Speculative Interference Attacks: Breaking Invisible Speculation Schemes

Mohammad Behnia
UIUC, USA

Prateek Sahu
UT Austin, USA

Riccardo Paccagnella
UIUC, USA

Jiyong Yu
UIUC, USA

Zirui Neil Zhao
UIUC, USA

Xiang Zou
Intel Corporation, USA

Thomas Unterluggauer
Intel Corporation, USA

Josep Torrellas
UIUC, USA

Carlos Rozas
Intel Corporation, USA

Adam Morrison
Tel Aviv University, Israel

Frank Mckeen
Intel Corporation, USA

Fangfei Liu
Intel Corporation, USA

Ron Gabor
Toga Networks, Israel

Christopher W. Fletcher
UIUC, USA

Abhishek Basak
Intel Corporation, USA

Alaa Alameldeen
Simon Fraser University, Canada

ABSTRACT

Recent security vulnerabilities that target speculative execution (e.g., Spectre) present a significant challenge for processor design. These highly publicized vulnerabilities use speculative execution to learn victim secrets by changing the cache state. As a result, recent computer architecture research has focused on *invisible speculation* mechanisms that attempt to block changes in cache state due to speculative execution. Prior work has shown significant success in preventing Spectre and other attacks at modest performance costs.

In this paper, we introduce *speculative interference attacks*, which show that prior invisible speculation mechanisms do not fully block speculation-based attacks that use cache state. We make two key observations. First, *mis-speculated younger instructions can change the timing of older, bound-to-retire instructions,* including memory operations. Second, changing the timing of a memory operation can *change the order of that memory operation relative to other memory operations*, resulting in persistent changes to the cache state. Using both of these observations, we demonstrate (among other attack variants) that secret information accessed by mis-speculated instructions can change the order of bound-to-retire loads. Load timing changes can therefore leave secret-dependent changes in the cache, even in the presence of invisible speculation mechanisms.

We show that this problem is not easy to fix. Speculative interference converts timing changes to persistent cache-state changes, and timing is typically ignored by many cache-based defenses. We develop a framework to understand the attack and demonstrate concrete proof-of-concept attacks against invisible speculation mechanisms. We conclude with a discussion of security definitions that are sufficient to block the attacks, along with preliminary defense ideas based on those definitions.

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446708

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**.

KEYWORDS

invisible speculation, speculative execution attacks, microarchitectural covert channels

ACM Reference Format:
Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. 2021. Speculative Interference Attacks: Breaking Invisible Speculation Schemes. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3445814.3446708

1 INTRODUCTION

Speculative execution attacks such as Spectre [31] and follow-on work [8, 12, 24, 30, 32, 36, 44, 56] have opened a new chapter in processor security. In these attacks, adversary-controlled transient instructions—i.e., speculative instructions bound to squash—access and then transmit potentially sensitive program data over *microarchitectural covert channels* (e.g., the cache [58], port contention [8]). For example in Spectre variant 1—`if (i < N) { j = A[i];`

`B[j]; }`—speculative execution bypasses a bounds check due to a branch misprediction, accesses an out-of-bounds value (`j = A[i]`) and transmits that value through a *cache-based covert channel* (`B[j]`), i.e., by forcing a cache fill to occur in a set that depends on `j`. In this paper, we consider the illegally accessed value `j` to be the *secret*. Here, the attacker controls the value of `i`, thus `j` can be any value in program memory and the covert channel can reveal arbitrary program data.

While a variety of covert channels can be used to leak secret values under mis-speculation, cache-based covert channels [31, 35, 37, 57–59] make the fewest assumptions on the attacker and have therefore received the most attention. This is for two reasons. First, secret-dependent cache fills leave a persistent footprint in the cache which is observable long after speculation squashes. Second, certain levels of modern cache hierarchies are globally shared by all cores in the system, enabling attackers to observe said persistent state changes from other physical cores [35, 57]. By contrast, many other covert channels (e.g., arithmetic port contention [6, 8]) leave only intermittent side effects that must be monitored before the squash, and/or require that the attacker share hardware resources on the same physical core (e.g., branch predictor channels [4, 13–15])—both of which can be easily blocked (e.g., disabling SMT).

The above view of the covert channel landscape has led to a surge of architecture-level "*invisible speculation*" proposals to block cache-based covert channels due to mis-speculations (e.g., InvisiSpec [56], SafeSpec [28], Delay-on-Miss [39], Conditional Speculation [33], MuonTrap [5]). Invisible speculation schemes add hardware to prevent mis-speculated loads from making persistent state changes to the memory subsystem. To maintain the performance benefits of caching, only non-speculative loads that are bound to retire are allowed to modify the cache state. To maintain the performance benefits of out-of-order execution, loads are allowed to "invisibly" execute (i.e., bring data directly to the core without filling the cache) and forward their results to dependent instructions.

This Paper. In this paper we introduce *speculative interference attacks*, which show that invisible speculation schemes do not fully block speculation-based attacks that use the cache state. Our attacks are based on two key observations. First, that mis-speculated instructions can influence the timing of older, bound-to-retire operations. Second, if changing the timing of a memory operation changes the *order* of that memory operation *with respect to other memory operations*, the resulting reordering can cause persistent cache-state changes. Putting these together, we show (among other attack variants) how secret information accessed in a mis-speculated window influences the order of bound-to-retire loads, leaving secret-dependent state changes in the cache—even if invisible speculation is enabled.

To explain these ideas, consider a simple but representative invisible speculation scheme: *Delay-on-miss* (DoM) [39]. DoM issues a speculative load and (a) on an L1 cache hit, forwards the load result to dependent instructions, or (b) on an L1 cache miss, delays servicing the miss and re-issues the load when it becomes non-speculative. In case (a), DoM does not update any replacement state (e.g., replacement bits) in the L1 cache until the load becomes non-speculative. For simplicity, we explain ideas assuming only branch instructions cast speculative shadows [39], i.e., a load is considered non-speculative/safe iff it is older than the oldest unresolved branch. We discuss attacks on more conservative DoM variants in § 3.

