	0x117000000 0x117002000	4341 5250 ... 4143 1032 4143 5052 ... 2301 5250
14	0x117002000 0x117006000	4143 5052 ... 2301 5250 5052 0123 ... 5250 4341
	...	
33	0x186600000 0x386600000	5052 4143 ... 5250 2301 3210 4341 ... 1032 4143

Table 4.3: Sequence mappings for a 6-core Intel Core i7-9850H with a non-linear function.

Bit k	Adjacent Addresses	Slice Mapping	XOR Map
6	0x268600000 0x268600040	0 1	1
7	0x268600040 0x2686000c0	1 3	2
	...		
33	0x5ce800000 0x7ce800000	2 2	3

Table 4.4: Example XOR-Reduction for bits 13, 14 and 33 using the same Intel Core i7-6700K as Table 4.2.

this can be divided into a series of masks to allow easy comparison, and provide further insights into Intel's design choices for these hash functions.

4.5.1 Linear Functions

For a linear function, retrieving the mapping is trivial. The only operation required is to XOR the slices for a and a' for each bit k:

$$xor_map_k = Slice(a) \oplus Slice(a')$$

Applying this to the results in Table 4.2, we see the resulting XOR in Table 4.4 used when bit k is set in any given address for this specific processor. Therefore by collecting all XOR operations for the map we retrieve the entire linear function. This allows for calculating the slice index for an arbitrary address a.

$$\begin{aligned} xor_map(a_n...a_0) &= xor_map_k \oplus a_n \; ... \; \oplus xor_map_0 \oplus a_0 \\ Hash(a) &= xor_map(a) \end{aligned}$$

Bit k	Adjacent Addresses	Sequence Mapping	ID	XOR Map
13	0x117000000	4341 5250 ... 4143 1032	0x00	0x45
	0x117002000	4143 5052 ... 2301 5250	0x45	
14	0x117002000	4143 5052 ... 2301 5250	0x45	0x4f
	0x117006000	5052 0123 ... 5250 4341	0x0a	
		...		
33	0x186600000	5052 4143 ... 5250 2301	0x23	0x0e
	0x386600000	3210 4341 ... 1032 4143	0x2d	

Table 4.5: Example XOR-Reduction for bits 13, 14 and 33 using Sequence IDs, from the same Intel Core i7-9850H as Table 4.3.

4.5.2 Non-Linear Functions

Retrieving the XOR-reduction for non-linear hash functions involves more steps than linear functions. Recall the method from [YGL+15] where each adjacent address defines the start of a sequence, which we term here as A. The adjacent address's slice begins the sequence at offset 0, with offset 1 corresponding to the following 64–byte cache-line and so on. This sequence has a corresponding permutation ID as determined by a 'root' comparison sequence. In our approach, we use the first adjacent address sequence as the root sequence, resulting in this having $ID = 0$. Therefore, to find the correct permutation, the tool brute forces the value for ID until the following occurs for each slice's offset o in the given sequence A and root sequence R:

$$R_o = A_{o \oplus ID}$$

Here, the maximum possible ID corresponds with the maximum sequence length found earlier in Section 4.2.1. With the ID computed for a pair of adjacent address sequences A and A', the XOR-reduction can now be recovered. For each addressable bit k we find the XOR of the two IDs:

$$xor_map_k = ID(A) \oplus ID(A')$$

This results in the sample results from a 6-core processor in Table 4.5, completing the retrieval of the XOR-reduction on each addressable bit of memory for the non-linear function. Instead of finding the slice directly for an address, we instead compute its sequence ID.

$$xor_map(a_n...a_0) = xor_map_n \oplus a_n \dots \oplus xor_map_0 \oplus a_0$$
$$ID(a) = xor_map(a)$$

4.5.3 Correctness

To confirm results, the tool can be configured to collect as many pairs of adjacent addresses as required. In doing so, it averages the retrieved XOR value and rounds

this to the closest integer. This provides a form of error correction if the HPC or timing methods incorrectly determine an address's slice value, resulting in an incorrect XOR-reduction stage. The tool's output reports such occurrences to the user upon retrieval of the function.

4.6 Master Sequence Retrieval for Non-Linear Functions

Overall, the non-linear functions can be described as a reduction of physical address bits into a sequence ID, followed by a further reduction on the value of ID into the specific slice values (in this case 0–5 for a 6–core processor):

$$Address\{0,1\}^{\texttt{ADDR_BITS}} \rightarrow ID\{0,1\}^{log_2(\texttt{SEQ_LEN})} \rightarrow Second\ Stage\{0,1,2,3,4,5\}$$

From previous work by Yarom et al. [YGL+15], due to the linearity of the sequence ID mapping each possible ID always maps to the same slice, allowing for the secondary stage to be inferred. The tool collects a large sample of addresses, several times the sequence length and finds their sequence IDs (via XOR-reduction) and slice mappings (from Section 4.1). This creates a set of pairs for each ID mapping to the same slice, in turn creating a 'master' or 'base' sequence (as in [McC21]). We refer to this as M. This sequence represents the output of a logical circuit in the processor further reducing the XOR-reduction bits, as described by Yarom et al. Table 4.6 shows an example M from a 6–core processor.

ID	0x00	0x01	0x02	0x03	...	0x7c	0x7d	0x7e	07f
Slice	0	1	2	3	...	3	4	1	4

Table 4.6: Excerpt from the master sequence retrieved from an Intel Core i7-9580H.

This master sequence may be transferred between processors with the same hash function, and used to determine an address a's slice by using its sequence offset bits:

$$offset(a) = a\{0,1\}^{log_2(\texttt{SEQ_LEN})...log_2(\texttt{LLC_CACHELINE})}$$
$$Hash(a) = M_{ID(a) \oplus offset(a)}$$

4.6.1 Correctness

Similar to Section 4.5, the tool takes averages for each $ID \rightarrow Slice$ mapping in order to remove any possible errors and determine M properly.

4.7 Output

Once the tool has completed the retrieval of the mapping function, it outputs several pieces of information for the user. It provides architectural information about the CPU

first, and following this all the the adjacent address pairs found, along with their slice or sequence mappings. From this, the tool prints the values for XOR-reduction along with information about the averaging process, indicating any incorrect address slice mappings. The tool presents these XOR-reductions in a variety of formats for use in future research. These are:

- XOR map C array.
- XOR permutation mask C array.
- Mathematical XOR of each address bit to get either $slice_k$ or ID_k.
- Visualisation using blocks to indicate XOR on an address bit.

