# I See Dead $\mu$ops: Leaking Secrets via Intel/AMD Micro-Op Caches

Xida Ren
*University of Virginia*
renxida@virginia.edu

Logan Moody
*University of Virginia*
lgm4xn@virginia.edu

Mohammadkazem Taram
*University of California, San Diego*
mtaram@cs.ucsd.edu

Matthew Jordan
*University of Virginia*
mrj3dd@virginia.edu

Dean M. Tullsen
*University of California, San Diego*
tullsen@cs.ucsd.edu

Ashish Venkat
*University of Virginia*
venkat@virginia.edu

***Abstract*—Modern Intel, AMD, and ARM processors translate complex instructions into simpler internal micro-ops that are then cached in a dedicated on-chip structure called the *micro-op cache*. This work presents an in-depth characterization study of the micro-op cache, reverse-engineering many undocumented features, and further describes attacks that exploit the micro-op cache as a timing channel to transmit secret information. In particular, this paper describes three attacks – (1) a same thread cross-domain attack that leaks secrets across the user-kernel boundary, (2) a cross-SMT thread attack that transmits secrets across two SMT threads via the micro-op cache, and (3) transient execution attacks that have the ability to leak an unauthorized secret accessed along a misspeculated path, even before the transient instruction is dispatched to execution, breaking several existing invisible speculation and fencing-based solutions that mitigate Spectre.**

## I. Introduction

Modern processors feature complex microarchitectural structures that are carefully tuned to maximize performance. However, these structures are often characterized by observable timing effects (e.g., a cache hit or a miss) that can be exploited to covertly transmit secret information, bypassing sandboxes and traditional privilege boundaries. Although numerous side-channel attacks [1]–[10] have been proposed in the literature, the recent influx of Spectre and related attacks [11]–[19] has further highlighted the extent of this threat, by showing how even transiently accessed secrets can be transmitted via microarchitectural channels. This work exposes a new timing channel that manifests as a result of an integral performance enhancement in modern Intel/AMD processors – *the micro-op cache*.

The x86 ISA implements a variety of complex instructions that are internally broken down into RISC-like micro-ops at the front-end of the processor pipeline to facilitate a simpler backend. These micro-ops are then cached in a small dedicated on-chip buffer called the micro-op cache. This feature is both a performance and a power optimization that allows the instruction fetch engine to stream decoded micro-ops directly from the micro-op cache when a translation is available, turning off the rest of the decode pipeline until a micro-op cache miss occurs. In the event of a miss, the decode pipeline is re-activated to perform the internal CISC-to-RISC translation; the resulting translation penalty is a function of the complexity of the x86 instruction, making the micro-op cache a ripe candidate for implementing a timing channel.

However, in contrast to the rest of the on-chip caches, the micro-op cache has a very different organization and design; as a result, conventional cache side-channel techniques do not directly apply – we cite several examples. First, software is restricted from directly indexing into the micro-op cache, let alone access its contents, regardless of the privilege level. Second, unlike traditional data and instruction caches, the micro-op cache is designed to operate as a stream buffer, allowing the fetch engine to sequentially stream decoded micro-ops spanning multiple ways within the same cache set (see Section II for more details). Third, the number of micro-ops within each cache line can vary drastically based on various x86 nuances such as prefixes, opcodes, size of immediate values, and fusibility of adjacent micro-ops. Finally, the proprietary nature of the micro-op ISA has resulted in minimal documentation of important design details of the micro-op cache (such as its replacement policies); as a result, their security implications have been largely ignored.

In this paper, we present an in-depth characterization of the micro-op cache that not only allows us to confirm our understanding of the few features documented in Intel/AMD manuals, but further throws light on several undocumented features such as its partitioning and replacement policies, in both single-threaded and multi-threaded settings.

By leveraging this knowledge of the behavior of the micro-op cache, we then propose a principled framework for automatically generating high-bandwidth micro-op cache-based timing channel exploits in three primary settings – (a) across code regions within the same thread, but operating at different privilege levels, (b) across different co-located threads running simultaneously on different SMT contexts (logical cores) within the same physical core, and (c) two transient execution attack variants that leverage the micro-op cache to leak secrets, bypassing several existing hardware and software-based mitigations, including Intel's recommended LFENCE.

The micro-op cache as a side channel has several dangerous implications. First, it bypasses all techniques that mitigate caches as side channels. Second, these attacks are not detected by any existing attack or malware profile. Third, because the micro-op cache sits at the front of the pipeline, well before execution, certain defenses that mitigate Spectre and other transient execution attacks by restricting speculative cache updates still remain vulnerable to micro-op cache attacks.

Most existing invisible speculation and fencing-based solutions focus on hiding the unintended vulnerable side-effects of speculative execution that occur at the back end of the processor pipeline, rather than inhibiting the source of speculation at the front-end. That makes them vulnerable to the attack we describe, which discloses speculatively accessed secrets through a front-end side channel, before a transient instruction has the opportunity to get dispatched for execution. This eludes a whole suite of existing defenses [20]–[32]. Furthermore, due to the relatively small size of the micro-op cache, our attack is significantly faster than existing Spectre variants that rely on priming and probing several cache sets to transmit secret information, and is considerably more stealthy, as it uses the micro-op cache as its sole disclosure primitive, introducing fewer data/instruction cache accesses, let alone misses.

In summary, we make the following major contributions.

- We present an in-depth characterization of the micro-op cache featured in Intel and AMD processors, reverse engineering several undocumented features, including its replacement and partitioning policies.
- We propose mechanisms to automatically generate exploit code that leak secrets by leveraging the micro-op cache as a timing channel.
- We describe and evaluate four attack variants that exploit the novel micro-op cache vulnerability – (a) cross-domain same address-space attack, (b) cross-SMT thread attack, and (c) two transient execution attack variants.
- We comment on the extensibility of existing cache side-channel and transient execution attack mitigations to address the vulnerability we expose, and further suggest potential attack detection and defense mechanisms.

## II. Background and Related Work

This section provides relevant background on the x86 front-end and its salient components, including the micro-op cache and other front-end features, in reference to Intel Skylake and AMD Zen microarchitectures. We also briefly discuss related research on side-channel and transient execution attacks.

### A. The x86 Decode Pipeline

Figure 1 shows the decoding process in an x86 processor. In the absence of any optimizations, the instruction fetch unit reads instruction bytes corresponding to a 16-byte code region from the L1 instruction cache into a small fetch buffer every cycle. This is then read by the predecoder that extracts individual x86 instructions, also called *macro-ops*, into a dedicated FIFO structure called the *macro-op queue*, which in the Skylake microarchitecture consists of 50 entries. The predecoding process is highly sensitive to composition of the fetch buffer (number and type of macro-ops it constitutes), and in some cases it may also incur an additional penalty of three to six cycles when a length-changing prefix (LCP) is encountered, as the decoded instruction length is different from the default length. As the macro-ops are decoded and placed in the macro-op queue, particular pairs of consecutive macro-op entries may be fused together into a single macro-op to save decode bandwidth.

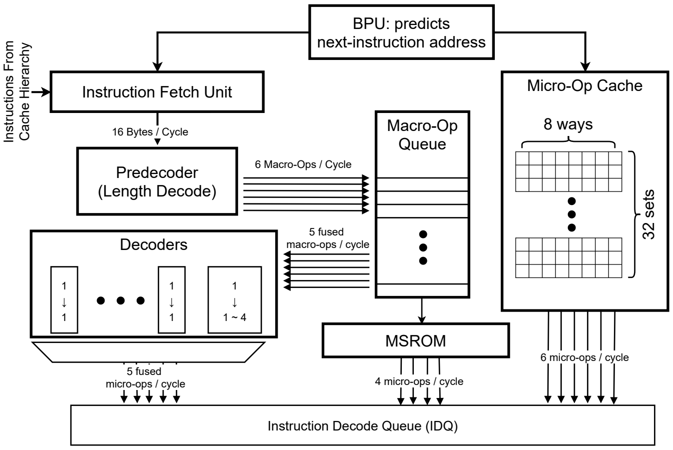


Fig. 1: x86 micro-op cache and decode pipeline

Multiple macro-ops are read from the macro-op queue every cycle, and distributed to the decoders which translate each macro-op into internal RISC (Reduced Instruction Set Computing)-like micro-ops. The Skylake microarchitecture features: (a) multiple 1:1 decoders that can translate simple macro-ops that only decompose into one micro-op, (b) a 1:4 decoder that can translate complex macro-ops that can decompose into anywhere between one and four micro-ops, and (c) a microsequencing ROM (MSROM) that translates more complex microcoded instructions, where a single macro-op can translate into more than four micro-ops, potentially involving multiple branches and loops, taking up several decode cycles. The decode pipeline can provide a peak bandwidth of 5 micro-ops per cycle [33]. In contrast, AMD Zen features four 1:2 decoders and relegates to a microcode ROM when it encounters a complex instruction that translates into more than two micro-ops.

