Stone-Skimming: Attacking Big-Data Analytics Applications with Page Table Walk

Jiajun Luo[1†], Yu Jin[1†], Yufeng Gu[2], Jinyi Deng[3], Yang Hu[3], Shuwen Deng[1*]
[1]Department of Electronic Engineering, Tsinghua University [2]University of Michigan, Ann Arbor
[3]School of Integrated Circuits, Tsinghua University
{luo-jj23,jiny25}@mails.tsinghua.edu.cn,dengjinyi@mail.tsinghua.edu.cn
{hu_yang,shuwend}@tsinghua.edu.cn,yufenggu@umich.edu

Abstract

This paper proposes the Stone-Skimming microarchitectural attack, the first page table walk (PTW)-based attack targeting large-scale sensitive memory applications. Stone-Skimming leverages the correlation among multiple signals, which are derived from a victim application's original sensitive data access that triggers PTW, to establish novel side channel attacks. Using Stone-Skimming, we develop the first microarchitectural attack targeting big data applications in DNA sequence reconstruction. Multiple associated PTW entries generated by the victim are cross-validated, enabling robust DNA reconstruction where traditional data-based attacks fail. Moreover, we discover a new stateful side effect of PTW on prefetch instructions that can be exploited to break KASLR, even with advanced defenses such as KPTI on. We also introduce the early-instruction-fetch (early-IF) PTW attack capable of bypassing LFENCE and CPUID defense.

ACM Reference Format:
Jiajun Luo[1†], Yu Jin[1†], Yufeng Gu[2], Jinyi Deng[3], Yang Hu[3], Shuwen Deng[1*]. 2026. Stone-Skimming: Attacking Big-Data Analytics Applications with Page Table Walk. In *63rd ACM/IEEE Design Automation Conference (DAC '26), July 26–29, 2026, Long Beach, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770743.3804222

1 Introduction

The recent advent of big data analytics and healthcare technologies has led to a significant shift in the landscape of sensitive data. For example, DNA sequencing algorithms [33] determine the order of nucleotides (bases) in an individual's genome. The key feature of such applications is that the data are both large-scale and sensitive. These applications contain identification with more complex data structures like DNA sequences [4, 21]. Thus, the protection of sensitive data needs to go beyond traditional applications.

The large memory footprint of big-data applications opens up a unique attack opportunity for paging. Based on virtual memory, large-scale data accesses are converted to accesses to one or more *pages* of page tables. Contemporary x86 systems use 4 or 5-level paging structures to translate virtual addresses into physical addresses. When a page's corresponding translation is not found in the Translation Lookaside Buffer (TLB), a page table walk (PTW) is triggered to derive the physical address from the virtual address. During PTW, potentially multiple relevant page table entries (PTEs) are brought into the cache, which can create architectural side-channels.

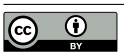
As a widely implemented and classical system mechanism, virtual memory or PTW has been the root cause of multiple attacks [14, 35–37, 39, 45]. However, for data leak attacks, these attacks focus on small-scale sensitive data, such as personally identifiable information or cryptographic key reconstruction [35, 36, 45], or require write operations [45] in real applications—impractical for big-data applications; or rely on attacker's [36], rather than victim's PTW; or require privilege permission [35]. AnC [14] and RevANC [37] reconstruct ASLR based on PTW-induced cache accesses. However, these designs focus on the last-level cache (LLC) and assumes an inclusive property between cache levels, which is not adopted in most current processors [2, 18]. Moreover, AnC uses the EVICT+TIME [28] method to exploit cache side channels [9, 11, 12, 16, 20, 28], requiring multiple identical accesses equal to the number of cache sets that cannot apply to real applications. Thus, none of them have explored the effectiveness of PTW in leaking big data applications [33].

In this paper, we introduce the *Stone-Skimming* attacks, the first PTW-based attacks that explore and target large-scale sensitive analytics applications. When multiple PTEs are brought into the cache, their presence and location within the cache can reveal information. By monitoring which cache sets are accessed during a PTW, attackers can effectively recover the virtual address information.

To make the attack effective, Stone-Skimming specifically focuses on exploiting ***correlation among multiple PTEs' cache signals*** triggered by victim application's ***original data access***, and uses such signals to establish novel side channel attacks. Using Stone-Skimming as the core mechanism, we develop the first novel μarch attack targeting DNA sequence reconstruction. This attack extracts the bases ('A','C','G','T') of the private and sensitive DNA samples. The key observation is that, when a data memory access involves the PTW, *multiple* associated PTW entries are brought into the cache together with the *single* accessed data item. This "$1 + x$" property (1 for the data access itself, x for the multiple PTW entries) provides the unique possibility of cross-validating accessed data and PTW entries to significantly enhance the signal and reduce the error rate to nearly zero. This property is crucial for attacks sensitive to errors, the case for DNA sequence reconstruction attack: a single error can crush all subsequent reconstructions, because sequencing algorithms rely on strict base-by-base dependency.

Beyond the first demonstration of DNA sequence reconstruction through PTW-induced side channels, this paper also reports the

[†]Co-first authors [*]Corresponding author.



