ZenLeak: Practical Last-Level Cache Side-Channel Attacks on AMD Zen Processors

Han Wang[†], Ming Tang[†‡], Quancheng Wang[†], Ke Xu[†], Yinqian Zhang[§]
[†]Key Laboratory of Aerospace Information Security and Trusted Computing,
Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan, China
{han.wang, m.tang, wangquancheng, kexuwhu}@whu.edu.cn
[§]Southern University of Science and Technology, yinqianz@acm.org

2025 62nd ACM/IEEE Design Automation Conference (DAC) | 979-8-3315-0304-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/DAC63849.2025.11133253

***Abstract*—While Last-Level Cache (LLC) side-channel attacks often target inclusive caches, directory-based attacks on non-inclusive caches have been demonstrated on Intel and ARM processors. However, the vulnerability of AMD's non-inclusive caches to such attacks has remained uncertain, primarily due to challenges in reverse-engineering cache addressing, constructing eviction sets, and evicting private cache lines.**

This paper addresses these challenges and demonstrates the feasibility of conducting LLC side-channel attacks on AMD's non-inclusive caches. We first reverse-engineer the cache addressing functions for the L2 set index, L3 slice, and L3 set index. Leveraging this insight, we construct the first eviction sets on AMD processors. We then introduce the first LLC side-channel attack on AMD's Zen series CPUs. The effectiveness of our approach is validated by attacking OpenSSL's AES T-table.

I. Introduction

Encryption is essential for data confidentiality, but side-channel attacks exploit physical characteristics of cryptographic implementations—such as execution time [1]–[3] and power consumption [4]—to extract secret information like cryptographic keys. These attacks leverage subtle hardware implementation details without software vulnerabilities.

The LLC has become a key target for side-channel exploits, with the Prime+Probe attack [1] being particularly notable. This method is especially concerning because it does not require the attacker to share a core or memory with the victim, lowering prerequisites and increasing the likelihood of success.

Prime+Probe attacks have traditionally targeted Intel's inclusive LLCs. Techniques for inclusive caches are not directly applicable to non-inclusive caches. In non-inclusive caches, evicting data from L3 does not necessarily evict corresponding data from L1 and L2, reducing attack effectiveness. Specifically, an attacker on one core cannot evict data from another core's private cache (Figure 1).

It has been demonstrated that in non-inclusive cache architectures, Intel's directory structure can be exploited to conduct conflict-based cache attacks on the LLC [5]. This approach enabled the first cross-core Prime+Probe attack on non-inclusive caches by exploiting the shared directory (Figure 2).

In contrast, this method is ineffective on AMD's non-inclusive cache architecture due to fundamental differences in directory structures. Intel uses a shared, capacity-constrained directory across all cores, whereas AMD assigns a core-partitioned shadow tag [6] in the L3 cache for each L2 cache entry. Consequently, AMD's cache directories prevent attackers from inferring a victim's cache activities through a shared directory.

To date, no LLC-based cache attacks without shared resources have been successfully carried out against AMD's non-inclusive caches. This raises a critical research question:

> **Research Question**: Is AMD's non-inclusive cache architecture inherently secure and immune to last-level cache side-channel attacks?

Challenge: Addressing this question involves three key challenges. *First*, AMD's cache addressing structure is undocumented, complicating the mapping of memory addresses to specific cache sets and slices. *Second*, no methods exist for constructing eviction sets on AMD's non-inclusive caches, which are essential for cache side-channel attacks. *Third*, cache lines cannot be evicted from private caches on other cores, limiting cross-core attacks.

Experiment Setup: All experiments were conducted on an AMD Ryzen 9 5900X (Zen 3). Starting with the Zen series, AMD has employed non-inclusive caches [7]. While addressing functions may vary between CPUs, the overall cache structure remains consistent, allowing our reverse engineering and attack methods to be adapted across the AMD Zen lineup.

Contribution: Our contributions are threefold. *First*, we comprehensively reverse-engineer the cache addressing functions for the L2 set index, L3 slice, and L3 set index on AMD's Zen series processors, revealing how physical addresses map to cache sets. Our methods are architecture-independent and not limited to AMD CPUs. *Second*, building on this reverse engineering, we propose the first effective algorithm for constructing eviction sets on AMD processors. *Third*, we introduce a method for cross-core eviction of private cache lines, enabling LLC side-channel attacks on

[‡]Corresponding author. This work was supported in part by the National Key R&D Program of China (No. 2022YFB3103800).