The translated micro-ops are then queued up in an *Instruction Decode Queue* (IDQ) for further processing by the rest of the pipeline. In a particular cycle, micro-ops can be delivered to the IDQ by the micro-op cache, or if not available, the more expensive full decode pipeline is turned on and micro-ops are delivered that way.

In Intel architectures, the macro-op queue, IDQ, and the micro-op cache remain partitioned across different SMT threads running on the same physical core, while AMD allows the micro-op cache to be competitively shared amongst the co-located SMT threads, as are the rest of the structures in the decode pipeline, including the decoders and the MSROM.

### B. The Micro-Op Cache Organization and Functionality

The inherent complexity of the x86 decoding process has a substantial impact on the overall front-end throughput and

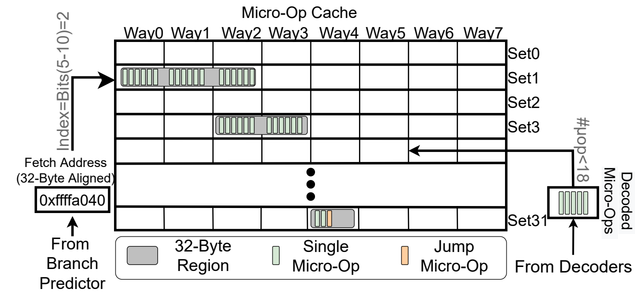


Fig. 2: Micro-Op Cache Organization and Streaming Functionality

power consumption. In fact, the x86 instruction decoding pipeline was shown to consume as much as 28% [34] of the overall processor power in the 1995 P6 architecture PentiumPro, and die shots of more recent processors including Skylake [35] indicate that the decoders and the MSROM continue to consume a significant chunk of the core area.

Modern Intel and AMD processors cache decoded micro-ops in a dedicated streaming cache, often called the *decoded stream buffer* or the *micro-op cache*, in order to bypass the decoder when a cached micro-op translation is already available. More specifically, it allows the front-end processing of hot code regions to be made faster and more energy efficient by sourcing micro-ops from the cache and powering down the decode pipeline [36]. In fact, when the micro-op cache was first introduced in Intel's Sandy Bridge microarchitecture, it was shown to provide a hit rate of 80% on average and close to 100% for "hotspots" or tight loop kernels [37], [38], resulting in large overall performance improvements and energy savings.

Figure 2 shows the Skylake micro-op cache that is organized as an 8-way set-associative cache with 32 sets. Each cache line is capable of holding up to 6 micro-ops, providing a maximum overall capacity of 1536 micro-ops. As decoded micro-ops arrive at the micro-op cache from one of the decoders or the MSROM, they are placed at an appropriate set identified using bits 5-9 of the x86 (macro-op) instruction address. This means any given cache line always contains micro-ops that correspond to macro-op instructions hosted within the same aligned 32-byte code region. Further, the following placement rules are observed.

- A given 32-byte code region may consume a maximum of 3 lines in the set (i.e., up to 18 micro-ops).
- Micro-ops delivered from the MSROM consume an entire line in the micro-op cache.
- Micro-ops that correspond to the same macro-op instruction may not span a micro-op cache line boundary.
- An unconditional branch (jump) instruction, if it appears, is always the last micro-op of the line.
- A micro-op cache line may contain at most two branches.
- 64-bit immediate values consume two micro-op slots within a given cache line.

In Section IV, we describe mechanisms to automatically generate code that exploits these placement rules to create arbitrary conflicts in the micro-op cache.

In contrast to traditional data and instruction caches, the micro-op cache is implemented as a streaming cache, allowing micro-ops that correspond to a 32-byte code region to be continuously streamed, from one way of the set after another, until an unconditional branch (jump) is encountered or a misprediction of a conditional branch interrupts the streaming process. Once all micro-ops of a 32-byte code region have been streamed, the appropriate way in the next set of the micro-op cache is checked for streaming the next statically contiguous 32-byte code region in the program. If the lookup fails (i.e., a micro-op cache miss occurs), a switch is made to the x86 decode pipeline to trigger the two-phase decoding process. This switch entails a one-cycle penalty. Therefore, the micro-op cache miss penalty is a combination of this one-cycle switch penalty and the variable latency of the 2-phase instruction decoding process that is highly sensitive to the composition of the 32-byte code region (number and type of macro-ops) being decoded, providing a sufficiently large signal to implement a timing channel.

Due to their high performance and energy savings potential, micro-op caches have been steadily growing in size. It is 1.5X larger in the latest Intel Sunny Cove microarchitecture, and the more recent AMD Zen-2 processors feature micro-op caches that can hold as many as 4K micro-ops.

Finally, the micro-op cache is inclusive with respect to the instruction cache and the instruction translation lookaside buffer (iTLB), which implies that any cache line evicted out of the instruction cache would also trigger an eviction in the micro-op cache, but not vice-versa. In the event of an iTLB flush, such as in the case of an SGX enclave entry/exit, the entire micro-op cache is flushed, and as a result, information is not leaked via the micro-op cache across enclave boundaries.

### C. Other Front End Optimizations

To maximize code density in structures beyond the instruction cache, Intel and AMD processors implement a bandwidth optimization called *micro-op fusion*. This feature allows decoded micro-ops that adhere to certain patterns to be fused together into a single micro-op. A fused micro-op takes up only one slot while being buffered in the micro-op cache and the micro-op queue, saving decode bandwidth and improving the micro-op cache utilization. However, it is later broken down into its respective component micro-ops at the time of dispatch into the functional units for execution.

In addition to the micro-op cache, Intel processors feature a *loop stream detector* (LSD) that is capable of identifying critical loop kernels and lock them down in the IDQ such that decoded and potentially fused micro-ops are continuously issued out of the IDQ, without even having to access the micro-op cache. While it is straightforward to violate these requirements for LSD operation and ensure that the micro-op cache is always operating, we did not need to use these techniques as it is already disabled in Skylake due to an implementation bug (erratum SKL150).

```
%macro UOP_REGION(lbl)
 lbl:
  nop15; 15 bytes +
  nop15; 15 bytes +
  nop2 ;  2 bytes
       ;  = 32 bytes
%endmacro

for samples in 1..3000{
  ; line 0
  UOP_REGION(region_0)
  ; line 1
  UOP_REGION(region_1)
  ...
  ; line n
  UOP_REGION(region_n)
}
```

Listing 1: Microbenchmark for determining the $\mu$op cache size.

### D. Microarchitectural Covert and Side Channels

The literature describes several covert- and side-channel attacks that use shared microarchitectural resources as their medium of information leakage [1]–[5]. The most prominent are the ones that exploit stateful resources such as the data cache [1]–[5], instruction cache [39], TLB [6], [7], and branch predictors [8], [9]. These stateful resources offer persistence over time, allowing attackers to periodically probe them to monitor the activity of a potential victim. While stateless resources [40]–[42] have also been exploited for side-channel leakage, the attack model on stateless resources are mostly limited to the scenarios that involve two co-located threads continuously contending for the shared stateless resource. Execution ports [41], functional units [42], [43], and memory buses [40] are among the stateless resources that are used as covert communication channels.

### E. Transient Execution Attacks

Transient execution attacks [11]–[15] exploit the side-effects of transient instructions to bypass software-based security checks, and further leak sensitive information to the attacker via a microarchitectural side channel. These attacks typically include four salient components [44] – (1) a *speculation primitive* that allows the attacker to steer execution along a misspeculated path, (2) a *windowing gadget* that sufficiently prolongs execution along the misspeculated path, (3) a *disclosure gadget* comprised of one or more instructions that transmit the transiently accessed secret to the attacker, and (4) a *disclosure primitive* that acts as the covert transmission medium. While most existing attacks mostly use data caches as their primary disclosure primitives [11]–[15], attacks like SMoTherSpectre [45] have shown that stateless resources such as execution ports are viable alternatives. The micro-op cache-based transient execution attack that we describe exploits a novel and a powerful disclosure primitive that is not easily blocked by existing defenses, as the information disclosure occurs at the front end of the pipeline, even before a transient instruction gets dispatched for execution.

```
%macro UOP_REGION (lbl, target)
 .align 1024
 lbl:
  jmp target
%endmacro

for samples in 1..3000{
 ; set 0, way 0
 UOP_REGION (region_0, region_1)
 ; set 0, way 1
 UOP_REGION (region_1, region_2)
 ...
 ; set 0, way n
 UOP_REGION (region_n, exit)
exit:
}
```

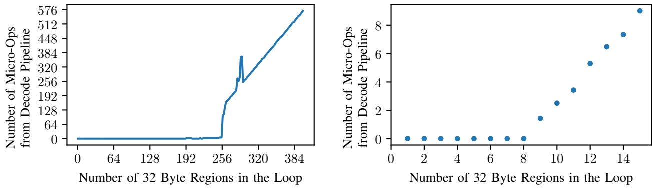
Listing 2: Microbenchmark for determining the $\mu$op cache associativity.