DAC '26, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3804222

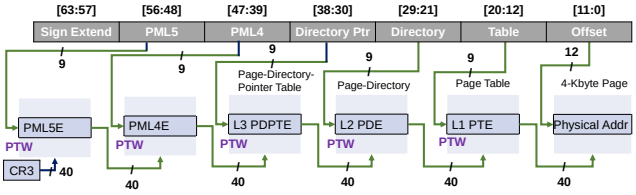


Figure 1: PTW process [3, 19]. In each level, from PML5E to L1 PTE, 9 bits of the virtual address are used to access the corresponding page table entry (PTE), referred to as a *PTW entry* for generality, which will be further cached into L1 data (L1D) cache once accessed. Ptr refers to pointer.

discovery of a new stateful side effect of PTW on the prefetch instruction that can be exploited to break KASLR even with advanced defenses such as KPTI [1], and a speculative early-instruction-fetch (IF) PTW attack that is not defended by CPUID, which [30] fails to bypass. The attack that breaks KASLR leverages a surprising and new observation that the PTWs triggered by the prefetch of mapped and unmapped addresses leave different traces and oscillation phenomena in the cache. The Early-IF instruction PTW attack leverages the fact that instruction-based PTW happens before execution serialization by either LFENCE or CPUID, circumventing existing mitigations.

In summary, this paper makes several key contributions:

- Construct a channel to monitor victim's PTW entries together with the victim's single data access in cache.
- Build, for the first time, a reliable side-channel attack on modern genome sequencing applications which can successfully reconstruct a long DNA read fragment.
- Discover a new characteristic of prefetch instructions' PTW that can be leveraged to break KASLR of systems equipped with state-of-the-art defense.
- Discover a new early-instruction-fetch PTW attack, that cannot be mitigated by `lfence` or `cpuid`.

2 Background & Related Work

2.1 PTW for x86

In modern computers, processors handle virtual addresses produced by the compiler, but the actual physical address is determined at runtime by the memory management unit (MMU). Memory is organized into pages to facilitate this process, with a typical page size of 4-KB (2^{12} bytes). An address is divided into two fields to convert from virtual to physical addresses: the byte address within a page (`11:0`) and the page address (the remaining high-order bits). The translation process involves replacing the virtual page address in the processor with the physical page address, according to a page table generated by the OS when creating a process.

Modern microprocessors adopt a multi-level page table organization. The key insight is to have a page that stores a set of consecutive PTEs at a certain level. Conceptually, the organization constitutes a tree structure, where a PTE at a higher level covers a larger range of virtual addresses, and the absence of the PTE saves the pages for storing PTEs of these addresses below the PTE in the tree.

Figure 1 demonstrates a 5-level hierarchical paging mechanism that translate 64-bit virtual address to 52-bit physical address in x86-64 [18] assuming 4KB page, which can store $2^{(12-3)} = 2^9 = 512$ 64-bit ($8 = 2^3$ Byte) addresses. If we have an L1 data cache with 64 sets and a 64-byte cache line, 12 bits combined are used to indicate the set index and block offset, the same as a byte address within a 4K page. This means that the cache line location is only relevant to the 9-bit address of a PTW entry within the page because the other 3 bits are used for addressing each byte in the 8B PTW entry, and together they add up to exactly 12 bits. Each cache line contains 8 PTW entries, so the set index is determined by the highest 6 bits of the 9-bit address of a PTW entry within the page table in each level in Figure 1: `56:51` of Page-Map Level-5 Entry (PML5E); `47:42` of Page-Map Level-4 Entry (PML4E); `38:33` of L3 Page-Directory-Pointer Table Entry (L3 PDPTE); `29:24` of L2 Page Directory Entry (L2 PDE); and `20:15` of L1 Page Table Entry (L1 PTE). The lowest 3 bits of the 9-bit address determine the PTW entry offset within the cache line. After mapping, `51:12` and the Offset (`11:0`) in the virtual address is concatenated to form the 52-bit physical address of the original 64-bit virtual address, concluding the translation and PTW.

The whole 5-level PTW in total generates and accesses five PTW entries, each of them will be brought to the L1 data cache, before the next PTW entry's address is generated and requested. It is the foundation of the side-channel of Stone-Skimming and will be discussed in detail in Section 3.

2.2 Related work

Besides AnC [14] and RevANC [37] discussed earlier, Binoculars attack [45] leverages the stateless contention and false dependencies between writes and reads to build attacks. The XLATE attack [36] uses traditional cache attacks such as PRIME+PROBE and replaces regular data in eviction sets with PTW entries to break the state-of-the-art defense. Stealthy page table-based attacks [35] attack enclave without encountering a page fault. Peek-a-walk [39] leverages PTW entries to leak an invalid pointer.

3 Stone-Skimming Side-Channel Attack

3.1 Motivation

Existing microarchitectural side-channel attacks focus on finding new channels [17, 29, 34, 42] and verifying their effectiveness with traditional applications, such as leaking cryptographic keys [1, 20]. In contrast, we seek to attack the popular and more complicated big-data applications. We specifically focus on cache side-channels [10, 12, 23, 43]. For big-data analytics applications, merely focusing on data accesses are hardly effective, because only the lower bits of the virtual address are reflected in the cache. To achieve our goal, it is crucial to explore PTW attacks, in which PTW entries cover a wide range of the virtual addresses of the memory access. More importantly, the correlation of multiple PTW entry signals provides the unique opportunity to reveal application-specific sensitive information with low noise.

3.2 Threat Model

We assume a victim process with access to large-scale sensitive memory and an attacker process trying to uncover the secret. The victim and attacker run on a multi-core x86 machine, with either Intel or AMD processor. The attacker is assumed to have knowledge of the victim's source code.

[1]Responsible Disclosure: We have disclosed our work to Intel and got acknowledged.

Table 1: Machine specifications in our evaluation.

