Rapid Reversing of Non-Linear CPU Cache Slice Functions: Unlocking Physical Address Leakage

Mikka Rainer, Lorenz Hetterich, Fabian Thomas, Tristan Hornetz,
Leon Trampert, Lukas Gerlach and Michael Schwarz
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

***Abstract*—Microarchitectural attacks are a growing threat to modern computing systems. CPU caches are an essential but complex element in many microarchitectural attacks, making it crucial to understand the inner workings. Despite progress in reverse-engineering techniques, non-linear cache-slice functions remain challenging to analyze, especially in recent Intel hybrid microarchitectures.**

In this paper, we introduce a novel approach towards reverse-engineering complex, non-linear cache-slice functions, particularly on modern Intel CPUs with hybrid microarchitectures. Our method significantly advances prior work by understanding the specific structure of microarchitectural hash functions, reducing the time required for reverse-engineering from days to minutes. In contrast to prior work, our technique successfully handles systems with 512 GB of memory and diverse slice configurations. We present 13 newly identified functions used for cache-slice addressing and extend existing functions to support systems with more DRAM for multiple CPU generations. Additionally, we introduce an unprivileged virtual-to-physical address oracle that is a direct consequence of the complexity of the non-linear slice functions. Our method is particularly effective on modern Intel hybrid CPUs, including Alder Lake and Meteor Lake, where previously used methods for measuring slices or leaking physical addresses are unavailable. In 3 case studies, we validate our approach, demonstrating its effectiveness in executing targeted Spectre attacks on non-attacker-mapped memory, enabling DRAMA attacks, and creating cache eviction sets. Our findings emphasize the increased attack surface introduced by complex cache-slice functions in modern CPUs.

1. Introduction

Microarchitectural attacks pose a significant threat to modern computing systems, which are constantly evolving in complexity and scale. These attacks exploit subtle nuances in the implementation details of CPUs, the so-called microarchitecture. As the complexity of modern CPUs grows, so does the surface area available for exploitation, leading to numerous microarchitectural attacks. Microarchitectural attacks have been shown on various parts of computing systems, such as side-channel attacks on caches [10], data leakage from internal CPU buffers [4], or fault attacks on the DRAM [47]. A frequent element in such attacks is the CPU cache: either directly as a covert or side channel [40] or indirectly by requiring access to the DRAM, circumventing the cache [31], [50]. Consequently, reverse-engineering the cache is often a critical preliminary step to understanding the implementation details and mounting effective attacks. Fortunately, as the number of microarchitectures is limited, such reverse-engineering is often a one-time effort, and the results can be applied to multiple CPUs. Previous work reverse-engineered various details of caches, such as addressing functions [11], [24], [25], [27], [40], [43], [44], [58], [68], internal structures [69], buses [5], [49], replacement policies [1], [17], [51], or predictors [38].

In this paper, we make two key observations that challenge common assumptions of previous works. First, while many aspects of caches have been reverse-engineered, *non-linear cache-slicing functions* remain challenging to reverse-engineer despite recent advancements [11]. These non-linear slice functions, common in modern Intel CPUs with hybrid microarchitectures, introduce additional complexity into the reverse-engineering process. Second, microarchitectural attacks often assume that physical address information is readily available. Physical addresses are often necessary for attacks targeting structures such as the DRAM or CPU caches, where precise indexing depends on the physical address used as input to various microarchitectural hash functions [11]. Although these two observations appear distinct, we demonstrate that a deeper understanding of cache-slice functions allows constructing an oracle for translating virtual addresses to physical addresses. We demonstrate how attackers, even without privileged code execution, can construct such an oracle using timing side channels.

Our primary contribution is a generic approach for the efficient reverse-engineering of *full* non-linear cache-slice functions for a wide range of Intel CPUs, including the new hybrid microarchitectures. We extend prior work [11] that used Gröbner bases to reverse-engineer cache slices by incorporating the common structures we identify in Intel cache-slice functions. This advancement eliminates previous limitations, such as being able to reverse-engineer only a limited number of bits. It drastically reduces the required reverse-engineering time from several days to mere minutes. Our approach is the only one that can reverse-engineer non-linear cache-slice functions for modern systems with more

than $4\,\mathrm{GB}$ of memory across various microarchitectures. We demonstrate this by reversing the non-linear cache-slice function for a system with $512\,\mathrm{GB}$ DRAM which is not feasible with previous approaches. Due to our efficient approach, we can provide reverse-engineering results for all major microarchitectures from the 5^{th} to the 13^{th} generation, including various slice counts from 2 to 24. We present 13 previously unknown functions used as components in cache-slice functions and extend known functions to be usable on systems with more DRAM.

Building on our reverse engineering of the *full* non-linear function, we introduce a novel method for translating virtual addresses into physical addresses. Physical address information is beneficial for several microarchitectural attacks, such as DRAMA [50], Rowhammer [35], [57], [67], Foreshadow [23], [56], [61], eviction-based cache attacks [13], [37], [65], [71], or ZombieLoad [54], and also kernel exploitation [29]. We extend previous side channels that determine the cache slice using timing measurements [11], [18], [68] to also work on hybrid microarchitectures. On these systems, a simple minimum is not sufficient, as cores have different frequencies, requiring dynamic thresholding and separation of Gaussian distributions. By measuring cache slice behavior within a target memory region and leveraging the full cache-slice function, we infer the corresponding physical address, given that slice functions operate directly on physical addresses. We adapt the Knuth-Morris-Pratt [33] algorithm to efficiently recover this information in linear time without storing the entire slice-function output in memory. Our analysis shows that we require less than $0.8\,\%$ of the available DRAM as a contiguous memory block to determine the physical address reliably. We demonstrate that this is a realistic assumption on real-world Linux systems.

Furthermore, we demonstrate that our technique is effective even on new Intel hybrid CPUs, such as those in the Alder Lake, Raptor Lake, and Meteor Lake series, which incorporate multiple different types of CPU cores. There is no direct mapping of slices to cores on these hybrid CPUs anymore, as cores might share cache slices [11]. On Meteor Lake and newer architectures, the previously-used performance counters—commonly used in reverse-engineering cache addressing functions [43]—are unavailable, rendering previous methods ineffective. We introduce a new method for automatically finding cache-based performance counters correlating with slice accesses, re-enabling measurements on these new architectures. Furthermore, this also generalizes to older architectures, making the measurement approach more generic than previous work [11], [43], [44].

We demonstrate the 3 case studies to validate our results. We confirm the accuracy of our reverse-engineered functions, and the *v2p*-oracle, by generating perfect cache eviction sets for an AES T-table attack, implementing a DRAMA covert channel, and executing a targeted Spectre attack by letting the victim reuse the memory for which we obtain the physical address. They demonstrate the practicality of our approach and emphasize the additional attack surface introduced by non-linear cache-slice functions.

Contributions. We summarize our contributions as follows.

- We introduce a generic approach to efficiently reverse-engineer complete non-linear cache-slice functions, extending prior work and reducing the time required from days to minutes.
- We show non-linear cache-slice functions for systems with up to $512\,\mathrm{GB}$ of memory and for hybrid CPU microarchitectures, spanning cache configurations from 2 to 24 slices across multiple microarchitectures, presenting the largest collection of full cache-slice functions.
- We introduce a novel method for translating virtual addresses to physical addresses using an unprivileged timing side channel to determine the active cache slice.
- We demonstrate the effectiveness of our technique on new Intel hybrid CPUs, where previously used performance counters are unavailable, validating our results with 3 case studies, including cache eviction sets, a DRAMA attack, and a targeted Spectre attack.

Responsible Disclosure. We responsibly disclosed our findings to Intel on November 15, 2024. Intel acknowledged the findings.

Availability. The source code of our reverse-engineering framework, the *v2p*-oracle and our case studies are open-sourced at https://github.com/CISPA/LLCSliceReversing.

Outline. The remainder of this paper is organized as follows. Section 2 provides the background required for the remainder of the paper. Section 3 introduces our novel algorithm to reverse-engineer non-linear cache-slice functions in linear time. Section 4 demonstrates how an unprivileged attacker can combine non-linear cache-slice functions with timing side channels to build a virtual-to-physical address oracle. Section 5 evaluates the *v2p*-oracle for different systems. Section 6 demonstrates that the new reverse-engineered slice functions and the *v2p*-oracle can be used for various attacks relying on physical addresses. Section 7 introduces additional use cases for the *v2p*-oracle and slice functions, discusses related work and mitigations. Section 8 concludes.

2. Background

In this section, we introduce the required background to understand the remainder of this work.

Caches. Caches are essential components of modern CPUs, providing fast access to frequently used data. Typically, caches are organized in a hierarchical structure, with caches being smaller but faster the closer they are to the CPU core. In Intel CPUs, the last-level cache (L3) is the largest cache and shared among all CPU cores. This makes the L3 an attractive target for microarchitectural attacks.

Cache Slices. On modern Intel CPUs, the L3 is divided into multiple slices. Typically, there is one slice per CPU core that is accessed directly, while the other slices are accessed via a bus. However, with the introduction of performance and efficiency cores on recent Intel CPUs, direct access to one slice is not limited to one CPU core but is possible for multiple cores. Dividing the L3 into slices increases its memory bandwidth, as multiple slices can be accessed in parallel. Accesses to the L3 cache are distributed across

slices using a microarchitectural hash function $H : p \mapsto s$ implemented in hardware. The hash function takes a physical address p and maps it to a slice s. It is designed to distribute accesses uniformly to slices with minimal latency.

Cache Eviction. Since caches are limited in size, they must evict data to accommodate new data. This process, called cache eviction, may follow different replacement policies, such as least-recently-used (LRU) [1]. Fine-grained control over cache eviction requires constructing minimal eviction sets, which can be used to evict specific cache sets [17]. Previous work constructed such sets via prior profiling or leveraging knowledge of replacement policies and physical addresses [17], [37], [65].

DRAM. DRAM is organized in channels, DIMMs, ranks, banks, and rows, where each row stores $8\,\mathrm{kB}$ with one capacitor per bit. The memory controller uses a microarchitectural hash function [50] to map from physical addresses to the channel, DIMM, rank, bank, and row. While these hash functions are not disclosed, previous work has shown that they can be successfully reverse-engineered via side channels such as timing, physically probing the address lines, or performance counters [22], [50]. When a DRAM row is accessed, it needs to first be loaded into the so-called row buffer, from which data can be transferred to the CPU. If another row is accessed, the row buffer must be written back to the respective row before the next row can be loaded into the row buffer. Such collisions are called row conflicts and introduce timing differences, which can then be used to leak if two physical addresses fall into the same DRAM bank. These conflicts can be utilized for so-called DRAMA attacks, which include leaking secret-dependent memory accesses from victims and a fast cross-CPU covert channel [50]. For mounting these *row conflict* side channels, virtual-to-physical address translation enables mapping addresses to a specific *bank*.