[§]Yinqian Zhang is in part supported by National Natural Science Foundation of China under grant No. 62361166633, National Key R&D Program of China under grant No. 2023YFB4503902, Shenzhen Science and Technology Program under grant No. JSGG20220831095603007

AMD's non-inclusive caches. We successfully demonstrate this approach by attacking OpenSSL's AES T-table, revealing potential vulnerabilities in these systems.

Disclosure: We have responsibly disclosed our findings to the AMD PSIRT team. AMD has acknowledged our work.

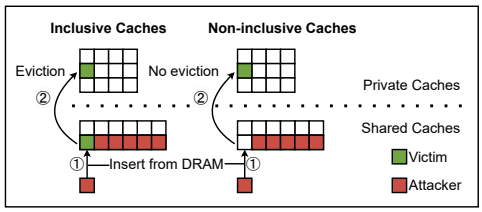

Fig. 1: Eviction attempts in (non-)inclusive cache architectures. In inclusive caches, inserting a new cache line into the LLC (step ①) evicts the corresponding line from the private cache (step ②). In non-inclusive caches, the victim cache line remains in the private cache despite the attacker's insertion.

II. Background and Related Work

LLC Side-Channel Attacks: Prime+Probe [1] introduced the concept of performing cache attacks on the LLC without shared memory. An attacker first fills specific cache sets with their data (prime) and then measures access latencies after the victim's execution (probe) to reveal memory access patterns. This method applies only to inclusive caches. Yan et al. [5] conducted directory-based attacks on Intel non-inclusive caches by targeting the Extended Directory (ED). By constructing eviction sets mapped to the same cache slice and the ED set, they force the victim's cache line from the private cache to the shared LLC, allowing detection of victim accesses through increased latencies. A similar attack has been performed on ARM processors [8]. Evict+Spec+Time [9] targets speculative attacks on Intel and AMD processors but replaces evict with the flush instruction in their implementation. This changes the threat model, as flush requires shared memory between attacker and victim, unlike eviction. On non-inclusive caches, flush and evict differ fundamentally, as eviction involves constructing sets and removing private cache lines—issues not fully explored in prior work.

Hardware Defenses Hardware approaches such as cache randomization [10] and partitioning [11] have not yet been widely adopted in commercial processors, leaving systems vulnerable to cache side-channel attacks. AMD has recommended software-based mitigation strategies. However, achieving universal and efficient protection remains challenging. Applying these techniques [12], [13] to arbitrary software is inherently complex and imposes a substantial burden on developers to safeguard sensitive data across diverse microarchitectures.

Reverse Engineering of Cache Addressing Functions: Initial efforts to reverse-engineer Intel LLC slice functions were presented in prior work [14], [15]. Specifically, two methods were proposed [15]: one utilizing the Performance Monitoring Unit (PMU) and another using latencies of the flush instruction. Subsequent research reverse-engineered the L2 set index of the Apple M1 processor using an eviction set [16]. An automated method for reverse-engineering nonlinear functions was also introduced in later studies [17]. However, these approaches have limitations when applied to AMD's non-inclusive caches, as discussed in Section III.

Construction of Eviction Sets: In inclusive caches, addresses are iteratively removed from a set, and if removing one stops evicting the target address, it is part of the eviction set [1]. Several improvements [18]–[20] have accelerated this pruning process. Various algorithms [5], [21] have also been proposed for Intel's non-inclusive caches; however, due to a lack of understanding of AMD's cache addressing, these algorithms have not been implemented on AMD processors. Specific details are discussed in Section IV.

III. Reverse Engineering

In this section, we address the first challenge: reverse engineering the cache addressing structure of AMD processors.

A. Cache Slicing

Two methods for reverse-engineering cache slicing on Intel CPUs were proposed: one based on Performance Monitoring Units (PMUs) and another on timing measurements of the flush instruction [15]. However, the PMU-based method is not applicable on AMD platforms due to the lack of relevant PMUs (e.g., LLC_LOOKUP events).

The principle of the timing-based method is that performing a flush operation on the slice closest to the core typically takes the least time. We found that this method cannot be directly applied to AMD CPUs because the time differences for flush operations across different slices are insufficient to provide conclusive results, as demonstrated in previous research [17]. We conducted a detailed analysis to understand the underlying reasons and subsequently enhanced the method.

