Data-Out Instruction-In (DOIN!): Leveraging Inclusive Caches To Attack Speculative Delay Schemes

Abstract—**Although the cache has been a known side-channel for some time, it has gained renewed notoriety with the introduction of speculative side-channel attacks such as Spectre. Because the cache continues to be one of the most exploitable side channels, it is often the primary target to safeguard in secure speculative execution schemes. One of the simpler secure speculation approaches is to delay speculative accesses whose effect can be observed until they become non-speculative. Delay-on-Miss, for example, delays all observable speculative loads, i.e., the ones that miss in the cache, and preserves the majority of the performance of the baseline (unsafe speculation) by executing speculative loads that hit in the cache, which were thought to be unobservable.**

In this work, we present a new cache attack, *DOIN!*, that leverages *cache inclusivity* to modify the *data cache* through interference caused by instruction fetch at a shared cache level. The attack uses *instructions* that conflict with *data* in a shared cache, e.g., the LLC, causing data evictions in the L1D, thus, creating observable timing differences. We demonstrate *DOIN!* on real commercial cores. Additionally, we show how a speculative version of *DOIN!* challenges assumptions about cache side-channels by breaking the security guarantee provided by Delay-on-Miss. Furthermore, we propose a simple defense to maintain the security Delay-on-Miss at the cost of negligible performance degradation for Delay-on-Miss executing the SPEC06 workloads.

Index Terms—**Speculative side-channels, cache side-channels, Spectre, security**

I. Introduction

Caches—a well-known side channel [17]—provide one of the key methods by which the memory hierarchy can be exploited to leak information. The introduction of Spectre [12] and other speculative side-channel attacks, has demonstrated how a core can be tricked into accessing secrets through erroneous speculative execution and leak them through side channels. The combination of speculative execution attacks and caches as a side-channel has become a serious design problem for processor designers, due to the necessity of fast caches and the variety of speculative side-channel attacks. Caches as a side-channel have been explored in many works [9], [10], [15], [31], and the discoveries have given speculative side-channel attacks [7] easier methods by which to leak data through the memory hierarchy. Several mitigations have been proposed, both for speculative side-channel attacks [3], [4], [11], [19], [21], [28], [32] and for caches as a generic side-channel [14], [18], [26], [29].

Speculative side-channel attacks exploit transient instructions, instructions that are erroneously executed and are guaranteed to be squashed. These instructions, resulting from speculative wrong path execution or delayed exception handling, are able to perform potentially dangerous memory accesses before speculation is resolved, and they are squashed. Although misspeculation is always eventually detected and architectural state may be fully reverted (e.g., registers), the microarchitectural state is not reverted (e.g., locations of cache lines in memory hierarchy). Side-channels that are able to expose this information non-speculatively can then leak secrets by, for example, observing timing on cache lines.

Instructions that create observable microarchitectural changes are called transmitters. Loads are some of the most important transmitters due to how easy it is to observe their changes and the relatively high bandwidth they enable for covert side-channel communication. There are different approaches to mitigate these observable effects: some schemes focus on hiding speculation [4], [28], others on delaying execution [21], [32], while others implement undo-based speculation, allowing speculation to proceed and undoing the effects [19]. While hiding speculation and undo-based speculation schemes are theoretically elegant solutions, they are costly to implement due to requiring many changes to the memory hierarchy. Delaying speculative execution is appealing, as selectively delaying instructions can limit performance loss. However, complexity varies from proposal to proposal.

Delay-on-Miss is a *delay* approach that focuses exclusively on preventing information leakage through the speculative cache side-channel. Delay-on-Miss, as the name suggests, delays all speculative loads that miss in the L1D cache. The key idea is that misses in the cache hierarchy are the only accesses that create observable timing differences. Accesses that hit in L1D cache are allowed to execute, since their side effects (e.g., updates to the replacement policy) can be deferred until after the speculation has been verified. Delay-on-Miss introduces allowing unobservable (with respect to the caches) execution to proceed, achieving notable performance gains, compared to the earlier InvisiSpec [28]. The design principles behind Delay-on-Miss have been adopted and extended (e.g., DOLMA [16]), and different optimizations have been investigated (e.g., InvarSpec [33], Clearing the Shadows [24]). Additionally, it has also been the target for new attacks (e.g., Speculative Interference [5], InvarSpec+Reorder Buffer Contention [2]).

However, despite the protection that Delay-on-Miss promises to offer for the data cache side-channel and for speculative loads (not allowing them to change the L1D *cache), it is unable to completely close the data cache side-channel as it allows the indirect modification of the L1D via instruction fetch, as we demonstrate.*

In this paper, we present a new variant of last level cache (LLC) attacks, called Data-Out–Instruction-In *DOIN!* that exploits inclusive caches. We show how *instruction fetching* can create conflicts with *data cache lines* in the LLC, resulting in the invalidation of cache lines in the *L1D Cache* and creating observable timing differences. The attacker fills caches with data ("prime" step), and waits for a period of time. The victim then misses while attempting to fetch instructions in the L1I Cache, resulting in the processor fetching cache lines from main memory to L1I and LLC (due to cache inclusivity). These instruction cache lines map to the same set of cache lines in LLC as the data that the attacker used to prime the L1D Cache (and were placed in the LLC because of inclusivity), leading to an invalidation from the LLC to the L1D Cache. The attacker then measures the data access ("probe" step) and detects whether it was a cache hit or a cache miss, discovering the behavior of the victim.

We extend this attack to use speculative side-channels, so that we can leak secrets that were accessed from memory speculatively. The attacker behaves the same as before by priming and probing the cache. This time, the secret is loaded speculatively, and a secret-dependent branch is executed, forcing a secret-dependent path to load an instruction that conflicts in the LLC with the data the attacker primed, leading to an evicting from the attacker's L1D cache. Afterwards, even as the transient executions have been squashed, the attacker can perform the probe step to observe a cache hit or a miss, leaking the binary value of the secret.[1]

We demonstrate the attack on an AMD Ryzen 9 processor and show how this attack breaks Delay-on-Miss. Delay-on-Miss allows control flow to depend on a speculatively accessed secret (as long as the access is a hit in the cache), and this proves to be its weakness as instruction fetch can affect changes in the data caches.

Finally, we present *DONOT!*, a mitigation to *DOIN!* that restores Delay-on-Miss security guarantees. *DONOT!* extends the Delay-on-Miss data delay premise and applies it also to instructions: instructions that miss in cache are delayed until they are guaranteed not to be squashed. The proposed mitigation introduces negligible performance slowed own for the SPEC06 compared to the unmodified Delay-on-Miss.

II. Background

Cache side-channels have been a focal point for security research for many years, while speculative side-channel attacks were only revealed in 2018. In this section, we introduce the background for both caches and speculative side-channel attacks, which contributed to the creation of *DOIN!*.

[1]An analogous attack can be mounted by swapping the role of instructions and data, but this form of attack would not work with the Delay-on-Miss defenses.