## III. Characterizing the Micro-Op Cache

In this section, we perform a detailed characterization of the micro-op cache. We describe mechanisms and microbenchmarks we use to study its timing effects under different execution scenarios. The importance of this characterization study is two-fold. First, it allows us to reverse-engineer several important micro-op cache features that are not documented in the manuals, in addition to throwing further light on the timing effects of various documented features and nuances such as the placement rules. Second and more importantly, it enables the development of a principled framework for automatically generating code that produces certain timing effects, which could then be exploited for constructing high-bandwidth timing channels.

We perform our characterization study on the Intel i7-8700T processor that implements the Coffee Lake microarchitecture (which is a Skylake refresh). We use the *nanoBench* [46] framework for our initial analysis, which leverages performance counters exposed by Intel. We use those to understand the structure, organization, and policies of the micro-op cache. However, we are also able to repeat our findings on purely timing-based variances alone, as measured by Intel's RDTSC instruction, which is available in user space.

**Size.** To infer the size of the micro-op cache, we use a microbenchmark shown in Listing 1 that executes loops of varying lengths, in increments of 32 byte code regions. We then use performance counters to measure the number of micro-ops being delivered from the decode pipeline (and thus not the micro-op cache). Note that the microbenchmark is crafted such that each 32-byte region in the loop consumes exactly one line in the micro-op cache, and exactly three micro-op slots within each line. The result of this experiment is shown in Figure 3a. Clearly, we observe a jump in the number of micro-ops delivered from the decode pipeline as the size of the loop increases beyond 256 32-byte code regions (each consuming one cache line in the micro-op cache), confirming our understanding that the micro-op cache contains 256 cache lines and no more. Note that the curve has a gradual slope and that we consistently observe a spike at around 288 regions. We



(a) Cache Size.

(b) Cache Associativity

Fig. 3: (a) Measuring $\mu$op cache size by testing progressively larger loops. (b) Measuring the size of one set in the $\mu$op cache to determine its associativity.

```
%macro
    UOP_REGION(
    lbl,target)
; X nops + 1 jmp
; = 32 bytes
.align 1024
lbl:
 nop; one byte
 nop
 ...
 nop
 jmp target
%endmacro
```

```
for samples in 1..3000
{
 ; set 0, way 0
 UOP_REGION (region_0, region_1)
 ; set 0, way 1
 UOP_REGION (region_1, region_2)
 ...
 ; set 0, way n
 UOP_REGION (region_n, exit)
exit:
}
```

Listing 3: Microbenchmark for Determining Placement Rules.

hypothesize that this is an artifact of a potential hotness-based replacement policy of the micro-op cache, which we study in detail later in this section.

**Associativity.** We next turn our attention to associativity. Attacks on traditional caches rely heavily on knowledge of associativity and replacement policy to fill sets. Listing 2 shows our microbenchmark that allows us to determine the associativity of a micro-op cache, by leveraging regularly spaced jumps hopping between regions that all map to the same set, again within a loop of variable size. In this case, the size of the loop targets the number of ways in a set rather than the number of lines in the cache.

More specifically, all regions in the loop are aligned in such a way that they get placed in the same cache set (set-0 in our experiment), and each region only contains an unconditional jump instruction that jumps to the next aligned region. Recall that the micro-op cache's placement rules forbid hosting micro-ops beyond an unconditional jump in the same cache line; thus in our microbenchmark, each region within the loop consumes exactly one cache line and exactly one micro-op slot in that cache line. Figure 3b shows the result of this experiment. The number of micro-ops being delivered from the decode pipeline clearly rises as the number of ways within each set increases beyond eight. This confirms our understanding that the micro-op cache has an 8-way set associative organization (and thus contains 32 sets).

**Placement Rules.** To understand the micro-op cache placement rules, we use a variant of the microbenchmark described above, as shown in Listing 3, where we jump from one aligned region (different way of the same set) to another in a loop. The difference here is that we not only vary the number of 32-byte code regions in the loop, but we also vary the number of micro-ops within each region. Since we only use one-byte NOPs and an unconditional jump instruction, the number of micro-ops in a given region can vary anywhere between 0 and 31. By varying these parameters, we are able to infer (1) the maximum number of cache lines (ways per set) that can be used by any given 32-byte code region and (2) the number of micro-op slots per cache line.

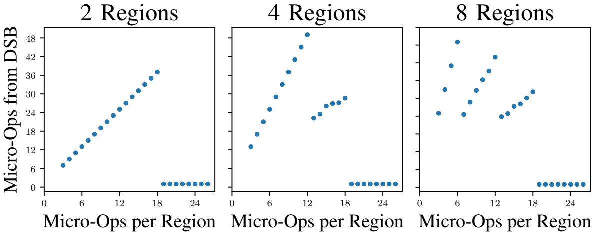
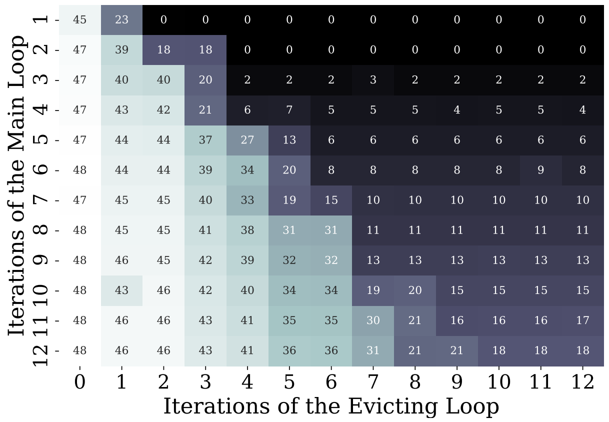


Fig. 4: Micro-Op Cache Placement Rules: Each 32-byte code region maps to a maximum of 3 micro-op cache lines (18 micro-ops). Note that in Intel's performance counter documentation the micro-op cache is called the *DSB*.

Figure 4 shows our findings in three scenarios – when the number of regions within the loop are 2, 4, and 8. Note that, in this experiment, we report the number of micro-ops delivered from the micro-op cache rather than the decode pipeline. We make the following major inferences. First, when we constrain the loop to contain only two 32-byte code regions, we observe that the number of micro-ops being delivered from the micro-op cache rises steadily as the code regions get larger, up until the point where it is able to host 18 micro-ops per region. Beyond 18 micro-ops, this number drops suddenly, indicating that the micro-op cache is unable to host 32-byte code regions that take up more than 18 micro-op slots. Second, when we constrain the loop to contain four or eight regions, we observe that the micro-op cache is able to host and steadily deliver micro-ops for four regions consuming 12 micro-op slots each or eight regions consuming 6 micro-op slots each. This confirms our understanding that the 8-way set-associative micro-op cache in Skylake allows at most 6 micro-ops to be held in any given cache line, and that a given 32-byte code region may consume a maximum of 3 ways per set (i.e., 18 micro-op slots). If it exceeds that, the line is not cached at all.

**Replacement Policy.** We next extend our characterization study to reverse-engineer undocumented features of the micro-op cache. We begin with identifying the replacement policy/eviction criteria used in Skylake's micro-op cache design.

In the absence of any information about the replacement policy, we would expect it to be similar to that of either a traditional cache (LRU-like policy), where the decision to evict a cache line is based on recency, or a trace cache [47] where this decision is more hotness-driven. To gain insight into the replacement policy used in the micro-op cache, we leverage a variant of the microbenchmarks we use to determine



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 45 | 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 47 | 39 | 18 | 18 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 47 | 40 | 40 | 20 | 2 | 2 | 2 | 3 | 2 | 2 | 2 | 2 | 2 |
| 4 | 47 | 43 | 42 | 21 | 6 | 7 | 5 | 5 | 5 | 4 | 5 | 5 | 4 |
| 5 | 47 | 44 | 44 | 37 | 27 | 13 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 48 | 44 | 44 | 39 | 34 | 20 | 8 | 8 | 8 | 8 | 8 | 9 | 8 |
| 7 | 47 | 45 | 45 | 40 | 33 | 19 | 15 | 10 | 10 | 10 | 10 | 10 | 10 |
| 8 | 48 | 45 | 45 | 41 | 38 | 31 | 31 | 11 | 11 | 11 | 11 | 11 | 11 |
| 9 | 48 | 46 | 45 | 42 | 39 | 32 | 32 | 13 | 13 | 13 | 13 | 13 | 13 |
| 10 | 48 | 43 | 46 | 42 | 40 | 34 | 34 | 19 | 20 | 15 | 15 | 15 | 15 |
| 11 | 48 | 46 | 46 | 43 | 41 | 35 | 35 | 30 | 21 | 16 | 16 | 16 | 17 |
| 12 | 48 | 46 | 46 | 43 | 41 | 36 | 36 | 31 | 21 | 21 | 18 | 18 | 18 |

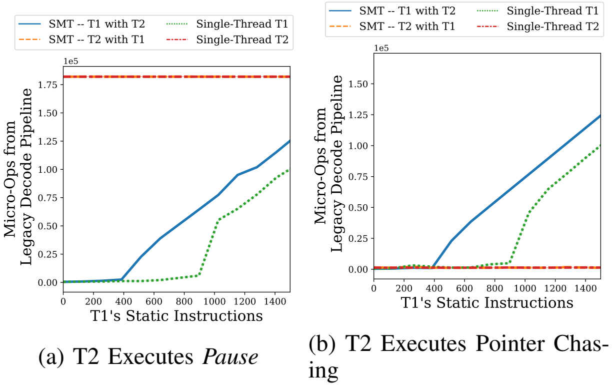
Fig. 5: Number of micro-ops from micro-op cache when measuring one loop and evicting it with another.

the associativity and placement rules of the micro-op cache. However, in this case, we interleave two loops – a *main loop* and an *evicting loop* that each jump through eight ways of the same set (set 0 in this experiment), for a controlled number of iterations. We then use performance counters to measure the number of micro-ops being delivered from the micro-op cache for the *main loop*, while varying the number of iterations executed by both loops.