For non-linear functions we provide the master sequence in numerical format, and also using the 'block' visualisation method as used for XOR-reductions. We present example command line outputs for both for linear and non-linear functions in the appendix. After this, several random address slice index values print for a sanity check on the calculated slice compared to the computed slice.

4.7.1 Use of XOR Permutation Masks

In previous work, splitting the XOR-reductions into several distinct XOR masks concisely represents the operations carried out on the physical address bits for each bit of the calculated reduction value. McCalpin [McC21] terms these *permutation select masks* as they permute the address linearly into distinct slice or sequence ID values. These XOR masks determine bit k of either the slice or sequence ID of an address. We utilise these through the following method, by finding whether each mask would determine the $slice_k$ to be 0 or 1 based off count of the address bit. Listing 4.2 shows our implementation of their use for quickly determining the XOR-reduction.

$$comparison(Address) = mask_k\{0,1\}^{\texttt{ADDR_BITS}} \; \& \; Address\{0,1\}^{\texttt{ADDR_BITS}}$$

$$XOR_Reduction\{0,1\}^k = count_bits(comparison(a)) \mod 2$$

```
1  /* For an Intel Core i7-6700K */
2  uint64_t mask[2] = {0x035f575440ULL, 0x06b5faa880ULL};
3
4  void permute_address(uint64_t physical_address, uint64_t
       *mask)
5  {
6      uint64_t result = 0;
7      for (int i = 0; i < sizeof(mask); ++i)
8          result = (result << 1) | (count_bits(mask[i] &
               physical_address) % 2);
9      return result;
10 }
```

Listing 4.2: C code to permute an address, adapted from [McC21] and [VG20].

5 Results

We present our findings on a range of Intel's consumer processors, highlighting some recent changes made to the slice mapping functions. To represent the results, we use a combination of block visualisations and permutation masks for a concise method to compare XOR-reductions across processors. Processors with the same function are grouped together. To further describe the slice hash functions, the first 128 cache line slices for each processor are included to visualise the slice mapping. For non-linear functions we observe this corresponds to the master sequence.

5.1 Retrieved Linear Functions

We provide the results from nine different Intel Core processors, each implementing the same linear hash function across several generations, with the exception of the latest 11th generation of processors. Upper bits of the hash function may be missing due to a lack of RAM in the block visualisations, which causes the upper hexadecimal characters of the XOR masks to similarly appear different.

5.1.1 2–Core Results

We tested the tool on two dual-core Kaby Lake mobile processors, an Intel Core i7-7500U and an i5-7200U. These processors require only a single permutation mask to select between slice 0 and 1, shown in Table 5.1. We provide the block visualisation of this function in Figure 5.1.

5.1.2 4–Core Results

Our tool retrieved the hash function from several 4–core processors across the Haswell (4th), Skylake (6th) and Kaby Lake (7th) processor generations. We found Intel Core i7-4790, i5-7600K, i7-6700, i7-6700K, i7-7700 and QKYP engineering sample (corresponding to an i7-7700K) processors all feature the same hash function. As these models contain 4 physical cores, the linear function splits into two permutation masks as shown in Table 5.2. We provide the block visualisation of this function in Figure 5.2. From this, we can confirm the hash function did not change for 6th and 7th generation processors which were released after the work by Maurice et al. [MSN+15]. Further, the permutation mask for $slice_0$ in Table 5.2 is the same as in Table 5.1, further confirming their results.

5.1.3 8–Core Results

Regrettably we lack results from 8th, 9th or 10th generation 8–core models as we found the slice mapping changed for 11th generation Rocket Lake processors. Tested on an Intel Core i7-11700KF, the permutation masks in Table 5.3 differ from previous functions, apart from $slice_0$. However, there exists some similarities in recent work by McCalpin [McC21]. Their 16–core Xeon reverse engineered function features the same $slice_1$ mask as their $slice_3$ mask, highlighting the common ground between the old and new functions. We provide the block visualisation of this function in Figure 5.3.

Permutation Mask	
$slice_0$	`0x15f575440`

Table 5.1: 2–Core permutation mask.

Permutation Masks	
$slice_0$	`0x35f575440`
$slice_1$	`0x6b5faa880`

Table 5.2: 4–Core permutation masks.

Permutation Masks	
$slice_0$	`0xb5f575440`
$slice_1$	`0x1aeeb1200`
$slice_2$	`0x6d87f2c00`

Table 5.3: 8–Core permutation masks.

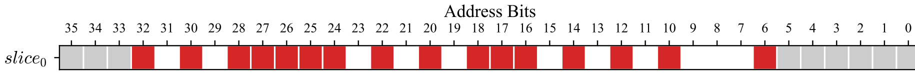

Figure 5.1: Block visualisation of the 2–core processor hash function.

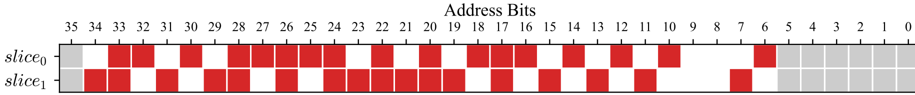

Figure 5.2: Block visualisation of the 4–core processor hash function.

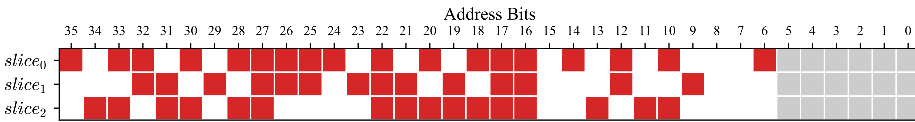

Figure 5.3: Block visualisation of the 8–core processor hash function.

5.2 Retrieved Non-Linear Functions

We retrieved two different types of non-linear functions from four different processors, one common 6-core hash function and another for 10-core processors. Most notably, both hash functions use the same XOR-reduction step with exact same permutation masks. The second stage master sequence however differs between them.

5.2.1 6–Core Results

Our 6–core hash function results come from three processors, an Intel i7-8850H, i7-9850H and an i7-10710U. We provide the permutation masks for these in Table 5.4. These masks reduce physical address bits 13 and up into an ID between 0 and 128. A block visualisation accompanies this in Figure 5.4, displaying a similar function as reverse engineered by Yarom et al. [YGL+15] from an Intel Xeon E5-2430. Table 5.5 shows the raw master sequence values, with a block visualisation provided in Figure 5.5. The master sequence distributes addresses lower for slices 0–3, with 21/128 going into each of these four LLC slices, and 22/128 being placed into both slices 4 and 5.