DoM's (and other invisible speculation schemes') stated security goal is to only focus on blocking cache state changes due to mis-speculations, while leaving other covert channels un-blocked. This is reflected in DoM's design. On the one hand, DoM prevents mis-speculated loads from directly changing the cache state. On the other hand, DoM allows mis-speculated loads to forward their results to dependent instructions, which can clearly form covert channels through intermittent state changes. In fact, this is exactly the basis for forming, e.g., arithmetic unit port contention covert channels [6, 8].

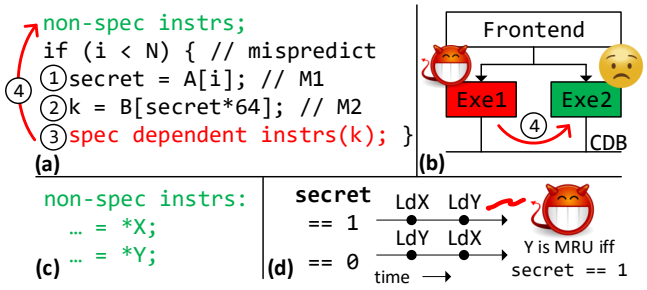


Figure 1: Speculative interference example. `secret` is 0 or 1. (a) Assume the code snippet is run on a processor protected by invisible speculation such as DoM and that `&B[0]` is cached while `&B[64]` is not cached. (b) This results in *speculative dependent instructions* conditionally contending for execution resources with *non-speculative instructions*, depending on the value of `secret`. (c), (d) If the non-speculative instructions are two loads, the contention can influence the order in which the loads are issued. Finally, the attacker can infer the secret based on the cache replacement state after the loads both issue.

This paper demonstrates how instructions that cause intermittent state changes can be leveraged to create persistent state changes in the cache. Consider the example in Figure 1, modeled after Spectre variant 1. Suppose this code is run on a processor using DoM. In Figure 1 (a), a mis-speculated load M1 forwards secret data `secret` to a second load M2 (①). A normal Spectre attack would monitor the cache state change left by M2 to deduce `secret`. To prevent this leak, DoM would prevent M2 from changing the cache state, specifically by allowing it to access and return data from the L1 if there is an L1 hit and delaying its execution otherwise. While this blocks the cache state change due to M2, M2 is allowed to forward its result when it completes (②), meaning that dependent instructions execute at a time that depends on `secret` (③). This has the potential to create a traditional non-cache based covert channel, e.g., through execution unit port usage, which DoM ignores.

Our key observation is that secret-dependent timing effects caused by the dependent instructions can be monitored *indirectly* through how they interact with the execution of *older non-speculative instructions* (④). Although the non-speculative instructions come before the speculative dependent instructions in program order, out-of-order execution could have both of them executing concurrently and contending for resources. In our example

(Figure 1 (b)), the non-speculative and speculative dependent instructions use execution units EXE1 and EXE2, respectively, and contend for the common data bus (CDB) in the same cycle. We call this *speculative interference*.

Next, we show how speculative interference can be used to bootstrap a change in the cache state. Specifically, suppose the non-speculative instructions are made up of two independent loads to addresses X and Y in different cache lines mapped to the same set, shown in Figure 1 (c). Since these loads are older than the mispredicted branch in program order, they are not protected by DoM. We show how, depending on the timing changes caused by the speculative dependent instructions, the order in which the load to X is issued with respect to the load to Y can change. *That is, depending on a secret, the processor issues either loads to X followed by Y or Y followed by X.* To finish the attack (Figure 1 (d)), we show how changing the order of memory operations can be used to create persistent changes in the cache state, the intuition being that state in the cache (e.g., replacement bits) depends on not just what requests are made, but also their order.

This issue is not easy to fix. The crux of the problem is that timing changes can be converted to persistent state changes. These timing changes can arise due to interference through a large number of microarchitectural structures, through different instructions, etc. Further, while our example reorders two loads that originate from the same thread, there are many other memory address streams through which to interleave operations, e.g., interleaving instruction and data cache accesses, accesses made across threads and security domains, etc.—which further widens the attack surface.

The rest of the paper expands on the above ideas as follows:

- We introduce and provide a framework to reason about *speculative interference attacks*, whereby subtle secret-dependent microarchitectural interference influences the behavior of older non-speculative instructions. We show how this can be used to create cache-based covert channels, even in the presence of invisible speculation schemes.
- We implement proof-of-concept exploits for three such attacks on a real machine, exploiting interference in the processor's out-of-order issue logic, MSHR usage and frontend queues. All attacks are cache-based and work across physical cores.
- We provide a sufficiently strong security definition to block the attacks, and also provide a starting point defense and discussion to set a research agenda for more efficiently and comprehensively addressing the problem.

2 BACKGROUND

2.1 Threat Model

We adopt the standard threat model used by invisible speculation schemes [5, 28, 33, 39, 56]. Such schemes care about preventing "persistent" side effects that are observable due to mis-speculated loads. For example, which lines are in the cache, replacement and coherence state of each line, etc.

Invisible speculation schemes further distinguish from where the attacker is monitoring the covert channel (i.e., where the receiver [29] runs). In particular, one of the first invisible speculation schemes by Yan et al. [56] specifies several such attacker models:

SameThread model: Here we consider untrusted code interleaved with trusted code, as in the case of a sandbox.

CrossCore model: Here, the idea is that the system prevents untrusted code from running on a sibling hyper-thread. However, the receiver may run on another core and monitor a cache-based covert channel through a shared cache.

We will show attacks against these models. Yan et al. [56] also specifies an **SMT** model where the receiver runs in an adjacent hyper-thread. This gives the attacker more power, thus we will focus on the former two models.

2.2 OoO Processor Pipeline

Dynamically scheduled processors execute instructions out of order (OoO) to improve performance [22, 48]. An instruction is *fetched* by the processor frontend, *dispatched* to reservation stations (RS) for scheduling, *issued* to execution units (EUs) in the processor backend, and finally *retired* when it updates the machine's architected state. Instructions proceed through the frontend, backend and retirement stages in order, possibly out of order and in order, respectively. In-order retirement is implemented by queueing instructions in a reorder buffer (ROB) [27] in program order and retiring a completed instruction when it reaches the ROB head.