3. Reverse-Engineering of Address to Slice Mapping in Linear Time

In this section, we describe a novel divide-and-conquer-based method to reverse-engineer the cache-slice function of Intel CPUs. In contrast to prior work on the generic recovery of hash functions [11], our method exploits the structure of the cache-slice function, increasing reverse-engineering efficiency. Consequently, we are not limited in the number of input bits to the function, enabling the recovery of functions on systems utilizing more than $4\,\mathrm{GB}$ of memory. Our method is the first to completely and generically reverse-engineer non-linear cache-slice functions on real-world systems. Additionally, our targeted approach leads to a performance improvement of several orders of magnitude. This speedup makes reverse engineering practical for a large number of systems, as the required time is reduced from multiple days [11] even for small functions (23 input bits) to mere minutes for large functions (33 input bits).

3.1. Limitations of Prior Work

Prior work on reverse-engineering cache-slice functions has primarily focused on recovering linear functions [24], [27], [40], [43], [58], with only a few partial and manual results for non-linear functions [25], [44], [68]. Gerlach et al. [11] were the first to demonstrate a generic reverse-engineering approach for non-linear cache-slice functions. While their work proposes a generic method for recovering arbitrary non-linear hash functions, the presented approach scales exponentially in the number of input bits. For example, for the Intel Core i9-12900K, they only manage to reverse-engineer the slice function for 31 output bits while taking $5\,\mathrm{d}$, making it infeasible to infer significantly more bits of the function. This limits the applicability of their approach to systems with at most $4\,\mathrm{GB}$ of DRAM, less memory than typically present in modern computer systems.

The main limitation of the Gerlach et al. [11] approach is that it assumes next to no structural information about the reverse-engineered hash function, except that it is a non-complex shallow circuit. While this assumption makes the approach highly generic, it is limited to functions with few input bits. Consequently, the approach is not practical on modern CPUs with typically at least $16\,\mathrm{GB}$ of DRAM.

3.2. Structure Analysis

Our goal is to vastly improve the reverse-engineering time for many input bits. While this is not generically possible, we aim to reduce the search space by exploiting the hash function's specific structure. For this, we manually analyze linear hash functions [43] and partially-reversed non-linear hash functions [11], [44], [68].

Based on the observation that parts of linear hash functions have been the same for more than 10 years [11], [43], and also other microarchitectural hash functions often stay the same over multiple generations [11], we make the following assumption:

Assumption 1 Entire or partial functions for cache slices are re-used across different microarchitectures and CPU generations.

Moreover, the function must be fast and energy-efficient, as it is evaluated for every L3 access. Additionally, it should balance the loads equally across all slices, for which many input bits are required. Based on Assumption 1, we assume that several bits are combined using XORs and then used as inputs to non-linear circuits. Previously known partial slice functions confirm this assumption [11], [44].

Assumption 2 Cache-slice functions effectively have 2 stages: a linear combination of bits, and a non-linear "merge" of the linear output.

Finally, CPUs from the same generation and with the same microarchitecture exist with linear and non-linear slice functions. Thus, we assume that CPU manufacturers do not want to have a large set of entirely different function designs.

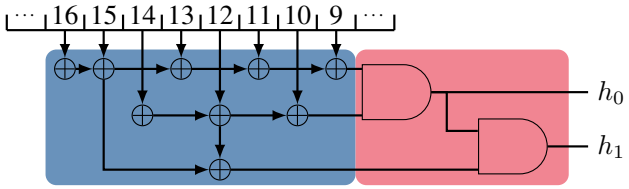


Figure 1: Linear XOR chains of address bits are fed into a non-linear mixer-circuit resulting in non-linear hash-function outputs h_1, h_2.

Assumption 3 Cache-slice functions are mainly defined by the number and types of cores, not by the microarchitecture generation.

Based on these assumptions and the existing partial slice functions, we recognize that all slice functions combine two building blocks: linear *XOR chains*, and non-linear *chain mixers*.

XOR Chain. We define an XOR chain as a function

$$L_{\text{id}}(x) = \bigoplus_{j=0}^{n} (x_j \wedge a_j) \tag{1}$$

Where x denotes the input (i.e., physical address), n the highest used input bit, a_j a binary mask that decides whether the j^{th} input bit is part of the function, and 'id' a unique name for the chain. The output is a single bit.

Chain Mixer. We define a chain mixer $C_n(x)$ as a function that takes at least $\lceil \log_2(n) \rceil$ input bits and uniformly[1] distributes the input number across n output buckets, i.e., the output is a number between 0 and $n-1$. For example, Figure 1 illustrates a (non-real) concept of a C_3 chain mixer. It takes an input between '0' and '7' (in the form of one XOR chain per bit) and maps it to an output number between '0' and '2', i.e., the 2 bits h_0 and h_1.

By viewing the cache slice function as a combination of multiple smaller building blocks, the number of XOR chains and mixers is limited and easier to find. Further analyzing how the mixer is applied leads to another insight into the structure: Chain mixers are typically only used if necessary. For example, in a CPU with 6 cache slices, these cache slices can also be thought of as 2 groups of 3 slices. Thus, one linear XOR chain uniformly distributes addresses among the 2 groups, and the non-linear chain mixer C_3 distributes addresses among the 3 slices. Similarly, for a CPU with 12 cache slices, there are 4 hypothetical groups of 3 slices each. This results in 2 linear XOR chains and a chain mixer C_3 that distributes addresses among the 3 slices within the groups. We can easily infer which chain mixer C_n for a CPU with x slices (with x not a power of two) is required:

$$n = \frac{x}{2^l} \quad \text{where} \quad l = \max\left\{k \in \mathbb{Z}_{\geq 0} \mid \frac{x}{2^k} \in \mathbb{Z} \text{ and } \frac{x}{2^k} \equiv 1 \pmod 2\right\}$$

1. It is not a perfect uniform distribution, but close to one [44]. This is the result of using a fast, shallow circuit.

Observation 1 Chain mixers are required for the remaining odd factor when dividing the number of slices by the highest power of two smaller than the number of slices. Other parts can use linear XOR chains.

Consequently, only a small number of chain mixers are needed, especially as Intel CPUs with an odd number of cores are rare.[2] For example, only chain mixers C_3 and C_5 are required to build a cache-slice function for CPUs with 1, 2, 4, 6 (2×3), 8, 10 (2×5), and 12 (4×3) cores. By looking at all the odd factors of all CPU core counts of available Xeon CPUs [26], it becomes apparent that only 8 different chain mixers are required for them: C_3, C_5, C_7, C_9, C_{11}, C_{13}, C_{15}, and C_{19}. Given that 4 P-cores share one slice of the last-level cache, the Intel Core CPUs since Ivy Bridge [26] have 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 14, 16, and 18 slices, which can be covered by C_3, C_5, C_7, C_9, and C_{11}. Thus, an even lower number of chain mixers is required.

3.3. Divide-and-Conquer: Component Splitting

In addition to the insights about the structure of the function, there is another observation that helps reverse-engineer the function:

Observation 2 XOR gates allow creating a shallow circuit that evenly distributes inputs to a number of buckets that is a power of two. Chained with a nonlinear mixer, the output bit distribution can be skewed in the desired direction, while also imposing activation constraints to the output bits.

For example, looking at the C_5 mixer in Figure 12, we see that if the most significant output bit is one, the lower 2 output bits are fixed to zero. Given that the XOR chains output 1 in 50 % of the cases, this happens in $\frac{3}{16} \approx \frac{1}{5}$ of the cases. If the upper bit is zero, the lower bits each have a 50 % probability of being one. This means that the function hits each of the 5 buckets evenly often, for a uniformly distributed input space. While it would be possible, to use individual input bits as input to the mixer, this would not distribute well in a shallow circuit. Therefore, the chain mixers do not use individual input bits, but only the output of the XOR chains.

We can split the reverse-engineering process into two parts, independently recovering the chain mixer and the XOR chains and combining them into the final cache-slice function. Based on the structure, we even know the type of chain mixer we have to reverse-engineer. Reverse-engineering the linear XOR chain is trivial and can be done within seconds [43]. If solved independently of the XOR chains, the remaining linear chain mixer is a small linear function with a few input bits—in our analysis, the number of input bits never exceeds 10. To solve this small function, we use a combination of approaches designed to synthesize loop-free programs [19], [20]. Such approaches

2. Some exceptions exist, e.g., the Intel Xeon E7-2870 v2 from 2014 with 15 cores.

systematically search for a small function representation given a function's input-to-output mapping (truth table). As the problem is complex, all state-of-the-art approaches scale exponentially; however, our problem instances remain small and, therefore, solvable. These approaches are exact and reconstruct a minimal version of the function while being efficient enough to recover our small mixer functions.

3.4. Efficient Reverse-Engineering Algorithm

Algorithm 1: Reverse-engineering Algorithm for Slice Addressing

Input: Address size n, threshold $0 \leq m \leq n$
Output: Function for all bits up to n
Step 1: Obtain the compact form of the slice addressing function for bits 1 to m;
  Record a pattern of tuples (address, slice) for all addresses with m bits;
  Create a truth table t from the pattern;
  Reduce t with Espresso [3];
  Synthesize function for t;
Step 2: Record a small pattern for each unknown bit $m+1$ to n;
Step 3: Brute-force the upper bits bitwise from $m+1$ to n;
Step 4: Obtain a final function for all bits up to n;

Due to the independence of the linear and non-linear parts, we design a highly efficient reverse-engineering algorithm, as shown in Algorithm 1. To summarize, we measure the cache-slice output for a small memory region. This is sufficient to recover the non-linear chain mixer using counter-example guided synthesis for boolean formulas [21]. For this chain mixer, we know the number of XOR chains that are used as the input. Additionally, we know the first few bits of the XOR chains, similar to Gerlach et al. [11].

However, instead of using a mathematical approach to solve the XOR chain, we brute-force the chains bitwise. We add every physical address bit to the XOR chain and test whether the output of this reconstructed guessed function agrees with the measured slice for randomly sampled addresses. More formally, for every a_j in Equation (1), we test whether setting it to '0' or '1' results in the same output as the measured slice, i.e., the ground truth. As all bits in Equation (1) are independent, the number of tries is n, i.e., linear in the number of used input bits.

3.5. Complexity Analysis

Our new method has a time complexity of $\mathcal{O}(n+c)$ compared to $\mathcal{O}(2^n)$ with the state of the art from Gerlach et al. [11]. The runtime of our method is linear with the number of relevant bits n of the physical address. The time required to obtain a compact form of the chain mixer is independent of n and thus constant $O(c)$, as we obtain the function for a fixed number of bits. Brute-forcing the upper bits of the physical address requires a fixed time per additional bit and is, therefore, linear to the number of bits $O(n)$.

3.6. Implementation

We measure all required patterns with a small C program. We implement our reverse-engineering algorithm in Python and use an SMT-based variant of counter-example guided synthesis [19], [20] using the Z3 solver [6] to solve the non-linear part of the functions. We brute-force the XOR chains in a bit-by-bit fashion to obtain the final function.

Measurement Interface. The prerequisite for reverse-engineering the slice function is to measure the accessed cache slice of a chosen address. Although this is a solved challenge on older microarchitectures, we have to reverse-engineer the interface for new microarchitectures. Previous work relied on performance counters documented by Intel to measure the accesses to every cache slice [11], [43], [44]. The used performance counters are either the L3/CHA read counters [44] for Xeon CPUs or the L3/CBox counters [43] for Core CPUs. Unfortunately, all these counters are undocumented for the newest generation of Intel CPUs, i.e., Meteor Lake (Core) and Sapphire Rapids (Xeon). Moreover, the counter configuration differs from previous generations and is no longer applicable.