5.2.2 10–Core Results

The i9-10900K is the most recent consumer processor to feature a 10–core, and thus a 10 slice LLC layout. We find its permutation masks to be the exact same as the 6-core processors, as in Table 5.4. The master sequence is the main difference here, which includes higher values to allow indexing into slices 7, 8 and 9. These permutation masks distribute addresses for 13/128 IDs into slices 0–7, falling to 12/128 for slices

8 and 9. These distributions correspond to those found in [IGI+15]. Similarities also exist between the 6 and 10–core master sequences. M_0 has the same bits set for each value of ID across both sequences, as shown in Figure 5.5.

Permutation Masks	
ID_0	0x1ae7be000
ID_1	0x35cf7c000
ID_2	0x717946000
ID_3	0x62f28c000
ID_4	0x45e518000
ID_5	0xbca30000
ID_6	0xd73de000

Table 5.4: Permutation masks for various 6–core and 10–core processors.

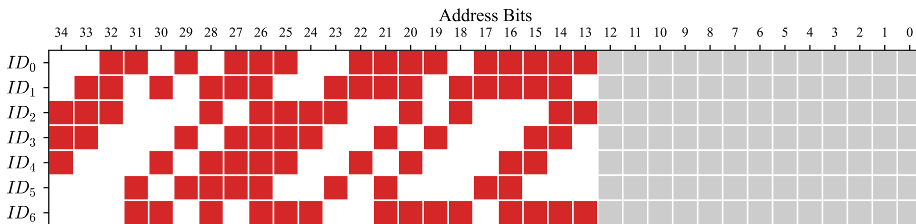

Figure 5.4: Block visualisation of the 6 and 10–core XOR-reduction.

	Master Sequence M
6-core	0123 1434 1032 0525 1032 0525 0525 1434
	0123 5052 5052 4143 1032 4143 4143 5052
	2301 5250 3210 4341 3210 4341 4341 5250
	2301 3414 3414 2505 3210 2505 2505 3414

Table 5.5: Master sequence for various 6–core processors.

	Master Sequence M
8-core	0505 3636 1414 2727 1414 2727 0589 3698
	4141 7272 5098 6389 5050 6363 4189 7298
	4141 7272 5050 6363 5050 6363 8941 9872
	0505 3636 9814 8927 1414 2727 8905 9836

Table 5.6: Master sequence for the 10–core hash function.

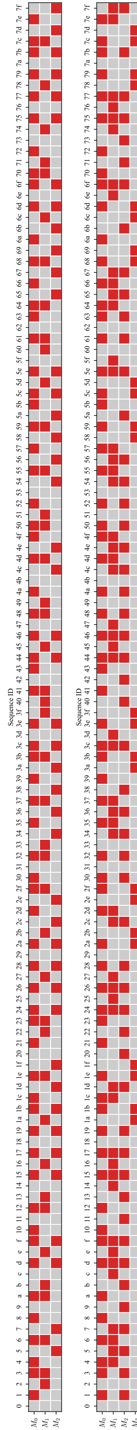

Figure 5.5: Block visualisation comparison between 6–core and 10–core master sequences.

5.3 Discussion and Future Work

5.3.1 Common Ground between Linear and Non-Linear Functions

Most of the functions retrieved using our tool come from top of the line processors which did not experience failure during fabrication. To save costs, CPU manufacturers 'bin' partially failed CPUs into lower specification models by disabling poorly functioning cores. This may influence the hash function and create a slice mapping different to that of other models.

To facilitate processor binning during fabrication, it makes sense for Intel to design the hash function with the ability to adapt dealing with disabled cores and slices. Therefore, any common ground between different types of linear and non-linear functions may expose the hash function's underlying design.

As identified in Section 5.1.3, there exists a common permutation masks between the linear functions. Specifically, `0x31aeeb100` feature in both Skylake and Rocket Lake linear functions for $slice_1$ and $slice_3$ respectively, from [McC21]. However, their use differs between hash functions due to an overall change in slice mappings for the more recent Rocket Lake architecture. We could not find any common or 'root' XOR masks which the hash function could be using to generate these masks for linear functions. However, for the retrieved 6 and 10-core non-linear functions, we find all of the XOR masks to be generated from permutations of the value `0x35cf7c`.

$$
\begin{aligned}
ID_0 &= \texttt{0x35cf7c} \ll \texttt{0xB} \\
ID_1 &= \texttt{0x35cf7c} \ll \texttt{0xC} \\
ID_2 &= (\texttt{0x35cf7c} \oplus (\texttt{0x35cf7c} \gg \texttt{0x2})) \ll \texttt{0xD} \\
ID_3 &= (\texttt{0x35cf7c} \oplus (\texttt{0x35cf7c} \gg \texttt{0x2})) \ll \texttt{0xE} \\
ID_4 &= (\texttt{0x35cf7c} \oplus (\texttt{0x35cf7c} \gg \texttt{0x2})) \ll \texttt{0xF} \\
ID_5 &= (\texttt{0x35cf7c} \oplus (\texttt{0x35cf7c} \gg \texttt{0x2})) \ll \texttt{0x10} \\
ID_6 &= \texttt{0x35cf7c} \ll \texttt{0xA}
\end{aligned}
$$

Storing a root value such as this allows the hash function to quickly calculate the permutation masks, reducing total space required in the uncore section of the processor.