2.3 Invisible Speculation Schemes

Invisible speculation aims to block covert channels from forming through the cache due to mis-speculated loads [5, 28, 33, 39, 56]. While speculative execution attacks can leverage both cache-based and non-cache-based covert channels, invisible speculation schemes are only concerned with attacks using the cache because cache-state changes are relatively simple to monitor. Specifically, cache-state changes *persist* after the squash and can be measured across physical cores (as in the CrossCore model). By contrast, non-persistent covert channels, such as those through execution units [21] and ports [8, 20], are more difficult to monitor and place additional restrictions on the attacker (e.g., limit the attacker to the SMT model [8]).

While there have been multiple invisible speculation proposals, they all share several common traits. For security, they prevent state changes to the cache due to mis-speculated loads, until those loads are deemed safe/become non-speculative. Performance hinges on several optimizations. First, speculative loads should—subject to the constraint of not updating cache state—be allowed to issue and forward their results to dependent (speculative) instructions. This allows these schemes to maintain efficiency in the common case that speculation turns out to be correct. Second, loads should be able to update cache state when they become non-speculative (safe). Together, these performance optimizations enable such schemes to reap the benefits of out-of-order execution and caching.

Different invisible speculation schemes differ in their exact policies for allowing speculative loads to issue. We describe the "Delay-On-Miss" (DOM) [39] scheme here, as it is simple and illustrates the main ideas.

Delay-On-miss (DOM) allows loads that hit in the L1 cache to execute and forward their results to dependent instructions (which are themselves allowed to execute). Any cache-state change that would have been made as a by-product of the L1 cache hit (e.g.,

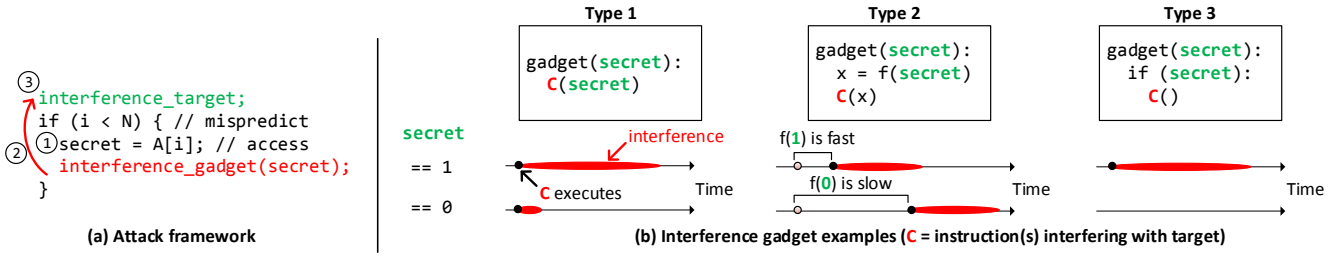


Figure 2: Speculative interference attack framework (a) and classification of interference gadgets (b).

modifying replacement bits) is deferred until the load becomes non-speculative. If the load misses the L1 cache, it is delayed and re-executed when it becomes non-speculative.

3 SPECULATIVE INTERFERENCE ATTACKS

We now describe *speculative interference attacks*. These attacks exploit the fact that while invisible speculation blocks direct cache-state changes by issued speculative loads, it does not restrict how secrets returned by those loads flow through the pipeline and impact the execution of non-memory instructions (§ 2.3). Our key insight is that the secret-dependent resource usage patterns of such non-memory speculative instructions *can be transformed into* cache-based covert channels. Specifically, a speculative interference attack exploits intermittent resource contention to influence the timing of *non-speculative* parts of the pipeline (§ 3.1). These timing effects are then used to make the non-speculative cache access pattern depend on the secret, creating a cache covert channel (§ 3.2).

3.1 Making Mis-speculation Influence the Timing of Non-speculative Actions

We first present a framework for leaking a secret bit by making the secret determine the time at which an unprotected *victim* memory operation accesses the cache, modifying its state. See Figure 2(a) for a visualization. ① A mis-speculated *access* load reads a secret into the pipeline and forwards that secret to a sequence of mis-speculated instructions, called the *interference gadget*, which creates secret-dependent pressure on some microarchitectural resource(s). ② Contention on these resource(s) influences the timing of actions performed by a non-speculative part of the pipeline, called the *interference target*, making them encode the secret. ③ The target is chosen so that changing its timing creates a pipeline "ripple effect" that ultimately delays an unprotected victim memory access. That is, how quickly the target executes determines when the unprotected victim's memory operation is issued and accesses the cache, thereby modifying the cache state.

Interference gadgets can be classified as one of three types, determined by how the secret is used to create resource contention interfering with the target's execution. Figure 2(b) shows examples of the gadget types.

A *Type 1* gadget forwards the secret to instruction(s) with operand-dependent resource usage patterns, called *transmitters* [62], which issue at a secret-independent time. The resulting secret-dependent resource pressure during the transmitter's execute stage interferes with the target. Examples of transmitters include data-dependent arithmetic [21] or loads. Notice that a (speculative) load is considered a transmitter despite not modifying the cache state under invisible speculation. The reason is that its usage pattern of other resources, such as MSHRs, depends on its address operand.

In a *Type 2* gadget, the secret is encoded through instructions issuing at a secret-dependent time. (As opposed to a Type 1 gadget, in which the interfering instructions issue at secret-independent times but have secret-dependent resource usage during their execute stage.) Secret-dependent issue time is achieved by having the interfering instructions be data-dependent on variable-latency instruction(s) that data-depend on the secret. Importantly, while these variable-latency instruction(s) are necessarily transmitter(s), here their secret-dependent resource usage does not interfere with the target. It is only used to influence *when* subsequent interfering instructions (which can be transmitters or non-transmitters) are issued for execution.

In a *Type 3* gadget, whether interfering instructions execute at all depends on the secret. For example, if the gadget contains a branch with a secret-dependent predicate, then resolution of the branch "poisons" the branch predictor's state with secret-dependent information. As a result, the branch's prediction can become secret-dependent and determine whether the contending instructions execute in subsequent executions of the gadget.