We devised an automated approach to avoid relying on recurring reverse-engineering efforts for current and future microarchitectures. The main idea is to find *any* uncore performance counter that *correlates* with the number of slice accesses. We rely on the Linux `intel_uncore_pmu` driver to get the base of the uncore events and iterate over all possible 256 event selectors. For every event selector, we iterate over the masks from 0 to 255. While masks can be larger, previous relevant events did not use larger masks. For each of the 2^{16} event-mask combinations, we measure the number of increments for every uncore counter when accessing the same uncached memory location 2^{12} times. Theoretically, this should result in 2^{12} accesses in one slice and no access in any other slice. However, due to unrelated memory accesses, both of the measuring application (e.g., stack and code) and other applications sharing the L3, we accept up to $100\,\%$ more accesses as expected per slice, and also $100\,\%$ more accesses in total over all slices.

Table 2 shows the automatically inferred events we use for the measurement. As a quick verification, we test whether the event results in the same slice over multiple measurements within an arbitrarily chosen cache line and in different (but consistent) slices for the adjacent cache lines. Interestingly, while most events match the documented events, for some CPUs, the inferred events are undocumented or used for a different event, according to the documentation. On the Intel Core Ultra 7 155h (Meteor Lake), we use the listed but not documented counter `UNC_HAC_CBO_TOR_ALLOCATION.DRD`, which we assume counts uncore (UNC) cache-coherency events via the table of requests (TOR) for demand reads (DRD). While counters are documented for older generations (e.g., Broad-

well), all microarchitectures since the 10^{th} generation use undocumented counters that we automatically inferred.

Measurement. Our measurement program to record the cache-slice pattern for a memory region is written in C. A pattern consists of tuples of physical addresses and the corresponding cache slice for a small memory region, i.e., 1024 cache lines ($64\,\mathrm{kB}$). We record a pattern at the start of the physical memory that is later used to obtain the non-linear mixer function. We further measure the tuples for the start of regions where a new bit of the physical address is used. This data is used to recover the upper parts of the XOR chains. We use PTEditor [53] to map the target physical memory to a virtual address region.

Non-linear Solving. We solve the non-linear chain mixer over a small pattern of 1024 cache lines ($64\,\mathrm{kB}$) at the start of the physical memory. In practice, we observe that 1024 addresses are sufficient to recover the mixer function on all tested CPUs. Keeping the number of addresses small is essential to ensure that the XOR chains remain as small as possible and that we recover only the non-linear mixer. We use counter-example-guided synthesis [19] combined with generic optimizations [20] to recover the mixer function. We use the implementation from Sebastian Hack [21] for our implementation. This implementation uses the Z3 solver [6] to repeatedly query for short programs that satisfy the input-output behavior specified by our function.

To reduce the problem's complexity, we find a function for each output bit separately, starting with the most significant bit. We create a truth table from the initial pattern as an Espresso [3] PLA. This PLA is minimized with Espresso to make the problem more tractable for the boolean-formula synthesis [21]. As a last step, we use the counter-example-guided synthesis [21] to recover the function. The input bits are the lower bits of the physical address and the output bit is the value of the current bit of interest of the slice index. The functions for the lower bits are found by adding the function of the most significant bit to the input bits as an additional input bit, as this is a typical mixer structure (cf. Section E). As we can fix the number of input bits to the mixer function for this step, the time complexity is constant.

Linear-chain Recovery. Once the non-linear mixing function is known, efficiently recovering the linear XOR chains is possible. Based on the following observation, the bits used in the XOR chains can be found within seconds using brute-force search. We know the function up to bit j. We try each combination of appending bit $j+1$ to the XOR chains. For example, for 6 chains, there are $2^6 = 64$ possible extensions of the known function, i.e., bit $j+1$ is either used or unused for each chain. We use the recorded pattern of (physical address, slice) tuples to verify the correctness of the extended XOR chain and only accept the correct extension. This process is repeated for all bits up to the maximum number of physical address bits.

3.7. Results

We evaluate our method on 18 CPUs, as listed in Table 1. The components of the reverse-engineered functions are shown in the last column, where L_{id} denotes the linear XOR chains and C_{id} the non-linear chain mixers. These boolean functions are presented in Section E. From the evaluated CPUs, 8 CPUs have a linear function, i.e., the number of slices is a power of two, and 10 CPUs have a non-linear function, i.e., the number of slices is not a power of two. Our method successfully reverse-engineers the non-linear slice functions for DRAM higher than $4\,\mathrm{GB}$.

Mixers are regularly reused across microarchitecture generations. For example, the C_3 (cf. Figure 11) mixer is used on the Intel Xeon E-2176M (Coffee Lake), the Intel Core i7-10710U (Comet Lake), and the Intel Core i9-13900K (Raptor Lake). The only change from Intel Core i7-10710U to Intel Core i9-13900K is the addition of a new XOR chain L_{9b} to cover the increased number of cores. We discover 2 new mixers, C_2 and C_9, that were not observed by any previous work [11], [43], [44]. Our experiments further indicate that the same non-linear slice function is reused for all CPUs with the same number of slices. For example, the function composed of $L_{6b}|C_3$ is used in all 4 CPUs with 6 slices, independent of the number of cores. The same applies for the function $L_{6b}|L_{9b}|C_3$, which is used in both CPUs with 12 slices.

XOR chains are also heavily reused across microarchitecture generations, e.g., XOR chain L_{6c} (cf. Figure 10) is used as a mixer input in all non-linear cache-slice functions. In our reverse-engineered functions, we report 23 different XOR chains, of which 11 were previously unknown, and 12 were already observed in previous work [11], [43], [44]. Still, we extended 9 known ones to support more DRAM.

Observation 3 Even though the cache architecture changed significantly on hybrid microarchitectures (non-inclusive last-level cache and shared slices), the slice functions did not change.

Notably, we found that on the Intel Xeon E5-2697 v4 (Broadwell) there are two linear XOR chains that are combined with a C_2 mixer. Even though this does not change the output distribution, it might be used to balance the load across the slices more evenly. With the introduction of the hybrid microarchitectures, we see interesting combinations of the number of slices and cores, which do not have a direct relationship anymore. Note that the Intel Core i9-12900H (Alder Lake) has 14 cores and 8 slices, and therefore has a linear function, while the Intel Core i9-12900K (Alder Lake) has 16 cores and 10 slices, and therefore has a non-linear function. This can be explained by the fact that typically, 4 E-cores share one slice of the last-level cache, while 1 P-core has its own slice. We observe outliers in the Intel Xeon Gold 6346 (Ice Lake) with 24 slices and 16 cores of the same type and the Intel Core i5-13420H (Raptor Lake) with 4 E-cores and 4 P-cores. Here, the number of slices does not directly relate to the number of cores.

Observation 4 The common assumption that the slice function is linear if the number of cores is a power of two and vice versa does not hold anymore with the hybrid microarchitectures.

TABLE 1: Our tested systems with non-linear and linear cache-slice functions, including our reverse-engineered slice functions. Highlighted (yellow) cells indicate non-power-of-two values, illustrating that all combinations of power-of-two and non-power-of-two values for cores and slices exist on modern Intel CPUs.

	CPU	μarch	Cores	Slices	Memory	Components
Non-linear	Intel Xeon E-2176M	Coffee Lake	6	6	4 GB	$L_{6b}\|C_3$
	Intel Core i7-8700	Coffee Lake	6	6	16 GB	$L_{6b}\|C_3$
	Intel Core i7-10710U	Comet Lake	6	6	16 GB	$L_{6b}\|C_3$
	Intel Core i5-13420H	Raptor Lake	8	6	8 GB	$L_{6b}\|C_3$
	Intel Core i9-12900K	Alder Lake	16	10	96 GB	$L_{6b}\|C_5$
	Intel Core i9-13900K	Raptor Lake	24	12	16 GB	$L_{6b}\|L_{9b}\|C_3$
	Intel Core Ultra 9 285K	Arrow Lake	24	12	64 GB	$L_{6b}\|L_{9b}\|C_3$
	Intel Xeon E5-2697 v4	Broadwell	18	18	512 GB	$C_2\|C_9$
	Intel Xeon Gold 6526Y	Emerald Rapids	16	20	128 GB	$L_{6b}\|L_{6f}\|C_5$
	Intel Xeon Gold 6346	Ice Lake	16	24	64 GB	$L_{6b}\|L_{6f}\|L_{8d}\|C_3$
Linear	Intel Xeon E5-2683 v4	Broadwell	16	16	32 GB	$L_{6a}\|L_{7a}\|L_{8a}\|L_{9a}$
	Intel Core i7-9700K	Coffee Lake	8	8	64 GB	$L_{6a}\|L_{9a}\|L_{10a}$
	Intel Core i7-10510U	Comet Lake	4	4	16 GB	$L_{6a}\|L_{7a}$
	Intel Core i3-1005G1	Ice Lake	2	2	16 GB	L_{6a}
	Intel Core i7-1185G7	Tiger Lake	4	4	32 GB	$L_{6a}\|L_{9a}$
	Intel Core i7-11700	Rocket Lake	8	8	16 GB	$L_{6a}\|L_{9a}\|L_{10a}$
	Intel Core i9-12900H	Alder Lake	14	8	32 GB	$L_{6a}\|L_{9a}\|L_{10a}$
	Intel Core Ultra 7 155H	Meteor Lake	16	8	32 GB	$L_{6a}\|L_{9a}\|L_{10a}$

We benchmark our approach against previous work from Gerlach et al. [11]. For the Intel Core i9-12900K, we improve the measurement duration for reverse engineering from 5 d to 8.19 s. For the Intel Xeon E-2176M, the improvement is from 23 h to 23.52 s. Moreover, we recover the function up to 96 GB, whereas Gerlach et al. [11] only recover it up to 4 GB, which is unrealistic for real-world attacks. Our approach significantly outperforms the state of the art, especially when required to reverse-engineer the entire function and not a limited set of input bits. This is because the approach of Gerlach et al. [11] scales exponentially with the number of input bits, while ours scales linearly. The speedup is in the range of 3500 to 53 000 for systems with up to 4 GB of DRAM—we cannot compare for more DRAM, as previous approaches are infeasible.

3.8. Slice Function Correctness

To evaluate the correctness of the reverse-engineered slice function, we rely on spot-checking [9]. As Gerlach et al. [11] show, measuring all slices for a system with 4 GB of DRAM takes 5 days. For our systems with up to 512 GB installed DRAM, this approach is infeasible with an estimated measurement time of 1.75 *years* per system. Thus, we only check for a small random subset of addresses whether the slice index measured with performance counters and the output of the reverse-engineered slice function agree.

We rely on the sample size formula $n = \frac{Z^2 \times p \times (1-p)}{E^2}$, where we choose $Z = 3$ for 99.9 % confidence, $p = 0.5$ as we do not know the error rate, and $E = 0.01$ for 1 % error margin. Thus, we have to sample 22 500 random addresses for our test. If all of these addresses are correct, we have a 99.9 % confidence level with 1 % error margin that the reverse-engineered slice function is correct. All our reverse-engineered functions passed this test. Additionally, Section 6 shows the correctness in practical use cases. By demonstrating that we can mount real-world attacks using the reverse-engineered slice functions, we show that even if errors were left, they are negligible for attackers.