Note that each 32-byte code region used here maps to exactly 6 micro-ops, taking up all micro-op slots in the cache line it occupies. Since we jump through eight ways of the same set within each loop iteration, in the absence of any interference (and thus, micro-op cache misses), we should observe a total of 48 micro-ops being streamed through the micro-op cache, per loop iteration. Figure 5 shows the result of our experiment. In this figure, the iterations of the two loops are on the axes, and the value/intensity at each x-y coordinate represents the number of micro-ops delivered by the micro-op cache. As expected, we find that the number of micro-ops delivered from the micro-op cache stays at about 48 when there is no interference, but this quickly drops as we increase the number of iterations in the *evicting loop*, while executing four or fewer iterations of the *main loop*. However, when the *main loop* executes eight or more iterations, we observe that this degradation is gradual, tending to retain micro-ops from the *main loop* in the micro-op cache.

These results show that the eviction/replacement policy is clearly based on hotness (how many times the interfering lines are accessed in the loop, relative to the target code) rather than recency (where a single access would evict a line). We see this because lines are only replaced in significant numbers when the interfering loop count exceeds the targeted loop count.

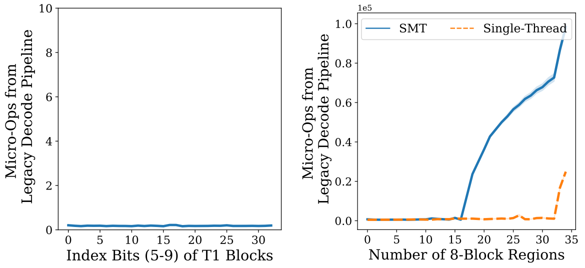
Hotness as a replacement policy potentially leaks far more information than traditional cache replacement policies, which always evict lines on first miss, and represents a dangerous attack vector. That is, a traditional cache attack can only detect whether an instruction block was accessed. The micro-op cache potentially reveals exactly how many times a conflicting instruction block was accessed.



(a) T2 Executes *Pause*

(b) T2 Executes Pointer Chasing

Fig. 6: Micro-Op Cache Usage of Sibling Threads in SMT-Mode in Intel Processors. T1 executes Listing 1, while T2 executes (a) PAUSE or (b) pointer chasing loads.



(a) One 8-Block Region

(b) Multiple 8-Block Regions

Fig. 7: Deconstruction of SMT Sharing Mechanism of Intel Processors: (a) an 8-way region (all blocks within a region are mapped to the same set) is moved across different sets to detect a possible contention or the lack thereof, (b) determining the number of potential 8-way sets in SMT mode

**Partitioning Policy.** We next devise an experiment where two logical threads running on the same SMT core contend for the micro-op cache. The threads each execute varying number of 8-byte NOP instructions, to measure the size of the micro-op cache. We observe in Intel (Sandy Bridge, Ivy Bridge, Haswell, and Skylake) processors that there is no measurable interference between threads, and in fact, we find that the effective size visible to each thread is exactly half the physical size.

To confirm if the micro-op cache is indeed statically partitioned, we examine a scenario where the threads contending for the micro-op cache execute the same number of micro-ops, but one thread is inherently slower than the other; if this results in unequal sharing of the micro-op cache, we conclude that it is dynamically partitioned. We examine two different options to slow down a thread: (1) using Intel's PAUSE instruction and (2) using a pointer chasing program that frequently misses in the data cache. More specifically, in these experiments, one of the threads, T1, continues to execute the NOP sequence, while the other thread, T2 executes a fixed number of pause (Figure 6a) or pointer chasing loads (Figure 6b). Clearly, from the figures, we observe that, in SMT mode, T1 always gets allocated exactly half of the micro-op cache, regardless of the instructions being executed by T2. Interestingly, we also find that PAUSE instructions don't get cached in the micro-op cache. This indicates that, in Intel processors, the micro-op cache is statically partitioned between the threads, and no lines are dynamically shared.

We next deconstruct the partitioning mechanism used by Intel processors by devising an experiment in which both threads try to fill set zero of the micro-op cache by repeatedly fetching and executing regions with 8 micro-op cache blocks that are aligned to a 1024-byte address, similar to the microbenchmark in Listing 3. During this experiment, the threads monitor the number of micro-ops being delivered from the decode pipeline. The somewhat surprising result of this experiment is that both partitions of the cache remain 8-way set associative. To analyze this further, we have T1 shift through different sets of the micro-op cache by aligning its code to addresses with different index bits, while T2 continues to fetch code mapped to set zero, as shown in Figure 7a. We observe that both threads deliver all of their micro-ops from the micro-op cache regardless of the set being probed by T1, still showing 8-way associative behavior. Further, when T1 accesses multiple such 8-way regions mapped to consecutive sets (as shown in Figure 7b), we observe that it is able to stream exactly 32 such regions in single-threaded mode and exactly 16 such regions in SMT mode. Thus, we hypothesize that the micro-op cache is not way-partitioned [48] into 32 4-way sets for each thread, but is rather divided into 16 8-way sets available to each thread.

Note that we use performance counters to obtain measurements throughout this characterization study to gain insights about the various features and nuances of the micro-op cache. However, we are able to extend our microbenchmarks to craft exploits that rely only on timing measurements, as shown in Section V. This is important because those experiments use techniques very similar to those an attacker, who may not have access to these performance counters, will use to covertly transmit secrets over the micro-op cache.

## IV. Creating and Exploiting Contention in the Micro-Op Cache

In this section, we leverage and extend the microbenchmarks from our characterization study to create a framework for automatically generating exploit code that can create arbitrary conflicts in the micro-op cache. Our goal is to exploit the timing effects associated with conflicting accesses in the micro-op cache to construct a timing channel that (1) can covertly transmit secrets through the micro-op cache and (2) cannot be detected by performance counter monitors unless they specifically target the micro-op cache. To accomplish this, we craft an attack such that strong signal is observed through the micro-op cache, but no noticeable perturbation is observed in instruction cache activity, data cache activity, or backend resource-related stalls (including those from the load/store buffers, reservation stations, ROB, BOB, etc.) [49].

In particular, for any given code region, we want to be able to automatically generate two types of exploit code – one that can evict a given code region by replicating its micro-op cache footprint (by occupying the same sets and ways), generating a clear timing signal, and another that is able to occupy a mutually exclusive subset of the micro-op cache such that no conflicts occur and consequently no timing signals are observed. Henceforth, we will refer to the former as *tiger* and to the latter as *zebra*. Thus, two *tigers* contend for the same sets and ways, but a *zebra* and a *tiger* should never contend with each other, always occupying the cache in a mutually exclusive manner.

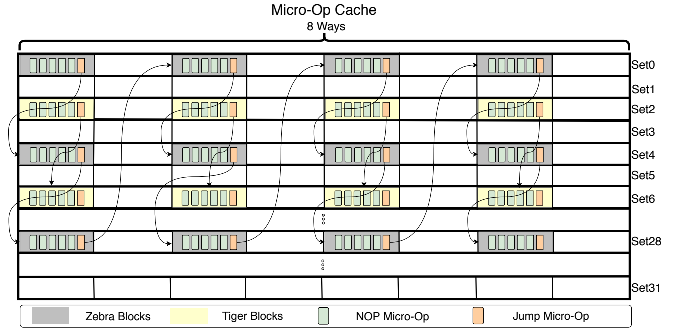


Fig. 8: Striping occupation of the micro-op cache. Zebras and Tigers are devised to be mapped into mutually exclusive sets in the micro-op cache. Jump targets are shown by arrows.

Although the microbenchmark in Listing 3 is designed to determine the associativity of the micro-op cache, it also provides us with the requisite framework for creating code that can arbitrarily jump from any given cache line to another, where each cache line hosts a set of NOPs followed by an unconditional jump instruction. Thus, given a code region, we can automatically generate the corresponding *tiger* and *zebra* code by ensuring that they jump through the micro-op cache, touching the appropriate set of cache lines. Figure 8 shows an example of a *striped* occupation of the micro-op cache, where the *zebra* and the *tiger* jump through four ways of every fourth set of the micro-op cache; the *zebra* starts at the very first set, while the *tiger* starts at the third set, thereby interleaving each other by one set.