5.3.2 Restricted Address Slice Mappings

Table 5.7 displays the slice mappings for the first 128 memory slice mappings from each type of function we have encountered. To retrieve these we start from address zero and increment by 64-bytes due to the hash functions disregarding cache line offset bits. However as such addresses cannot be measured directly due to their memory location in the upper kernel, they are found by directly computing the slice index values from the retrieved hash functions. This provides another insight into the operation of the functions, and the variations between each. We clearly see the effect of the

	First 128 Cache Line Slice Mappings
2-core	0101 0101 0101 0101 1010 1010 1010 1010 0101 0101 0101 0101 1010 1010 1010 1010 1010 1010 1010 1010 0101 0101 0101 0101 1010 1010 1010 1010 0101 0101 0101 0101
4-core	0123 0123 0123 0123 1032 1032 1032 1032 2301 2301 2301 2301 3210 3210 3210 3210 1032 1032 1032 1032 0123 0123 0123 0123 3210 3210 3210 3210 2301 2301 2301 2301
6-core	0123 1434 1032 0525 1032 0525 0525 1434 0123 5052 5052 4143 1032 4143 4143 5052 2301 5250 3210 4341 3210 4341 4341 5250 2301 3414 3414 2505 3210 2505 2505 3414
8-core	0101 0101 2323 2323 5454 5454 7676 7676 4545 4545 6767 6767 1010 1010 3232 3232 3232 3232 1010 1010 6767 6767 4545 4545 7676 7676 5454 5454 2323 2323 0101 0101
10-core	0505 3636 1414 2727 1414 2727 0589 3698 4141 7272 5098 6389 5050 6363 4189 7298 4141 7272 5050 6363 5050 6363 8941 9872 0505 3636 9814 8927 1414 2727 8905 9836

Table 5.7: First 128 slice mappings from address 0 for all retrieved functions.

XOR operations applied to the physical address bits for each of the linear functions. However, the mapping becomes more complicated for the non-linear functions due to further permutation of ID with the master sequence.

5.3.3 Further Improvements and Research

Our performance counter interface does not currently support Xeon processors due to differences in performance monitoring hardware, thus we cannot monitor the CHAs in these processors to find the slice mappings of any physical address. Therefore, we cannot determine hash functions in such CPUs at this time. Also, the interface is limited to x86-64 architecture only, with further possible development to allow its use on ARM processors.

There are several pathways to improve retrieval of the hash function. When searching for addresses, huge pages could be dynamically resized to allocate larger portions of contiguous memory than 2MB, completing the search faster than our current method.

In [YGL+15] the second stage of the hash function was not represented as a master sequence, but rather as a logical circuit operating on the bits of an address's sequence ID. Determining this circuit, which is most likely to be most representative of the hash function in the processor could be accomplished with an SAT solver, however we leave

this for future work.

We have not tested whether dual CPU systems share the hash function between their two separate uncores, or if the hash functions are specific to each processor. Intel Core processors cannot function in a dual CPU configuration, therefore work to enable retrieval of the slice mapping on Xeon processors is required.

6 Conclusion

In this work, we explore the nature of an undisclosed last-level cache hash function in several of Intel's consumer processors. In doing so, we mitigate any related hindrance to further cross-core side-channel research through our contribution of an automatic hash function retrieval tool. This function distributes memory accesses evenly across several last-level cache slices, stemming from Intel's design decision to use a sliced cache to increase overall bandwidth to the processor uncore. However, the lack of disclosure surrounding this negatively affects offensive security research, especially in the realm of cross-core side-channel attacks [LYG+15] which are significantly more difficult to perform without knowledge of address slice indexes. Understanding these attacks is crucial for their effective mitigation. Although through obscurity the slice mapping function implicitly hinders such low-level attacks, reverse engineering and retrieval of the function completely removes any protection provided.

To allow low-level reverse engineering of Intel processors, we present an interface to measure the micro-architectural effects of a user defined function using in-built hardware performance counters. This interface underpins our hash function retrieval tool, allowing us to determine which LLC slice an arbitrary memory address maps to, giving insights into the hash function implemented for many processors. It provides an easy method to access both x86-64 and uncore micro-architectural events on finely executed code sections.

With our automatic hash function retrieval tool, we successfully reverse engineer and retrieve the slice mappings from 2, 4, 6, 8 and 10-core processors. Drawing from select previous works, we consider the hash functions as a series of XOR operations on physical memory address bits, reducing the large memory address value into a smaller 2^n number. By searching for addresses which differ on one bit, our tool retrieves this operation successfully from all processors tested. For non-linear

functions, the tool recovers a secondary stage in the form of a master sequence the processor uses to index addresses into a LLC with non-2^n slices. From this, we learn the changes in Intel's linear slice mapping functions since last publications on this topic from 2015 [MSN+15]. Additionally, we show further reuse of similar functions for 6 and 10–core processors over the past decade, due to similarity with these when compared to other previous works [YGL+15, IGI+15]. From this, we describe how the XOR-reduction masks from non-linear functions relate to each other via a root XOR mask.

Through our contributions of a simple to use performance counter interface as well an automated slice mapping retrieval tool, we provide effective methods to discover the internal design of Intel processors. This brings forth a conducive environment for further research on securing their shared last-level caches and ensuring their trustworthiness in the face of low-level hardware side-channel attacks.