A. Last Level Cache Prime+Probe

Prime+Probe [17] is a cache side-channel attack that is able to observe changes in specific cache lines. The idea behind the attack is that the attacker loads cache lines into a cache set (step prime), waits for a certain period of time, and then measures the access time to the cache lines in this set (step probe). The cache set conflicts with a potential cache line that the victim may or may not access. Depending on the time required for the accesses (hit or miss), the attacker is able to say if the victim accessed this set while the attacker was waiting, as any access from the victim will evict a cache line from the cache set and change timing.

Last level cache Prime+Probe [15] applies the Prime+Probe attack to the LLC. This technique enables the attacker to attack a victim operating on a different core, as the LLC is shared between different cores. Information leakage occurs when the victim evicts data that the attacker primed in the LLC.

B. Eviction sets

Cache lines are replaced according to the size of an *eviction set*. To replace all the addresses in a single set we need to map as many addresses as *ways* to that specific set. *Eviction sets* [23], [25] is a technique that does exactly this: provides as many virtual addresses as *ways* in a cache set that map to the same set in the unified (data+instruction) cache. When all those virtual addresses are accessed, they will clear all other cache lines that were mapped to the same address. Thus, an *eviction se*t is guarantees that the all previous contents in that particular LLC set have been cleared. The attacker has full control of the cache state, and can later probe that set to examine if the victim accessed it, as any access would evict at least one of the attackers cache lines.

C. Speculative Side-Channel Attacks

Speculative side-channel attacks leverage speculative execution to leak data that would otherwise be inaccessible. A processor can access data it is normally not able to during speculation, due to misspeculation allowing incorrect execution. In this class of attacks, the attacker exploits predictors or exception handling to make the processor execute incorrect instructions, and access data that is inaccessible in non-speculative execution.

Figure 1 illustrates how a typical speculative attack using a predictor works. The attacker goes through the setup phase: trains the branch predictor to always mispredict and flushes a probe array from the cache. Probe array is the array that will be used to transmit the secret and create observable timing differences into the non-speculative worlds. Because the branch predictor is trained to assume the illegal access is allowed, the attacker makes an out-of-bounds illegal access, fetching a value from an otherwise inaccessible address, i.e., acquiring a secret. To leak the value of the secret, the attacker passes it as an address into the probe_array, accessing a specific cache line. Depending on the secret value, a specific cache line will be moved from lower in the memory hierarchy (L2, L3) upwards, closer to the core (L1). Measuring the access latency of that

```
void access_array(int index){
 if(index < array_size)
  secret = array[index];
}

void train(){
 for(i=0; i<100; i++)
  access_array(0);
}

void attack() {
 char probe_array[N * CACHE_LINE_SIZE];
 train();
 flush(&probe_array);
 secret = access_array(secret_location);
 x = probe_array[secret * CACHE_LINE_SIZE];
 probe(&probe_array);
}
```

Fig. 1. Spectre V1.

cache line even after speculative execution has been squashed, the value of the secret can be acquired. The cache line that was accessed using the value of the secret will take significantly less time than the others, as it will hit, in contrast to the other cache lines which were flushed before the attack and are still missing from the L1 cache. Thus, when we time all the possible cache lines of the probe array after misspeculation is verified, the cache line with an address corresponding to the secret *value* would hit (lower access time) and the rest would miss.

D. Delay-on-Miss

Delay-on-Miss [21] is a safe speculation scheme that aims to block the cache hierarchy as a speculative side-channel. Its threat model covers only the speculative cache side channels. Delay-on-Miss modifies the execution of speculative load instructions, delaying all speculative loads that miss in the L1D cache, while allowing hits to execute, but always delaying any observable side effects, such as updating replacement policies, a speculative load may impose on the microarchitecture.

Delay-on-Miss uses *speculative shadows* to track the speculative state of instructions efficiently. Every instruction that might trigger a speculative state, either prediction or delayed exception handling, casts a shadow. Every instruction placed in the ROB after a shadow-casting instruction is said to be under a shadow. A shadow is lifted once the instruction that casts the shadow is guaranteed to commit and not squash the following instructions. The following shadows are introduced:

- `E-Shadows`: are cast by instructions that may cause an exception, such as memory operations with unknown address and arithmetic operations.
- `C-Shadows`: are cast by speculative control-flow instructions, such as branches and jumps.
- `D-Shadows`: are cast by store instructions with unresolved addresses that may have memory dependencies. If undetected aliasing occurred, this triggers an exception and requires the processor to rollback.
- `M-Shadows`: are cast by load instructions when they may be violating load ordering in some memory models (e.g, under TSO).

Delay-on-Miss is an elegant concept that requires limited hardware modifications. Following Delay-on-Miss, several other works block more side-channels [16], [22] or propose performance optimizations [24], [33].

III. Threat Model

In this section, we describe our threat model, and the conditions under which *DOIN!* can successfully attack. For non-speculative *DOIN!*, we assume the same threat model as Prime+Probe. For this version of *DOIN!*, a successful attack involves observing timing differences in the cache hierarchy and being able to gleam information about the execution of the program of the victim.

For speculative *DOIN!*, we assume that the attacker is executing under normal user-permissions on an out-of-order processor and wishes to access data not belonging to its process, i.e., a secret. We assume a lenient threat model in favor of Delay-on-Miss and mitigations: the attacker cares only about the cache as a side-channel. For evaluation, we evaluate only explicit speculation through control-flow instructions, i.e., C-shadows, but *DOIN!* works with any kind of speculation. We use a strict definition of leakage as the persistence of secrets after transient execution has been squashed. This means that the secret must be recoverable after speculation has been squashed and normal execution resumed.

For speculative *DOIN!*, a successful attack involves misspeculation to access a secret, and using this secret to trigger dependent control flow that results in either a conflicting eviction or no eviction, which reveals the value of the secret. The secret can be recovered after squashing by timing the access to the potentially conflicted cache line.

Cache side-channels are used as the only side-channel consideration for these attacks due to being among the most noise-resistant and high-bandwidth side-channels. *DOIN!* demonstrates how previous secure speculative execution schemes are unable to comprehensively eliminate speculative side-channel attacks, due to their interactions with other aspects of the cache hierarchy, such as inclusivity and instruction fetch. This is likely to complicate mitigation efforts against attacks such as Prime+Probe, as direct accesses to data caches are not necessary to observe the victim program.

Other side-channels, such as port contention, timing-inversion, or physical attribute (e.g., power or EMF) side-channels are not considered for this work as they also fall outside the scope of the original Delay-on-Miss strategy. Delay-on-Miss only attempts to mitigate timing differences in the memory hierarchy (including coherence directories and DRAM) as the prime speculative side-channel, due to its ubiquity and relative ease-of-use. Other side-channels are known to be able to leak data under Delay-on-Miss, but at a reduced bandwidth compared to the unsafe baseline. We achieve similar results here, but directly use the timing differences in the cache hierarchy, which Delay-on-Miss was designed to protect against.

IV. Data-Out Instruction-In (DOIN!)

Lower level caches are often inclusive, containing copies of higher level cache elements, and evicting the copy from the lower level cache also evicts it from the higher level cache. Inclusive caches are simpler to design than non-inclusive or exclusive caches in terms of coherence and in particular with respect to invalidation. Because they avoid complications and cost associated to the coherence implementation, inclusive caches are appealing —and typically found— in cost-effective commercial products for the consumer market (e.g., laptop and desktop processors). *DOIN!* leverages cache inclusivity to perform a combined data and instruction attack. In this section, we describe how *DOIN!* functions non-speculatively and speculatively.