CPU	Xeon Silver 4314	Xeon Gold 6226R	Xeon Gold 6438Y+	EPYC 7543
μarch	Ice Lake	Cascade Lake	Sapphire Rapids	Zen3
#Core	16	32	32	64
L1D Cache	64 sets, 12-way, 64-B line	64 sets, 8-way, 64-B line	64 sets, 12-way, 64-B line	64 sets, 8-way, 64-B line
OS	Ubuntu 18.04.6	CentOS-7	Ubuntu 22.04	Ubuntu 22.04.2
Mem (GB)	256	128	512	256

The attacker can call the victim library's Application Programming Interface (API) but does not have the authority to access any data encapsulated by the API.

The attacker can use the user-level RDTSC instruction for timing measurement, and a time counting thread synchronized with the attacker program that also does not require privilege level access.

3.3 PTW-induced Attack Primitive

Reveal the vulnerability. As illustrated in Figure 1, the PTW procedure based on the 5-level paging accesses *five* PTW entries from each level before generating the physical address of the data and accessing it. The PTW entries are brought to the cache hierarchy in the same way as the data, and thus the presence or absence of a PTW entry in cache leads to the timing-side channel, which can be used to detect the location of the cache line that contains a PTW entry. Further, the location of the cache line can reveal part of the physical address of the PTW entry, which is determined by certain bits in the virtual address of the originally requested data. In a nutshell, the virtual address of data access can be partially recovered by the locations of PTW entries during PTW.

Attack Validation. We have tested through 4 different machines as shown in Table 1. We maintain two entities. One is the victim that holds large-scale sensitive memory. The other one is the attacker that deploys the attack and observes timing to reconstruct the sensitive information. Our attack can be divided into four steps.

- *Preliminary preparation*: The goal is to evict all targeted TLB entries so that the memory access will trigger PTW. We use a series of memory accesses to build an eviction set of pages to evict targeted TLB entries. It is noteworthy that the size of the eviction set is also large enough to eliminate the effects of page table caches.
- *Attacker setup*: In this step, the attacker fills targeted α ($1 \leq \alpha \leq 64$ according to different types of attack) number of L1D cache sets with its own data.
- *Victim encoding*: The victim performs a large-scale sensitive memory access. The missing TLB entry will trigger a PTW, in which the accessed PTW entries and data are placed into the cache, replacing the data primed in the previous step.
- *Attacker decoding*: the attacker will probe the corresponding set with the same data primed in the second step and measure its timing to monitor potential data replacement caused by the victim in the encoding step.

Figure 2 illustrates the results on the timing side channel due to PTW. Higher count indicates some entries in the corresponding cache set are evicted by the victim. We carry out the measurement 10,000 times and remove outliers. We can detect 5 signals (1 for data and 4 for PTW entries) when the victim performs an access to virtual address `0x00007f299cc3c0c0`. These signals indicate not only data access in cache set 3 as determined by bits 11:6, but also PTW entries' access traces.

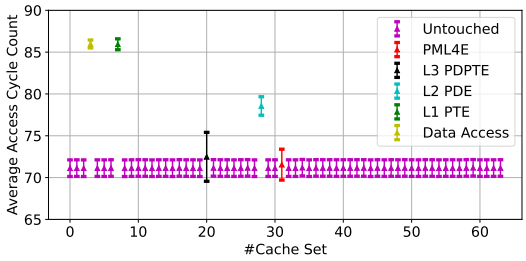


Figure 2: Average timing signals when the victim accesses virtual address `0x00007f299cc3c0c0`. Bits `47:42` of the address equal 31 in decimal which is the set of PML4E, L3 PDPTE's set number is 20 from bits `38:33`, L2 PDE's set number is 28 from bits `29:24`, L1 PTE's set number is 7 from bits `20:15`, and data's set number is 3 from bits `11:6`.

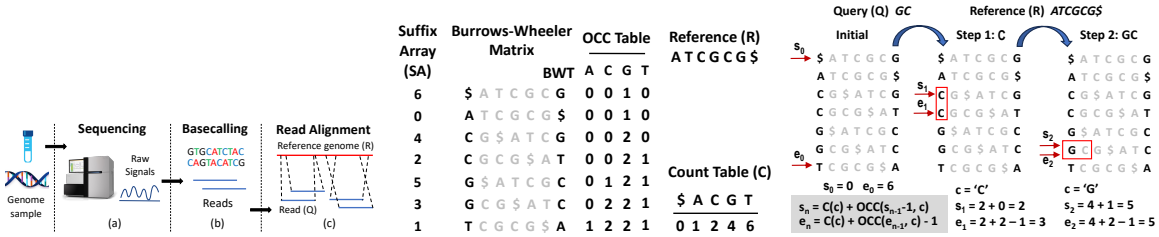
4 Stone-Skimming on Genome Sequencing

4.1 Genome Sequencing Algorithms

As shown in Figure 3a, genome sequencing starts with raw signals from the genome sample. Sequencers read the raw signals by fragmenting it into billions of short substrings, called *reads*. This process is called *basecalling*. The reads typically have 100-150 base pairs [26]. After that, the sample genome is reconstructed by aligning the reads with an existing reference genome sequence, known as *read alignment*, which performs approximate string matching. This step produces multiple positions in the reference that may match the read exactly. Then, a dynamic algorithm is performed to compare the similarity between the read and these positions and determine which position has minimal difference [33]. FM-Index Search (FMI) is a common indexing algorithm used in *read alignment* [25, 38]. The backward search based on FMI can efficiently find the number of occurrences for a pattern and locate their positions.