Our analysis revealed that the execution time of the first 1,500 flush operations on each core is relatively short, but time increases significantly afterward (Figure 3). If we follow the original algorithm and frequently switch cores, the flush operations remain within the shorter time range, leading to insufficient differentiation between slices.

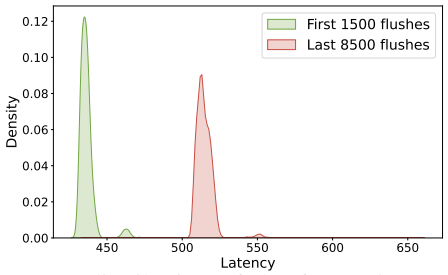

Fig. 3: Latency distribution of the first 1,500 and last 8,500 flushes on AMD cores.

To address this issue, we modified the algorithm to incorporate a warm-up phase and complete all address tests on each core before switching. This adjustment improves the reliability of the results. Although Gerlach et al. [17] proposed an eviction-based method to collect data on AMD CPU, it

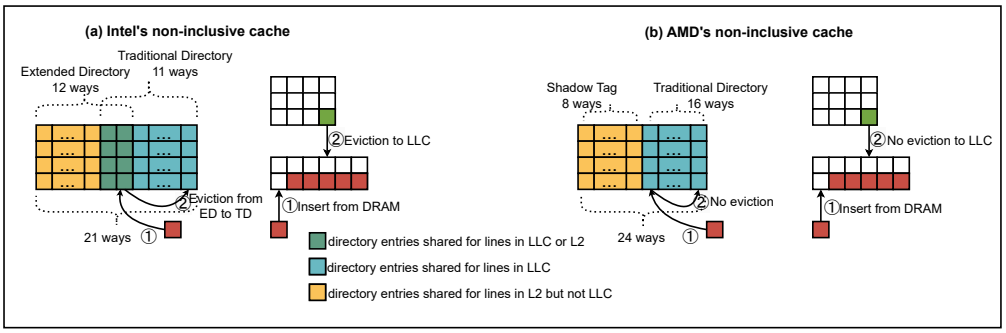

Fig. 2: Intel and AMD use distinct directory structures for non-inclusive caches. (a) Intel's shared directory allows LLC eviction through directory conflicts. (b) AMD's per-L2-line shadow tags prevent LLC eviction through the directory.

requires 23 hours to gather sufficient data for analyzing the slice-addressing hash function, whereas our method requires only 30 minutes.

On the Ryzen 9 5900X processor, we faced a unique challenge: the processor has six physical cores but eight cache slices in one CCX (CPU Complex). During our timing measurements, we observed that if a core flushes an address and the time is the shortest, we increment a count corresponding to that core-slice mapping. For certain addresses, we noticed that the counts between two cores were approximately equal, indicating that these addresses correspond to the extra two slices, as they do not uniquely map to a single core.

With the mapping between addresses and slice indexes established, we iterated through memory addresses, flipping the i-th bit (starting from bit 6, as bits 0–5 correspond to the cache line offset) in each address (`addr1`) to generate a new address (`addr2`). By comparing their index bits, we incremented the corresponding position in the count matrix if a difference was detected. This process tracks how flipped address bits affect index bits, revealing their relationship, as shown in Table I.

The threshold is calculated as:

$$\text{Threshold} = \left\lfloor \frac{N_{\text{addr}}}{2^B} \right\rfloor$$

where N_{addr} is the total number of addresses, and B denotes the number of address bits involved in hashing. This threshold determines whether a specific address bit contributes to the hash computation. The same formula is used to determine the involvement of address bits in Section III-B and III-C.

TABLE I: Correspondence between physical address bits and slice value bits.

Slice Bit	b6	b7	b8	b9	b10	b11	b12
0	23162	0	0	0	0	0	0
1	7216	11412	22290	6	8	13	10
2	11234	22448	97	7	14	19	16

Analyzing the count matrix, we derived the corresponding hash function. Previous studies [8], [15], [17] indicate that cache-addressing hash functions predominantly use XOR operations, so we focused exclusively on XOR-based functions in the analysis. Unlike previous work using an AMD Ryzen 9 5900HX [17], we found no bits involved in CCX selection on the Ryzen 9 5900X.