Appendix

Sample Tool Output for i7-6700K Linear Function

```
1  ...
2  Bit: 06 | Total: 002 | Count: 02 | ID_Rounded: 0x1 | ID:
       1.000000
3  Bit: 07 | Total: 004 | Count: 02 | ID_Rounded: 0x2 | ID:
       2.000000
4  Bit: 08 | Total: 000 | Count: 00 | ID_Rounded: 0x0 | ID: -nan
5  Bit: 09 | Total: 000 | Count: 00 | ID_Rounded: 0x0 | ID: -nan
6  Bit: 10 | Total: 002 | Count: 02 | ID_Rounded: 0x1 | ID:
       1.000000
7  Bit: 11 | Total: 004 | Count: 02 | ID_Rounded: 0x2 | ID:
       2.000000
8  Bit: 12 | Total: 002 | Count: 02 | ID_Rounded: 0x1 | ID:
       1.000000
9  Bit: 13 | Total: 004 | Count: 02 | ID_Rounded: 0x2 | ID:
       2.000000
10 Bit: 14 | Total: 002 | Count: 02 | ID_Rounded: 0x1 | ID:
       1.000000
11 Bit: 15 | Total: 004 | Count: 02 | ID_Rounded: 0x2 | ID:
       2.000000
12 Bit: 16 | Total: 002 | Count: 02 | ID_Rounded: 0x1 | ID:
       1.000000
13 Bit: 17 | Total: 006 | Count: 02 | ID_Rounded: 0x3 | ID:
       3.000000
14 Bit: 18 | Total: 002 | Count: 02 | ID_Rounded: 0x1 | ID:
       1.000000
15 Bit: 19 | Total: 004 | Count: 02 | ID_Rounded: 0x2 | ID:
       2.000000
16 Bit: 20 | Total: 006 | Count: 02 | ID_Rounded: 0x3 | ID:
       3.000000
17 Bit: 21 | Total: 004 | Count: 02 | ID_Rounded: 0x2 | ID:
       2.000000
18 Bit: 22 | Total: 006 | Count: 02 | ID_Rounded: 0x3 | ID:
       3.000000
19 Bit: 23 | Total: 004 | Count: 02 | ID_Rounded: 0x2 | ID:
       2.000000
20 Bit: 24 | Total: 006 | Count: 02 | ID_Rounded: 0x3 | ID:
       3.000000
21 Bit: 25 | Total: 002 | Count: 02 | ID_Rounded: 0x1 | ID:
       1.000000
22 Bit: 26 | Total: 006 | Count: 02 | ID_Rounded: 0x3 | ID:
       3.000000
```

```
23 Bit: 27 | Total: 002 | Count: 02 | ID_Rounded: 0x1 | ID:
       1.000000
24 Bit: 28 | Total: 006 | Count: 02 | ID_Rounded: 0x3 | ID:
       3.000000
25 Bit: 29 | Total: 004 | Count: 02 | ID_Rounded: 0x2 | ID:
       2.000000
26 Bit: 30 | Total: 002 | Count: 02 | ID_Rounded: 0x1 | ID:
       1.000000
27 Bit: 31 | Total: 004 | Count: 02 | ID_Rounded: 0x2 | ID:
       2.000000
28 Bit: 32 | Total: 002 | Count: 02 | ID_Rounded: 0x1 | ID:
       1.000000
29 Bit: 33 | Total: 006 | Count: 02 | ID_Rounded: 0x3 | ID:
       3.000000
30 Bit: 34 | Total: 004 | Count: 02 | ID_Rounded: 0x2 | ID:
       2.000000
31
32 The following XOR reduction map can be used to get the
       sequence ID of an address
33 int xor_map[35] = {0, 0, 0, 0, 0, 0, 1, 2, 0, 0, 1, 2, 1, 2,
       1, 2, 1, 3, 1, 2, 3, 2, 3, 2, 3, 1, 3, 1, 3, 2, 1, 2, 1,
       3, 2};
34
35 The following mask can be used to get the sequence ID of an
       address
36 uint64_t mask[2] = {
37 0x035f575440ULL,
38 0x06b5faa880ULL};
39
40 Showing the XOR bits for each bit of the address to find the
       sequence ID
41 ID0 = A06 ^ A10 ^ A12 ^ A14 ^ A16 ^ A17 ^ A18 ^ A20 ^ A22 ^
       A24 ^ A25 ^ A26 ^ A27 ^ A28 ^ A30 ^ A32 ^ A33
42 ID1 = A07 ^ A11 ^ A13 ^ A15 ^ A17 ^ A19 ^ A20 ^ A21 ^ A22 ^
       A23 ^ A24 ^ A26 ^ A28 ^ A29 ^ A31 ^ A33 ^ A34
43
44 Showing ID mapping split into binary (MSB->LSB) (with the
       bits belonging to cache line index stripped)
45 ID0 = | ## # ##### # # ### # # #   #|
46 ID1 = |## # ## # ###### # # # #   # |
47
48 No master sequence, number of processor cores is a power of
       2.
49
50 Testing first 128 cache lines starting from address 0x0:
51 0123 0123 0123 0123 1032 1032 1032 1032 2301 2301 2301 2301
       3210 3210 3210 3210 1032 1032 1032 1032 0123 0123 0123
       0123 3210 3210 3210 3210 2301 2301 2301 2301
52
53 Testing found slice mapping function with some random values:
54 Phys Addr: 0x59bcb4567 | Calc Slice: 1 | Real Slice: 1
55 Phys Addr: 0x54adb23c6 | Calc Slice: 1 | Real Slice: 1
56 Phys Addr: 0x710dc9869 | Calc Slice: 2 | Real Slice: 2
57 Phys Addr: 0x6c8334873 | Calc Slice: 2 | Real Slice: 2
58 Phys Addr: 0x14790dc51 | Calc Slice: 0 | Real Slice: 0
59 Phys Addr: 0x049a95cff | Calc Slice: 0 | Real Slice: 0
```

```
60 Phys Addr: 0x5e8e8944a | Calc Slice: 1 | Real Slice: 1
61 Phys Addr: 0x8171558ec | Calc Slice: -1 | Real Slice: 3
62 Phys Addr: 0x6884e1f29 | Calc Slice: 3 | Real Slice: 3
63 Phys Addr: 0x24a087ccd | Calc Slice: 2 | Real Slice: 2
64 Phys Addr: 0x17bfb58ba | Calc Slice: 2 | Real Slice: 2
65 Phys Addr: 0x5751ed7ab | Calc Slice: 1 | Real Slice: 1
66 Phys Addr: 0x7763141f2 | Calc Slice: 0 | Real Slice: 0
67 Phys Addr: 0x5aa771efb | Calc Slice: 3 | Real Slice: 3
68 Phys Addr: 0x7ef42a9e3 | Calc Slice: 0 | Real Slice: 0
69 Phys Addr: 0x692c5e146 | Calc Slice: 0 | Real Slice: 0
70 Phys Addr: 0x16c3f007c | Calc Slice: 1 | Real Slice: 1
71 Phys Addr: 0x7461062c2 | Calc Slice: 0 | Real Slice: 0
72 Phys Addr: 0x021400854 | Calc Slice: 1 | Real Slice: 1
73 Phys Addr: 0x64a5127f8 | Calc Slice: 1 | Real Slice: 1
74 Phys Addr: 0x06a76231b | Calc Slice: 1 | Real Slice: 1
75 Phys Addr: 0x1b8f6e9e8 | Calc Slice: 1 | Real Slice: 1
76 Phys Addr: 0x059d0cde7 | Calc Slice: 1 | Real Slice: 1
77 Phys Addr: 0x682cf438d | Calc Slice: 1 | Real Slice: 1
78 Phys Addr: 0x04ece0f76 | Calc Slice: 1 | Real Slice: 1
79 Phys Addr: 0x7cfd2255a | Calc Slice: 0 | Real Slice: 0
80 Phys Addr: 0x017dcf92e | Calc Slice: 1 | Real Slice: 1
81 Phys Addr: 0x06f4d7263 | Calc Slice: 1 | Real Slice: 1
82 Phys Addr: 0x807bcc233 | Calc Slice: -1 | Real Slice: 1
83 Phys Addr: 0x02a0fd79f | Calc Slice: 2 | Real Slice: 2
84 Phys Addr: 0x5aa67c4c9 | Calc Slice: 0 | Real Slice: 0
85 Phys Addr: 0x58148079a | Calc Slice: 0 | Real Slice: 0
```

Listing 1: Shortened raw output from running the slice mapping retrieval tool on an i7-6700K.