FMI search process. Figure 3b depicts the FMI for an example reference (R) that contains 6 bases. Real-world length of R is very large, in the order of 2^{30}. FMI has four main components. *Count table (C)* stores the number of characters that are smaller than the current character in the lexicographically sorted reference, e.g., "$ACCGGT" for the example with "$" having the smallest value. In its *count table*, the count for "G" is 4 since four characters "$ACC" are smaller. The *Burrows-Wheeler Matrix (BWM)* contains all cyclic shifts of R. Examples are "TCGCG$A" and "G$ATCGC". These shifted sequences are sorted in lexicographic order to form the BWM. Based on BWM, *Burrows-Wheeler Transform (BWT)* is simply BWM's last column, and *suffix array (SA)* is the distance of "left cyclic shifts" from R. For example, the SA of the first row is 6, because R is cyclically left shifted 6 times to get this sequence. Finally, each row of *Occurrance table (OCC)* indicates the number of occurrences for each character up to the row in BWT. In the example, we can see that the fourth row of OCC contains $[0, 0, 2, 1]$ because there are 2 'G's, 1 'T', and no 'A' or 'C' in the first four rows of the BWT.

The backward search algorithm based on FMI is an iterative procedure. For a read with length L, the algorithm executes for L steps, updating the start-end pair (s, e). We denote (s, e) after step i as (s_i, e_i). The calculation of (s_i, e_i) involves accessing C and OCC



(a) Genome Sequencing Pipeline **(b) FMI for reference sequence (R).** **(c) Backward search using the FMI.**

Figure 3: Genome sequencing process.

described earlier and the base, denoted by c or c_i. The reads are iterated in reverse order. The recursive formulas to calculate (s_i, e_i) are: $s_i = C(c) + OCC(s_{i-1} - 1, c)$ and $e_i = C(c) + OCC(e_{i-1}, c) - 1$. After L steps are executed, s_L and e_L converge to a range, which gives all possible locations of the read in the reference sequence R.

We explain the algorithm using an example in Figure 3c with a short query Q="GC". Before step 1, $(s_0, e_0) = (0, 6)$. The first character in reverse order is 'C', based on the formulas, $s_1 = C('C') + OCC(-1, 'C')$. The "-1" row of OCC is defined as all zeros. Thus, we have $s_1 = 2$. Similarly, we calculate $e_1 = 2 + 2 - 1 = 3$. The same computation is performed in step 2, after which s_2 and e_2 converge to 5, which indicates the location in R where "GC" is matched.

4.2 DNA Sequence Reconstruction Attack

Threat Model. The bases of the reads are secrets. They are maintained inside the library and can only be indirectly accessed by the library function. The function caller has no permission to access the read samples or the matching positions. He can be a third party who provides the reference template and an FMI algorithm running environment but is not authorized to access those matching reads.

Root cause. Based on the threat model, reference R and its relevant data structures OCC and C are public. And the FMI search involves the access to OCC, a very large table whose size is proportional to the length of R (in the order of 2^{30}). Each step i accesses two rows of OCC to compute s_i and e_i, which will bring PTW entries into the L1D cache. In principle, our side channel can reveal the index of the rows. However, the *secret* our attack aims to extract is the character of each position (c_i) in the read processed in each step i. Thus, we need to establish the relation between the OCC row access indices and the character in the read.

From the recursive formulas, we see that the index of OCC rows accessed in step i are determined by (s_{i-1}, e_{i-1}) but does *not* depend on c_i, which only decides the value to access within a row. (s_{i-1}, e_{i-1}) is further determined by both (s_{i-2}, e_{i-2}) and c_{i-1}. Thus, by transitivity of dependence, the index of OCC rows accessed in step i depends on c_{i-1}. We have established the relation. The attack is challenging as the successful recovery in step i depends on the success of previous recoveries, requiring zero fault tolerance. Because we need to know (s_{i-2}, e_{i-2}) so that (s_{i-1}, e_{i-1}) is solely dependent on c_{i-1}. Fortunately, at the start, (s_1, e_1) is solely dependent on c_1 because the "-1" row of OCC is zero.

Attack method. The next question is how to extract the secret. Fortunately, for each c_{i-1}, there are only four choices: "A", "C", "G", and "T". We can regard each choice as a *hypothesis* and compute its respective OCC row index in step i, which corresponds to a set of L1D cache sets, denoted as $HSet_{i,h}$, where $h \in \{A, C, G, T\}$, that

```
1. #include "genomic_bench.h"
2. fmi_search(...);     // Genome library call
3. for (size_t i = read_length - 2; i < read_length; i--) {
4.     load(TLB_eviction_sets);
5.     prime();
6.     fmi_search(...); // Genome library call
7.     size_t *timings = probe();
8.     for (size_t j = 0; j < 4; j++)
9.         for (size_t k = 0; k < 3; k++)
10.            if (timings[sets[j][k]] > threshold)
11.                // Each row of 2D array have three accesses
12.                // for L2 PDE, L1 PTE, and data, respectively
13.                // map to 1 out of 4 of the value of base[i + 1]
14.                scores[j]++;
15.    reconstructed_bases[i + 1] = get_max_id(scores);
16. }
```

```
1. fmi_search (...) {
2.     ...
3.     GET_OCC(size_t s, base_t c, ...); // OCC table access
4.     // s is one element of the matching pair
5.     // c is the secret genome sequence base
```

Figure 4: Attacker program on Genomicbenches. *fmi_search* is a public library call accessing secrets internally.

contains PTW entries and data entry for the index's address in step i. These cache sets can be identified with our side channel.