The specific hash function is as follows:

$$\{s_2, s_1, s_0\} = \{b_6 \wedge b_7,\ b_6 \wedge b_7 \wedge b_8,\ b_6\}$$

B. L2 Index Addressing

It is commonly assumed that cache set index bits correspond to the base-2 logarithm ($\log_2$) of the number of sets, as in Intel processors [5]. However, we found that on AMD processors, L2 and L3 cache indexing may involve the most significant bits (MSBs) of the address, contrary to this assumption.

This prompted us to investigate which bits contribute to L2 cache addressing on AMD processors. Similar patterns exist on other architectures; for instance, previous work showed that the Apple M1 CPU also uses high-order address bits for cache indexing [16]. Their method involved constructing eviction sets—a technique unavailable for AMD processors before our work. Moreover, our eviction set construction algorithm relies on insights gained from reverse-engineering these addressing functions. Similarly, the reverse engineering of snoop filter addressing on ARM processors relied on specific PMU events [8], which are not available on AMD processors. Our method does not depend on architecture-specific knowledge, making it broadly applicable to other architectures.

We conducted experiments to identify the bits involved in L2 cache indexing. Sixteen nodes were added to a linked list with physical addresses spaced at regular intervals. By adjusting the interval between addresses, we ensured that the physical addresses differed by only four bits (see Figure 4). If these bits do not contribute to the L2 set index computation, access latency increases due to multiple addresses mapping to the same cache set, exceeding the 8-way associativity. As shown in Figure 5, when the interval was set to 2^{16} and 2^{17}, access latency increased; when the interval was increased to 2^{18}, latency decreased, indicating that the addresses no longer mapped to the same cache set. We deduced that bits 6–15 and bits 21–27 of the physical address collectively determine the L2 cache index on AMD processors.

To further investigate the impact of higher-order address bits, we reverse-engineered the L2 cache addressing function using Algorithm 1. We first established two addresses:

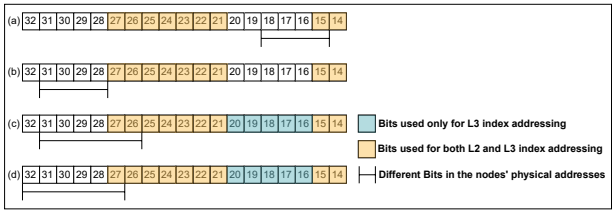

Fig. 4: Using a sliding window to control the bits of physical address differences among nodes in a linked list.

Addr1: Defined by a base address, an offset (gap), and varying cache sets.

Addr2: Defined by the base address and different cache sets.

Algorithm 1: Algorithm for determining the relationship between the L2 set index and the physical address.

```
Input : Memory region mem, range from begin to end, step size
        gap, number of nodes num
Output: Results of each test
base ← starting address of mem;
for offset from begin to end step gap do
    for j from 0 to number_of_L2_set do
        Addr1 ← base + offset + (j ≪ 6);
        AppendNodes(Addr1, num/2);
        for k from 0 to number_of_L2_set do
            count ← 0;
            Addr2 ← base + (k ≪ 6);
            AppendNodes(Addr2, num/2);
            ShuffleList();
            for trial from 0 to 9 do
                avg_time ← TraverseList() ;
                if avg_time exceeds threshold then
                    count ← count +1;
            if count >5 then
                same_set ← 1;
```

For each address, we constructed linked lists containing eight nodes, each separated by a gap of 0x10000. This setup ensured that the nodes within each list were identical in the bits relevant to the L2 cache index.

We measured the access latency by traversing these linked lists. If the combined latency exceeded a predefined threshold, it indicated that both lists mapped to the same L2 cache set, confirming how bits 21-–27 affected bits 6—15.

This analytical approach mirrors the method used in our cache-slicing analysis. After mapping addresses to L2 set indexes, we flipped specific address bits to generate new addresses and recorded the relationship between differing index bits and address bits, producing a count matrix (Table II).