A. Non-speculative DOIN!

Non-speculative *DOIN!* extends previous LLC attacks such as Prime+Probe, by offering a new method of observing memory access patterns of a victim through instruction interference. Non-speculative *DOIN!* consists of an attacker and a victim, in which the attacker wishes to gleam information about the execution of the victim's program. These sorts of attacks have implications for the security of encryption implementations, amongst other concerns. Non-speculative *DOIN!* consists of the following four main steps: HMS: MINOR, but is it clear how an eviction set is loaded since the L1D commonly has less associativity than the LLC.

1) Prime the cache by accessing an entire eviction set, loading the set into the L1D cache (and the LLC due to inclusivity).
2) Wait for the victim to execute its program, which includes a secret-dependent instruction fetch.
3) In the case of an instruction fetch, it will cause a conflict in the eviction set, and evict data from the L1D cache, creating a timing difference.
4) The attacker periodically probes the cache by timing the entire eviction set in the L1D. A longer access time on any part of the set indicates secret = 1, else secret = 0.

Unlike other LLC attacks, here, the attacker avoids directly accessing the same memory types as the victim, and can instead merely conflict indirectly through the instruction-data conflicts occurring in the LLC. The attack's advantage is that previous detection mechanisms that might mitigate observable timing differences on conflicting data accesses, might not be designed to detect such conflicts originating from the fetch part of the processor. This is, to the best of our knowledge, the first time that a combined attack using both data and instructions to create conflicts in the LLC that result in timing differences in a separate L1 cache has been presented. In the past, there have been exploits that use the data cache through data [17] and instruction cache through instructions [1], but no works that exploit the data cache using instructions and the instruction cache using data

B. Speculative DOIN!

In this section, we focus on speculative *DOIN!*, as this particular speculative attack is able to break the security guarantees of Delay-on-Miss. The speculative and non-speculative versions consists of many similarities, but the non-speculative version does not require a victim program, and rather uses *DOIN!* to avoid the mitigation introduced by Delay-on-Miss.

The attack consists of five main steps:

1) Prime the cache by accessing an entire eviction set, and loading the set into the L1D cache (and the LLC due to inclusivity).
2) Access a secret during speculation and perform a secret-dependent instruction fetch. Secret-dependent control flow does not use the cache directly, and therefore avoids Delay-on-Miss.
3) In the case of an instruction fetch, it will cause a conflict in the eviction set, and evict data from the L1D cache, creating a timing difference.
4) Let the misprediction be resolved and the squash to complete, relinquishing the secret, but otherwise not affecting the cache hierarchy.
5) Probe the entire eviction set in the L1D cache. A longer access time on any part of the set indicates secret = 1, else secret = 0.

Figure 2 illustrates each step of a successful attack. The attacker makes a read request and brings the data `X` into the cache hierarchy, both in the L1D cache and the LLC, due to inclusivity. A conflicting instruction that misses in the L1I cache is fetched, and brings `Y` into the cache hierarchy (both the L1I cache and the LLC). `Y` conflicts with `X` on the same set in LLC, so `X` sends an invalidation to the L1D cache, which then evicts `X` out of the L1D cache. The attacker now probes `X`, and observes a miss.

V. Breaking Delay-on-Miss

The goal of secure speculative execution schemes is to protect against information leakage under speculation. Delay-on-Miss proposed a complexity-effective solution, noting that speculative loads that *miss* are the memory accesses that create observable timing differences through the memory hierarchy, covering a threat model only considering cache side-channels. Because Delay-on-Miss prevents all data cache misses while under speculation, it was assumed that control-flow would be unable to create cache-side channels through implicit channels. An implicit channel uses the secret value to create differences in control flow, and thus observable timing differences [32]. Delay-on-Miss allows speculatively accessed secrets that hit in the cache to be used for the formation of implicit side-channels.

However, speculative interference [5] has shown that it is possible for an implicit channel to affect the timing (and ordering) of non-speculative instructions that precede the speculation. This case was not considered by Delay-on-Miss and although the instructions that are affected are non-speculative, it is an inherent weakness, because it enables speculatively

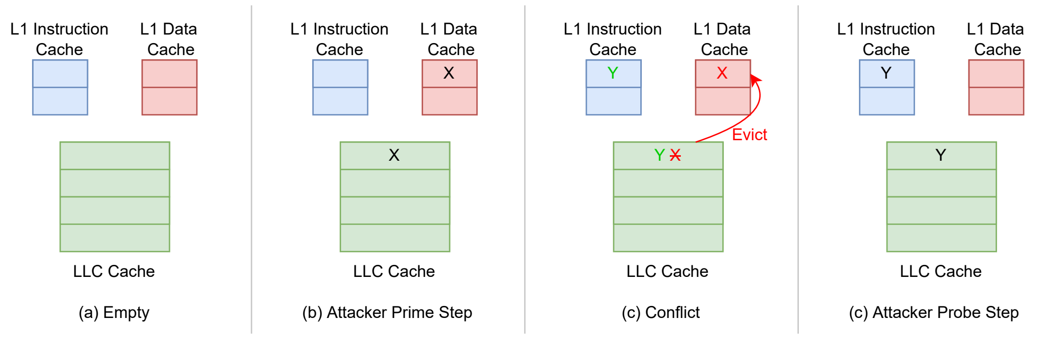


Fig. 2. Steps of *DOIN!*: (a) The cache is empty. (b) Attacker primes the L1D cache. (c) Conflict with the previously cached data in the LLC. The instruction replaces the data in the LLC and evicts the data from the L1D cache. (d) Attacker probes the data in the L1D cache.

accessed secrets to be used in implicit side-channels. These issues have been addressed in a follow-up work that still enables Delay-on-Miss to use secrets in implicit side-channels, but prevents speculative interference by preserving priority among younger and older instructions [22].

Understanding this, we show that allowing secrets to be used in implicit side channels can still create observable differences in data caches, thus breaking Delay-on-Miss in its own threat model. More specifically, we show that although it does not allow misses in the data cache, such misses can be indirectly caused by instruction fetches, which in turn can be driven by a secret dependent implicit side-channel.

DOIN! is an attack that, instead of forming an explicit channel, forms an implicit channel and leaks information through interference in the memory hierarchy. As discussed in Section IV-B, the instruction can contest the same cache set as the data, and evict it both from the LLC and the L1D cache. An attack using only the instruction cache as a side channel would also be able to break Delay-on-Miss, but an instruction only focused attack is considered outside the scope of Delay-on-Miss. Instead, *DOIN!* uses instructions, but still leaks secret values through observable timing differences in the L1D cache, which Delay-on-Miss explicitly is supposed to protect against.

VI. Attack Demonstration

HMS: Avoid consecutive titles by adding some text between them.

A. Actual Processor: AMD Ryzen 9

We now delve into the details on how we make the attack function on a state-of-the-art processor, the AMD Ryzen 9, which has an inclusive L2 cache. The attack is able to leak the value of a secret a single bit at a time under speculation using a secret-dependent branch. The attack is designed to control precisely which instructions are executed under speculation, and aims to conflict with a specific eviction set in the L2. Caches lower than the L1 are big enough to hold whole pages, and we aim to create a conflict between instructions and data at an exact position in a page.