To choose a hypothesis, we compute the *score* for each one. Specifically, after step i, we can obtain a set of L1D cache sets filled, i.e., sets accessed during the iteration of s_i and e_i, and denote as $WSet_i$. Then we compare the intersect of each $HSet_{i,h}$ and $WSet_i$, and the size of the intersected set is added to $score_h$. Due to the system noise, i.e., cache filled by other activities, $WSet_i$ is always larger than any of $HSet_{i,h}$. Thus, a higher score indicates a higher likelihood of the hypothesis h being correct. In our attack, we run the attack 9 times to mitigate noise. We also need to decide the size of $HSet_{i,h}$. $HSet_{i,h}$ for each OCC row access may contain PTW entries of up to 4 levels and 1 data entry. However, we find that only using PTW entries of 2 levels (PDE and PTE) and the data entry is already sufficient to distinguish different hypotheses. Moreover, while computing e_n also accesses OCC, we find it enough to just focus on s_n computation and treat the other as noise. Based on the two considerations, the size of $HSet_{i,h}$ is at most 3. With 9 runs in total, the maximum score for a hypothesis is $9 \times 3 = 27$.

For a read sample of length L, $L - 1$ bases can be reconstructed, except base c_{151}, because s_{151} decided by it is not used for the OCC table lookup. However, with typically 30-50X coverage of reads [6], the single absent base can still be acquired from other reads.

Figure 4 highlights the method to attack Genomicsbench [33] leveraging Stone-Skimming based on the insights discussed earlier. The backward search function *fmi_search* is provided as a library call which is akin to cryptographic encryption functions. The first step of backward search is done outside the loop because it involves no OCC table access (line 2), and then the attacker runs the remaining 150 rounds for the 151-base long read. In each round,

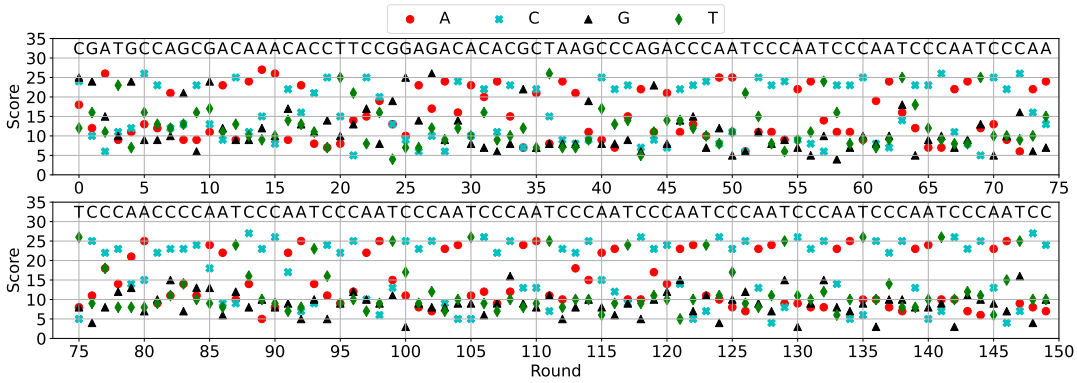


Figure 5: Attack results for a 151-base-long read. The scores for each hypothesis in each step are shown with the ground truth. The extracted bases by the score are all consistent with the ground truth (with a special case in the first round).

Stone-Skimming evicts TLB entries, primes the L1D cache, calls *fmi_search* forwarding a step, and probes to obtain the timings. After step i, the collected timings are used to decide all the accessed cache sets $WSet_i$, which are compared with the each $HSet_{i,h}$. The *score* is updated accordingly. Finally, the hypothesis (base) with the highest score is chosen to be the recovered base (secret).

Attack evaluation. As shown in Figure 5, by completing the 150 rounds, we can reconstruct 150 bases of a read. It shows the score of each hypothesis for each step. Note that in the first round, there are two hypotheses with equally high scores. This is caused by a characteristic in step 1 that we have $e_1 = C(\text{`}C\text{'}) + OCC(e_0, \text{`}C\text{'}) - 1$ which is precisely equal to $C(\text{`}G\text{'}) - 1$, exactly the index for hypothesis 'G'. This is because $e_0$ resolves to the last line of $OCC$ and thereby $OCC(e_0, \text{`}C\text{'})$ indicates the total quantity of C in the reference. Therefore, $C(\text{`}C\text{'}) + OCC(e_0, \text{`}C\text{'}) = C(\text{`}G\text{'})$ and $e_1 = C(\text{`}G\text{'}) - 1$. This characteristic only holds in step 1 and influences the first round of the following attack. Therefore, if the actual base is 'C', its $HSet_{2,\text{`}C\text{'}}$ is accessed due to $OCC(s_1 - 1, c)$ but $HSet_{2,\text{`}G\text{'}}$ is also accessed due to $OCC(e_1, c)$. With this in mind, we know that the hypothesis with a smaller lexicographical value is the secret.

Comparison with attacks without utilizing PTW. We also conduct the experiment using only data entry without PTW entries, where the size of $HSet_{i,h}$ is 1. This attack can only reconstruct the *first* 4 *bases*. Since the attack relies on the correctness of previous rounds to ensure subsequent success, the attack fails. The reason is that is has only one signal which is easily overwhelmed by noise. This problem can not be mitigated by repeating because the attacker is not allowed to control the victim to repeatedly execute one step.