Using the count matrix, we derived a hash function that employs XOR operations. Given the address bits:

$$b_i = (\text{addr} \gg i)\,\&1 \quad \text{for } i \in [6, 15] \cup [21, 27]$$

Each bit of the L2 index is defined as follows:

$$index_i = \begin{cases} b_6 & \text{if } i = 0, \\ b_{i+6} \oplus index_{i-1} & \text{if } 0 < i < 3, \\ b_{i+6} \oplus b_{i+18} \oplus index_{i-1} & \text{if } 3 \le i \le 10. \end{cases}$$

C. L3 Index Addressing

Reverse engineering the index addressing for the L3 cache follows the same principles as for the L2 cache but involves differences in identifying specific bits in the calculation. A linked list of 64 nodes was constructed, with physical addresses spaced at regular intervals. By adjusting the intervals, we ensured that the physical addresses differed by exactly six bits, as shown in Figure 4. Cache lines were evicted from private caches before each traversal, as detailed in Section V-A. As depicted in Figure 5, the results showed that with a 2^{26} interval, the 64 nodes spanned four L3 sets, each containing 16 cache lines. In contrast, with a 2^{27} interval, the nodes occupied only two L3 sets, exceeding the associativity limit and increasing access latency. These observations indicate that L3 set indexing is determined by bits 9–27.

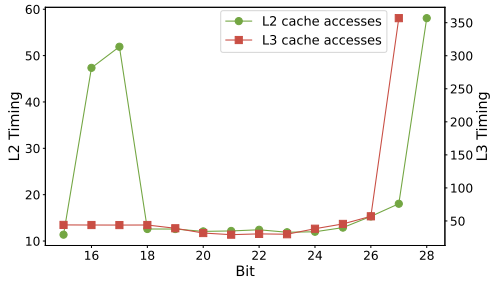

Fig. 5: Relationship between the interval in physical addresses of linked list nodes and memory access timing.

Similar to reversing the L2 set index, given the address bits:

$$b_i = (\text{addr} \gg i)\,\&1 \quad \text{for } i \in [9, 27]$$

Each bit of the L3 index is defined as follows:

$$index_i = \begin{cases} b_9 \oplus b_{21} & \text{if } i = 0, \\ b_{i+9} \oplus b_{i+21} \oplus index_{i-1} & \text{if } 0 < i \le 7, \\ b_{i+9} \oplus index_{i-1} & \text{if } 7 < i \le 12. \end{cases}$$

IV. Eviction Set Construction

An eviction set is a group of memory addresses that map to the same cache set; in cache attacks, accessing these addresses forces the eviction of specific cache lines through the cache replacement policy. In this section, we analyze why previous eviction set construction algorithms fail on AMD Zen processors and introduce a new method based on reverse-engineering cache addressing.

Firstly, algorithms designed for inclusive caches are not effective on AMD's non-inclusive cache architecture. An eviction set construction algorithm for inclusive caches [1] iteratively removes addresses from a candidate set U and observes whether the target address x still experiences eviction. If removing an address x' prevents eviction of x, then x' is necessary for the eviction set. However, on non-inclusive caches like those in AMD processors, some addresses in the minimal eviction set U may reside only in private caches and not in the LLC. They cannot conflict with x in the LLC, which results in false negatives when constructing eviction sets [5].

TABLE II: Correspondence between the physical address bits and L2 set index bits.

Index Bit	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15	b16	b17	b18	b19	b20	b21	b22	b23	b24	b25	b26	b27
0	130946	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	65410	130818	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	32640	65280	130562	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	16256	32512	65024	130050	0	0	0	0	0	0	0	0	0	0	0	130050	0	0	0	0	0	0
4	8064	16128	32256	64514	129026	0	0	0	0	0	0	0	0	0	0	64512	129026	0	0	0	0	0
5	3968	7936	15872	31746	63490	126978	0	0	0	0	0	0	0	0	0	31744	63490	126978	0	0	0	0
6	1920	3840	7680	15362	30722	61442	122882	0	0	0	0	0	0	0	0	15360	30722	61442	122882	0	0	0
7	896	1792	3584	7170	14338	28674	57346	114690	0	0	0	0	0	0	0	7168	14336	28672	57344	114690	0	0
8	384	768	1536	3073	6145	12289	24577	49153	98306	0	0	0	0	0	0	3072	6144	12288	24576	49153	98305	0
9	128	256	512	1024	2048	4096	8192	16384	32769	65538	0	0	0	0	0	1024	2048	4096	8192	16384	32768	65536

Secondly, existing algorithms for non-inclusive caches also face limitations on AMD processors. The L2 occupy set was introduced to address false positives in non-inclusive caches [5]. It evicts addresses from private caches to ensure that all candidate addresses reside in the LLC and can conflict with x. It is effective on Intel's non-inclusive caches because flipping the 16th bit of the physical address allows the L2 occupy set and the L3 eviction set to be mapped to different slices, preventing false positives. However, our reverse-engineering of AMD's cache addressing revealed that all cache lines in the same L2 cache set are mapped to the same LLC slice, making this method inapplicable to AMD architecture.