HMS: We perform? at what point of time? During the execution of the attack itself? I assume it's how to calculate the

```
touch = data[0];
if(mispredict){
    if(secret){
        jmp inst_addr2; // PC: 0x(inst_addr1)
        nops; // until 0xinst_addr2 is created
        jmp inst_addr3; // PC: 0x(inst_addr2)
        nops; // until 0xinst_addr3 is created
        jmp inst_addr4; // PC: 0x(inst_addr3)
         ....
    }
}
measure(data[0]);
```

Fig. 3. Pseudo-code of the speculative *DOIN!* attack, on AMD Ryzen 9.

addresses to be used in an attack but this is done offline. We perform bitwise operations by isolating the index and offset bits of the primed data, and calculate all possible addresses that can map to the same position in a page, by iterating through the values of the most significant bits. With this, it is possible to navigate to the same place on all pages appearing in different cache sets. HMS: "same place on all pages" but different cache sets? My intuition says that the same page offset would be mapped to a single cache set (might of course be wrong). To ensure instruction addresses map correctly in the attack program, *nop* instructions are inserted until the actual conflicting addresses match the program counter. Then, using a chain of jumps, it is possible to link the conflicting addresses. By jumping from one conflicting address to the next, the processor fetches only instruction addresses that conflict and will eventually evict the primed data from the L2 cache.

HMS: I still don't understand the attack. Why is multiple conflicting instructions needed? Is it such that it is difficult to target one specific eviction set in the cache so one need to touch as many as possible, i.e., at least one of them will conflict in the eviction set? Else a single instruction fetch should be enough.

Figure 3 presents the pseudo-code behind the attack. *line 4* is the first instruction address that may conflict with data in L2. *nop* instructions are used so that the program has instructions with specific addresses that conflict (every time a *nop* is inserted the program counter changes). For example, the instruction address of *line 6* will be fetched, after a certain

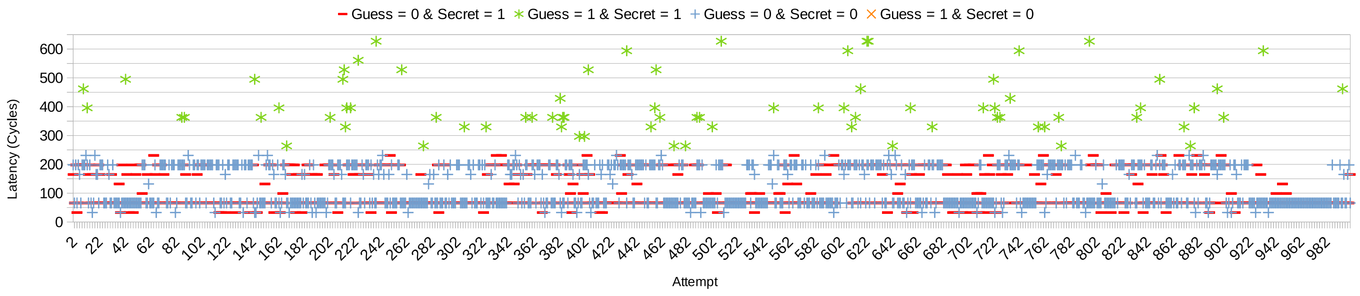


Fig. 4. Speculative *DOIN!* attack: one thousand attempts to guess the secret, on AMD Ryzen 9.

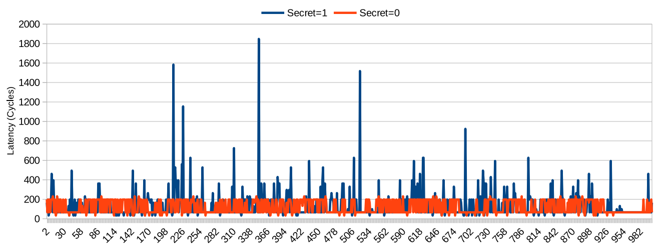


Fig. 5. Speculative *DOIN!* all one thousand attempts, on AMD Ryzen 9.

number of *nop* instructions. This specific address (*inst_addr2*) may conflict with the primed data in the L2. During the program execution when the *jmp* from *line 4* is executed, the next fetched instruction is at address (*inst_addr2*) that may also conflict with data, and so on. The jump chain continues until all possible instruction addresses that can conflict with the data are fetched in the front-end. Eventually this leads to the data being evicted from the L2 and, consequently, from the L1.

The asymmetry between the taken path and the non-taken (fall-through) path leaks information. Note that this is the simplest form of the attack. More complex attacks can be mounted where each path generates a different conflict miss. For simplicity, in this paper, we discuss the simple form of the attack, but it is straightforward to generalize.

1) Success Rate: Figure 4 shows the results of using the attack with a thousand repetitions. A threshold of 231 cycles differentiates between LLC hits and misses, as the noise between two consecutive *Read Time-Stamp Counter and Processor ID* cycle measurement function calls can be 33, 66, 198, or 231 cycles on AMD Ryzen 9. In this graph, accesses with latency longer than 650 cycles are omitted, so that the scaling of the graph is more visually intuitive. Figure 5 shows latencies for all one thousand attempts, including longer latencies. The results show the success on guessing the secret value by its latency on a single try. There are no false-positives when `secret` is equal to zero, as it always hits in the cache (no eviction takes place). On the other hand, when `secret` is equal to one, the success rate is 8.3% on a single access. The lack of false-positives when the value is zero, makes it possible to enhance the signal of the attack by replaying it repeatedly.

Since there is a $\frac{91.7}{100}$ chance of guessing wrong when secret is equal to one, it is necessary to try many times. If the attacker performs x number of attempts, the probability to succeed at least once is $p = (1 - (\frac{91.7}{100})^x)$. That means, that if the attack is performed 9 times then the chance of seeing a correct high value is $p = 0.541$. To ensure a strong signal, it is necessary to have a high likelihood of evaluating the secret correctly. For this purpose, if $x = 50$, the probability of getting a positive signal is $p = 0.986$. We consider 50 repetitions as a safe number of attempts before guessing the value of the secret. Within those repetitions, if the attack observes at least one cache miss, it assumes that the secret was one, otherwise it assumes that the secret was zero.

2) Bandwidth: To leak secrets with a high success rate, the bandwidth is calculated based on the execution time of the attack with a repetition factor of 50. Replaying the attack 50 times, *DOIN!* can leak secrets at a rate of 1.17 KiB/sec on an AMD Ryzen 9 running at XXXX GHz.