5 Stone-Skimming on KASLR with prefetching

The KASLR [15] protects the kernel from code reuse attacks such as return-oriented programming (ROP) [31] with minimal performance overhead and has been widely adopted in modern operating systems [7]. In Linux's x86-64 implementation of KASLR with 5-level page tables, bits 29 to 21 are randomized, providing 9 bits of entropy (512 choices), which are located in the L2 PDE index.

Threat model and setting. We consider an unprivileged attacker who can execute arbitrary instructions on a KASLR-enabled local machine. The attacker's goal is to determine the actual addresses of the kernel image to conduct code reuse attacks. They know the CPU model and the constant offsets of kernel functions. The attack

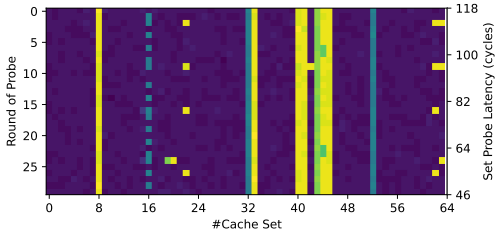


Figure 6: Probing latency after prefetching kernel space (`0xffffffff90a00000`). L2 PDE's set is 16 by bits `29:24`, where oscillation is observed, indicating the address is unmapped.

can be divided into four stages: (1) evict the TLB; (2) prime the L1D cache; (3) execute prefetch instruction target to the kernel address, which will trigger PTW because TLB has been evicted; (4) probe the L1D cache and observe the PTW traces.

New Observation to Break KASLR. We notice that prefetch a kernel address after evicting its TLB triggers a PTW, which leaves a PTW trace in the L1D cache on all tested CPUs. The surprising observation is that, on Intel Xeon processors, if the virtual address is physically unmapped, timings observed in its L2 PDE's set oscillate between large and small as we increase the number of test rounds. The oscillating behavior can be captured by the mean and standard deviation of the probed latency of L2 PDE's co-located set. As shown in Figure 6, prefetching virtual addresses with 9-bit entropy, which corresponds to 512 distinct choices, results in significant and consecutive oscillations in the timing for the cache sets for L2 PDE if the kernel address is unmapped in the KASLR environment. Specifically, the `prefetchnta` and `prefetcht1` instructions do not consistently evict data mapped to the same cache sets, causing an oscillating pattern, as seen in cache set #16 in Figure 6. In contrast, when prefetching mapped kernel addresses, the PTW maintains a stable L2 PDE's set eviction pattern, e.g., set #32. The mean and standard deviation (std) of the probing times are more consistent for mapped kernel addresses, with minimal noise affecting the distribution. By analyzing the mean probing values for the 512 choices, we can successfully break Linux's KASLR within 10 seconds.

Against the state-of-art defense. The ideal KPTI should completely separate the user and kernel page tables. But in reality, as the user needs to have kernel trampolines for proper OS functionality [24], there are still some kernel pages left on the user page table. Our experiment shows that even with KPTI enabled and FLARE [8]

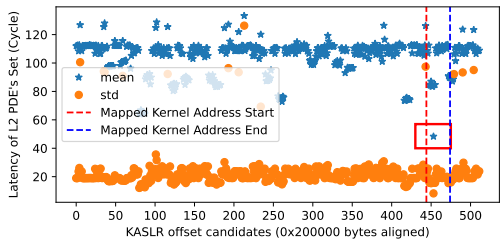


Figure 7: Breaking KASLR with KPTI enabled. In this case, the 9bit entropy is 444, which means the kernel image address starts at `0xffffffffb7800000`.

```
1. #define SET(set) \
2. (((uint8_t *)l1i->memory) + PAGE_SIZE * PAGE_OFFSET * (set))
3. char *key = "ae1lpq7355608dsav31jnnk"
4. void fc_victim (size_t x, l1i_struct l1i) {
5.     asm volatile("lfence"); // barrier
6.     asm volatile("cpuid"); // another barrier
7.     (*((fptr) SET(key[x]))) ();
8. }
```

Figure 8: Stone-Skimming early-IF attack's victim gadget.

deployed, there is still a constant address with a distinctive PTW signal. When we examine the actual kernel address, we notice that this signal is always located at the 0x1200000 offset of the kernel (10^{th} mapping in the L2 PDE of the kernel image). This means we can exploit it to leak the kernel address. As demonstrated in Figure 7, the PTW of prefetching to that address consistently does not evict cache sets co-located with the L2 PDE's set. Remarkably, enabling KPTI even helps reduce noise, making it easier to break KASLR. To measure the attack accuracy with KPTI enabled, we rebooted the system eleven times and mounted the attack five times per boot. We achieved 87.9% accuracy on the Intel Xeon Sapphire Rapids Server. We also conducted experiments on the Intel Xeon Ice Lake Server and successfully broke KASLR.

6 Early-Instruction-Fetch (Early-IF) Attack

Threat Model. We assume an intra-domain attack scenario, where the attacker resides within the same thread, for instance, as sandboxed code wherein the disclosure gadget is executed.

Root Cause. The key observation is that the instruction-based speculative PTW happens before execution serialization due to either LFENCE or CPUID, which is attributed to instruction predictor [41]. Note the instruction PTW entries are also cached by L1D.

Attack preparation. Our proof-of-concept victim is shown in Figure 8 which leaks each character of the *key*. To enable the attack, we create 64 series of JUMP instructions, each series include JUMP instructions jumping to the next instruction in the series. We ensure the L1 PTEs of all JUMP instructions from the same series are mapped to the same L1D cache set, while different series are mapped to distinct sets. In Figure 8, the JUMP address is encoded using a function pointer *SET* defined in line 1-2.