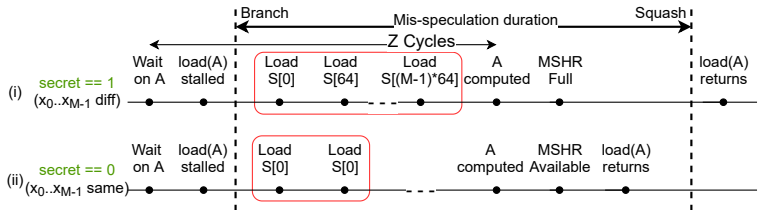
All gadgets exploit the fact that invisible speculation does not "protect" a transmitter's resource usage pattern, execution time, or branch prediction. Although the resulting secret-dependent execution behavior cannot be directly observed by the attacker (which monitors the cache), it can be indirectly observed through its influence on the non-speculative interference target, as discussed next.

3.1.1 Interference Gadgets & Targets. Here, we design several interference gadget/targets, which illustrate Type 1 and 2 gadgets that delay D- and I-Cache accesses. § 4 implements the attack variants described in this section and shows that they lead to observable interference in practice. Our goal here is not to exhaustively enumerate all possible gadgets/targets, but to illustrate the problem. Exploring if (and which) other microarchitectural resources can be used to build interference gadgets is future work.

We first present two gadgets that delay an unprotected D-Cache access. We exploit the fact that in all invisible speculation designs,

```
1 A = ...    // takes Z cycles
2 y = load(A)  // Interference Target
3 if (i < N): // mispred. taken (miss on N)
4    secret = load(&TargetArray[i]) // access
5    // Interference Gadget
6    x0   = load(&S[secret * 64 * 0])
7    x1   = load(&S[secret * 64 * 1])
8     ...
9    x_{M-1} = load(&S[secret * 64 * (M-1)])
```

(a)

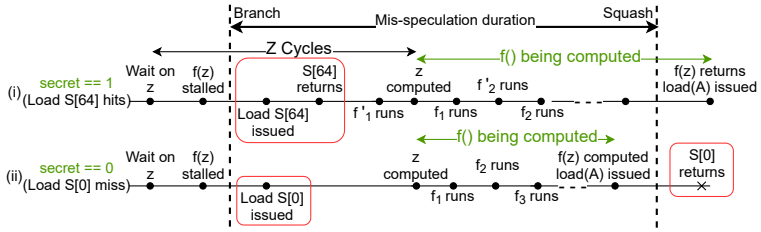


(b)

Figure 3: Delaying a load using miss status holding registers (G^{D}_{MSHR}). M is the number of L1 D-Cache MSHRs. Transmitters in the interference gadget are boxed in both the pseudo-code and timeline to show correspondence.

```
1 z = ...    // takes Z cycles
2 A = f(z)
3 y = load(A)  // Interference Target
4 if (i < N): // mispred. taken (miss on N)
5    secret = load(&TargetArray[i]) // access
6    // Interference Gadget
7    x = load(&S[secret * 64])
8    f'(x)
```

(a)



(b)

Figure 4: Delaying a load using contention on a non-pipelined EU (G^{D}_{NPEU}). Instruction sequences `f` and `f'` use the same non-pipelined EU. Transmitters in the interference gadget are boxed in both the pseudo-code and timeline to show correspondence.

a load that executes only when it reaches the head of the ROB performs its D-Cache access unprotected. Our gadgets therefore require interference targets in which a victim load's address, or *target address*, becomes available just as it reaches the head of the ROB. Our interference gadgets either delay the victim load from reaching the ROB head or delay its execution (cache access) after it has reached the ROB head.

Next, we present a gadget that delays an unprotected I-Cache access. Such accesses are performed by InvisiSpec and DoM. We acknowledge that an unprotected I-Cache access can form a cache-based covert channel in and of itself but describe this gadget due to its interesting property of interfering with the frontend and not with some instruction's execution.

G^{D}_{MSHR}: Delay data access with MSHR contention. This is a Type 1 gadget that delays the execution time of a load at the head of the ROB by a secret-dependent amount of time. Figure 3(a) shows the gadget and target. The target consists of the victim load whose address operand, `A`, takes Z cycles to generate. The value Z is such that the gadget's instructions can issue while the target address is being generated. The gadget consists of M independent loads, where M is the number of L1 D-Cache miss status handling registers (MSHRs), each of which holds information on all the outstanding misses for some cache line. (Here and elsewhere, the gadget executes in the shadow of a slow-to-resolve mispredicted branch, due to a cache miss on `N`.) The gadget's goal is to create secret-dependent MSHR pressure, to delay when the victim load obtains its data after reaching the head of the ROB and issuing. This gadget targets invisible speculation designs that issue speculative L1 D-Cache misses, i.e., InvisiSpec, SafeSpec, and MuonTrap. None of these designs specify changes to the MSHR allocation policy, so we assume they use the standard policy of allocating an MSHR to a missing load based on issue order. Figure 3(b) shows the attack timeline.

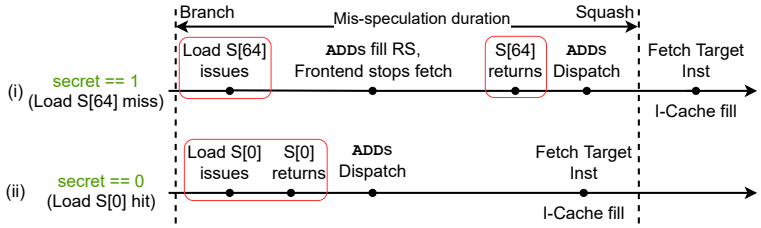
① If `secret` = 1, each gadget load accesses a different cache line. The attacker primes the cache so that each of these accesses is an L1 D-Cache miss. The result is that each load allocates a distinct MSHR, exhausting the available MSHRs. Thus, the victim load (assumed to be an L1 D-Cache miss) cannot issue and is delayed until one of the gadget loads completes or the mis-speculation is squashed.
② If `secret` = 0, all the gadget loads access the same cache line. They therefore use the same MSHR, which leaves MSHRs available for the victim load once it reaches the head of the ROB. The victim load can issue and is not delayed.