Through reverse-engineering in Section III, we determined that in Zen 3 processors, the L2 set index is derived from bits 6—15 and 21—27 of the physical address, LLC slice addressing is determined by bits 6—8, and L3 set indexing is based on bits 9—27. Addresses within the same L2 set and slice must also belong to the same L3 set. Therefore, we can adapt the L2-driven Candidate Address Filtering method [21] for AMD processors.

To construct the L3 eviction set, we first identify the L2 eviction set EV'. After obtaining the L2 eviction set EV', we filter the address set U to select those addresses that share the same L2 set as the target address, using them as candidates for the L3 eviction set. The L3 eviction set EV is then constructed by ensuring that all candidate addresses are evicted into the L3 cache, avoiding false negatives. Finally, we construct the L3 eviction set from these candidates. The whole process is shown in Algorithm 2.

In this approach, the L2 eviction set serves as the occupy set. Since it is a subset of the full eviction set, this structure prevents false positives and enables effective eviction set construction on AMD's non-inclusive caches.

Algorithm 2: Eviction set construction algorithm

Input : Target address x, Address set U
Output: Eviction set EV
$EV' \leftarrow \texttt{FindEV}(x, U, L2\_Threshold, \varnothing)$;
$U' \leftarrow \texttt{CheckConflict}(U, EV')$;
$EV \leftarrow \texttt{FindEV}(x, U', L3\_Threshold, EV')$;

In our experiments, the `FindEV` function used the $O(n^2)$ algorithm, proposed in previous work [1], but more efficient algorithms can be applied. Figure 6 shows the cumulative success rates: 98% after 7 attempts on 1GB pages and 78% after 8 attempts on 4KB pages.

Fig. 6: Cumulative success rate of eviction set construction on 1GB and 4KB pages.

V. Side Channels

A. Eviction from Private Caches

It was observed that a context switch on Intel processors may flush cache data to memory [22]. This allows an attacker to evict a victim's cache lines to memory by inducing a context switch, for example, by sending a signal. If both processes share the same user ID, no special privileges are required to send a signal to a process running on a different core.

However, our experiments on AMD CPUs revealed different behavior. As shown in Figure 7, we measured the cache latency of accessing cache lines from the same L2 and L3 cache sets, both under normal conditions and after the process received a signal. Under normal conditions, when the number of entries was below 24 (the combined associativity of the L2 and L3 caches), accesses hit the cache, resulting in low latency. When the number of entries exceeded 24, cache misses occurred, leading to a significant increase in latency.

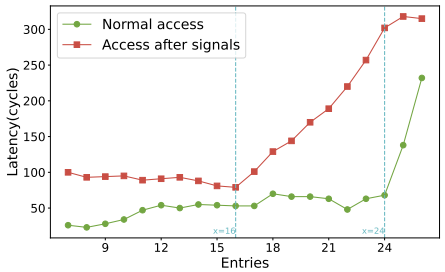

Fig. 7: Comparison of cache access latency under normal conditions and after receiving signals.

Interestingly, after receiving a signal, no significant increase in latency was observed, suggesting that the data remained in the L3 cache. When the number of entries exceeded 16 (the L3 cache associativity), latency increased, indicating that the

L2 set was not involved. This suggests that the signal caused cache lines to be evicted only from private caches.

Thus, we conclude that on AMD platforms, the signal mechanism can be exploited to flush the victim's private caches while retaining data in the L3 cache, presenting a unique opportunity for cache-based side-channel attacks.

B. Prime+Signal+Probe

Based on these findings, we propose a cache attack targeting the LLC in AMD's non-inclusive cache architecture. In this attack, the attacker first identifies the victim's target LLC sets, utilizing power spectral density to detect periodic accesses to the LLC sets in the frequency domain [21]. Once the target LLC sets are identified, the attack proceeds as follows.