4. Virtual-to-Physical Address Oracle

In this section, we demonstrate how we can exploit the complexity of non-linear cache-slice functions for building an unprivileged virtual-to-physical address translation oracle. For the sake of readability, we refer to this oracle as *v2p*-oracle. In the following, we denote a sequence generated by applying the slice function to all cache lines in a memory region as a *slice index sequence*. The *v2p*-oracle builds on the observation that, in contrast to linear slice functions, non-linear slice functions lead to slice index sequences in which sub-patterns rarely repeat. Sufficiently long sub-patterns in this sequence only occur once in the physical address space, starting at a unique base input. Given a slice pattern for a sufficiently long physically contiguous memory range, we can uniquely infer this input, i.e., the range's base physical address. Figure 2 illustrates this concept for a toy non-linear slice function. In this example, every slice index sequence of length 3 is unique for the memory range. Thus, by knowing the sequence and the slice function, it is possible

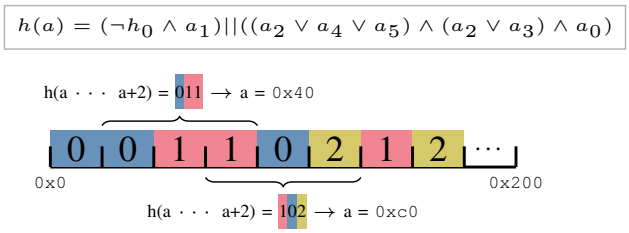


Figure 2: Example illustrating the physical-address recovery. When measuring a sufficiently large slice index sequence, this sequence does not repeat in the physical address space. Finding the slice index sequence hence results in the input to the slice function, i.e., the physical address.

to infer the input to the slice function. While the concept is intuitive, we discuss 4 main challenges when practically implementing it as an unprivileged attacker.

4.1. Threat Model

We assume an unprivileged attacker with native code execution on the system. The attacker runs as a single unprivileged process on the system. The goal of the attacker is to retrieve the physical address of an (accessible) virtual address. The CPU requires a non-linear slice function, which is nowadays a common property of Intel CPUs. 55.9 %, i.e., the majority of CPUs since the introduction of the hybrid microarchitecture (12^{th} generation) have a non-linear slice function. This ratio is similar when looking at large-scale deployments in the cloud. For example, on AWS, we analyzed the slice numbers for all bare-metal Intel instances. 17 of the 34 (50 %) available bare-metal instances fulfill our requirement. Even considering all 1704 Intel Core and Xeon CPUs since the introduction of cache slices, 35.3 % have a non-linear cache-slice function. We assume that there is no direct architectural access to physical addresses (e.g., via the now restricted `/proc/self/pagemap` interface). We expect that there is no CPU vulnerability that directly extracts page-table entries and thus leaks physical addresses [39], [61], [63]. We do not rely on software vulnerabilities or Spectre gadgets in the kernel or other processes.

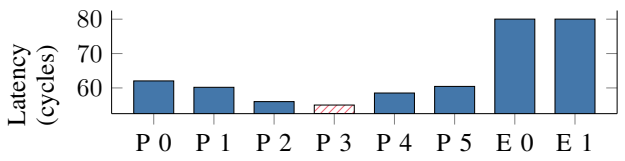
4.2. Challenges

To use the *v2p*-oracle in a realistic setting with an unprivileged attacker, we must solve the following challenges.
$\mathcal{C}_1$: Unprivileged Slice Measurement. We require a technique to measure the slice number of an address that works in unprivileged code. The techniques used for measuring the slice number during reverse-engineering require access to performance counters of shared microarchitectural elements. For security reasons, these performance counters are only available to privileged users on modern Linux distributions. Thus, while they are extremely useful for reverse-engineering the slice function (a one-time effort), they cannot be used for the *v2p*-oracle within our threat model. Section 4.3 describes how we rely on a timing side channel in combination with unprivileged Linux APIs to reliably retrieve this information as unprivileged user.
$\mathcal{C}_2$: Contiguous Physical Memory. Inferring the physical address from the slice index sequence using our approach requires the underlying memory to be physically contiguous. However, an unprivileged user does not have any control over the allocation of physical memory. This is done transparently by the memory allocator in the kernel. Section 4.4 describes how we can reliably get physically contiguous memory by exhausting the fragmented memory blocks and by increasing the chances of getting 2 MB pages.
$\mathcal{C}_3$: Slice Function Identification. We have to know the slice function used by the CPU to infer the underlying physical address from the slice measurements. However, an attacker cannot reverse-engineer this function, as access to slice measurements (which we solve by solving $\mathcal{C}_1$) and knowledge of physical addresses are required. This leads to the circular dependency where physical address knowledge is required for inferrence. Hence, Section 4.5 introduces an unprivileged side-channel-based approach for identifying which (known) slice function is used by the CPU.
$\mathcal{C}_4$: Physical Address Reconstruction. The final challenge is to infer the correct physical address from the measurements (cf. $\mathcal{C}_1$) and the identified slice function (cf. $\mathcal{C}_3$). As the slice function is a lossy compression function, it is not possible to fully recover the input from the output. Thus, Section 4.6 introduces an efficient algorithm to reconstruct the input (i.e., the physical address) to the slice function from multiple measurements.

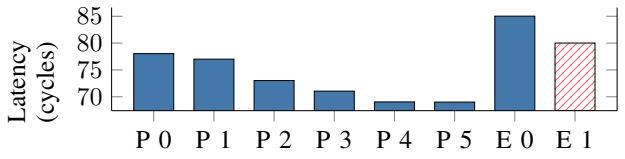
4.3. $\mathcal{C}_1$: Mapping Virtual Addresses to Slices

As an unprivileged user cannot access the required performance counters, we instead rely on a timing side channel. While a timing side channel has already been used in prior work for measuring the timing differences induced by cache slices [11], [18], it does not provide the slice number on hybrid microarchitectures. The reason is that the access latencies vastly differ between performance (P) and energy-efficient (E) cores. Thus, the approach by Gerlach et al. [11] relying on the minimum access time only sometimes results in the correct slice. Figure 3 illustrates the problem on an Alder Lake i9-12900H with 8 slices. If the slice belongs to a P core, the minimum access (or flush) latency corresponds to the slice from which the measurement originates. However, if the slice belongs to an E core, the "fast" access time to the local slice is still slower than the "slow" access time to the remote slice from a P core.

Instead of looking for the minimum, we measure a per-core threshold to determine if a memory access is to the local slice or a remote slice. For this, we require at least one address that maps to the local slice of a core. As we do not have any ground truth, i.e., we do not know any address that fulfills that property, we have to rely on timing measurements again. We measure the access time to a large number of random addresses. Since the number of slices



(a) L3 access times to an address from different cores if the slice belongs to a performance (P) core, in this example, to P3. The further away the other performance core, the higher the timing. The ring-bus architecture leads to a nearly symmetric distribution of timings. Efficiency (E) cores are slower and thus show much higher timings.



(b) L3 access times to an address from different cores if the slice belongs to an efficiency (E) core, in this example, to E1. Performance (P) cores are significantly faster, making remote slice access from P cores even faster than local slice accesses from E cores.

Figure 3: Comparison of L3 access times from performance and efficiency cores.

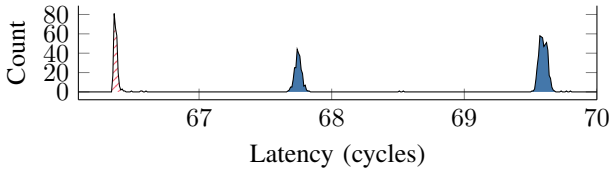


Figure 4: The Gaussian distributions of access times of one core to addresses on different slices. The local slice is always the fastest (marked in red).

is comparably small, several of these addresses are in the local slice and thus have the lowest latency. However, as timings always have some measurement error, we cannot simply choose the minimum.

As the measurements result in multiple normal distributions, depending on how long it takes to access the slice, our measurements are a Gaussian mixture model [8]. Such a measurement is illustrated in Figure 4. Note that the figure is recorded on a machine where the distributions are easily distinguishable—on other machines, they are far more interleaved. Thus, we can use the EM algorithms [45] to infer the parameters of each Gaussian distribution. The Gaussian distribution with the smallest mean is then used as the threshold.

Solution 1 Per-core thresholds extracted from Gaussian mixture models are required for unprivileged slice measurements on hybrid CPU microarchitectures.

4.4. C_2: Recording a Pattern

For recording the slices, we use a contiguous virtual memory region allocated using `mmap`. As shown in previous work [28], [47], [62], choosing a sufficiently-large memory block (e.g., 4 GB) leads to a contiguous physical memory range after the kernel runs out of fragments in the allocator. We achieve the best result by using the `mmap` flag `MAP_LOCK`, which tries to keep the physical pages in memory, in combination with a read- and writable mapping. A side effect of this flag is that, similar to `MAP_POPULATE`, the physical memory is allocated nearly atomically, increasing the chance of getting contiguous memory that stays in memory and is not swapped. By allocating multiple gigabytes and only using the last part, we reliably have physically-contiguous memory (cf. Section 5.2). While simple memory allocations were sufficient in our experiments, more advanced techniques to obtain physically contiguous memory have been outlined in recent research [36], [59].

Transparent Huge Pages. Alternatively, we can use different tricks for physically contiguous memory in specific scenarios. On several Linux distributions, transparent huge pages are enabled by default [7]. We have verified this for several distributions, as shown in Table 3 in the appendix. On such systems, we can first exhaust small pages by using a dummy allocation of several megabytes up to gigabytes. Afterward, we `mmap` a private anonymous memory range with a size that is a multiple of 2 MB. Immediately following the allocation, we initialize the memory range, call `madvise` with `MADV_HUGEPAGE`, and sleep for 2 s. The sleep is essential to give the operating system time to convert the underlying physical pages to transparent huge pages. The resulting huge pages have a significant chance of being physically contiguous—even on our system with only 16 GB of DRAM, we get on average 62.4 ($\pm$ 6.9) contiguous huge pages with this trick when allocating a 512 MB range.

Page Cache. In specific scenarios, especially with Rowhammer [15], it can be beneficial to know the physical address of a page residing in the page cache. If this is either desired or not a limitation, we demonstrate another possibility that we use to get physical contiguous memory reliably. Instead of using an anonymous memory mapping using `mmap`, we map a dummy file, e.g., containing zeros. This results in the pages being taken from the page cache [14], which might feature lower fragmentation.

Solution 2 Reliably getting Physically-contiguous memory is achievable with `mmap` and `madvise` flags.

Noise Reduction. Independently of how we allocate the contiguous physical memory, we require essentially noise-free measurements. Since we rely on a timing side channel, we inherently have measurement inaccuracies. To increase the likelihood of detecting errors, we measure the slice of different offsets within the same cache line. As cache lines are never distributed over slices, all offsets must result in the same measured slice. We repeat this process until the measurements agree. Our measurements result in a slice index sequence over the physically contiguous memory.