G^{D}_{NPEU}: Delay data access using non-pipelined EU. This is a Type 2 gadget that creates a secret-dependent delay of the victim's target address generation. Figure 4(a) shows the gadget and target. The target consists of a victim load whose address operand, `A`, is generated by a dependent chain of instructions, denoted `f`. The gadget consists of a load (transmitter) and a sequence of independent instructions, denoted `f'`, that depend on the load. Each instruction in `f'` uses the same execution unit (EU) as the target. This must be a non-pipelined EU, so that a gadget instruction being issued to the EU blocks a target instruction from issuing.

Figure 4(b) shows the attack timeline. The value `z`, which the target address `A` depends on, takes Z cycles to compute. Z is such that before `z` gets computed, there is enough time for the attack's access load to read the secret, forward it to the interference gadget, and for the gadget's transmitter load to return (if it is a cache hit).
① The transmitter load accesses a secret-dependent cache line. This load executes under invisible speculation protection, but it

```
1 if (i < N): // mispredict taken (miss on N)
2     secret = load(&TargetArray[i])   // access
3     // Interference Gadget
4     x = load(&S[secret * 64])
5     // Congest RS
6     sum += x;
7     ...
8     sum += x; // many times
9 target instr . // Target instruction
```

(a)



(b)

Figure 5: Back-throttling the Fetch Unit by contending for RS for G^{I}_{RS} gadget. `sum+=x` repeats for N (number of RS slots) times. Transmitters in the interference gadget are boxed in both the pseudo-code and timeline to show correspondence.

still retrieves the data from some level of the memory hierarchy. The attacker can therefore orchestrate for its execution time to be secret-dependent, by appropriately priming the cache prior to the attack.

② If `secret` = 1, the transmitter load returns quickly, just before the value `z` is produced. This makes the instruction sequence `f'` in the gadget ready, and its first instruction f_1' is issued to the EU before f_1, the first instruction in `f`. Thus, when f_1 becomes ready, it is blocked from using the EU. When f_1' completes, f_1 is issued (due to age-ordered scheduling). However, once f_1 completes, f_2—which depends on f_1—does not immediately become ready, due to f_1's writeback delay. In contrast, f_2', which depends only on the transmitter, is already ready and so is issued to the EU. This creates a cascading effect in which each instruction f_i gets delayed, delaying the target address' computation until the mis-speculation is squashed.

③ If `secret` = 0, the transmitter load does not return before `z` is produced. (In delay-based invisible speculation designs [39], the load is never executed. In other designs [56], the load simply takes a long time to return compared to the hit case.) As a result, the target address is computed before the gadget's interfering instructions execute (if they execute), and the victim load can issue.

G^{I}_{RS}: Delay instruction fetch with RS contention. This is a Type 1 gadget that fills the RS for a specific EU, which causes head-of-line blocking in the dispatch queue and forces the Fetch unit to stop I-Cache accesses for fetching instructions. Figure 5(a) shows the gadget; the target is an instruction fetch by the frontend, so does not appear in the code. (The gadget's goal is to change when or if the target instruction is fetched.) The gadget consists of a load (transmitter) and a long sequence of arithmetic (ADD) instructions that depend on the load. Figure 5(b) shows the attack timeline.

① The transmitter load accesses a secret-dependent cache line, which is setup to hit or miss in the cache hierarchy, depending on the secret.

② If `secret` = 1, the transmitter load is a miss in the D-Cache. The dependent ADD instructions are fetched and fill up the RS slots, but do not issue. This leads to the RS getting filled up. This consequently creates pressure on the Fetch Unit and the frontend stalls. Hence, the target instruction is not fetched. Once the branch resolves, execution continues and the target instruction gets fetched.

③ If `secret` = 0, the transmitter load hits in the D-Cache. Its output is then quickly available to the dependent ADD instructions, which issue as soon as EU resources are available, hence freeing up the RS slots. Since the RS does not fill up, the frontend does not stop fetching. Consequently, the cache line holding the target instruction is fetched into the I-Cache.

3.2 From Timing to Cache State Changes

We now show how to transform the basic attack primitive (§ 3.1), which creates a secret-dependent delay for an unprotected victim memory access, into a cache covert channel. The insight here is that we can transform a secret-dependent *timing* change—the delay in the victim's access—into a secret-dependent *cache state* change, by using the delay to *reorder* the victim access with another (unprotected) *reference* memory access, which occurs at a fixed, secret-independent time. Conceptually, the reference access acts as a kind of "clock," helping the attacker to observe whether the victim load issues before or after some point in time.

Crucially, the only property required from the reference memory access is that its issue time does not depend on the secret. In particular, the reference access can be issued by the victim or by the attacker, depending on the specific attack. In what follows, we denote the victim memory access load A (which has address A) and the reference memory access load B (which has address B). Then, we arrange for load A (A for short) to be issued before or after load B (B for short), depending on the value of the secret. More precisely: The cache state, σ, is determined by the sequence of memory accesses to the cache, α. We assume that σ is not commutative, i.e., that $\sigma(\alpha\,\mathrm{A}\,\mathrm{B}) \neq \sigma(\alpha\,\mathrm{B}\,\mathrm{A})$. Formally, therefore, making the order in which A and B access the cache secret-dependent makes the cache state secret-dependent, creating a cache covert channel.

Non-commutativity of cache-state updates holds for most cache architectures as long as both memory accesses target different cache line addresses that map to the same cache set. For example, the memory access order impacts the set's *replacement state* (e.g., LRU bits), and can be observed by inducing evictions and monitoring which lines get evicted (by timing memory accesses).

Blocking replacement state-related leakage is explicitly in the scope of invisible speculation (e.g., [33, 39, 56]). However, we are not aware of such attacks being demonstrated in practice.[1] In § 4, we demonstrate a covert channel based on the ordering of two LLC accesses on a commercial CPU with a sophisticated replacement

[1] Recent work [55] shows information leakage through cache LRU states, but its channels rely on more than the ordering of two accesses.

Table 1: Invisible speculation designs vulnerability matrix.