Sample Tool Output for i7-9850H Non-Linear Function

```
1  ...
2  Bit: 13 | Total: 138 | Count: 02 | ID_Rounded: 0x45 | ID:
       69.000000
3  Bit: 14 | Total: 158 | Count: 02 | ID_Rounded: 0x4f | ID:
       79.000000
4  Bit: 15 | Total: 182 | Count: 02 | ID_Rounded: 0x5b | ID:
       91.000000
5  Bit: 16 | Total: 230 | Count: 02 | ID_Rounded: 0x73 | ID:
       115.000000
6  Bit: 17 | Total: 070 | Count: 02 | ID_Rounded: 0x23 | ID:
       35.000000
7  Bit: 18 | Total: 140 | Count: 02 | ID_Rounded: 0x46 | ID:
       70.000000
8  Bit: 19 | Total: 146 | Count: 02 | ID_Rounded: 0x49 | ID:
       73.000000
9  Bit: 20 | Total: 174 | Count: 02 | ID_Rounded: 0x57 | ID:
       87.000000
10 Bit: 21 | Total: 214 | Count: 02 | ID_Rounded: 0x6b | ID:
       107.000000
11 Bit: 22 | Total: 038 | Count: 02 | ID_Rounded: 0x13 | ID:
       19.000000
12 Bit: 23 | Total: 076 | Count: 02 | ID_Rounded: 0x26 | ID:
       38.000000
13 Bit: 24 | Total: 152 | Count: 02 | ID_Rounded: 0x4c | ID:
       76.000000
14 Bit: 25 | Total: 186 | Count: 02 | ID_Rounded: 0x5d | ID:
       93.000000
15 Bit: 26 | Total: 254 | Count: 02 | ID_Rounded: 0x7f | ID:
       127.000000
16 Bit: 27 | Total: 118 | Count: 02 | ID_Rounded: 0x3b | ID:
       59.000000
17 Bit: 28 | Total: 236 | Count: 02 | ID_Rounded: 0x76 | ID:
       118.000000
18 Bit: 29 | Total: 082 | Count: 02 | ID_Rounded: 0x29 | ID:
       41.000000
19 Bit: 30 | Total: 164 | Count: 02 | ID_Rounded: 0x52 | ID:
       82.000000
20 Bit: 31 | Total: 194 | Count: 02 | ID_Rounded: 0x61 | ID:
       97.000000
21 Bit: 32 | Total: 014 | Count: 02 | ID_Rounded: 0x7 | ID:
       7.000000
22 Bit: 33 | Total: 028 | Count: 02 | ID_Rounded: 0xe | ID:
       14.000000
23
24 The following XOR reduction map can be used to get the
       sequence ID of an address
25 int xor_map[34] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       69, 79, 91, 115, 35, 70, 73, 87, 107, 19, 38, 76, 93,
       127, 59, 118, 41, 82, 97, 7, 14};
26
27 The following mask can be used to get the sequence ID of an
       address
28 uint64_t mask[7] = {
29 0x01ae7be000ULL,
```

```
30  0x035cf7c000ULL,
31  0x0317946000ULL,
32  0x022f28c000ULL,
33  0x005e518000ULL,
34  0x00bca30000ULL,
35  0x00d73de000ULL};
36
37  Showing the XOR bits for each bit of the address to find the
        sequence ID
38  ID0 = A13 ^ A14 ^ A15 ^ A16 ^ A17 ^ A19 ^ A20 ^ A21 ^ A22 ^
        A25 ^ A26 ^ A27 ^ A29 ^ A31 ^ A32
39  ID1 = A14 ^ A15 ^ A16 ^ A17 ^ A18 ^ A20 ^ A21 ^ A22 ^ A23 ^
        A26 ^ A27 ^ A28 ^ A30 ^ A32 ^ A33
40  ID2 = A13 ^ A14 ^ A18 ^ A20 ^ A23 ^ A24 ^ A25 ^ A26 ^ A28 ^
        A32 ^ A33
41  ID3 = A14 ^ A15 ^ A19 ^ A21 ^ A24 ^ A25 ^ A26 ^ A27 ^ A29 ^
        A33
42  ID4 = A15 ^ A16 ^ A20 ^ A22 ^ A25 ^ A26 ^ A27 ^ A28 ^ A30
43  ID5 = A16 ^ A17 ^ A21 ^ A23 ^ A26 ^ A27 ^ A28 ^ A29 ^ A31
44  ID6 = A13 ^ A14 ^ A15 ^ A16 ^ A18 ^ A19 ^ A20 ^ A21 ^ A24 ^
        A25 ^ A26 ^ A28 ^ A30 ^ A31
45
46  Showing ID mapping split into binary (MSB->LSB) (with the
        bits belonging to cache line index stripped)
47  ID0 = | ## # ###  #### #####       |
48  ID1 = |## # ###  #### #####        |
49  ID2 = |##   # ####  # #   ##       |
50  ID3 = |#   # ####  # #   ##        |
51  ID4 = |   # ####  # #   ##         |
52  ID5 = |  # ####  # #   ##          |
53  ID6 = |  ## # ###  #### ####       |
54
55  Master Sequence:
56  0123 1434 1032 0525 1032 0525 0525 1434 0123 5052 5052 4143
        1032 4143 4143 5052 2301 5250 3210 4341 3210 4341 4341
        5250 2301 3414 3414 2505 3210 2505 2505 3414
57
58  Master Sequence (split into binary, to better visualise its
        effect on each bit of the XOR reduction/mask)
59  M0 = | # ## # # #  # ## #  # # # ## #  # ## # # #  # ## #  #
        # # ## #  # ## # # #  # ## #  # # # ## #  # ## # # #  #
        ## #  # # # ## # |
60  M1 = |  ##  #   ##  #   ##  #   #   #   ##   #   #   #  ##
        #   #   ###   #  ##   #  ##   #   #   #  ##  #   #   #
        ##  #   #   #   |
61  M2 = |     # #     # #     # # # # # #    # # # # # #     #
        # # # # #     # #     # #     # # # # # #      # # # # #
        #     # # # # # #|
62
63  Testing first 128 cache lines starting from address 0x0:
64  0123 1434 1032 0525 1032 0525 0525 1434 0123 5052 5052 4143
        1032 4143 4143 5052 2301 5250 3210 4341 3210 4341 4341
        5250 2301 3414 3414 2505 3210 2505 2505 3414
65
66  Testing found slice mapping function with some random values:
67  Phys Addr: 0x02b8eb52b | Calc Slice: 1 | Real Slice: 1
```

```
68 Phys Addr: 0x3f8fbcb8c | Calc Slice: 2 | Real Slice: 2
69 Phys Addr: 0x3bad0d6b6 | Calc Slice: 4 | Real Slice: 4
70 Phys Addr: 0x03ae8cf87 | Calc Slice: 1 | Real Slice: 1
71 Phys Addr: 0x08b2fc41b | Calc Slice: 5 | Real Slice: 5
72 Phys Addr: 0x031a1c464 | Calc Slice: 5 | Real Slice: 5
73 Phys Addr: 0x3e3923bf5 | Calc Slice: 0 | Real Slice: 0
74 Phys Addr: 0x04f769861 | Calc Slice: 5 | Real Slice: 5
75 Phys Addr: 0x3c4b956ab | Calc Slice: 3 | Real Slice: 3
76 Phys Addr: 0x06b5be41c | Calc Slice: 3 | Real Slice: 3
77 Phys Addr: 0x33a1a15e7 | Calc Slice: 4 | Real Slice: 4
78 Phys Addr: 0x3fc5d6690 | Calc Slice: 0 | Real Slice: 0
79 Phys Addr: 0x0667b5c5b | Calc Slice: 2 | Real Slice: 2
80 Phys Addr: 0x4465e5490 | Calc Slice: -1 | Real Slice: 5
81 Phys Addr: 0x040a4be1e | Calc Slice: 2 | Real Slice: 2
82 Phys Addr: 0x025dbb7e5 | Calc Slice: 0 | Real Slice: 0
83 Phys Addr: 0x41ab68302 | Calc Slice: -1 | Real Slice: 5
84 Phys Addr: 0x3dc25c7a8 | Calc Slice: 5 | Real Slice: 5
85 Phys Addr: 0x3acf41011 | Calc Slice: 3 | Real Slice: 3
86 Phys Addr: 0x36e03c777 | Calc Slice: 3 | Real Slice: 3
87 Phys Addr: 0x0571c6e4d | Calc Slice: 1 | Real Slice: 1
88 Phys Addr: 0x3fa4a56cd | Calc Slice: 0 | Real Slice: 0
89 Phys Addr: 0x402f86ee1 | Calc Slice: -1 | Real Slice: 1
90 Phys Addr: 0x07654e53b | Calc Slice: 3 | Real Slice: 3
91 Phys Addr: 0x19f800487 | Calc Slice: 2 | Real Slice: 2
92 Phys Addr: 0x0962b1b60 | Calc Slice: 5 | Real Slice: 5
93 Phys Addr: 0x4306dd574 | Calc Slice: -1 | Real Slice: 3
94 Phys Addr: 0x42ee3808a | Calc Slice: -1 | Real Slice: 5
95 Phys Addr: 0x083d57276 | Calc Slice: 3 | Real Slice: 3
96 Phys Addr: 0x3c96681db | Calc Slice: 0 | Real Slice: 0
97 Phys Addr: 0x02846c974 | Calc Slice: 3 | Real Slice: 3
98 Phys Addr: 0x00cc427a1 | Calc Slice: 2 | Real Slice: 2
```