B. Simulation Proof-of-Concept

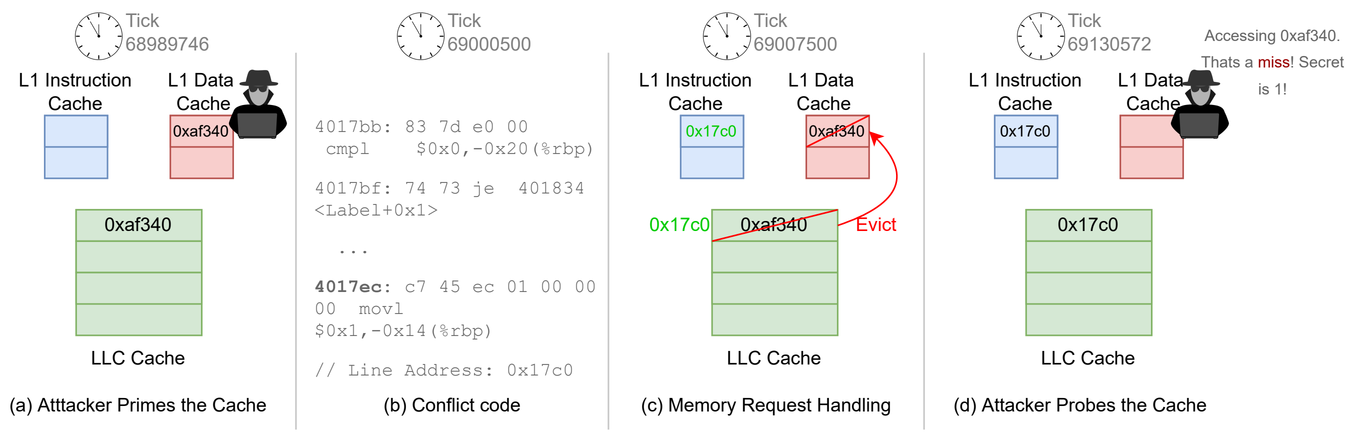
We also demonstrate our attack on the gem5 [6] simulator using the available debug flags. We use the ruby memory system with the MESI_Two_Level protocol, which includes two levels of inclusive cache hierarchy.

Examining the simulator output trace, it is possible to illustrate exactly how the attack behaves. We demonstrate the attack based on the following instructions:

```
(a) Prime
 // touch = data[0];
  4017ae: 8b 05 9c eb 0a 00 mov 0xaeb9c(%rip),%
      eax   # 4b0350 <data> // Line Address: 0
      xaf340
  4017b4: 89 45 e8  mov %eax,-0x18(%rbp)

(b) Conflict
 // if(secret){
  4017bb: 83 7d e0 00    cmpl    $0x0,-0x20(%rbp)
  4017bf: 74 73 je  401834 <Label+0x1>
  ...
  4017ec: c7 45 ec 01 00 00 00   movl    $0x1,-0
      x14(%rbp) // Line Address: 0x17c0
  ...
 // }

(c) Probe
```



```
4017bb: 83 7d e0 00
 cmpl    $0x0,-0x20(%rbp)

4017bf: 74 73 je  401834
<Label+0x1>

  ...

4017ec: c7 45 ec 01 00 00
00  movl
$0x1,-0x14(%rbp)

// Line Address: 0x17c0
```

Fig. 6. Proof-of-Concept on gem5: (a) The attacker primes the L1D cache with cache line 0xaf340. (b) The conflict code executes code using a secret-dependent branch fetching instruction 4017ec, with cache line 0x17c0. (c) Memory handles the requests and sends back-invalidation to L1D cache for 0xaf340, to be able and cache 0x17c0. (d) The attacker probes L1D cache and observe changes in cache line 0xaf340.

```
  // touch = data[0];
   401839: 8b 05 11 eb 0a 00 mov 0xaeb11(%rip),%
           eax    # 4b0350 <data> // Line Address: 0
           xaf340
   40183f: 89 45 e8  mov %eax,-0x18(%rbp)
```

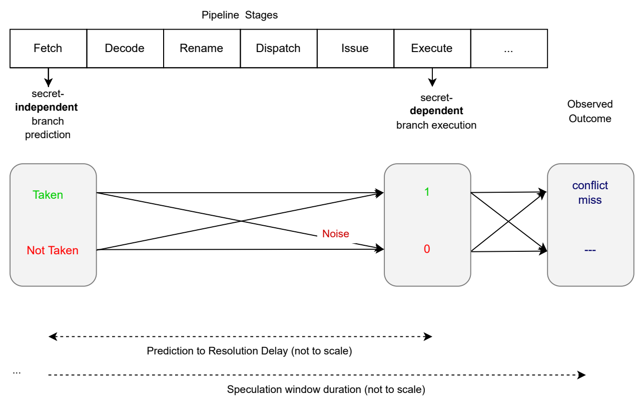
Figure 6 demonstrates the behavior of a processor as seen on the simulator. First, (a) the attacker primes the cache using his data. The data[0] (*PC: 4017ae*) has the physical address: *0xaf350*, which belongs to the cache line with physical address: *0xaf340*. Accessing the data and bringing the value into the cache hierarchy completes on tick `68989746`. (b) The attacker waits for the conflict to access a cache line, which conflicts with the data in LLC. The conflict fetches the instruction *PC: 4017ec* with physical address: *0x17c0*. The request to the memory hierarchy starts on tick `69000500`. The LLC (L2 on MESI_Two_Level protocol) sends an invalidation on *0xaf340*, which evicts it from the L1D cache on tick `69002094`, from the directory on tick `69006000`, and finally from the LLC on tick `69007500`. Finally, (c) the attacker probes the data (*PC: 401839*) on tick `69130572`, which misses in the memory hierarchy. HMS: (d) is not mentioned, which is the probe.

VII. What Really Happens: Interaction with Branch Prediction

Figure 3 implies that the attack fetches instructions in a secret-dependent manner. *In reality, this is not exactly what happens.* Speculative execution, and in particular branch prediction, firstly fetches instructions in a secret *independent* manner. Fetch is first predicted (using non-speculative and therefore secret-independent data) and only subsequently becomes secret-dependent, once the speculation is resolved, which leads to the results we observed in the previous section. Here, we give a more in-depth explanation of how the attack interacts with branch prediction and explain the noise in the results.

When an instruction is in the fetch stage, branch prediction based on branch history takes place, independent of the secret value.[2] Since branch prediction may or may not take the branch, not only the results, but also the speculation window is affected. Recall, that the secret-dependent branch (line 3 in Figure 3) comes after the branch that triggers speculation (line 2 in Figure 3), and when it is fetched the speculation window is already running. If the speculation window was large enough, and fetch only occurred as a result of the secret value, there would be much less noise and fewer repetitions would be needed.

In general, attacks that rely on a secret-dependent branch may experience two kinds of noise. First, noise induced by branch prediction in the fetch stage, and second, noise induced by limited speculative window length. The following section summarizes how these two features can influence the results of *DOIN!*, and similar ideas can be applied to explain the behavior of various other speculative attacks.


Pipeline Stages
Fetch
Decode
Rename
Dispatch
Issue
Execute
...
secret-independent branch prediction
secret-dependent branch execution
Observed Outcome
Taken
Not Taken
Noise
1
0
conflict miss

Prediction to Resolution Delay (not to scale)
...
Speculation window duration (not to scale)

Fig. 7. Attack in the instruction pipeline. Before the branch is resolved and the value of the secret is used, a prediction is made, affecting the results.

Figure 7 summarizes how the attack proceeds in the pipeline. The branch instruction, conditional on the secret

[2]We assume that the branch predictor is trained non-speculatively which is a logical assumption post-Spectre.