Gadget	Accesses With Secret-Dependent Order		
	V^D-V^D & V^I-V^D	V^D-A^D	V^I-A^D
G^D_{NPEU}	InvisiSpec (Spectre), DoM (non-TSO), SafeSpec (WFB)	**All**	**All**
G^D_{MSHR}	InvisiSpec (Spectre), SafeSpec (WFB)	InvisiSpec, SafeSpec, MuonTrap	InvisiSpec, SafeSpec, MuonTrap
G^I_{RS}	–	–	InvisiSpec, DoM

policy. Thus, for the following discussion, we assume that achieving secret-dependent cache access order is equivalent to forming a covert channel.

3.2.1 Completing the Attacks. We now combine the speculative interference gadgets (§ 3.1.1) with various types of reference memory accesses to obtain several complete attacks on different points in the invisible speculation design space. Each attack creates a cache covert channel by making the secret determine the order of two unprotected LLC accesses, which may be a victim data access (V^D), victim instruction fetch (V^I), or an attacker data access (A^D). (We use V and A to specify whether the victim or attacker thread, respectively, performs the access.) Table 1 summarizes which defenses are vulnerable to which attack combinations.

V^D-V^D ordering. This attack targets invisible speculation designs that may have multiple unprotected loads executing concurrently. For example, InvisiSpec and SafeSpec have modes that only defend against control-flow mis-speculation. In these modes, any load that becomes ready to execute when there are no unresolved branches older than it in the ROB, performs an unprotected access [28, 56]. A similar case exists with DoM on architectures with a non-TSO memory consistency model. In this case, any load can execute without protection if all older branches have resolved and all older stores and loads have their addresses resolved [39].

We show how to base the attack on the G^D_{MSHR} or G^D_{NPEU} gadgets, by modifying the gadget's interference target so that the victim load A is followed (in program order) by a retirement-bound reference load B, whose issue time is not affected by the gadget. Due to space constraints, we fully describe the attack based on the G^D_{NPEU} gadget; the G^D_{MSHR}-based attack is similar. Figure 6 shows the modified target and the original gadget (Figure 4). Both A and B's address generation depend on `z`. If `secret = 0` (i.e., no speculative interference), load A accesses the D-Cache before the reference load B, since the sequence of instructions that generates B, `g(z)`, takes longer to complete than `f(z)`. However, if `secret = 1`, there is speculative interference, so A's generation is delayed while B's is not, and load B accesses the D-Cache first.

V^I-V^D ordering. Modifying the target in the V^D-V^D attack so that the branch condition `N` depends on load A makes the delay of load A also delay the branch's resolution time, i.e., when the squash occurs. This can change the order of a post-squash instruction fetch—which is unprotected, as it is of the correct execution path—with respect to load B.

```
1  z = ...      // takes Z cycles
2  A = f(z)     // takes F cycles
3  y = load(A)
4  B = g(z)     // takes G > F cycles
5  v = load(B)
6  if (i < N): // mispredict taken (miss on N)
7     secret = load(&TargetArray[i])
8     // Interference Gadget
9     x = load(&S[secret * 64]) // secret=1->hit, secret=0->miss
10    f'(x)
```

Figure 6: Reordering victim loads by exploiting contention on a non-pipelined EU. Instruction sequences `f` and `f'` use the same non-pipelined EU. Instruction sequence g uses a different EU.

V^D-A^D ordering. Many invisible speculation designs unprotect a load only when it becomes the oldest load or the oldest instruction in the ROB. This is the case in InvisiSpec's Futuristic mode [56], SafeSpec's wait-for-commit mode [28], Conditional Speculation [33], and MuonTrap [5]. These designs make it impossible to reorder unprotected victim loads, as no two such loads can execute concurrently. As noted above, however, the same effect—secret-dependent order—can be achieved if the attacker performs the reference access. For this, the attacker simply needs to issue an LLC access to the same set accessed by the V^D load from another core, at a fixed time after inducing the mis-speculation. This attack can be based on either of the G^D_{MSHR} or G^D_{NPEU} gadgets.

V^I-A^D ordering. As in the V^I-V^D case, the G^D_{MSHR} and G^D_{NPEU} gadgets can be used to target the branch condition, delaying a post-squash instruction fetch on the correct execution path. This can be measured using the attacker's LLC access as a reference clock. In contrast, the G^I_{RS} gadget only impacts the timing of instruction fetches in the mis-speculated path. Hence, the delay it introduces for instruction fetches can only be observed if I-Cache accesses are not protected by the invisible speculation scheme, as in InvisiSpec and DoM.

Attack landscape summary. *Every* invisible speculation design we have evaluated is vulnerable to at least one of the attacks described above. Table 1 summarizes which designs are vulnerable to which attack combinations. The differences in security manifest in whether an attacker can reorder unprotected victim accesses or must rely on its own access as a "reference clock."

3.3 Existence of Interference Gadgets/Targets (Senders)

We refer to the combination of an interference gadget and a target as a *sender*, i.e., the sending side of the cache covert channel. It is natural to ask if senders exist in "the wild," given their specific structure. There are several real-world attack settings [25] in which the attacker has some control over the instruction stream and can craft senders. These settings include (1) the *in-domain* setting, where a software sandbox executes attacker-controller code, as in the case of in-browser JavaScript code or user-supplied Linux eBPF kernel extensions [40]; and (2) the *domain-bypass* setting, where the attacker runs its own program, attempting to use its mis-speculated

execution to steal secrets from another hardware protection domain, e.g., Meltdown [34]. Finding whether speculative interference senders exist in victim programs is interesting future work.

Conceptually, however, even the fact that senders *might* exist creates uncertainty about security on an invisible speculation system. Users and developers cannot *know* if their program contains a sender without performing program analysis to verify their non-existence. Having to rely on such analysis to guarantee security undermines the efficacy of invisible speculation as a software-transparent hardware defense.

4 ATTACK DEMONSTRATIONS

In this section, we demonstrate concrete proof-of-concept (PoC) speculative interference attacks based on the ideas from § 3 on a commercial machine. Although invisible speculation schemes are not implemented today, we can emulate their behavior by arranging for loads that would be made 'invisible' to return data in secret-dependent amounts of time. At the same time, by evaluating on real hardware, we must address many details in real machines that are simplified in simulators (e.g., LLC replacement policies, RS limits).