We found that our best tigers and zebras (1) occupy evenly spaced sets across the 32-sets of the micro-op cache, (2) leave two ways of each set empty to allow subsequent code to enter the micro-op cache without obfuscating the signal, and (3) are made of long sections of code that include as few micro-ops as possible by padding no-ops and jumps with length-changing prefixes (LCP). The instruction composition described in point (3) facilitates creating a sharp timing signal: the lack of back-end execution, combined with the heavy use of LCP, ensures that the bottleneck is in the decode pipeline and creates a observable difference between micro-op cache hits and misses. This allows us to obtain a clearly distinguishable a binary signal (i.e., hit vs. miss) with a mean timing difference of 218.4 cycles and a standard deviation of 27.8 cycles, allowing us to reliably transmit a bit (i.e., one-bit vs. zero-bit) over the micro-op cache.

TABLE I: Bandwidth and Error Rate Comparison

| **Mode** | **Bit Error Rate** | **Bandwidth (Kbit/s)** | **Bandwidth with error correction** |
|---|---:|---:|---:|
| Same address space* | 0.22% | 965.59 | 785.56 |
| Same address space (User/Kernel) | 3.27% | 110.96 | 85.20 |
| Cross-thread (SMT) | 5.59% | 250.00 | 168.58 |
| Transient Execution Attack | 0.72% | 17.60 | 14.64 |

*results are taken over 32 samples, with standard deviation of 6.91Kbit/s

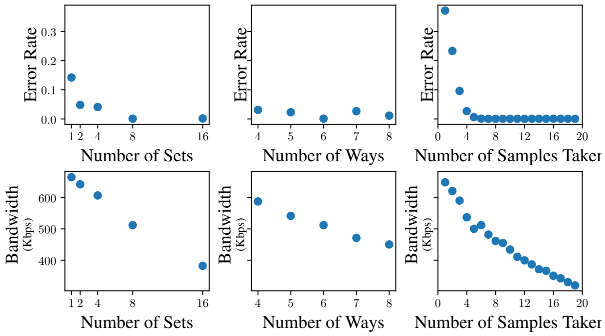


Fig. 9: Tuning Set/Way Occupancy and Sample Count for Transmission Accuracy and Bandwidth. We vary one parameter while keeping the other two constant. The constant parameters are (nways=6, nsets=8, samples=5).

## V. The Micro-Op Cache Disclosure Primitive

In this section, we leverage the framework described above to demonstrate two information disclosure methods that rely solely on exploiting the timing effects of the micro-op cache to covertly transmit secrets – (a) across the user/kernel privilege boundary within the same thread, and (b) across different SMT threads that are co-located on the same physical core, but different logical cores. These make no observable modifications to any data/instruction cache in the traditional L1/L2/L3 cache hierarchy, and are therefore not only far less detectable, but have the ability to bypass several recently proposed cache side-channel defenses that rely on deploying randomized cache indexing mechanisms. We describe and evaluate each of these in detail.

### A. *Cross-Domain Same Address Space Leakage*

The goal of this attack is to establish a communication channel between two code regions within the same address space. We first describe a proof-of-concept attack that is able to leak information over the micro-op cache, across code regions within the same address space and the same privilege levels. We further discuss techniques to improve the bandwidth and the error rate of this channel, and then extend this attack to detect secrets across different privilege levels.

**Proof-Of-Concept Implementation.** This section describes a proof-of-concept implementation that exploits the timing effects due to conflicting micro-op cache accesses in two different code regions, albeit operating at the same privilege level. To this end, we leverage three distinct functions made available from our framework described in Section IV – two versions of *tiger* and one *zebra*. Recall that two *tiger* functions contend with each other in the micro-op cache as they occupy the same sets and ways, while the *tiger* and the *zebra* occupy mutually exclusive cache lines in the micro-op cache. This then allows us to mount a conflict-based attack where the receiver (spy) executes and times a *tiger* loop, while the sender (Trojan) executes its own version of the *tiger* function to send a one-bit or the *zebra* function to send a zero-bit.

We verify that the information transmitted through this timing channel is indeed a manifestation of the timing effects of the micro-op cache (i.e., not the instruction cache, which would also create measurable performance variation) by probing performance counters that show a jump in the number of instructions delivered from the decode pipeline rather than the micro-op cache and further indicate that there is no noticeable signal due to misses in the L1 (data/instruction), L2, and L3 caches during execution.

**Bandwidth Optimization.** To improve the bandwidth and accuracy of our timing channel, we tune a number of different parameters, including the number of micro-op cache sets and ways probed by the *tiger* and *zebra* functions, and the number of samples gathered for measurements.

Figure 9 shows the impact on bandwidth and error rate as we vary sets occupied, ways occupied, and the number of samples taken. We make several observations. First, as expected, the covert channel bandwidth increases as we reduce the number of samples and the number of set and ways we probe. We reach the highest bandwidth of over 1.2 Mbps, while incurring an error rate of about 15%, when we only probe four ways of just one set and gather only five samples. Second, the number of ways probed do not have a significant impact on the accuracy, which means we are able to create conflicts in the micro-op cache that produce noticeable timing effects when our *tiger* functions contend for just four ways of a set. Third, the error rate drastically drops to less than 1% as we increase the number of sets probed to eight.

We reach our best bandwidth (965.59 Kbps) and error rates (0.22%) when six ways of eight sets are probed, while limiting ourselves to just five samples. We further report an error-corrected bandwidth by encoding our transmitted data with Reed-Solomon encoding [50] that inflates file size by roughly 20%, providing a bandwidth of 785.56 Kbps with no errors (shown in Table I).

**Leaking Information across Privilege Boundaries.** We next extend the proof-of-concept attack described above to enable information disclosure across the user-kernel boundary. We leverage the same functions as described above, where the spy makes periodic system calls to trigger a kernel routine that makes secret-dependent call to another internal kernel routine. The secret can then be inferred by the spy in a probe phase, by executing and timing a corresponding *tiger* version of that internal routine. This timing channel has a bandwidth of 110.96 Kbps with an accuracy of over 96%, or 85.2 Kbps with error correction. This experiment not only confirms that the micro-op cache is not flushed across privilege boundaries, but shows that our channel is tolerant to noise and interference due to the additional overhead of making a system call.

TABLE II: Tracing Spectre Variants using Performance Counters

| Attack | Time Taken | LLC References | LLC Misses | $\mu$op Cache Miss Penalty |
|---|---|---|---|---|
| Spectre (original) | 1.2046 s | 16,453,276 | 10,997,979 | 5,302,647 cycles |
| Spectre ($\mu$op Cache) | 0.4591 s | 3,820,847 | 3,756,310 | 74,689,315 cycles |

### B. Cross-SMT Thread Leakage

We also examine cross thread micro-op cache leakage in SMT processors. Since the threads of an SMT processor share the same micro-op cache, they may be vulnerable to micro-op cache attacks, depending on how they share it. However, since our characterization study suggests that the micro-op cache in Intel processors is statically partitioned, we turn our attention to the AMD Zen processor, where the micro-op cache is competitively shared among the threads.

In particular, on AMD Zen processors, micro-ops of one thread could evict the micro-ops of another thread as they compete for the micro-op cache. Based on this observation, we construct a cross-thread covert channel on AMD Zen processors. The Trojan (sender) thread encodes "one" by executing a large number of static instructions that contend for a wide number of micro-op cache sets. The spy (receiver) thread constantly executes and times a large number of static instructions that touch all the sets of the micro-op cache. Since we choose instructions whose execution time is sensitive to their micro-op cache hit rate, the spy's execution time considerably increases when the Trojan sends "one". This enables us to achieve a covert channel with a high bandwidth of 250 Kbps with an error rate of 5.59% or 168.58 Kbps with error correction, as shown in Table I.

## VI. I See Dead $\mu$ops: Transient Execution Attack

In this section, we describe and evaluate two transient execution attack variants that exploit the novel micro-op cache vulnerability we expose. The first attack is similar to Spectre-v1 [11] in that it bypasses a bounds check to perform an unauthorized read of secret data, but leverages the micro-op cache side channel as its disclosure primitive. The second attack also bypasses a software-based authorization check, but performs a secret-dependent indirect function call that automatically triggers a micro-op cache access to the predicted target of that function call, thereby leaving an observable footprint in the micro-op cache even before the instructions at the predicted target are renamed and dispatched for execution. The proof-of-concept attacks described in this section were carried out on an Intel i7-8700T processor that implements the Coffee Lake microarchitecture.