The attacker constructs an eviction set, either through an eviction set construction algorithm or by selecting cache lines from a 1GB large page. The relevant cache sets are primed to establish a known state. A signal is sent to the victim process, triggering a context switch that evicts the victim's cache lines from private caches. The attacker then probes the cache sets by measuring access latency. By analyzing these latency measurements, the attacker can determine whether the victim accessed specific cache lines. If the victim accessed cache lines containing secret data, the attacker's corresponding cache lines would be evicted, leading to increased access latency. This process is illustrated in Figure 8.

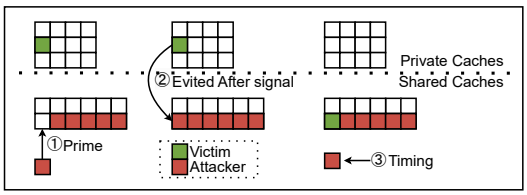

Fig. 8: Overview of the Prime+Signal+Probe attack. The attacker primes the cache set with their data (step ①), and sends a signal to the victim, triggering a context switch that evicts the victim's cache lines from the private L2 cache to the shared LLC if they were accessed by the victim beforehand (step ②). Finally, the attacker probes for increased latency to detect if specific lines were accessed by the victim (step ③).

We selected the AES T-table implementation as our side-channel attack target. Despite countermeasures [23]–[25] that mitigate most cache side channels in AES implementations, the T-table approach remains widely used for examining emerging side-channel attacks compared to earlier methods [3], [26], [27]. Drawing from similar attack scenarios [1], [21], our Prime+Signal+Probe technique also applies to other cryptographic algorithms, including RSA, ElGamal, and ECDSA.

The T-table implementation optimizes AES by converting SubBytes, ShiftRows, and MixColumns into 16 memory lookups from four precomputed tables. In the first round, table accesses are made to entries $T_j[p_i \oplus k_i]$, where $i \equiv j \mod 4$ and $0 \leq i < 16$, allowing attackers to infer possible key-byte values (k_i) when the plaintext (p_i) is known.

We conducted a known-plaintext attack on the AES T-table implementation in OpenSSL. By analyzing whether the victim accessed specific T-table entries, we inferred possible key-byte values. For each round, we recorded the results of 1024 AES encryptions. Figure 9 illustrates a cache template generated from 1024 encryptions, revealing a discernible pattern.

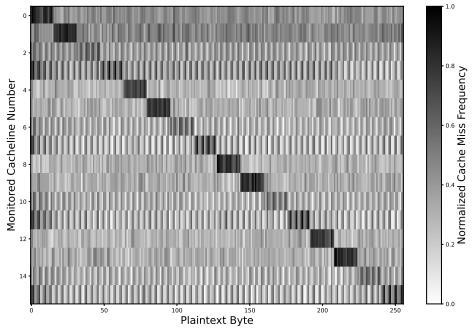

Fig. 9: Cache templates showing the cache miss frequency of the eviction set.

We performed multiple rounds of the attack, applying majority voting to the results. Scenarios included selecting cache lines from the same L3 set on a 1GB page and constructing eviction sets with the proposed algorithms on 1GB and 4KB pages. Figure 10 shows the accuracy of majority voting: on 4KB pages, accuracy reached 100% with zero standard deviation when the vote count exceeded 16, allowing key-byte inference. For 1GB pages, achieving 100% accuracy required a majority vote count of 30 when constructing eviction sets and 22 when selecting cache lines directly.

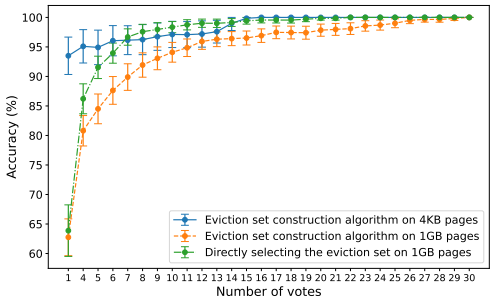

Fig. 10: Accuracy of majority voting on 1GB and 4KB pages.

VI. Conclusion

We analyzed the security of AMD's non-inclusive caches. By reverse-engineering the cache addressing functions, developing an eviction set construction algorithm, and employing signals to evict cache lines from private caches, we successfully execute the last-level cache side-channel attack on AMD processors. Given the challenges of applying software mitigation to arbitrary applications, our findings highlight the need for hardware-based defenses in commercial processors.