4.5. $\mathcal{C}_3$: Determining the Slice Function

To identify the employed slice function as an unprivileged attacker, we use a side-channel-based approach to detect which known slice function parts are used by the CPU. As an unprivileged user can read the CPU brand, it is theoretically possible to build a database containing all CPUs since the introduction of cache slices and their slice function. However, with more than 1700 CPUs implementing cache slices [70], this is infeasible. Such an approach could only be used if the target set of CPUs is small enough, e.g., for all CPUs in the AWS cloud.

We rely on a generic method that only requires knowledge of the linear parts L_x of the cache-slice functions. As discussed in Section 3.7, these functions are heavily reused across CPUs and microarchitectures, resulting in only a few distinct functions (cf. Figure 10). The main idea is to measure whether two addresses that only differ in bit i where $i \in \{0, ..., \log_2(\text{pagesize}) - 1\}$ have different L3 access latencies. These bits of the virtual address are also the same in the physical address. Thus, by flipping these bits, we directly influence the physical address, and, with that, the slice. We observe that it is in many cases sufficient to only detect which bits of a $4\,\text{kB}$ page, i.e., the 12 least-significant address bits, are used in a function to detect which linear chain is used. For $2\,\text{MB}$ pages, this is possible for all linear chains, since none of them use the same bits from 0 to 20.

Solution 3 The linear chains of the slice function can be inferred by measuring the influence of the address bits in the page offset on the L3 latency.

For CPUs with more than 2 slices, flipping a bit used in a slice function results in a different slice depending on the slice function that uses this bit. However, as slice timings can be distinguished (cf. Section 4.3), we can also infer to which slice function a bit belongs with the timing difference. We use the access-time difference between the address v and $v \oplus 2^i$ to group the bits using 1-dimensional k-means clustering, with k = number of slices.

4.6. $\mathcal{C}_4$: Mapping a Pattern to the Physical Address

The remaining challenge is to infer the physical address based on a slice index sequence for an arbitrary contiguous memory range and knowledge of the slice function. Depending on the resources available to the attacker, we choose one of two different approaches. If an attacker can use a large amount of memory or outsource the computation to a different machine with a large amount of memory, we employ pre-computed suffix and longest common prefix (LCP) arrays to recover the physical address with a time complexity of $\mathcal{O}(|P| + log(n))$, where P is the recorded pattern and n is the amount of DRAM in the target system. Such an implementation also allows fuzzy matching if we can accept higher runtimes. Otherwise, we use an adapted version of the Knuth-Morris-Pratt (KMP) pattern-matching algorithm with a sliding window over the slice function, with a time complexity of $\mathcal{O}(n)$ but without the requirement to store the output of the entire slice function.

Suffix and LCP Arrays. A string's suffix array is a sorted array of its suffixes [42]. The main benefit of this data structure is that it allows for locating all occurrences of a substring P (needle) in a string S (haystack) with only binary searches. Combined with the LCP array, which stores the lengths of the longest common prefixes for consecutive entries in a suffix array, this achieves a search time of only $\mathcal{O}(|P| + log(|S|))$. Furthermore, constructing the suffix and LCP arrays only requires $\mathcal{O}(|S|)$ time [48], making this approach ideal for large strings.

To utilize these properties, we treat the slice index sequence as the haystack, with its length depending on the amount of DRAM in the target system. As the slice function is constant, we only need to compute the suffix array once for every CPU and DRAM configuration. Given a slice pattern, we can then efficiently locate its occurrences, retrieving its physical address if it is unique or a list of candidate addresses otherwise. Note that the LCP array's maximum corresponds to the longest repeating substring, giving us an upper bound for the contiguous memory we need to uniquely recover the physical address. If we permit errors, suffix arrays can also improve the performance of fuzzy matching [34]. Due to the memory requirements, we mainly use this approach for the evaluation of slice functions and not for the end-to-end attacks.

Adapted KMP. For attacks, we adapt the KMP algorithm to use a sliding window for the haystack to avoid storing all possible outputs of the slice function in memory. The KMP algorithm is another efficient ($\mathcal{O}(n)$) algorithm for finding the position of a substring (needle) within a long string (haystack). The original KMP algorithm expects that both the haystack and needle are in memory, which is unrealistic on the target system as the haystack requires $16\,\text{kB}$ per $1\,\text{MB}$ of DRAM. Thus, on larger systems, the haystack would require multiple gigabytes of memory.

In contrast, our sliding window adaption only needs to keep the prefix array in memory, which only depends on the length of the needle, i.e., our measurements. All haystack values are computed on the fly within the sliding window with the additional optimization that every haystack value is only computed once. Additionally, we change the KMP algorithm such that it does not only return the first match but optionally all matches of the needle within the haystack.

Solution 4 An adapted window-based KMP algorithm can recover the physical address from a slice index sequence with linear runtime and negligible memory overhead.

5. Evaluation

In this section, we evaluate the *v2p*-oracle. The effectiveness of the *v2p*-oracle depends on 3 key points: The minimum length of a slice index sequence that does not repeat on the machine (Section 5.1), the average size of a contiguous physical memory block allocated by the operating system (Section 5.2), and the reliablity of the slice

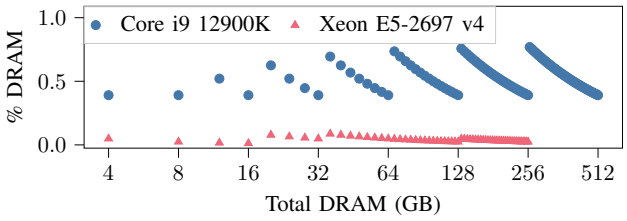


Figure 5: Length of the longest repeating sub-pattern relative to DRAM size.

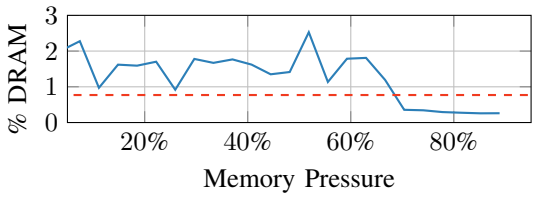


Figure 6: Average amount of contiguous memory for allocations under varying memory pressure (blue) and maximum amount of required physically contiguous memory (dashed).

function measurement (Section 5.3). While all three points are interdependent, i.e., a larger allocation size results in a longer slice index sequence which is easier to recognize if measured precisely, we also focus on all points individually. This is useful, as the allocator changes with different operating systems and versions, while the slice function can change with new CPU generations. Similarly, improvements in slice-function measurements or fuzzy matching for the slice index sequence can be implemented independently. We evaluate the *v2p*-oracle on 2 systems and are able to recover the physical address of a virtual address whithin 22 min on an Intel Core i7-10700 and within 30 min on an Intel Xeon E-2176M, both with 16GB of RAM.

5.1. Slice Index Sequence Evaluation

We determine the smallest amount of physically contiguous memory for the recovered physical address to be unique as well below 1 % of the system's DRAM. This bound is given by the longest repeating sub-pattern in the function's output sequence, which we can efficiently locate with LCP arrays (cf. Section 4.6). We evaluate the output sequences of all DRAM configurations that are a multiple of 4 GB in the ranges for which we reverse-engineered the function. Figure 5 shows our results for the Core i9-12900K and Xeon E5-2697 (cf. Section A for the remaining functions). First, we observe that the size of the longest repeating sub-pattern is always well below 1 % of the system's DRAM. This holds for all non-linear slice addressing functions we analyze, including those not shown in Figure 5. With the Core i9-12900K, we require at most 0.77 % of the installed DRAM as contiguous memory to recover the physical address. For the Xeon E5-2697 v4, which has more cache slices, we only require 0.08 %. Second, the longest repeating sub-pattern often occurs near the highest power of 2 in the evaluation range. Hence, this pattern's relative size is typically lowest when the DRAM size is a power of 2, as the next-largest repetition is just out of range. To illustrate this, we need at most 2052 kB for the Xeon E5-2697 v4 with 16 GB of DRAM, which amounts to roughly 0.01 %. With 20 GB, we need approximately 16 MB, or roughly 0.08 %. In contrast, the contiguous memory required with linear slice addressing functions is significantly larger. With the Xeon E5-2683 v4, we require up to 50 % of the system's DRAM as contiguous memory. Nevertheless, the *v2p*-oracle may still be practical for some configurations with linear functions. If the Xeon E5-2683 v4 runs with 8 GB of DRAM, we only need 3.1 % (256 MB), for instance.

5.2. Memory Allocator Evaluation

We experimentally determine how much physically contiguous memory an unprivileged user can typically get from the operating system with a simple allocation under varying memory pressure. For this, we recreate the *Memory Utilization and Contiguity* experiment from Islam et al. [28]. However, instead of recording the fraction of allocations with a physically contiguous part of at least 520 kB, we measure the average size of the largest physically contiguous part of each allocation over 25 executions.

The result is shown in Figure 6. For memory pressures up to 66.7 % (77.8 % including the measured 1 GB allocation), the average amount of physically contiguous memory obtained by allocations exceeds even the maximum requirement of 0.77 % of physical memory. Thus, a simple call to `mmap` usually suffices to obtain enough physically continguous memory.

5.3. *v2p*-oracle Evaluation

The *unprivileged* measurement to determine the cache slice of one cache line takes, on average, around 5.6 million CPU cycles. This is below 1 ms on all tested machines. The time already includes repeating the measurement 30 times to make it essentially noise-free on all our tested machines. We evaluate our approach on the Intel Core i7-10710U by measuring the slice for 1 000 000 cache lines. We achieve a success rate above 99.99 %. To rule out the remaining errors, we record the pattern multiple times and use majority voting to create a noise-free pattern. Additionally, the dynamic threshold calibration for hybrid systems is a one-time overhead of 13 s. However, as we must determine the slice for multiple thousand cache lines (cf. Section 5.1), this one-time overhead is negligible for the entire runtime. The *v2p*-oracle is evaluated on the Intel Core i7-10710U and Intel Xeon E5-2176M, both with 16 GB of DRAM. A complete translation on the Intel Core i7-10710U takes 22 min with a pattern length of 8 MB and on the Intel Xeon E5-2176M 30 min with a pattern length of 25 MB.

6. Case Studies

In the following, we present 3 case studies showing how access to physical addresses can improve attacks. We show how a victim can be coerced to reuse previously attacker-controlled memory for which the attacker inferred the physical address using the *v2p*-oracle. Based on this, we demonstrate a case study with a Spectre gadget leaking memory via the kernel physmap, i.e., the kernel mapping of the entire physical memory. Additionally, we show that an attacker can mount DRAMA attacks using the *v2p*-oracle. To demonstrate the correctness of our non-linear slice function, we efficiently generate eviction sets for attacks such as Evict+Reload and Prime+Probe on a server system with $512\,\mathrm{GB}$ memory.

6.1. Spectre Attack

In this case study, we use the *v2p*-oracle for a victim page by "massaging" the memory allocator. We demonstrate that it is possible to infer a page's physical address, unmap it, and trick a victim into reusing this page with known physical address. Combined with a Spectre gadget targeting the physmap (allowing to leak memory from arbitrary *known* addresses), this leaks victim memory.