### A. Variant-1: The $\mu$op Disclosure Primitive

**Attack Setup.** Listing 4 shows the code for a vulnerable library that contains a secret in its memory space (in this case, a character array `secret`) and exposes an API `victim_function` to read an element of a non-secret array. The `victim_function` performs a bounds check on the index, and if the index is within bounds of the public array, it returns the non-secret value at the index. However, as described in the Spectre-v1 attack, the bounds check can be bypassed by mistraining the branch predictor such that it is always predicted to land on the in-bounds path. This can be accomplished by repeatedly calling the `victim_function` with in-bounds indices so the predictor gets sufficiently trained to emit a predicted taken outcome.

```
char array[1024];
int array_size = 1024;
...
char secret[1024];
extern uint8_t victim_function(size_t i) {
    // bounds check:
    if (i >= 0 && i < array_size) {
        // misspeculation of this branch
        // bypasses the bounds check
        return array[i];
    }
    return -1;
}
```

Listing 4: Victim Method for our Variant-1 Attack

The next step is to set up the micro-op cache for covert transmission. Again, the attacker leverages three distinct functions from our framework described in Section IV – two versions of *tiger* (one used by the sender and the other used by the receiver) and one *zebra*. The attacker first primes the appropriate sets in the micro-op cache using the receiver's version of the *tiger* function and then invokes the `victim_function` with a maliciously computed out-of-bounds index `i` such that the transient array access, `array[i]`, actually evaluates to a target location in the `secret` array. This results in the `victim_function()` returning the speculatively accessed secret byte to the caller.

**Secret Transmission.** The attacker is then able to infer this secret via the micro-op cache side-channel as follows. The attacker first extracts each bit of the byte using bit manipulation (masking logic), and then calls a *zebra* to transmit a zero-bit or a *tiger* (the sender's version) to transmit a one-bit.

**Side-Channel Inference.** In the next step, the attacker waits until after the transiently executed instructions get squashed and the `victim_function` returns, throwing an out-of-bounds error. At that point, it executes and times the receiver's version of *tiger*. Timing will show whether it is read from L1 ICache and decoded, or if it is streamed directly from the micro-op cache, allowing the attacker to infer the secret bit. The process is repeated until all bits of the secret are inferred.

**Bandwidth and Stealthiness of the Attack.** Note that the attack is similar in spirit to Spectre-v1 in that both attacks exploit transient execution to bypass bounds checks and gain unauthorized access to secret information. However, there are two key differences in the way our attack uses the micro-op cache to transmit that information to non-speculative, committed attacker code, instead of transmitting it over the LLC.

First, our attack is much faster. Table II shows both time measured in seconds and a few other relevant performance counter measurements taken while running our attack and the original Spectre-v1 attack, both reading the same number of samples and revealing the same secret data. Notice that the timing signal has clearly shifted from the LLC to the micro-op cache – the number of LLC references and misses have decreased significantly (by about 5X and 3X respectively), whereas the micro-op cache miss penalty (the decode overhead and the switch penalty) has substantially increased (by about 15X). Overall, our attack is 2.6X faster than the original Spectre-v1 attack. This difference is even more dramatic considering the fact that our attack leaks a secret on a bit-by-bit basis while the Spectre-v1 attack leaks it on a byte-by-byte basis, leaving significant additional room for bandwidth optimizations (for example, using a jump table) that could further expand this gap. Furthermore, our attack is more stealthy and far less detectable by traditional cache monitoring techniques [51], [52], given that we make fewer references to not just the LLC, but to data and instruction caches across the hierarchy.

Second, and likely more significantly, the gadgets for our attack occur more naturally than those of Spectre-v1 as they only involve looking up a single array access with an untrusted index that is guarded by a security check. In fact, a value-preserving taint analysis on the LGTM security analysis web platform counts 100 of our gadgets in the `torvalds/linux` repository, compared to only 19 for Spectre-v1.

We also identify 37 gadgets in the Linux kernel (version 5.11-rc7) that also have the ability to perform a bit masking operation on the retrieved secret followed by a dependent branch, to trigger a micro-op cache access. To demonstrate the exploitability of such gadgets, we replace the gadget in our proof-of-concept exploit code with one of these gadgets (specifically, a gadget in the PCI driver routine $pci\_vpd\_find\_tag$), and further automatically generate the appropriate *tiger* functions for side-channel inference. We observe that our attack is able to reliably leak bits of the transiently accessed secret, and our measurements indicate a clear signal from the micro-op cache. Further, by combining our attack with Spectre-v2 (Branch Target Injection) [11], we are also able to arbitrarily jump to these gadgets while we are in the same address space. More comprehensive gadget analysis and chaining experiments are subject of future work.

### *B. Variant-2: Bypassing LFENCE*

In this section, we describe a novel transient execution attack variant that not only leverages the micro-op cache as a disclosure primitive, but exploits the fact that indirect branches and calls trigger a micro-op cache access to *fetch* micro-ops at a predicted target. If these micro-ops are transmitter instructions (that carry secrets and further transmit them via a suitable disclosure primitive), they would leave a footprint in the micro-op cache, even before they get dispatched to execution.

This not only breaks existing invisible speculation-based defenses [20], [21], [28] that track potential transmitter instructions in the instruction queue and prevent them from leaking information, but also bypasses a restrictive solution recommended by Intel – the LFENCE [53], a synchronization primitive that forbids younger instructions from being dispatched to execution. This is the first transient execution attack to demonstrate that a transmitter instruction can produce observable microarchitectural side-effects before a hardware-based defense strategy gets the opportunity to decode its operands, analyze information flow, and prevent the instruction from executing.

```c
char secret;
extern void victim_function(ID user_id) {
  // authorization check bypassed by mistraining
  if (user_id is authorized) {
    asm volatile("lfence");
    // LFENCE: stall the execution of
    // younger instructions

    // transmitter: indirect call
    fun[secret]();
  }
}
```

Listing 5: Victim Method for our Variant-2 Attack

**Proof-Of-Concept Attack.** The goal of our proof-of-concept attack is to consistently leak a secret bit-by-bit across the LFENCE. To demonstrate this, we consider a victim function (shown in Listing 5) that contains a gadget guarded by an authorization check, similar to the one used in Spectre-v1 [11]. However, the key difference is that our gadget relies on a transmitter instruction that performs a secret-dependent indirect jump/function call, rather than a secret-dependent data access. This allows for the implicit transmission of secret information at *fetch* rather than execute, if the indirect branch predictor is sufficiently well-trained to accurately predict the outcome of the transmitting indirect branch instruction. This training is possible via previous legitimate invocations of the victim function by an authorized user passing the security check, essentially encoding the secret implicitly in the indirect branch predictor.

In the setup phase, we sufficiently mistrain the authorization check guarding our Spectre gadget similar to Spectre-v1. Note that, this step is specifically targeted at bypassing the authorization check and this does not influence the outcome of the secret-dependent indirect jump in our experiments. We then prime the entire micro-op cache to ensure that our timing measurements are not attributed to lingering data in the micro-op cache.

In the next step, for each potential target of the indirect jump, we automatically generate and re-prime appropriate micro-op cache sets using a corresponding *tiger* version of the code at the predicted target using the methodology described in Section IV. We then invoke the victim by providing an untrusted input that would otherwise result in the authorization check to fail. However, due to our mistraining step above, execution proceeds speculatively beyond the authorization check, essentially bypassing it, similar to Spectre-v1.

Once execution reaches the LFENCE, all younger instructions, including our transmitter, are prevented from being

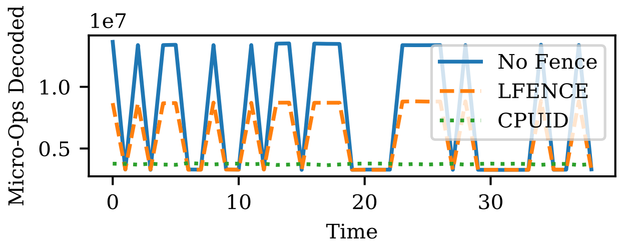


Fig. 10: Micro-Op Cache Timing Signal (with CPUID, LFENCE, and no fencing at the bounds check)

dispatched to functional units, until the LFENCE is committed. However, this does not prevent younger instructions, including those at the predicted target of our secret-dependent indirect branch, from being fetched, thereby allowing them to leave a trace in the micro-op cache. Note that the secret-dependent indirect branch does not need to be evaluated, it only needs to be accurately predicted so we fetch the instructions at the appropriate target, filling appropriate sets in the micro-op cache. Once execution recovers from misspeculation and control is transferred back to the attacker, the secret bit can be inferred by probing the micro-op cache with the appropriate *tiger* routines. The timing signal will reveal if they were evicted by the victim function or if they remained untouched, ultimately disclosing the secret.

**Results.** To evaluate the effectiveness of our attack in the presence of different synchronization primitives, we consider three different victim functions where the authorization check and the transmitter instructions are separated by – (1) no synchronization primitive, (2) an LFENCE that prevents younger instructions from being dispatched to execution, and (3) a CPUID that prevents younger instructions from being fetched. Figure 10 shows the results of this experiment. We observe a clear signal when execution is not protected by any synchronization primitive. We also don't observe any signal when speculation is restricted via CPUID. However, we continue a observe a signal when an LFENCE is deployed to serialize execution, unlike the case of Spectre-v1 that is completely mitigated by LFENCE.

## VII. Discussion

In this section, we discuss the effectiveness of our attacks in the presence of existing mitigations and countermeasures proposed in the literature against side channel and transient-execution attacks. We also discuss the extensibility of these mitigations to defend against our attack. We begin by categorizing recently proposed mitigations for Spectre and other related transient execution attacks into two major classes that – (1) prevent the covert transmission of a secret, and (2) prevent unauthorized access to secret data. We discuss the effectiveness of our attack in the presence of each of these classes of defenses below.