Attack process. For each character *key[x]*, we first evict the TLB entries and prime the entire L1D. Then, *key[x]* is passed to *SET*, which will trigger the execution of JUMP instructions in a series, whose PTWs evict cache lines from some sets. The character can be extracted by probing the cache sets.

Beyond the basic process, Figure 8 shows a scenario where the JUMP instructions in line 7 is *speculatively* fetched, and the attack cannot be prevented by LFENCE or CPUID before it (line 5 & 6). To achieve that, we first repeatedly execute the JUMP with *key[x]*. The purpose is to train the branch predictor with a specific target. This step is not shown in Figure 8. Then, at line 7, we call the JUMP with any index *x'*. However, due to the branch prediction, the speculative fetch still resolves to *SET*(*key[x]*). In this case, the speculative PTW can leak *x* even if we intend to execute *SET*(*key[x']*).

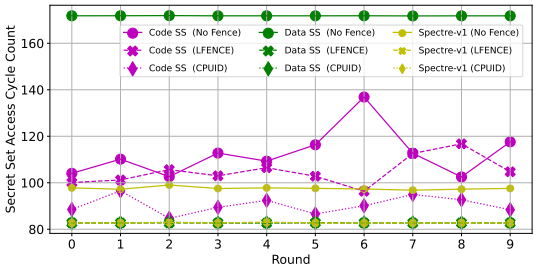


Figure 9: Set timings for Stone-Skimming (SS) early-IF attack.

Attack evaluation. We measure the time spent on probing the secret-locating set in three different settings: CPUID, LFENCE, and no fence, under three attacks: Code Stone-Skimming, Data Stone-Skimming and PRIME+PROBE-version Spectre-v1 [22]. Figure 9 shows the cycle counts for probing, where the value 80 means all cache hit. Values larger than 80 are considered signals. We can observe that if there is no fence, all three attacks have valid signals. Both CPUID and LFENCE can defend Spectre-v1 and Data Stone-Skimming, because the data access is during the execution stage in pipeline, blocked by the serialization instructions. For Code Stone-Skimming, neither LFENCE nor CPUID can provide ideal protection by eliminating the signal. Real-world gadgets, e.g., those found by SpecFuzz [27], can be effective. One example resides in the OpenSSL function *asn1_ex_i2c* [44] with a compact switch statement. It results in a jump table at compilation, facilitating our attack to leak history jump destinations.

7 Potential Defense

Stone-Skimming cannot be mitigated by partitioning schemes [5] and page coloring [13] targeting last-level cache [23]. Proposals for partitioning L1 caches [40] are also ineffective if the partitioning is based on process identifiers (PIDs), since PTW entries from the system are not assigned with PIDs. Direct mitigations by disabling caching of PTW entries will substantially degrade performance.

On the other hand, since the key feature used by Stone-Skimming for genomics is the multilevel PTW, it can be mitigated by hashing-based PTW design such as Cuckoo Page Tables [32]. However, it requires a vast amount of design and validation effort.

8 Conclusion

We introduce Stone-Skimming attacks, exploiting victims' PTWs for big-data analytics applications. We use correlations among signals to reconstruct DNA. We discover a new stateful side effect of PTW on prefetch instructions that can break KASLR with KPTI enabled. We also develop a speculative attack capable of bypassing CPUID.

Acknowledgments

This work was generously supported by NSFC (No. U24A6009, 62572265), the National Key Research and Development Program of China under Grant (2024YFB4405400), Beijing Municipal Science and Technology Project (Nos. Z241100004224028), Beijing Natural Science Foundation (L247013).

References

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*. Springer, 225–242.

[2] AMD. 2025. AMD Tech Docs: Programmer's Manual.

[3] AMD AMD. 64. Architecture Programmer's Manual: Volume 2: System Programming. *AMD Pub* 24593 (64).

[4] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure:{SGX} cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*.

[5] Improving Real-Time Performance by Utilizing. 2015. Cache Allocation Technology. *Intel Corporation, Apr* (2015).

[6] Marta Byrska-Bishop, Uday S Evani, Xuefang Zhao, Anna O Basile, Haley J Abel, Allison A Regier, André Corvelo, Wayne E Clarke, Rajeeva Musunuri, Kshithija Nagulapalli, et al. 2022. High-coverage whole-genome sequencing of the expanded 1000 Genomes Project cohort including 602 trios. *Cell* 185, 18 (2022), 3426–3440.

[7] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. 2020. KASLR: Break It, Fix It, Repeat. In *15th ACM Asia Conference on Computer and Communications Security (ASIACCS)*.

[8] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. 2020. KASLR: Break It, Fix It, Repeat. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (Taipei, Taiwan) *(ASIA CCS '20)*. Association for Computing Machinery, New York, NY, USA, 481–493. doi:10.1145/3320269.3384747

[9] Shuwen Deng, Nikolay Matyunin, Wenjie Xiong, Stefan Katzenbeisser, and Jakub Szefer. 2022. Evaluation of Cache Attacks on Arm Processors and Secure Caches . *IEEE Trans. Comput.* 71, 09 (Sept. 2022), 2248–2262. doi:10.1109/TC.2021.3126150

[10] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2018. Cache Timing Side-Channel Vulnerability Checking With Computation Tree Logic. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2.

[11] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2019. Analysis of Secure Caches Using a Three-Step Model for Timing-Based Attacks. *Journal of Hardware and Systems Security* 3, 4 (December 2019), 397–425.