Setup. In line with other work [41], [55], we introduce a Spectre gadget for demonstration. This gadget is a bit-wise Spectre-V1 gadget performing an out-of-bounds access to a pointer in the physmap like those obtained using *kmalloc*. We assume the attacker knows this pointer and can trigger the Spectre gadget via *ioctl*. The attacker can coarsely synchronize with the victim to run before the victim. In our proof of concept, the victim maps a single page.

Attack. The attacker first allocates a large buffer, resulting in physically contiguous memory (cf. Section 4.4) for which they infer the physical address. After inferring the physical address, the attacker frees the memory range, causing the victim to reallocate one of the previously freed physical pages. The victim then stores their secret on the allocated page. Consequently, the attacker knows the physical address of the victim page and uses the Spectre gadget to leak the targeted secret via the physmap.

Reallocation of Attacker Data. We analyze the reallocation patterns on multiple devices running *Linux 6.8.0* and *Linux 6.11.5*. The reallocated addresses are always located close to the end of the attacker-unmapped large buffer and consistently correspond to the same (virtual) offsets within the buffer. This offset stays consistent across different system configurations running the same kernel version. Thus, an attacker can measure this offset once for a given victim executable and kernel and then reuse this offset for attacks on other devices. Even if the victim-mapped buffer spans multiple pages, we find that all of its physical addresses are consistently predictable. This is in line with *Frame Feng Shui* described by Kwong et al. [36].

Evaluation. We evaluate our proof of concept on an Intel Core i7-8700 running *Linux 6.8.0*. We run the code $10\,000$ times and record the victim page offset relative to the predicted attacker buffer index and whether the Spectre attack correctly recovers all 20 bytes. In 7106 cases, the predicted page is allocated, and the complete secret is recovered without error. In 173 cases, the predicted page is allocated, but the Spectre attack fails to recover at least one bit correctly. The predicted attacker buffer index is one page off in the remaining 2721 cases. However, as this is a constant offset, the victim can be coerced into allocating one of two pages with a known physical address in all runs. In $72.79\,\%$ of executions, the physical address can be predicted.

Conclusion. While the *v2p*-oracle is limited to addresses mapped by an attacker, this limitation can be overcome by coercing a victim to reuse addresses with known physical addresses after they are unmapped. On devices running Linux, an attacker can accurately predict which pages are reused by the victim.

6.2. DRAMA Attack

In this case study, we demonstrate that the *v2p*-oracle can be used to mount a DRAMA covert channel [50].

Setup. We mount the covert channel on an Intel Core i7-8700 machine with $16\,\mathrm{GB}$ of memory. We assume the slice and DRAM addressing functions have been successfully reverse-engineered before setting up the covert channel. The sender and receiver processes allocate a large contiguous memory region and use the *v2p*-oracle to infer the physical base address. Since the *v2p*-oracle uses timing measurements on L3 slices and is, therefore, susceptible to noise from other processes, we start the receiver process, wait for the *v2p*-oracle to complete its work, and then start the sender process. We randomly select a virtual address that maps to the attacker-chosen DRAM set 0, which is used for the covert channel. Since the DRAMA covert channel relies on inducing row conflicts, the chosen virtual addresses must map to different rows; otherwise there would be no row conflicts. We rely on the fact that the *v2p*-oracle sets up two sufficiently big mappings that span many rows, removing the chance that we get the same row for sender and receiver.

For simplicity, the sender and receiver synchronize via the shared time stamp counter accessible via `rdtscp`. If the corresponding sender bit is '1', we introduce a row conflict on DRAM set 0 by keeping the row open. We alternatingly use `clflush` and `mov` to keep the data in the row buffer. The receiver measures the DRAM access time of its virtual address (DRAM set 0) and compares it against a threshold. If the access time exceeds the threshold, the receiver logs the current bit to be a '1', '0' in the other case.

Evaluation. The *v2p*-oracle takes approx. $30\,\mathrm{min}$ with a pattern length of $25\,\mathrm{MB}$ to set up a contiguous mapping with a known physical address. The total setup time of the covert channel is, therefore, $1\,\mathrm{h}$. Note that this is a one-time effort; after that, the covert channel can operate for an arbitrary time span. We send $1\,\mathrm{MB}$ of random data over the covert channel and observe a transmission rate of $1.04\,\mathrm{kbit/s}$ with a low error rate of only $0.1\,\%$. This case study shows that the

v2p-oracle can be successfully used for mounting an end-to-end DRAM row conflict covert channel from unprivileged processes. While the setup time is relatively high, this is an unoptimized one-time effort.

6.3. Perfect Cache Eviction for AES T-Tables

With the knowledge of physical addresses and the slice function, we can build perfect eviction sets without relying on collision-based eviction-set generation [64].

Setup. We run our case study on the Intel Xeon E5-2697 v4 with 18 slices and 512 GB memory. For such a system, no slice function was previously known, and the approach by Gerlach et al. [11] is not applicable as the measurement would take roughly 1.75 years. We demonstrate the perfect eviction set on the well-known example of AES T-tables from OpenSSL 3.4.0 (released October 2024). Note that the T-table implementation is largely unchanged since OpenSSL 1.0.1e, which is regularly used as a "benchmark" for side channels [10], [11], [18], [37], [51]. Our implementation is based on the code from Gruss et al. [18]. However, we reduce the number of encryptions to 1000 and use our reverse-engineered slice function.

Evaluation. We execute our attack 1000 times. Figure 9 (Section D) shows the typical heatmap for the T-table accesses, with the expected diagonal. We fully recover the correct key in 97 % of the runs, with only single-byte errors in the remaining runs. Thus, these remaining cases can be easily brute-forced in negligible time. Our minimal eviction set also outperforms a Flush+Reload-based implementation. On average, recovering the key with our eviction takes $328\,\mathrm{ms}\pm0.1$, while Flush+Reload takes $379\,\mathrm{ms}\pm0.09$.

7. Discussion

In this section, we discuss other CPU architectures, using additional side channels, related work and propose mitigations.

7.1. Other Architectures

In this paper, our focus is purely on Intel CPUs, as they have the most complex slice functions. As our reverse-engineering efforts show, server CPUs used in the cloud and new hybrid microarchitectures have complex non-linear slice functions. Thus, as evaluated in Section 5.3, they allow building an efficient virtual-to-physical address oracle. While AMD also has the concept of cache slices, the design is different and has significantly lower complexity. AMD uses so-called core complexes (CCX), which typically consist of 4 cores that share a part of the last-level cache. Thus, the slice function within the CCX is significantly simpler, using only bits within the page offset [11]. Consequently, the slice function on AMD does not depend on any unknown part of the physical address and can thus not be used for a virtual-to-physical address oracle. ARM also supports cache slices [2], but they are similarly limited. Moreover, we are unaware of any work reverse-engineering the cache-slice function on ARM devices or the existence of cache slices on any hardware RISC-V CPU.

7.2. Additional Side-channel Information

For the *v2p*-oracle, we solely rely on the cache-slice function. While this results in a practical oracle (cf. Section 4), the *v2p*-oracle could be improved further by considering additional side-channel information. For example, additional constraints for the physical address could be generated via the DRAM addressing function [50]. Previous work showed that the DRAM addressing function can be used on older Intel CPUs to infer the cache set from a virtual address [52]. We expect that additional effects, such as from TLB timings [12] or page-table-walk contention [73], can further improve the *v2p*-oracle. We leave the evaluation of how the combination of different side-channel leakages can be used to improve the *v2p*-oracle to future work.

7.3. Related Work

In the following, we cover related work on slice reverse engineering and virtual-to-physical address oracles.

Reverse Engineering of L3 Slice Addressing Functions. The microarchitectural hash functions that map physical addresses to L3 cache slices are linear if the number of slices is a power of two [24], [27], [43] and non-linear otherwise [11], [44], [68]. A generic approach for reverse-engineering linear slice-addressing functions was first shown by Maurice et al. [43]. The linear nature of these functions makes it possible to interpolate them from a small number of measurements. Gerlach et al. [11] presented the first general approach to reverse-engineer non-linear slice-addressing functions, advancing previous techniques that required manual reverse-engineering of the structure [68] or only partially reverse-engineered the functions by relying on look-up tables [44]. While their approach is generic and can be applied to infer minimal versions of arbitrary non-linear functions, the time complexity of their approach is exponential in the number of function input bits.

Virtual-to-Physical Address Oracles. Until 2015, unprivileged users could access physical addresses via the `/proc/self/pagemap` interface on Linux systems [32]. As a response to previous attacks, this interface was restricted to privileged users [29], [46]. Since then, researchers have developed various techniques to obtain information about address mappings. Gruss et al. [16] used the prefetch instruction to determine if two virtual addresses map to the same physical address. Mounting the approach on the physmap region, which maps the physical memory into the kernel address space, allowed for determining virtual-to-physical mappings. However, recent work by Schwarzl et al. [56] attributed the leakage to a Spectre gadget in the kernel, and not the prefetch instruction. Wikner et al. [66] and Trujillo et al. [60] also used Spectre gadgets in the kernel to perform a cache attack on the physmap region. Most notably, such Spectre-based approaches rely on the presence

of a Spectre gadget in the kernel. Such gadgets are, however, regularly patched. The SPOILER [28] side channel can leak the 8 least significant bits of a physical page number by using timing differences induced by the dependency resolution logic that serves speculative loads. Meltdown-type attacks [4], such as Meltdown [39] or RIDL [63] can leak physical addresses as page-table entries also end up buffers from which these attacks leak. However, modern CPUs used in this paper are unaffected by these vulnerabilities.

7.4. Mitigations

In our case studies (Section 6), we show that complex cache-slice functions increase the attack surface of modern CPUs. This motivates the need for suitable mitigations.

Software. In a system that uses a vulnerable cache-slice function, the operating system can use a modified memory allocator [72] that does not give large physically contiguous regions to user-space processes. This prevents an attacker from obtaining a sufficiently large "fingerprint" pattern to uniquely identify the physical base address. However, the attacker can still reduce the number of candidate physical base addresses.

Hardware. A possible hardware defense could be to add a 64-bit IV that is XORed with the physical address before feeding it to the cache-slice function. The IV is generated randomly upon boot. This would lead to different patterns every time a system is restarted. Given that the attacker has no method to infer the IV, attacks become significantly more difficult [11].

8. Conclusion

We provided a significant step forward in reverse-engineering complex, non-linear cache-slice functions on modern Intel CPUs, particularly those with hybrid architectures. By exploiting the structural characteristics of these microarchitectural hash functions, our method drastically reduces reverse-engineering time and handles large memory configurations and diverse slice structures, resulting in multiple new slice functions. We introduced a novel unprivileged virtual-to-physical address oracle, the *v2p*-oracle, which combines the slice function with a timing side channel to infer physical addresses. We demonstrated the practicality of our approach for targeted Spectre attacks, DRAMA attacks, and eviction-set creation, showing the attack surface introduced by complex cache-slice functions.

Acknowledgements

We want to thank our shepherd and the anonymous reviewers, for their comments and valuable suggestions. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 491039149. We thank Daniel Weber for helpful feedback on this work, and Katharina Buchthal for proposing the use of the KMP algorithm.

References

[1] Andreas Abel and Jan Reineke. Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[2] Arm Limited. Cache slice and master port selection, 2024. URL: https://developer.arm.com/documentation/100453/0300/functional-description/l3-cache/cache-slices-and-portions/cache-slice-and-master-port-selection?lang=en.