Prediction	Secret	Outcome	Comments
T	0	miss	Noise: False-positive miss from random wrong-path conflict — exceedingly rare
T	0	—	Attack: Wrong prediction but no miss occurs (short misprediction to resolution period) — likely outcome
T	1	miss	Attack: Correct prediction
T	1	—	Noise: Speculation window too short!
NT	0	miss	Noise: Random conflict from the correct path — rare
NT	0	—	Attack: Correct prediction
NT	1	miss	Attack: Wrong prediction delays the conflict, somewhat reducing the chances for a miss
NT	1	—	Noise: Speculation window too short!

TABLE I
ALL POSSIBLE OUTCOMES ACCORDING TO PREDICTION AND ACTUAL SECRET VALUE.

value (line 3 in Figure 3), is fetched. At this point, the secret value is unknown, so the branch decision cannot be secret-dependent: the branch predictor predicts the path of the instruction.

In case the prediction is `taken`, the next instructions in the attack path (lines 4–9 in Figure 3) will be fetched. Those next instructions that are fetched after the prediction and before the branch is resolved, can miss in the instruction cache and insert noise into the attack. While predicting `not taken` forces the execution to fetch instructions after the secret-dependent branch, not interacting with the secret-dependent branch instructions. Instructions that come after the reconvergence point and not in the branch, may be able to cause a conflict miss, but the chances of doing so are close to zero. The difference between those conflict misses, and the conflict misses happening through the secret-dependent *true* branch path is that the secret-dependent branch consists of instruction addresses dedicated to conflict with the data in the cache, while instructions after the reconvergence point are not.

The branch resolves in the execute stage, where it turns into a secret-dependent branch. Depending on the prediction, execution will either continue (prediction was correct), or will squash everything and start fetching new instructions (prediction was incorrect). In any case, from this point onward, all upcoming instructions are control-dependent on the secret value. If the secret is equal to one, instructions are fetched from the secret-dependent *taken path* (Figure 3). Instructions on this path are malicious, since they are destined to miss in the cache and cause information leakage. In contrast, if the secret is equal to zero, instructions are fetched from the secret-dependent *fall-through path* that is not designed to cause conflict misses (Figure 3).

The attack exhibits different behavior according to the prediction and the actual secret value. Table I presents all the possible outcomes depending on the branch prediction and the actual value of the secret. A successful attack must create an observable miss only when the $secret\ value == 1$. We summarize the behavior as follows:

a) Prediction: Taken & Secret value: 0: The prediction is incorrect ($secret\ value == 0$) and the branch is incorrectly taken. Instructions from the secret-dependent branch are incorrectly fetched. Those instructions can execute during the period between prediction and resolution, inducing noise to the attack, before getting squashed. Since `secret` can hit in cache, the prediction-to-resolution delay may not be enough to fetch and execute many instructions to evict the data. Although, false-positives can theoretically happen, they are exceedingly rare. Results show that in a thousand attempts, no such case occurred.

b) Prediction: Taken & Secret value: 1: The prediction is correct since $secret\ value == 1$ means that the branch should be taken. The attack starts executing as soon as the branch is in the fetch stage, as it will not be squashed when it resolves. If the speculation window lasts long enough, observable timing differences will be effected, otherwise a conflict miss will not appear, leading to a false-negative.

c) Prediction: Not Taken & Secret value: 0: The prediction is correct ($secret\ value == 0$) and the most likely outcome is that no conflict miss is observed. Although a false-positive can happen through coincidental interference, it is exceedingly rare: If an observable conflict miss occurs, it is most likely caused (at random) by an instruction fetch after the branch's reconvergence point. Results show that in a thousand attempts, no such case has occurred.

d) Prediction: Not Taken & Secret value: 1: The prediction is incorrect, since $secret\ value == 1$ means that the branch should be taken. However, this misprediction only delays the attack until the secret-dependent branch is resolved. Because of the prediction-to-resolution lost time being subtracted from the speculation window, the chances of a successful attack are lower. If the speculation window lasts long enough, observable timing differences will be created, otherwise a conflict miss will not be created.

Based on this analysis, it is highly unlikely for false-positives to appear when the secret is equal to zero, and highly likely to create observable timing differences when the secret is equal to one. A defining factor in both cases is the duration of the speculation window that is started by the mispredicted branch in line 2 of Figure 3. These conclusions are well corroborated by the results we see in Figure 5.

VIII. *DONOT!*: MITIGATING *DOIN!* ATTACKS.

A. Front-end stalls

The front-end of out-of-order processors is in-order. The processor has to wait for an instruction to be fetched, before fetching the next instructions. In the front-end, and more specifically the fetch stage, there are several factors that can cause delays when fetching an instruction, such as instruction squashes, BTB misses, L1I cache misses, and TLB misses. This can negatively impact performance, sometimes drastically, depending on the instruction pattern of the executing workload. Figure 8 shows the amount of time the processor front-end stalls. On average, the processor spends 11.3% of

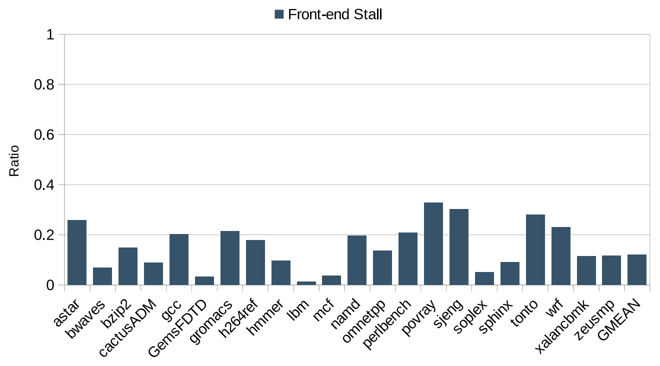


Fig. 8. Ratio of cycles that front-end stalls.

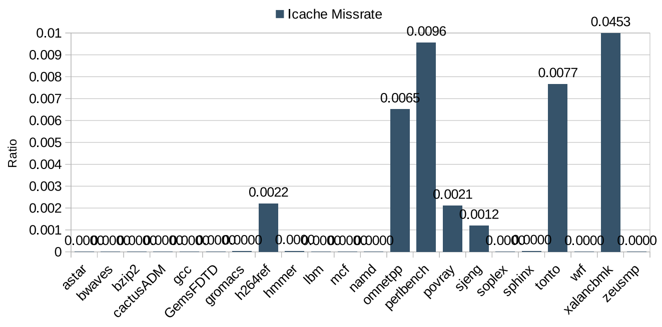


Fig. 9. L1I cache miss rate.

total execution cycles waiting for the front-end to finish until it continues fetching the next instructions.

Finding an efficient mitigation strategy against speculative *DOIN!* can be difficult, as restricting front-end execution can introduce large performance penalties. However, instruction caches are used ubiquitously in modern processors and, for many workloads, posses an extraordinarily low miss rate. Figure 9 shows the miss rate of the L1I cache running the SPEC2006 workloads. Even when using high precision (four decimal points), the miss rate is close to zero, showing that there are just a few to no instructions missing in the L1I cache. With 4 decimal digit precision on ratio (misses over accesses), only *xalancbmk* has an instruction miss rate higher than 1%. More specifically, it has a miss rate of 4.59%. *h264ref, omnetpp, perlbench, povray, sjeng, and tonto* have a miss rate between 1% and 0%, and the rest of the benchmarks 0%.