REFERENCES

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

[2] H. Wang, M. Tang, K. Xu, and Q. Wang, "Cache bandwidth contention leaks secrets," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2024.

[3] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: a fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*, pp. 279–299, Springer, 2016.

[4] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "Platypus: Software-based power side-channel attacks on x86," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 355–371, IEEE, 2021.

[5] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy (SP)*, (San Francisco, CA, USA), pp. 888–904, IEEE, 2019.

[6] S. Srinivasan and W. L. Walker, "Shadow tag memory to monitor state of cachelines at different cache level," Sept. 11 2018. US Patent 10,073,776.

[7] I. Cutress, "The amd zen and ryzen 7 review: A deep dive on 1800x, 1700x and 1700," *Anandtech, Mar*, vol. 2, 2017.

[8] Z. Kou, S. Sinha, W. He, and W. Zhang, "Attack directories on arm big.little processors," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, (San Diego California), pp. 1–9, ACM, 2022.

[9] S. H. W. Cheng, C. Chuengsatiansup, D. Genkin, D. McNeil, T. Murray, Y. Yarom, and Z. Zhang, "Evict+ spec+ time: Exploiting out-of-order execution to improve cache-timing attacks," *Cryptology ePrint Archive*, 2024.

[10] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "{ScatterCache}: thwarting cache attacks via cache set randomization," in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 675–692, 2019.

[11] G. Dessouky, A. Gruler, P. Mahmoody, A.-R. Sadeghi, and E. Stapf, "Chunked-cache: On-demand and scalable cache isolation for security architectures," *arXiv preprint arXiv:2110.08139*, 2021.

[12] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "{DATA}–differential address trace analysis: Finding address-based {Side-Channels} in binaries," in *27th USENIX Security Symposium (USENIX Security 18)*, pp. 603–620, 2018.

[13] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 406–421, 2017.

[14] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *2013 IEEE Symposium on Security and Privacy*, pp. 191–205, IEEE, 2013.

[15] C. Maurice, N. L. Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering intel last-level cache complex addressing using performance counters," in *Research in Attacks, Intrusions, and Defenses*, vol. 9404 of *Lecture Notes in Computer Science*, pp. 48–65, Springer International Publishing, 2015.

[16] J. Yu, A. Dutta, T. Jaeger, D. Kohlbrenner, and C. W. Fletcher, "Synchronization storage channels ({{{{{S2C}}}}}): Timer-less cache {Side-Channel} attacks on the apple m1 via hardware synchronization instructions," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 1973–1990, 2023.

[17] L. Gerlach, S. Schwarz, N. Faroß, and M. Schwarz, "Efficient and generic microarchitectural hash-function recovery," in *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 3661–3678, IEEE Computer Society, 2024.

[18] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 39–54, IEEE, 2019.

[19] Z. Xue, J. Han, and W. Song, "Ctpp: A fast and stealth algorithm for searching eviction sets on intel processors," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 151–163, 2023.

[20] T. Kessous and N. Gilboa, "Prune+ plumtree-finding eviction sets at scale," in *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 173–173, IEEE Computer Society, 2024.

[21] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Last-level cache side-channel attacks are feasible in the modern public cloud," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 582–600, 2024.

[22] Y. Chen, A. Hajiabadi, L. Pei, and T. E. Carlson, "Prefetchx: Cross-core cache-agnostic prefetcher-based side-channel attacks," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, (Edinburgh, United Kingdom), pp. 395–408, IEEE, 2024.

[23] S. Gueron, "Intel advanced encryption standard (aes) new instructions set," *Intel Corporation*, vol. 128, 2010.

[24] R. Könighofer, "A fast and cache-timing resistant implementation of the aes," in *Cryptographers' Track at the RSA Conference*, pp. 187–202, Springer, 2008.

[25] E. Käsper and P. Schwabe, "Faster and timing-attack resistant aes-gcm," in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 1–17, Springer, 2009.

[26] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "{RELOAD+ REFRESH}: Abusing cache replacement policies to perform stealthy cache attacks," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1967–1984, 2020.

[27] J. Lee, F. Sang, and T. Kim, "Prime+ retouch: When cache is locked and leaked," *arXiv preprint arXiv:2402.15425*, 2024.