We evaluate multiple D-Cache PoCs and a variation of the I-Cache PoC described in § 3.1.1 namely, G^{D}_{NPEU}, G^{D}_{MSHR} and G^{I}_{RS}. All the PoCs were successfully implemented and the attacks successfully leak secret bits to the attacker. We only show the G^{D}_{NPEU} and G^{I}_{RS} attacks for space and refer to them as the D-Cache PoC (§ 4.2) and I-Cache PoC (§ 4.3) respectively. Of independent interest, our D-Cache PoC requires constructing a novel receiver able to read changes in replacement state for the QLRU_H11_M1_R0_U0 replacement policy (§ 4.2.2). All attacks change cache state, with a receiver (attacker) that monitors execution from another physical core (CrossCore; § 2.1).

4.1 Methodology

Processor details. We evaluate on an Intel Core i7-7700 Kaby Lake CPU with 4 physical cores running at a base frequency of 3.6GHz, with hyper-threading enabled. Each core has a unified reservation station, that is shared across execution units, stores up to 97 micro-ops, and has 8 execution unit ports (numbered 0 through 7). Each core has two levels of private cache (a 32KB L1-instruction and 32KB L1-data cache, 256KB of combined L2) and 8MB of Shared L3 (LLC) cache [1, 2].

Tools borrowed from prior work. We trigger branch mispredictions by training the target branch in a given direction (similar to [31]). Likewise, we delay branch resolution by having the branch predicate be the result of a pointer chase. The attacks also use a Flush+Reload-style [58] receiver.

Finally, the D-Cache PoC uses standard techniques to construct eviction sets in the LLC [35], which are sets of cache lines that map to the same LLC set in the same LLC slice. By accessing lines in an eviction set, the attacker can efficiently evict other lines whose set and slice is known.

4.2 D-Cache PoC

Recall from § 3.1.1, the key principle in the G^{D}_{NPEU} attack is for the attacker to observe the reordering of two bound-to-retire loads. Our PoC measures this ordering by mapping the two loads to the same LLC set and measuring changes in replacement state.

To deploy the attack there are two ingredients that need to be developed. First (§ 4.2.1), an implementation of the G^{D}_{NPEU} sender, i.e., to reorder older bound-to-retire loads. Second (§ 4.2.2), a novel receiver capable of measuring differences in LLC replacement state. We consider both of these to be of independent interest, i.e., to reorder older non-load instructions to perform different speculative interference attack variants or to be used in entirely different attack settings (in the case of the replacement state-based receiver).

4.2.1 Sender (Load Reordering). We use the same notation to denote the victim load A and reference load B as in § 3.2. To reorder the loads to two addresses `A` and `B`, we follow the structure from Figure 6. Namely, there are two sequences of instructions, `f(z)` and `g(z)`, that generate addresses `A` and `B` respectively. An interference gadget only affects `f(z)`. In presence of the gadget, load A is delayed to issue after load B whereas regularly it would issue before load B.

First, we consider the address generation for load A and the interference gadget in isolation. We implement `f(z)` and `f'(x)` (Figure 6) as repeated sequences of same instructions, called the *target instruction* and *gadget instruction*, respectively.

We pick suitable instructions (i.e., that maximize the interference of the gadget on the target) as follows. We identify high latency, low-throughput instructions that use the same execution port. Low-throughput allows for an issued instruction in the interference gadget to block the execution port of ready-to-schedule instructions in the interference target; high latency maximizes the time it blocks instructions in the interference target. Finally, the gadget instruction should be composed of only a few micro-ops. This allows more instructions in the interference gadget to occupy RS simultaneously, which increases the likelihood of them getting issued concurrent to the target instructions.

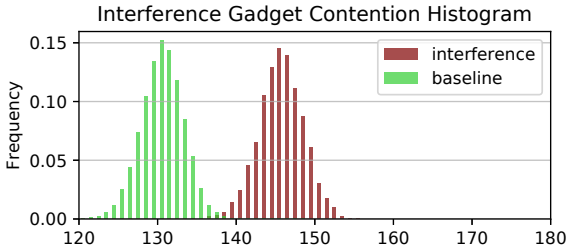


Figure 7: The average time (measured with a clock thread [42, 45]) to execute the interference target changes by ~ 16 clock ticks (80 rdtsc cycles) based on the presence or absence of the interference gadget.

Based on the above process, our PoC uses the `VSQRTPD` instruction for both gadget and target. `VSQRTPD` consists of only 1 micro-op executed on the core's execution port 0 and has observed latencies of 15–16 cycles and reciprocal throughput of 9–12 cycles [16]. We also verified that the attack is functional with `VDIVPD`. Figure 7 shows the time from the issue of the first instruction of `f(z)` to the completion of the load A in the presence (interference) and absence (baseline) of the interference gadget's

execution. The takeaway is there is a clear timing difference in the interference target's execution depending on presence/absence of the interference gadget's execution. This is the secret-dependent delay imposed by the gadget on the victim load.

4.2.2 Receiver (Monitoring Replacement State). With the capability to reorder two loads, the next ingredient for the attack is to translate a reordering of loads into a persistent cache-state change. We achieve this using the cache replacement state.[2] For the rest of the section, we use the notation A-B to indicate the order in time in which the loads are issued, i.e., A-B means A issues first and vice-versa. We also assume access to eviction sets (EV; § 4.1).

Our attack targets the replacement state because we are only changing the order of loads. Changing the order of loads is different than changing which loads are issued as in a normal cache-based attack. For example, a standard LLC Prime+Probe attack, without a very fine probe granularity, would observe both A and B in the cache, regardless of their order and be unable to distinguish A-B from the B-A case.

Translating load issue order into a persistent replacement state change is not difficult in textbook replacement policies, such as LRU, as the ordering directly influences replacement priority ranking. However, replacement policies in modern machines, such as our target processor, are more complex. The new technical challenge for the attacker is that fresh insertions of A and B are ranked equally.