[3] Robert K Brayton, Gary D Hachtel, Curt McMullen, and Alberto Sangiovanni-Vincentelli. *Logic minimization algorithms for VLSI synthesis*. Springer Science & Business Media, 1984.

[4] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*, 2019. Extended classification tree and PoCs at https://transient.fail/.

[5] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. Don't mesh around: Side-Channel attacks and mitigations on mesh interconnects. In *USENIX Security Symposium*, 2022.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[7] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. Smash: Synchronized many-sided rowhammer attacks from javascript. In *USENIX Security Symposium*, 2021.

[8] Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong. *Mathematics for machine learning*. 2020.

[9] Funda Ergün, Sampath Kannan, S Ravi Kumar, Ronitt Rubinfeld, and Mahesh Viswanathan. Spot-checkers. In *ACM Symposium on Theory of Computing*, 1998.

[10] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering*, 2016.

[11] Lukas Gerlach, Simon Schwarz, Nicolas Faroß, and Michael Schwarz. Efficient and Generic Microarchitectural Hash-Function Recovery. In *S&P*, 2024.

[12] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.

[13] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *USENIX Security Symposium*, 2017.

[14] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In *CCS*, 2019.

[15] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.

[16] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*, 2016.

[17] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*, 2016.

[18] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[19] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 2011.

[20] Winston Haaswijk, Mathias Soeken, Alan Mishchenko, and Giovanni De Micheli. Sat-based exact synthesis: Encodings, topology families, and parallelism. *IEEE TCAD*, 2019.

[21] Sebastian Hack. shack/Synth: Synthesis of Loop-free Programs, 2024. URL: https://github.com/shack/synth.

[22] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. Reliable reverse engineering of intel dram addressing using performance counters. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020.

[23] Lorenz Hetterich, Fabian Thomas, Lukas Gerlach, Ruiyi Zhang, Nils Bernsdorf, Eduard Ebert, and Michael Schwarz. ShadowLoad: Injecting State into Hardware Prefetchers. In *ASPLOS*, 2025.

[24] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*, 2013.

[25] 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.

[26] Intel. Product specifications, 2024. URL: https://ark.intel.com/content/www/us/en/ark.html.

[27] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *Euromicro Conference on Digital System Design*, 2015.

[28] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security Symposium*, 2019.

[29] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *USENIX Security Symposium*, 2014.

[30] The kernel development community. Transparent Hugepage Support, Kernel Version 6.12.0-rc6, 2024. URL: https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html.

[31] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*, 2014.

[32] Kirill A. Shutemov. Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace, 2015. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce.

[33] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 1977.

[34] Philipp Koehn and Jean Senellart. Fast Approximate String Matching with Suffix Arrays and A* Parsing. In *Conference of the Association for Machine Translation in the Americas*, 2010.

[35] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: Hammering From the Next Row Over. In *USENIX Security Symposium*, 2022.

[36] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P*, 2020.

[37] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[38] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *AsiaCCS*, 2020.

[39] 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 *USENIX Security*, 2018.

[40] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.

[41] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*, 2018.

[42] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990.

[43] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Complex Addressing Using Performance Counters. In *RAID*, 2015.

[44] John D McCalpin. Mapping addresses to l3/cha slices in intel processors. Technical report, 2021.

[45] Todd K Moon. The expectation-maximization algorithm. *IEEE Signal processing magazine*, 13(6), 1996.

[46] Andrew Morton. [RFC][PATCH 1/2] fs proc: make pagemap a privileged interface, 2015. URL: https://lore.kernel.org/all/20150312153533.d1c6083e4a9e7825b1a4bc64@linux-foundation.org/.

[47] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[48] Ge Nong, Sen Zhang, and Daricks Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference*, 2009.

[49] 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. In *USENIX Security Symposium*, 2021.

[50] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, 2016.

[51] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *CCS*, 2021.

[52] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[53] Michael Schwarz, Moritz Lipp, and Claudio Canella. misc0110/PTEditor: A small library to modify all page-table levels of all processes from user space for x86_64 and ARMv8, 2018. URL: https://github.com/misc0110/PTEditor.

[54] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[55] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*, 2019.

[56] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative Dereferencing of Registers: Reviving Foreshadow. In *FC*, 2021.

[57] Mark Seaborn. Exploiting the DRAM rowhammer bug to gain kernel privileges, March 2015. Retrieved on June 26, 2015. URL: http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[58] Mark Seaborn. L3 cache mapping on Sandy Bridge CPUs, April 2015. Retrieved on June 26, 2015. URL: http://lackingrhoticity.blogspot.com/2015/04/l3-cache-mapping-on-sandy-bridge-cpus.html.

[59] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shin. Spechammer: Combining spectre and rowhammer for new speculative attacks. In *S&P*, 2022.

[60] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. Inception: Exposing new attack surfaces with training in transient execution. In *USENIX Security*, 2023.

[61] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[62] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS*, 2016.

[63] 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*, 2019.

[64] Pepe Vila, Boris Köpf, and Jose Morales. Theory and Practice of Finding Eviction Sets. In *S&P*, 2019.

[65] Pepe Vila, Boris Köpf, and José Francisco Morales. Theory and practice of finding eviction sets. *arXiv:1810.01497*, 2018.

[66] Johannes Wikner and Kaveh Razavi. Retbleed: Arbitrary speculative code execution with return instructions. In *USENIX Security*, 2022.

[67] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. DeepHammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips. In *USENIX Security Symposium*, 2020.

[68] 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.

[69] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. *JCEN*, 2017.

[70] Dogan Yigit Yenigun. toUpperCase78/Intel-Processors: Datasets for All Processors Maufactured By Intel , 2024. URL: https://github.com/toUpperCase78/intel-processors.

[71] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based Fault Injection using Selective State Reset. In *USENIX Security*, 2024.

[72] Ruiyi Zhang, Tristan Hornetz, Lukas Gerlach, and Michael Schwarz. Taming the Linux Memory Allocator for Rapid Prototyping. In *DIMVA*, 2025.

[73] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. Binoculars: Contention-based side-channel attacks exploiting the page walker. In *USENIX Security Symposium*, 2022.

Appendix A. Longest Repeating Pattern

Figure 8 and Figure 7 show the length of the longest repeating sub-pattern for CPUs with linear and non-linear slice addressing functions, respectively. For non-linear functions, the length never exceeds 1 % of the system's DRAM size. Note that the values for the Core i9-12900K and Xeon E-2176M match perfectly despite these CPUs having different addressing functions. With linear functions, the length is significantly higher. For the Core i7-9700K, the longest repeating pattern frequently repeats within itself. Hence, we require more than 95 % of the system's DRAM as contiguous memory to uniquely determine the physical address with larger DRAM configurations.

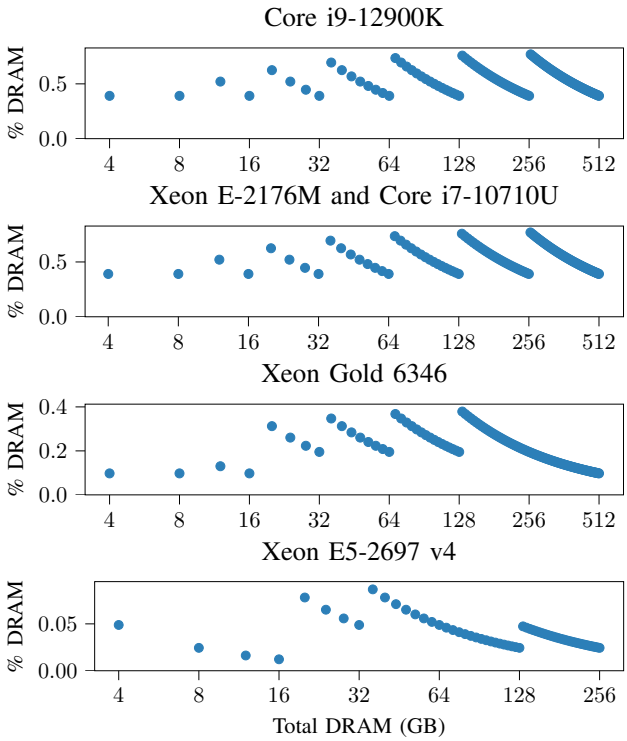


Figure 7: Length of the longest repeating sub-pattern relative to DRAM size for non-linear slice addressing functions.

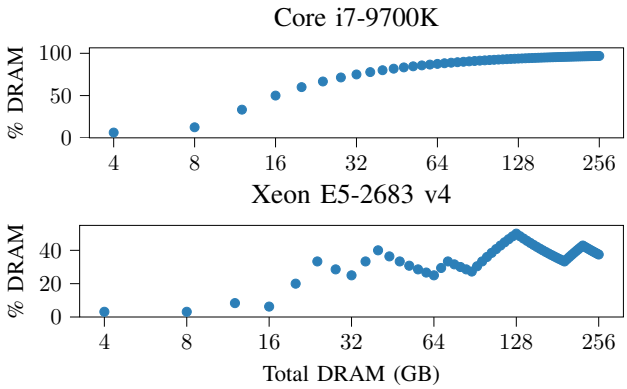


Figure 8: Length of the longest repeating sub-pattern relative to DRAM size for linear slice addressing functions.

Appendix B. Used Performance Counters

To measure the accessed slice for a physical address, we rely on performance counters. However, as the slice counters are not documented for every microarchitecture, we record all uncore performance counters and choose the one that has the best correlation with the number of expected slice accesses (cf. Section 3.6). Table 2 lists the counters we used for each microarchitecture based on this profiling step.

TABLE 2: The uncore performance counters used to measure the accessed slice.

CPU	μarch	Base	Event	Mask	Name
Intel Core i7-1185G7	Tiger Lake	0xb	0x37	0xf	*undocumented*
Intel Xeon E5-2697 v4	Broadwell	0x18	0x10	0x2	`UNC_H_TxR_BL.DRS_CORE`
Intel Xeon Gold 6526Y	Emerald Rapids	0x17	0x50	0x1	`UNC_CHA_REQUESTS.READS_LOCAL`
Intel Core i7-8700	Coffee Lake	0xd	0x2	0x1	*undocumented*
Intel Core i7-10510U	Comet Lake	0xc	0x24	0xf	*undocumented*
Intel Core Ultra 7 155H	Meteor Lake	0xe	0x22	0xf	`UNC_HAC_CBO_TOR_ALLOCATION.DRD`
Intel Core i9-12900H	Alder Lake	0xe	0x37	0xf	*undocumented*
Intel Core i9-13900K	Raptor Lake	0xe	0x35	0x1	*undocumented*
Intel Core i3-1005G1	Ice Lake	0xc	0x3c	0x7	*undocumented*
Intel Xeon Gold 6346	Ice Lake	0x16	0x32	0x7f	*undocumented*
Intel Core Ultra 9 285K	Arrow Lake	0x10	0x35	0x1	`UNC_HAC_CBO_TOR_ALLOCATION.DRD`
Intel Core i5-13420H	Raptor Lake	0xe	0x35	0x1	*undocumented*

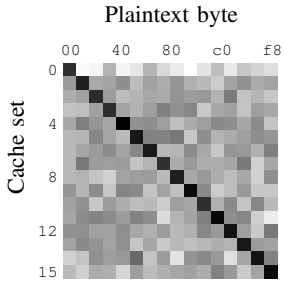


Figure 9: AES T-table cache-access pattern on an Intel Xeon E5-2697 v4 with 18 slices.