[12] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2020. A Benchmark Suite for Evaluating Caches' Vulnerability to Timing Attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 683–697. doi:10.1145/3373376.3378510

[13] Matthew Dillon. 2001. Design elements of the FreeBSD VM system. *Daemon News, Jan* (2001).

[14] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU.. In *NDSS*, Vol. 17. 26.

[15] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems*, Eric Bodden, Mathias Payer, and Elias Athanasopoulos (Eds.). Springer International Publishing, Cham, 161–176.

[16] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy*. 490–505. doi:10.1109/SP.2011.22

[17] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. 2022. Adversarial prefetch: New cross-core cache side channel attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1458–1473.

[18] Intel. 2025. Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3.

[19] Intel Intel. 64. IA-32 Architectures Software Developer's Manual. *Volume 3A: System Programming Guide, Part* 1, 64 (64), 64.

[20] Yu Jin, Minghong Sun, Dongsheng Wang, Pengfei Qiu, Yinqian Zhang, and Shuwen Deng. 2025. GhostCache: Timer- and Counter-Free Cache Attacks Exploiting Weak Coherence on RISC-V and ARM Chips. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei) *(CCS '25)*. Association for Computing Machinery, New York, NY, USA, 3795–3809. doi:10.1145/3719027.3744833

[21] Sowoong Kim, Myeonggyun Han, and Woongki Baek. 2022. DPrime+ DAbort: A High-Precision and Timer-Free Directory-Based Side-Channel Attack in Non-Inclusive Cache Hierarchies using Intel TSX. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 67–81.

[22] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting Speculative Execution. In *Symposium on Security and Privacy (S&P)*. 1–19.

[23] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. 605–622. doi:10.1109/SP.2015.43

[24] William Liu, Joseph Ravichandran, and Mengjia Yan. 2023. EntryBleed: A Universal KASLR Bypass against KPTI on Linux. In *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy* (Toronto, Canada) *(HASP '23)*. Association for Computing Machinery, New York, NY, USA, 10–18. doi:10.1145/3623652.3623669

[25] Ruibang Luo, Thomas Wong, Jianqiao Zhu, Chi-Man Liu, Xiaoqian Zhu, Edward Wu, Lap-Kei Lee, Haoxiang Lin, Wenjuan Zhu, David W Cheung, et al. 2013. SOAP3-dp: fast, accurate and sensitive GPU-based short read aligner. *PloS one* 8, 5 (2013), e65632.

[26] W Richard McCombie, John D McPherson, and Elaine R Mardis. 2019. Next-generation sequencing technologies. *Cold Spring Harbor perspectives in medicine* 9, 11 (2019), a036798. http://perspectivesinmedicine.cshlp.org/content/9/11/a036798.short

[27] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2020. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1481–1498. https://www.usenix.org/conference/usenixsecurity20/presentation/oleksenko

[28] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology* (San Jose, CA) *(CT-RSA'06)*. Springer-Verlag, Berlin, Heidelberg, 1–20. doi:10.1007/11605805_1

[29] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. 2021. Lord of the Ring (s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. *arXiv preprint arXiv:2103.03443* (2021).

[30] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. 2021. I See Dead μops: Leaking Secrets via Intel/AMD Micro-Op Caches. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 361–374. doi:10.1109/ISCA52012.2021.00036

[31] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *14th ACM Conference on Computer and Communications Security (CCS)*.

[32] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1093–1108.

[33] Arun Subramaniyan, Yufeng Gu, Timothy Dunn, Somnath Paul, Md Vasimuddin, Sanchit Misra, David Blaauw, Satish Narayanasamy, and Reetuparna Das. 2021. Genomicsbench: A benchmark suite for genomics. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 1–12.

[34] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. 2022. {TLB; DR}: Enhancing {TLB-based} Attacks with {TLB} Desynchronized Reverse Engineering. In *31st USENIX Security Symposium (USENIX Security 22)*. 989–1007.

[35] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling your secrets without page faults: Stealthy page {Table-Based} attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*. 1041–1056.

[36] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security Symposium*. 937–954.

[37] Stephan Van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. 2017. RevAnC: A framework for reverse engineering hardware page table caches. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.

[38] Md Vasimuddin, Sanchit Misra, Heng Li, and Srinivas Aluru. 2019. Efficient architecture-aware acceleration of BWA-MEM for multicore systems. In *2019 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 314–324.

[39] Alan Wang, Boru Chen, Yingchen Wang, Christopher W. Fletcher, Daniel Genkin, David Kohlbrenner, and Riccardo Paccagnella. 2025. Peek-a-Walk: Leaking Secrets via Page Walk Side Channels. In *2025 IEEE Symposium on Security and Privacy (SP)*. 3534–3548. doi:10.1109/SP61157.2025.00023

[40] Zhenghong Wang and Ruby B Lee. 2007. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 494–505.

[41] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. 2023. Phantom: Exploiting Decoder-detectable Mispredictions. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 49–61.

[42] Wenjie Xiong and Jakub Szefer. 2020. Leaking information through cache LRU states. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 139–152.

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

[44] Eric A Young, Tim J Hudson, and Ralf S Engelschall. 2001. OpenSSL. *World Wide Web, http://www. openssl. org/, Last visited* 9 (2001).

[45] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. 2022. Binoculars:{Contention-Based}{Side-Channel} Attacks Exploiting the Page Walker. In *31st USENIX Security Symposium (USENIX Security 22)*. 699–716.