B. DONOT!*: Delay-on-Miss for I-Caches*

In this work, we focus on control shadows (C-Shadow) as introduced by Delay-on-Miss Section II-D, more specifically on speculative side-channel attacks executed as caused by branch misprediction. *DOIN!* attack focuses on instructions that miss in cache. Inspired by Delay-on-Miss' idea of delaying loads that miss in the L1D cache, we propose *DONOT!* a mechanism that delays instructions that miss in the L1I cache.

TABLE II
THE SIMULATED SYSTEM PARAMETERS.

Parameter	Value
Core	out-of-order, 8-issue/execute/commit width, 192 Reorder Buffer Entries, 64 Instruction Queue Entries, 32 Load Queue, 32 Store Queue
Cache line size	64 bytes
L1 data cache size	32KiB, 8-way
L1 instruction cache size	32KiB, 8-way
L2 shared cache size	1MiB, 8-way
Warmup	3 billion instructions
Run	1 billion instructions

An instruction that is to be fetched, but misses in the L1I cache while branch instructions are unresolved is considered unsafe for the Spectre threat model, and the fetch request will not propagate through the cache hierarchy. Instruction fetches that miss are delayed until all previous branch instructions are clear from the pipeline, by either committing in the reorder buffer or getting dropped before reaching the reorder buffer. Once the pipeline is free of branches, the fetch stage can continue by repeating the initial memory request.

Restricting instruction misses from the front-end flow while an unresolved branch exists in the instruction pipeline is sufficient to tackle *DOIN!* attack, as no speculative evictions can occur. This is a conservative, yet effective way of mitigating the attack.

C. *Methodology*

To evaluate the performance impact of our proposed mitigation we use the gem5 [6] simulator, and we run SPEC2006 [8], which is consistent with the type of workloads one would find in a processor for consumer products. SPEC2006 is characterized by low ICache miss rates in contrast to datacenter and server workloads, which exhibit high instruction miss rates. However, the latter typically run on processors with non-inclusive or exclusive cache hierarchies and thus less relevant to our case. We implement *DONOT!* on top of Delay-on-Miss with only control shadows. Table II shows the configuration of the simulated processor and the parameters for the executed simulation. We warmup the simulation for 3 billion instructions (2 billion for tonto) and gather statistics for 1 billion instructions after the warmup.

D. *Evaluation*

DONOT! introduces negligible performance overhead for Delay-on-Miss. Figure 10 presents the IPC of both Delay-on-Miss and *DONOT!* normalized to an unsafe baseline. *DONOT!* introduces a performance overhead of 0.1% on Delay-on-Miss, slowing it down from 88.5% to 88.4%, as compared to the unsafe baseline. Most benchmarks are not observably affected by the mitigation, as they have a very small amount of L1I cache misses. *perlbench, povray, tonto, and xalancbmk* introduce 0.62%, 0.52%, 0.35%, and 0.61% overhead respectively. *h264ref, omnetpp, and sjeng* are the other benchmarks that have some very small number of instruction misses (Figure 9) and introduce 0.04%, 0.03% and 0.04% performance overhead

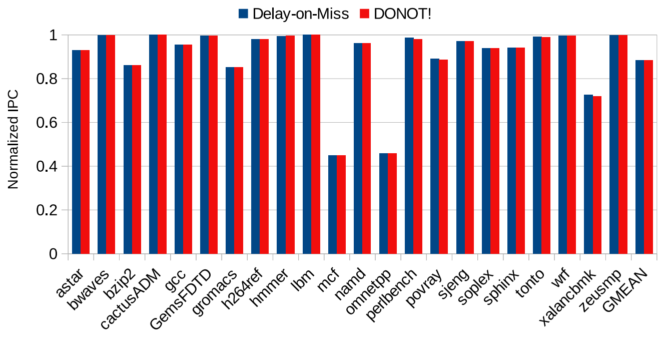


Fig. 10. Normalized IPC to unsafe baseline.

respectively. The rest of the benchmarks maintain less than 10000 ICache misses on over 100m ICache accesses.

Figure 11 illustrates the percentage of how many instructions out of all the instruction cache accesses were delayed. As expected from the performance results, only *h264ref, omnetpp, perlbench, povray, sjeng and tonto* delay a very small number of instructions, while the rest of the benchmarks delay from 0 to some decades or hundreds of instructions. This amount of delayed instructions is not enough to slow down the performance.

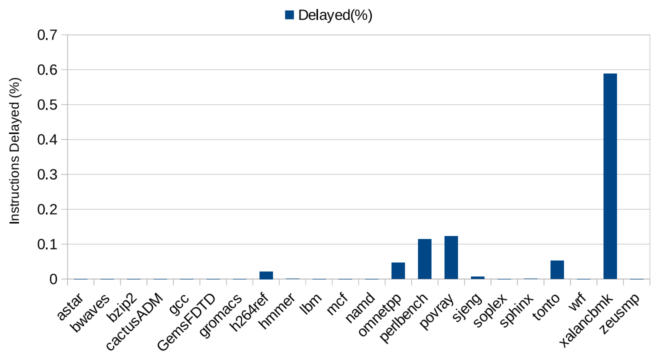


Fig. 11. Percentage of instructions that were delayed out of all the instruction accesses.

E. Discussion: Further Improvements

The scope of this paper is presenting a new cache side-channel attack and using it to expose a new vulnerability in Delay-on-Miss. The proposed mitigation is a basic solution to the problem and can be further improved. In this section, we discuss how *DONOT!* can be further optimized. Even though the mitigation introduces negligible performance slowdown, it remains conservative and can experience different performance behavior on workloads with significant front-end instruction misses.

One of the main reasons why *DONOT!* is conservative, is because it delays all instruction misses, when they are under a C-Shadow and miss in the L1I cache. In fact, an instruction miss in L1 does not always corresponds to a miss in L2 cache and beyond when caches maintain inclusive policy. An instruction that is already placed in L2 and the request hits in L2 cache is guaranteed not to cause any data eviction and thus is safe to execute. Instead of delaying instructions that miss in L1 cache, we can consider delaying instructions that miss in L2 cache and beyond, as far as those caches maintain inclusivity with L1 cache. A mitigation like this would result into a more complex hardware, as the packet can now be dropped deeper in the memory hierarchy.

Beyond that, currently *DONOT!* waits for all unresolved branch to retire from the reorder buffer before it proceeds with the instruction access. In fact, we know if a branch is going to commit or squash, once its operands are ready, and not when it reaches the head of the reorder buffer. This would enable instructions to be accessed and the front-end to continue earlier, before the branch reaches the head of the reorder buffer.