Appendix C. Transparent Huge Page Settings

Table 3 shows the default settings for transparent huge pages (THPs) on different Linux distributions. The setting `always` indicates that THPs are always allocated. Meanwhile, `madvise` indicates that THPs are only allocated when requested by the application. In this case, the application has to call `madvise` with the `MADV_HUGEPAGE` flag [30].

TABLE 3: Default settings for Transparent Huge Pages on different Linux distributions.

Distribution	Version	Default
Ubuntu	24.04 LTS Desktop	`madvise`
Fedora	41 Workstation	`madvise`
Linux Mint	22	`madvise`
Manjaro	24.1.1	`always`
Debian	12.7	`always`
openSUSE	Tumbleweed 20241031	`always`

Appendix D. AES T-Table Heatmap

Figure 9 shows the typical heatmap for the T-table accesses, with the expected diagonal when the first key byte is '0'.

Appendix E. Reverse-engineered Functions

Figure 10 shows the linear slice functions we reverse-engineered on our tested CPUs. Out of the 20 functions, 13 are new (L_{6d}, L_{6e}, L_{6f}, L_{6f}, L_{6h}, L_{7c} L_{8c}, L_{8d}, L_{9b}, L_{9d}, L_{9e}, L_{11b}, L_{12a}), and 9 known ones are extended with our approach (L_{6b}, L_{6c}, L_{7b}, L_{8a}, L_{8b}, L_{9a}, L_{9c}, L_{10b}, L_{11a}). We provide the structures of C_3 in Figure 11, and of C_5 in Figure 12. Additionally, the full structure of the slice addressing function of the Intel Xeon E5-2697 v4 is shown in Figure 13.

$$
\begin{aligned}
L_{6a} &= [b_6, b_{10}, b_{12}, b_{14}, b_{16}, b_{17}, b_{18}, b_{20}, b_{22}, b_{24}, b_{25}, b_{26}, b_{27}, b_{28}, b_{30}, b_{32}, b_{33}, b_{35}, b_{36}]_\oplus \\
L_{6b} &= [b_6, b_8, b_9, b_{10}, b_{14}, b_{15}, b_{17}, b_{18}, b_{20}, b_{23}, b_{27}, b_{30}, b_{31}, b_{34}, b_{36}, b_{38}]_\oplus \\
L_{6c} &= [b_6, b_{11}, b_{12}, b_{16}, b_{18}, b_{21}, b_{22}, b_{23}, b_{24}, b_{26}, b_{30}, b_{31}, b_{32}, b_{35}, b_{38}]_\oplus \\
L_{6d} &= [b_6, b_7, b_8, b_9, b_{10}, b_{12}, b_{13}, b_{14}, b_{15}, b_{18}, b_{19}, b_{20}, b_{22}, b_{24}, b_{25}, b_{30}, b_{32}, b_{33}, b_{34}, b_{38}]_\oplus \\
L_{6e} &= [b_6, b_9, b_{11}, b_{17}, b_{18}, b_{19}, b_{24}, b_{25}, b_{27}, b_{29}, b_{30}, b_{35}, b_{36}, b_{37}]_\oplus \\
L_{6f} &= [b_6, b_7, b_8, b_{12}, b_{16}, b_{17}, b_{20}, b_{21}, b_{22}, b_{23}, b_{24}, b_{25}, b_{26}, b_{28}, b_{30}, b_{33}, b_{35}]_\oplus \\
L_{6h} &= [b_6, b_7, b_{11}, b_{13}, b_{16}, b_{17}, b_{18}, b_{19}, b_{21}, b_{25}, b_{26}, b_{27}, b_{30}, b_{33}, b_{35}, b_{36}]_\oplus \\
L_{7a} &= [b_7, b_{11}, b_{13}, b_{15}, b_{17}, b_{19}, b_{20}, b_{21}, b_{22}, b_{23}, b_{24}, b_{26}, b_{28}, b_{29}, b_{31}, b_{33}, b_{34}, b_{35}, b_{37}]_\oplus \\
L_{7b} &= [b_7, b_{12}, b_{13}, b_{17}, b_{19}, b_{22}, b_{23}, b_{24}, b_{25}, b_{27}, b_{31}, b_{32}, b_{33}, b_{36}]_\oplus \\
L_{7c} &= [b_7, b_8, b_{11}, b_{12}, b_{13}, b_{19}, b_{22}, b_{23}, b_{24}, b_{27}, b_{31}, b_{32}, b_{33}, b_{36}]_\oplus \\
L_{8a} &= [b_8, b_{12}, b_{13}, b_{16}, b_{19}, b_{22}, b_{23}, b_{26}, b_{27}, b_{30}, b_{31}, b_{34}, b_{35}, b_{36}, b_{37}]_\oplus \\
L_{8b} &= [b_8, b_{13}, b_{14}, b_{18}, b_{20}, b_{23}, b_{24}, b_{25}, b_{26}, b_{28}, b_{32}, b_{33}, b_{34}, b_{37}]_\oplus \\
L_{8c} &= [b_8, b_{12}, b_{14}, b_{16}, b_{19}, b_{21}, b_{22}, b_{23}, b_{27}, b_{28}, b_{29}, b_{31}, b_{32}, b_{38}]_\oplus \\
L_{8d} &= [b_8, b_9, b_{12}, b_{15}, b_{16}, b_{18}, b_{19}, b_{21}, b_{23}, b_{25}, b_{26}, b_{28}, b_{30}, b_{34}, b_{35}]_\oplus \\
L_{9a} &= [b_9, b_{12}, b_{16}, b_{17}, b_{19}, b_{21}, b_{22}, b_{23}, b_{25}, b_{26}, b_{27}, b_{29}, b_{31}, b_{32}]_\oplus \\
L_{9b} &= [b_9, b_{11}, b_{12}, b_{13}, b_{14}, b_{17}, b_{19}, b_{21}, b_{22}, b_{26}, b_{27}, b_{29}, b_{32}]_\oplus \\
L_{9c} &= [b_9, b_{14}, b_{15}, b_{19}, b_{21}, b_{24}, b_{25}, b_{26}, b_{27}, b_{29}, b_{33}, b_{34}, b_{35}, b_{38}]_\oplus \\
L_{9d} &= [b_9, b_{13}, b_{14}, b_{17}, b_{18}, b_{19}, b_{20}, b_{21}, b_{22}, b_{23}, b_{25}, b_{27}, b_{30}, b_{32}, b_{36}, b_{38}]_\oplus \\
L_{10a} &= [b_{10}, b_{11}, b_{13}, b_{16}, b_{17}, b_{18}, b_{19}, b_{20}, b_{21}, b_{22}, b_{27}, b_{28}, b_{30}, b_{31}, b_{32}, b_{33}]_\oplus \\
L_{10b} &= [b_{10}, b_{15}, b_{16}, b_{20}, b_{22}, b_{25}, b_{26}, b_{27}, b_{28}, b_{30}, b_{34}, b_{35}, b_{36}]_\oplus \\
L_{11a} &= [b_{11}, b_{16}, b_{17}, b_{21}, b_{23}, b_{26}, b_{27}, b_{28}, b_{29}, b_{31}, b_{35}, b_{36}, b_{37}]_\oplus \\
L_{11b} &= [b_{11}, b_{12}, b_{15}, b_{20}, b_{21}, b_{25}, b_{26}, b_{27}, b_{28}, b_{29}, b_{32}, b_{34}, b_{35}, b_{36}, b_{37}, b_{38}]_\oplus \\
L_{12a} &= [b_{12}, b_{14}, b_{16}, b_{18}, b_{19}, b_{20}, b_{22}, b_{24}, b_{25}, b_{27}, b_{28}, b_{30}, b_{31}, b_{32}, b_{33}, b_{35}, b_{36}, b_{38}]_\oplus
\end{aligned}
$$

Figure 10: All reverse-engineered XOR chains.

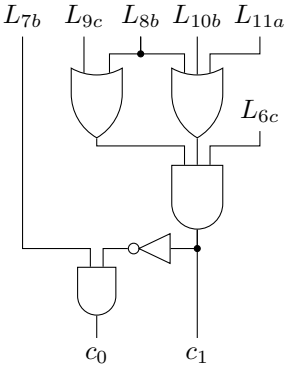


Figure 11: Structure of the C_3 mixer.

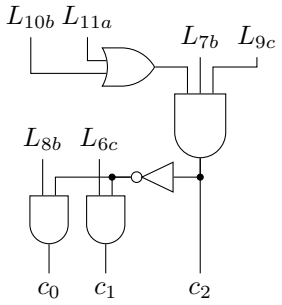


Figure 12: Structure of the C_5 mixer.

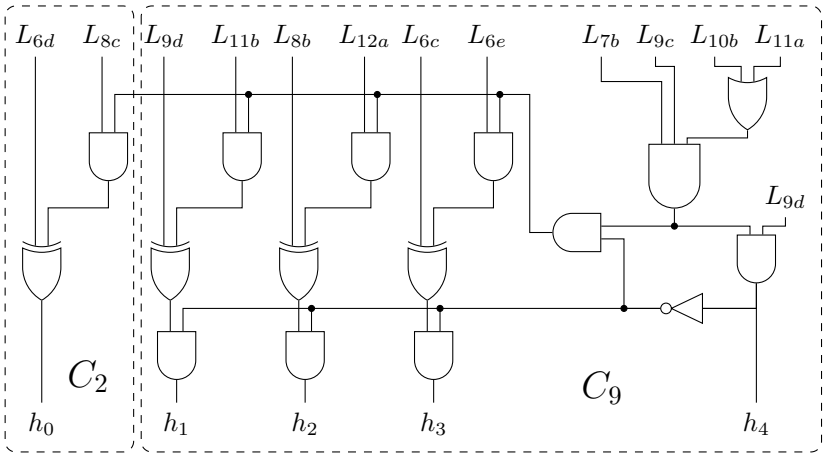


Figure 13: Cache slice function for the Intel Xeon E5-2697 v4

Appendix F.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

F.1. Summary

This paper presents a novel approach for reverse engineering the non-linear cache-to-slice functions on modern Intel machines. Understanding these functions is crucial for many types of microarchitectural side-channel attacks. The authors' proposed method applies some domain knowledge to the problem, resulting in a solution which slightly less generic than previous works, but which is drastically faster and more comprehensive. The knowledge of the mapping function is used to mount several practical attacks.

F.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

F.3. Reasons for Acceptance

1) The paper Provides a Valuable Step Forward in an Established Field and Creates a New Tool to Enable Future Science. The proposed method for reversing the cache to slice mapping is fast, robust and well-documented. It will be useful in understanding future architectures as well.
2) The paper Identifies an Impactful Vulnerability. The proposed attacks, in particular the v2p oracle, have offensive potential in themselves, and can also be used as building blocks in future attacks, as well as serving as inspiration for future defensive designs.