Listing 2: Shortened raw output from running the slice mapping retrieval tool on an i7-9850H.

Project Media

Poster for *Black Box CPU Reverse Engineering* presented at *Ingenuity 2021*
(Poster on next page)

THE UNIVERSITY of ADELAIDE

Black Box CPU Reverse Engineering

Brad Morgan

Supervisors
Defence Science and Technology Group
The University of Adelaide
Mr Nick Manser
Dr Chitchanok Chuengsatiansup | Dr Yuval Yarom

Australian Government
Department of Defence
Defence Science and Technology Group

Introduction

Motivation

Intel and AMD do not disclose important design details of their processors

- Some vulnerabilities exist for years before discovery (e.g. Spectre, Meltdown)
- We seek to reveal underlying hardware decisions made by Intel
- These are crucial for furthering research into micro-architectural security

Intel Processor Layout

Core 0: L1 Cache, L2 Cache | Core 1: L1 Cache, L2 Cache | .... | Core *n*: L1 Cache, L2 Cache

Uncore: Unknown Hash Function

Slice 0: Last Level Cache | Slice 1: Last Level Cache | .... | Slice *n*: Last Level Cache

Objectives

- ✓ Reverse engineer an undisclosed **hash function** within Intel processors
  - Researchers cannot control memory in an Intel CPU without this
- ✓ Create tool to **automatically retrieve this hash function**
- ✓ Use the tool on a variety of Intel CPUs
- ✓ Report results in a simple format for use in future research

Background

...on an entire field of research in 7 dot points or less

CPU Architecture Crash Course

Current processors use caches to speed up memory accesses

- Each processor core has L1, L2 and Last Level Caches (LLC)
- On access, an address is stored in the CPU's cache for future retrieval
- The LLC is split into several shared slices; **One core, one slice**

Previous Discoveries

- Hash function takes memory address and computes LLC slice
- Evenly distributes memory accesses between each shared slice
- Memory addresses differing by 1 bit reveal how the slice mapping changes
- Machines with a non-2^n number of slices use a **second reduction step**

Observations

Disclaimer:
No CPUs were harmed in the process of conducting this project

Intel's hash function behaves as an XOR operation on the computed slice

- We use pairs of 'adjacent' address **differing on one bit** to find a piece of the hash function
- If an address bit is 1, slice mapping changes in a determinable way

Note: Below experiment conducted on a quad-core Intel i7-6700K
This CPU has a slice range of 0-3 (00-11 in binary)

Adjacent Address (Hex)	Adjacent Address Bits																																		Observed Slice (S) [Binary]
	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x267a00040	1	1	0	0	1	1	0	0	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	S_a = 0 [00]
0x267a00840	1	1	0	0	1	1	0	0	1	1	1	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	S_b = 2 [10]

Methodology

for frying your computer

1) Find adjacent address pairs in available memory

Bit 11 Pair

0x267a00040
0x267a00840

2) Flush each address from CPU cache, observe slice access

CLFLUSH 0x267a0040

Unknown Hash Function

Slice 0: Flush! | Slice 1: No Flush | ... | Slice *n*: No Flush

3) In failure, fall-back to access time

- Each core has an associated LLC slice
- Measure address access time from each processor core; find minimum

S_a = 0 [00]
S_b = 2 [10]

4) Compute XOR operation for bit

- Compare slices with XOR
- Store result as slice mapping

$S_a \oplus S_b = 2$

Therefore, Bit 11 XOR operation is 2 [10]

Results

Hey give me a slice!