IX. Related Work

Besides *Prime+Probe* [15], [17] that we introduced in background, there are several other cache hierarchy side-channel attacks. *Flush+Reload* [31] uses software instructions (e.g., clflush()) to evict data from cache, and then reloads them while measuring the access latency. *Flush+Flush* [9] relies on the execution time of flush instructions, which varies depending on if the data is cached or not. *"LLC Attacks are Practical"* [15] shows how attacks can be implemented for the LLC. In addition to purely cache content focused attacks, there are other works regarding memory hierarchy functionality. *LRU State Attack* [27] shows how replacement policies can leak information about the data in caches, while *Attack directories, not caches* [30] shows how information leakage can be done through directories in non-inclusive cache hierarchies.

Speculative attacks have also evolved the last few years. After the introduction of Spectre [12], there has been a ping-pong game between mitigations and attacks. The first wave of mitigations focused on hiding observable speculation using buffers [20], [28]. The second wave focused on delaying the execution, e.g., either delaying transient loads that miss [21] or using a taint tracking mechanism to delay speculative transmitters [32], or undoing the speculation leakage [19]. Moreover, there were works trying to recover lost performance of speculative mitigations, such as InvarSpec [33], which lifts protections for specific speculative instructions, if they are guaranteed to commit regardless of the outcome of the speculation.

Hiding speculative execution [28] and delay execution schemes [21] were proven vulnerable by Speculative Interference [5]. Speculative Interference [5] uses speculative instructions in order to influence the timing, and thus the ordering, of non-speculative older instructions. Delaying execution using taint tracking [32] was proven to be vulnerable to using secret-dependent store instructions to leak information through the TLB. Undo-based speculation schemes [19] were found vulnerable by unXpec [13] as the speculation window varies according to the number of speculative loads. Lastly, optimizations such as InvarSpec [33] were proven to be vulnerable by Reorder Buffer Contention [2] using a variant of Speculative Interference, manipulating the reorder buffer in a secret-dependent manner and pushing in or out on demand a speculation. Attacks against memory hierarchy speculative defenses [2], [5], [13] share a similarity with *DOIN!*, information leakage through implicit channels. Those attacks do not leak the value of the secret explicitly, but by creating observable timing differences using the value of the secret.

X. Conclusion

This paper introduced *DOIN!*, an attack that leverages inclusive caches to leak information through the LLC using both data and instructions. The attack uses instructions to conflict and evict data from the shared cache, resulting in observable data cache evictions.

This attack is sufficient on breaking speculative side-channel defenses, such as Delay-on-Miss, by using a secret-dependent (that is illegal and can only be accessed speculatively) branch to load an instruction that conflicts with data in the LLC. To tackle the problem, we present *DONOT!*, a solution to mitigate the *DOIN!* attack under Delay-on-Miss speculative defense. *DONOT!* waits for previous branches to retire from the reorder buffer before fetching an instruction that misses in the L1I cache, providing security with negligible performance overhead.

References

[1] O. Aciiçmez, "Yet another microarchitectural attack: exploiting i-cache," in *Proceedings of the 2007 ACM workshop on Computer security architecture*, 2007, pp. 11–18.

[2] P. Aimoniotis, C. Sakalis, M. Själander, and S. Kaxiras, "Reorder buffer contention: A forward speculative interference attack for speculation invariant instructions," *IEEE Computer Architecture Letters*, vol. 20, pp. 162–165, Jul. 2021.

[3] S. Ainsworth, "GhostMinion: A strictness-ordered cache system for spectre mitigation," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, Oct. 2021, p. 592–606. [Online]. Available: https://doi.org/10.1145/3466752.3480074

[4] S. Ainsworth and T. M. Jones, "MuonTrap: Preventing cross-domain Spectre-like attacks by capturing speculative state," in *Proceedings of the International Symposium on Computer Architecture*, May 2020, pp. 132–144.

[5] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. 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," in *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, Apr. 2021, p. 1046–1060. [Online]. Available: https://doi.org/10.1145/3445814.3446708

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 1–7, May 2011. [Online]. Available: https://dl.acm.org/doi/10.1145/2024716.2024718

[7] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. v. Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *Proceedings of the USENIX Security Symposium*, 2019, pp. 249–266. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[8] S. P. E. Corporation, "SPEC CPU2006 benchmark suite," 2006. [Online]. Available: http://www.specbench.org/cpu2006/

[9] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 279–299.

[10] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive {Last-Level} caches," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 897–912.

[11] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Oct. 2018, pp. 974–987. [Online]. Available: https://ieeexplore.ieee.org/document/8574600/

[12] 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," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2019, pp. 1–19.

[13] M. Li, C. Miao, Y. Yang, and K. Bu, "unxpec: Breaking undo-based safe speculation," in *Proceedings of the International Symposium High-Performance Computer Architecture*, Apr. 2022, pp. 98–112.

[14] F. Liu and R. B. Lee, "Random fill cache architecture," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 203–215.

[15] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[16] 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 *Proceedings of the USENIX Security Symposium*, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/loughlin

[17] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*. Springer, 2006, pp. 1–20.

[18] K. Ramkrishnan, S. McCamant, P. C. Yew, and A. Zhai, "First time miss: Low overhead mitigation for shared memory cache side channels," in *49th International Conference on Parallel Processing-ICPP*, 2020, pp. 1–11.

[19] G. Saileshwar and M. K. Qureshi, "CleanupSpec: An "undo" approach to safe speculation," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, Oct. 2019, p. 73–86. [Online]. Available: https://doi.org/10.1145/3352460.3358314

[20] C. Sakalis, M. Alipour, A. Ros, A. Jimborean, S. Kaxiras, and M. Själander, "Ghost loads: What is the cost of invisible speculation?" in *Proceedings of the ACM International Conference on Computing Frontiers*. Association for Computing Machinery, Apr. 2019, p. 153–163. [Online]. Available: https://doi.org/10.1145/3310273.3321558

[21] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *Proceedings of the International Symposium on Computer Architecture*, Jun. 2019, pp. 723–735.

[22] C. Sakalis, M. Själander, and S. Kaxiras, "Seeds of SEED: Preventing priority inversion in instruction scheduling to disrupt speculative interference," in *Proceedings of the IEEE International Symposium on Secure and Private Execution Environment Design*, Sep. 2021, pp. 101–107.

[23] W. Song and P. Liu, "Dynamically finding minimal eviction sets can be quicker than you think for Side-Channel attacks against the LLC," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 427–442. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/song

[24] K.-A. Tran, C. Sakalis, M. Själander, A. Ros, S. Kaxiras, and A. Jimborean, "Clearing the shadows: Recovering lost performance for invisible speculative execution through HW/SW co-design," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. Association for Computing Machinery, Sep. 2020, p. 241–254. [Online]. Available: https://doi.org/10.1145/3410463.3414640

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

[26] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th annual international symposium on Computer architecture*, 2007, pp. 494–505.

[27] W. Xiong and J. Szefer, "Leaking information through cache lru states," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 139–152.

[28] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InvisiSpec: Making speculative execution invisible in the cache hierarchy," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, Oct. 2018, pp. 428–441.

[29] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 347–360.

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

[31] Y. Yarom and K. Falkner, "{FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack," in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 719–732.

[32] 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 *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, Oct. 2019, p. 954–968. [Online]. Available: https://doi.org/10.1145/3352460.3358274

[33] 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 *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Oct. 2020, pp. 1138–1152. [Online]. Available: https://ieeexplore.ieee.org/document/9251941/