**Prevention of Covert Transmission of Secrets.** Most existing defenses that focus on blocking covert transmission of secrets have primarily targeted the cache hierarchy rather than other side channels within the processor architecture.

Partitioning-based solutions [54]–[59] that defend against conflict-based side-channel attacks typically ensure isolation by organizing the cache (or any other microarchitectural structure) into multiple per-thread independent partitions, preventing conflicting accesses within the same partition. While this works well for a cross-core or a cross-SMT thread attack, it does not prevent attacks where information may be leaked across privilege levels, albeit within the same process. In fact, although the micro-op cache is partitioned in Intel processors, it remains vulnerable to most of our attacks, including our Spectre variants.

Encryption and randomization-based solutions [54], [60]–[64] randomize the indexing and cache filling procedure to impede eviction set construction. These defenses are specifically targeted at traditional caches, and don't directly apply to our attack. First, our attack does not cause any additional misses in the ICache or other levels of the cache hierarchy, and we confirm this through performance counter measurements. This is because micro-op cache misses only trigger a fetch from the ICache. As a result, randomized indexing of the ICache neither affects how the micro-op cache is accessed nor affects how it gets filled. Second, extending the randomized indexing process to the micro-op cache would not only provide very low entropy due to fewer sets, but can be extremely expensive as, unlike traditional caches, micro-op caches are implemented as streaming caches and the filling procedure is governed by numerous placement rules as described in Section II. Doing so in a cost-effective way while providing high levels of entropy is a challenge that needs to be addressed as part of future work.

More recent defenses against transient execution attacks prevent leaking secrets accessed along a misspeculated path. These fall into two major categories – (a) invisible speculation techniques, and (b) undo-based techniques. Solutions that leverage invisible speculation [20], [21], [24], [28] delay speculative updates to the cache hierarchy until a visibility point is reached (e.g., all older branches have been resolved). On the other hand, undo-based solutions rely on reversing speculative updates to the cache hierarchy once misspeculation is detected.

Our attack is able to completely penetrate all of these solutions as they primarily prevent covert transmission of information over traditional caches (and in some cases, arithmetic units, albeit with high performance overhead). More specifically, we never evict our *tiger* and *zebra* functions out of the ICache or any other lower-level cache, and thus we are able to send and receive information while leaving no trace in any of those caches.

**Prevention of Unauthorized Access to Secret Data.** The literature contains a vast amount of prior work on memory safety solutions [65]–[68] that prevent unauthorized access to secret data – these range from simple bounds checks to more sophisticated capability machines [69]–[74]. Transient execution attacks have the ability to successfully bypass software bounds checks by mistraining a branch predictor to temporarily override them. The remaining hardware-based solutions including capability machines that provide more fine-grained memory safety remain as vulnerable to our attack as they are to Spectre, since the offending instructions that make such unauthorized accesses get squashed in the pipeline, before an exception can be handled.

More recently, many hardware and software-based fencing solutions [23], [75], [76] have been proposed to inhibit speculation for security-critical code regions to prevent the unauthorized access of secret information. These solutions typically rely on injecting speculation fences before a load instruction that may potentially access secret data, thereby ensuring that the load instruction gets dispatched to execution only after all older instructions are committed. These defenses are particularly effective in preventing the unauthorized access of secret data in case of existing transient execution attacks. However, they do not defend against the transient attack variant-2 that we describe.

While taint analysis can detect data flow turning into control flow, existing defenses [21], [77], [78] prevent secret-dependent instructions from being dispatched to execution, but continue to allow them to be speculatively fetched, renamed, and added into the instruction queue, thereby leaving a footprint in the micro-op cache.

STT [21] and DOLMA [78] prevent instructions along a misspeculated path from updating the branch predictor, but they still allow instructions accessed along correctly speculated paths (those that eventually get committed) to update the branch predictor, as they consider tracking the information flow of non-transiently accessed secrets out of scope. However, as noted in our variant-2 attack, those instructions essentially implicitly encode the secret in the branch predictor. In general, we believe that this is hard to avoid without incurring a significant performance penalty, unless secret-dependent instructions are annotated as unsafe (either by the programmer or through an automatic annotation mechanism), so they don't accidentally get encoded in the predictor or any other microarchitectural structure.

Once the secret gets encoded in the predictor, our attack is able to leverage a single transient indirect branch instruction to first implicitly access the secret by reading the predicted target address from the predictor and then transmit it via the micro-op cache.

## VIII. Potential Mitigations

This section discusses potential attack mitigations that could block information leakage over the micro-op cache.

**Flushing the Micro-Op Cache at Domain Crossings.** Cross-domain information leakage via this side channel may be prevented by flushing the micro-op cache at appropriate protection domain crossings. This can be simply accomplished with current hardware by flushing the instruction Translation Lookaside Buffer (iTLB), which in turn forces a flush of the entire micro-op cache. Intel SGX already does this at enclave entry/exit points, and as a result both the enclave and the non-enclave code leave no trace in the micro-op cache for side-channel inference.

However, frequent flushing of the micro-op cache could severely degrade performance. Furthermore, given that current processors require an iTLB flush to achieve a micro-op cache flush, frequent flushing of both structures would have heavy performance consequences, as the processor can make no forward progress until the iTLB refills. While it is possible to selectively flush cache lines in the micro-op cache by flushing the corresponding lines in the instruction cache at appropriate protection domain crossings, that may not be tractable for two reasons. First, by flushing the cache lines in both the ICache and the micro-op cache, we would significantly slow down the fetch and decode process for hot code regions, making protection domain switches considerably more expensive than they already are. Second, selective flushing necessarily requires the continuous monitoring and attribution of micro-op cache lines in hardware, which may not be tractable as the number of protection domains increase.

**Performance Counter-Based Monitoring.** A more lightweight alternative to disabling and flushing the micro-op cache is to leverage performance counters to detect anomalies and/or potential malicious activity in the micro-op cache. For instance, sudden jumps in the micro-op cache misses may reveal a potential attack. However, such a technique is not only inherently prone to misclassification errors, but may also be vulnerable to *mimicry* attacks [79], [80] that can evade detection. Moreover, to gather fine-grained measurements, it is imperative that performance counters are probed frequently, which could in turn have a significant impact on performance.

**Privilege Level-Based Partitioning.** Intel micro-op caches are statically partitioned to allow for performance isolation of SMT threads that run on the same physical core, but different logical cores. However, this does not prevent same address-space attacks where secret information is leaked within the same thread, but across protection domains, by unauthorized means. A countermeasure would be to extend this partitioning based on the current privilege-level of the code, so for example, kernel and user code don't interfere with each other in the micro-op cache. However, this may not be scalable as the number of protection domains increase, and considering the relatively small size of the micro-op cache, such a partitioning scheme would result in heavy underutilization of the micro-op cache, negating much of its performance advantages. Moreover, it does not prevent our variant-1 attack, as both the priming and the probing operations are performed in user space, even though they might be triggered by a secret that is returned by kernel code.

## IX. Conclusion

This paper presents a detailed characterization of the micro-op cache in Intel Skylake and AMD Zen microarchitectures, revealing details on several undocumented features. The paper also presents new attacks that exploit the micro-op cache to leak secrets in three primary settings: (a) across the user-kernel boundary, (b) across co-located SMT threads running on the same physical core, but different logical cores, and (c) two transient execution attack variants that exploit the micro-op cache timing channel, bypassing many recently proposed defenses in the literature. Finally, the paper includes a discussion on the effectiveness of the attack under existing mitigations against side-channel and transient execution attacks, and further identifies potential mitigations.

## Acknowledgments

The authors would like to thank the anonymous reviewers for their many helpful suggestions and comments. The authors would also like to thank Paul Kocher for his insightful feedback and encouraging comments. This research was supported by NSF Grant CNS-1850436, NSF/Intel Foundational Microarchitecture Research Grants CCF-1823444 and CCF-1912608, and a DARPA contract under the agreement number HR0011-18-C-0020.

## References

[1] Y. Yarom, D. Genkin, and N. Heninger, “CacheBleed: A Timing Attack on OpenSSL Constant Time RSA,” in *Journal of Cryptographic Engineering*, 2016.

[2] Y. Yarom and K. Falkner, “FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack,” in *USENIX Security Symposium*, 2014.

[3] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, “Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX,” in *USENIX Security Symposium*, 2017.

[4] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications,” in *ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[5] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-Level Cache Side-Channel Attacks Are Practical,” in *IEEE Symposium on Security and Privacy*, 2015.

[6] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks,” in *USENIX Security Symposium*, 2018.

[7] R. Hund, C. Willems, and T. Holz, “Practical Timing Side Channel Attacks against Kernel Space ASLR,” in *IEEE Symposium on Security and Privacy (SP)*, 2013.