Collating each XOR reveals the slice function

- XOR operations are visualised as binary
- If an address bit is 1, an XOR is applied to the result
- Bit 11's XOR is [10] as found previously
- We have a live USB with the tool installed to try it!

Slice Hash Function	Address Bit XOR Operations - Intel i7-6700K																																	
	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit 0																																		
Bit 1																																		

Non-2^n Core Processors: Two Stage Slice Mapping

Intel's above method only works for CPUs with 2^n slices

- 6, 10 & 12 core CPUs **reduce a large 2^n value into a smaller range**
- Our tool adapts and finds *sequences* of slice mappings using the found pairs of adjacent addresses
- A sequence is made by finding the slices for a series of sequential addresses
- The XOR function in 6 & 10 core CPUs generates 128 unique sequences
- We assign each sequence an ID (0-127) to recover the initial XOR function

Sequence ID Hash Function	Address Bit XOR Operations - i9-10900K																				
	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
Bit 0																					
Bit 1																					
Bit 2																					
Bit 3																					
Bit 4																					
Bit 5																					
Bit 6																					

Each ID always maps to the same slice and creates a master sequence

- A logical circuit in the CPU reduces the initial output into slice values 0-9
- To find an address's slice, we initially determine its sequence ID
- Indexing this ID into the master sequence returns the address's slice value

ID Value	Master Sequence Slice Mapping - i9-10900K							
00	0505	3636	1414	2727	1414	2727	0589	3698
32	4141	7272	5098	6389	5050	6363	4189	7298
64	4141	7272	5050	6363	5050	6363	8941	9872
96	0505	3636	9814	8927	1414	2727	8905	9836

Source Code

Documentation and code related to this project is available on the 0xADE1A1DE Git Repository.

Bibliography

[AMD20] AMD. Developer Guides, Manuals & ISA Documents. https://developer.amd.com/resources/developer-guides-manuals/, November 2020.

[AR19] Andreas Abel and Jan Reineke. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. *CoRR*, abs/1911.03282, 2019.

[BT16] Edd Barrett and Laurence Tratt. The user_rdpmc Linux Kernel Module. https://github.com/softdevteam/user_rdpmc, 08 2016.

[CGG+19] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[DKL+17] Jack Doweck, Wen-Fu Kao, Allen Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Effi Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, 37:52–62, 03 2017.

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505, 2011.

[GRBG18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, Baltimore, MD, August 2018. USENIX Association. https://www.usenix.org/conference/usenixsecurity18/presentation/gras.

[HWH13] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, page 191–205, USA, 2013. IEEE Computer Society.

[IES15] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *In Proceedings of the 18th EUROMICRO Conference on Digital System Design*, 2015.

[IGI+15] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. Cryptology ePrint Archive, Report 2015/898, 2015. https://ia.cr/2015/898.

[Int13] Intel. msr-tools 1.3. https://guix.gnu.org/packages/msr-tools-1.3/, September 2013.

[Int15] Intel. Intel® Core™ i7-6700K Processor. https://www.intel.com.au/content/www/au/en/products/sku/88195/intel-core-i76700k-processor-8m-cache-up-to-4-20-ghz/specifications.html, 2015.

[Int16] Intel. 6th Generation Intel® Core™ Processor Family Uncore Performance Monitoring Reference Manual . https://software.intel.com/content/www/us/en/develop/download/6th-generation-intel-core-processor-family-uncore-performance-monitoring-reference-manual.html, 04 2016.

[Int17] Intel. Intel® Xeon® Processor Scalable Family Technical Overview. https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-technical-overview.html, 10 2017.

[Int20] Intel. Coffee Lake - Microarchitectures - Intel . https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake, 12 2020.

[Int21a] Intel. Difference of Cache Memory Between CPUs for Intel® Xeon® E5 Processors and Intel® Xeon® Scalable Processors. https://www.intel.com/content/www/us/en/support/articles/000027820/processors/intel-xeon-processors.html, 07 2021.

[Int21b] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html, 06 2021.

[Int21c] Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html#combined, 04 2021.

[KHC94] Lizyamma Kurian, Paul T. Hulina, and Lee D. Coraor. Memory latency effects in decoupled architectures. *IEEE Transactions on Computers*, 43(10):1129–1139, 1994.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[Kum17] Akhilesh Kumar. Intel's New Mesh Architecture: The 'Superhighway' of the Data Center. *Intel IT Peer Network*, 06 2017.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, page 605–622, USA, 2015. IEEE Computer Society.

[Mar02] Deborah Marr. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6, 2002.

[McC21] John McCalpin. Mapping Addresses to L3/CHA Slices in Intel Processors. Technical report, Texas Advanced Computing Center, 2021.

[Mor21] Bradley Morgan. user rdpmc. https://github.com/BMorgan1296/user_rdpmc, May 2021.

[MSN+15] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, page 48–65, Berlin, Heidelberg, 2015. Springer-Verlag.

[Per05] Colin Percival. Cache Missing for Fun and Profit. In *BSDCan 2005*, pages 1–3, 2005.

[PLF21] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical, 2021.

[Smi82] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[TJYD10] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High*

Performance Computing 2009, pages 157–173, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23(1):37–71, January 2010.

[Var09] Various. Perf. https://github.com/torvalds/linux/tree/master/tools/perf, July 2009.

[VG20] Pepe Vila and Enes Goktas. evsets. https://github.com/cgvwzq/evsets, 05 2020.

[vSM19] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *S&P*, May 2019.

[Yar16] Yuval Yarom. Mastik: A Micro-Architectural Side-Channel Toolkit. https://cs.adelaide.edu.au/~yval/Mastik/, 2016.

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[YGL+15] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel Last-Level Cache. Cryptology ePrint Archive, Report 2015/905, 2015. https://ia.cr/2015/905.