[8] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, “Jump over ASLR: Attacking Branch Predictors to Bypass ASLR,” in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[9] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, “BranchScope: A New Side-Channel Attack on Directional Branch Predictor,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[10] M. Taram, A. Venkat, and D. Tullsen, “Packet Chasing: Spying on Network Packets over a Cache Side-Channel,” in *International Symposium on Computer Architecture (ISCA)*, 2020.

[11] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre Attacks: Exploiting Speculative Execution ,” Tech. Rep., 2019.

[12] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading Kernel Memory from User Space,” Tech. Rep., 2018.

[13] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, “ZombieLoad: Cross-Privilege-Boundary Data Sampling,” in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[14] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, “RIDL: Rogue in-flight data load,” in *in IEEE Symposium on Security and Privacy (SP)*, 2019.

[15] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, “Fallout: Leaking data on meltdown-resistant cpus,” in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[16] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, “SGAxe: How SGX fails in practice,” https://sgaxeattack.com/, 2020.

[17] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution,” in *USENIX Security Symposium*, 2018.

[18] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. Abu-Ghazaleh, C. Song, and M. Payer, “Specrop: Speculative exploitation of ROP chains,” in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[19] V. Kiriansky and C. Waldspurger, “Speculative buffer overflows: Attacks and defenses,” 2018.

[20] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, “InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy,” in *International Symposium on Microarchitecture (MICRO)*, 2018.

[21] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, “Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data,” in *International Symposium on Microarchitecture (MICRO)*, 2019.

[22] G. Saileshwar and M. K. Qureshi, “CleanupSpec: An "Undo" Approach to Safe Speculation,” in *International Symposium on Microarchitecture (MICRO)*, 2019.

[23] M. Taram, A. Venkat, and D. Tullsen, “Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[24] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, “SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation,” in *Design Automation Conference (DAC)*, 2019.

[25] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, “SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels,” in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[26] S. Ainsworth and T. M. Jones, “MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State,” 2020.

[27] Z. N. Zhao, H. Ji, M. Yan, J. Yu, C. W. Fletcher, A. Morrison, D. Marinov, and J. Torrellas, “Speculation Invariance (InvarSpec): Faster Safe Execution Through Program Analysis,” in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[28] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, “Efficient Invisible Speculative Execution through Selective Delay and Value Prediction,” in *International Symposium on Computer Architecture (ISCA)*, 2019.

[29] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, “NDA: Preventing Speculative Execution Attacks at Their Source,” in *International Symposium on Microarchitecture (MICRO)*, 2019.

[30] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, “I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing,” in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[31] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, “Understanding Selective Delay as a Method for Efficient Secure Speculative Execution,” *IEEE Transactions on Computers*, 2020.

[32] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, F. Mckeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. Alameldeen, “Speculative interference attacks: Breaking invisible speculation schemes,” 2020.

[33] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, “Inside 6th-generation intel core: New microarchitecture code-named skylake,” *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[34] B. Solomon, A. Mendelson, D. Orenstien, Y. Almog, and R. Ronen, “Micro-operation Cache: a Power Pware Front-end for Variable Instruction Length ISA,” in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2001.

[35] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, “Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake,” 2017.

[36] S. Manne, A. Klauser, and D. Grunwald, “Pipeline gating: Speculation Control for Energy Reduction,” in *International Symposium on Computer Architecture (ISCA)*, 1998.

[37] Intel Corporation, *Intel*$^{®}$ *64 and IA-32 Architectures Optimization Reference Manual*, March 2009.

[38] I. Corporation, "Intel® 64 and IA-32 Architectures Optimization Reference Manual," Tech. Rep., 2019. [Online]. Available: http://www.intel.com/design/literature.htm.

[39] O. Aciiçmez, "Yet Another Microarchitectural Attack:: Exploiting I-cache," in *ACM Workshop on Computer Security Architecture (CSAW)*, 2007.

[40] Z. Wu, Z. Xu, and H. Wang, "Whispers in the Hyper-Space: High-Bandwidth and Reliable Covert Channel Attacks inside the Cloud," in *USENIX Security Symposium (USENIX Security)*, 2015.

[41] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri, "Port Contention for Fun and Profit," in *IEEE Symposium on Security and Privacy (SP)*, 2019.

[42] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On Subnormal Floating Point and Abnormal Timing," in *IEEE Symposium on Security and Privacy(SP)*, 2015.

[43] Z. Wang and R. B. Lee, "Covert and Side Channels Due to Processor Architecture," in *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[44] M. SWIAT, "Mitigating speculative execution side channel hardware vulnerabilities," Mar 2018. [Online]. Available: https://msrc-blog.microsoft.com/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/

[45] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting Speculative Execution through Port Contention," in *Conference on Computer and Communications Security (CCS)*, 2019.

[46] A. Abel and J. Reineke, "nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.

[47] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," in *International Symposium on Microarchitecture (MICRO)*, 1996.

[48] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *International Symposium on Microarchitecture (MICRO)*, 2006.

[49] Intel Corporation, *Intel® 64 and IA-32 Architectures Performance Monitoring Events*, March 2009.

[50] V. Guruswami, *Decoding Reed–Solomon Codes*. Springer US, 2008, pp. 222–226.

[51] S. Briongos, G. Irazoqui, P. Malagón, and T. Eisenbarth, "CacheShield: Detecting Cache Attacks through Self-Observation," in *Conference on Data and Application Security and Privacy (CODASPY)*, 2018.

[52] T. Zhang, Y. Zhang, and R. B. Lee, "CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds," in *Research in Attacks, Intrusions, and Defenses (RAID)*, 2016.

[53] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2B*, Intel Corporation, August 2007.

[54] Z. Wang and R. B. Lee, "New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks," 2007.

[55] K. T. Nguyen, "Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family," 2016, https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html.

[56] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating Last-level Cache Side Channel Attacks in Cloud Computing," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[57] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks," 2012.

[58] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: secure dynamic cache partitioning for efficient timing channel protection," in *Design Automation Conference (DAC)*, 2016.

[59] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors," in *International Symposium on Microarchitecture (MICRO)*, 2018.

[60] M. K. Qureshi, "CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping," in *International Symposium on Microarchitecture (MICRO)*, 2018.

[61] M. K. Qureshi, "New Attacks and Defense for Encrypted-Address Cache," in *International Symposium on Computer Architecture (ISCA)*, 2019.

[62] F. Liu and R. B. Lee, "Random Fill Cache Architecture," in *International Symposium on Computer Architecture (ISCA)*, 2014.

[63] F. Liu, H. Wu, and R. B. Lee, "Can randomized mapping secure instruction caches from side-channel attacks?" in *Fourth Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2015.

[64] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," 2016.

[65] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic Memory Safety for Unsafe Languages," in *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[66] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "HeapHopper: Bringing Bounded Model Checking to Heap Implementation Security," in *USENIX Security Symposium (USENIX Security)*, 2018.

[67] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[68] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack," in *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 2018.

[69] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI Capability Model: Revisiting RISC in an Age of Risk," in *International Symposium on Computer Architecture (ISCA)*, 2014.

[70] B. Davis, R. N. M. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka, A. Joannou, B. Laurie, A. T. Markettos, J. E. Maste, A. Mazzinghi, E. T. Napierala, R. M. Norton, M. Roe, P. Sewell, S. Son, and J. Woodruff, "CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[71] N. W. Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. T. Napierala, A. Richardson, J. Baldwin, D. Chisnall, J. Clarke, K. Gudka, A. Joannou, A. T. Markettos, A. Mazzinghi, R. M. Norton, M. Roe, P. Sewell, S. Son, T. M. Jones, S. W. Moore, P. G. Neumann, and R. M. Watson, "Cornucopia: Temporal Safety for CHERI Heaps," in *IEEE Symposium on Security and Privacy (SP)*, 2020.

[72] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware Support for Fast Capability-Based Addressing," in *ACM SIGOPS Operating Systems Review*, 1994.

[73] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, and A. DeHon, "Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-Based Security," in *SIGSAC Conference on Computer & communications security (CCS)*, 2013.

[74] R. Sharifi and A. Venkat, "CHEx86: Context-Sensitive Enforcement of Memory Safety via Microcode-Enabled Capabilities," in *Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[75] Intel, "Speculative Execution Side Channel Mitigations," 2018. [Online]. Available: www.intel.com/benchmarks.

[76] G. Wang, "oo7: Low-Overhead Defense against Spectre Attacks via Program Analysis," Tech. Rep., 2019.

[77] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[78] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, "DOLMA: Securing speculation with the principle of transient non-observability," in *USENIX Security Symposium (USENIX Security)*, 2021.

[79] D. Wagner and P. Soto, "Mimicry Attacks on Host-Based Intrusion Detection Systems," in *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2002.

[80] K. N. Khasawneh, N. Abu-Ghazaleh, D. Ponomarev, and L. Yu, "RHMD: Evasion-Resilient Hardware Malware Detectors," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[81] A. Fog, "The Microarchitecture of Intel, AMD and VIA CPUs: An Optimization Guide for Assembly Programmers and Compiler Makers." [Online]. Available: https://www.agner.org/optimize/microarchitecture.pdf