This new challenge can be overcome by providing a technique to extract replacement state data from the replacement policy on the Kaby Lake machine. To identify the replacement policy on our machine, we used a CacheAnalyzer tool by nanoBench [3]. The resulting replacement policy is approximately QLRU_H11_M1_R0_U0 ("Quad-age LRU") on specific cache sets [51].[3] QLRU is a Static-RRIP Replacement policy variant with a 2 bit field used for the age of a cache line [3, 26], summed up here:

- M1: Insertion policy. Inserts cache lines with age 1.
- H11: Hit promotion policy. Promotes a line of age 3 to age 1, age 2 to age 1, and age 1/0 to age 0 upon hit.
- R0: Eviction policy. Insert to leftmost location if cache set is not full; otherwise, evict block corresponding to the leftmost physical tag with age 3.
- U0: Age update policy. Increments age fields of all cache lines until there is a candidate ready for eviction (age = 3).

Attacker Receiver Protocol. We now describe how the attacker decodes from the replacement state whether A-B or B-A occured. At a high level, similar to a traditional cache attack, the attacker thread first primes the LLC set, waits for the victim to issue its secret-dependent ordering, and finally probes the LLC set to determine which ordering the victim issues. Due to the nature of QLRU, however, the details are different from conventional attacks. Specifically, the attacker first constructs two eviction sets of size LLC_ASSOCIATIVITY-1 elements, call these EVS1 and EVS2, which map to the same LLC set and slice as A and B. The attacker then uses the following access sequences to prime and probe the cache set:

- *Prime Sequence: Access EVS1 many times + Access A*
- *Probe Sequence: Access EVS2*

The attacker accesses EVS1 many times in order to saturate their age at 0, leaving A with an age of 3. To be able to access address A, our current PoC requires that the receiver share memory with the victim (hence the use of Flush+Reload).

For our machine, the targeted cache sets are 16-way associative. We will refer to elements in EVS1 as EV0-EV14, and elements in EVS2 as EV15-EV29. The resulting cache states for prime and probe with the A-B sequence is displayed in Figure 8. The main idea is that only A or B is still resident in the LLC by the end of prime+victim_accesses+probe sequence.

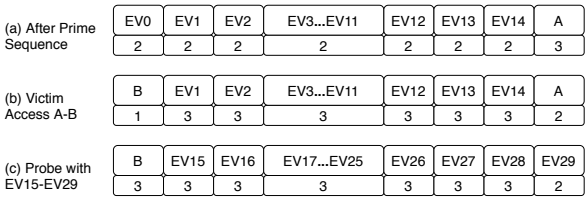


Figure 8: QLRU State for the targeted cache set. EV*N*, A, B represent addresses and numbers represent the age for each cache line. (a) shows the cache state after attacker primes the cache. (b) & (c) represent the cache states after the victim runs (with pattern A-B) and after the attacker completes the probe. A victim access pattern of B-A has analogous state changes.

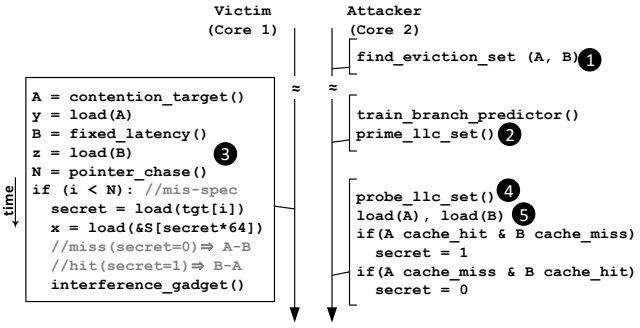


Figure 9: An End to End visualization of the D-Cache attack.

4.2.3 End-to-End Attack. In this section, we present the overall D-Cache PoC. The attack steps shown in Figure 9 are explained in detail below:

① Attacker initializes eviction sets based on addresses A, B.

② Attacker primes the LLC set replacement state (§ 4.2.2) and mis-trains the victim's branch predictor.

③ Victim issues loads A and B, where order depends on the secret (§ 3.1.1). If secret = 0, A-B is issued, and if secret = 1, B-A is issued.

[2]RELOAD+REFRESH [10] also uses replacement-state manipulation principles to execute a cache-based attack. The distinction in this work is we try to identify the victim's load issue order, whereas they try to identify the presence of a victim's access to a target address.

[3]It is likely the case that the LLC cache sets do not strictly abide by this replacement policy and have an adaptive replacement policy. However, for the purposes of this PoC, the attack strategy that creates observable replacement state changes on QLRU_H11_M1_R0_U0, also creates observable replacement state changes on our machine.

④ Attacker probes the LLC set replacement state (§ 4.2.2) and observes the residency of lines for addresses A or B in the LLC set. The residency is determined by issuing a timed access to address A and B and comparing it with a LLC cache miss threshold.
⑤ Attacker attempts to identify the secret bit. If the victim issues the load sequence A-B (secret = 0), the expectation is for load A to be a cache miss and load B to be a cache hit. If the victim issues the load sequence B-A (secret = 1), the expectation is for load A to be a cache miss and load B to be a cache hit. Cases where both accesses are cache misses can happen due to noise and are ignored.
⑥ Attacker repeats steps 2-5 as needed to increase confidence.

Note, while our PoC observes load-reordering through replacement state, other receivers not based on replacement state might be possible.

4.3 I-Cache PoC

We now describe a PoC for the G^{I}_{RS} attack from § 3.1.1. The attack works by creating contention on available reservation stations to create a ripple effect that eventually stops the frontend from fetching more instructions, causing changes to the I-Cache access pattern.

4.3.1 Experiment Setup. As with the D-Cache PoC, we show the I-Cache PoC given an attacker that monitors state from another physical core through the LLC. The pseudocode for the victim is described in Figure 5. Without loss of generality, the target instruction used at line 9 in Figure 5 is a shared library function call. For simplicity, our attack slightly differs from that explained in Figure 5. Specifically, we move the target instruction into the mis-speculated path (before the branch join point). Thus, in a correct execution, the target instruction will not be fetched (as opposed to fetched later). The receiver (attacker) on the adjacent physical core issues a load to a shared library function to perform the reload step.

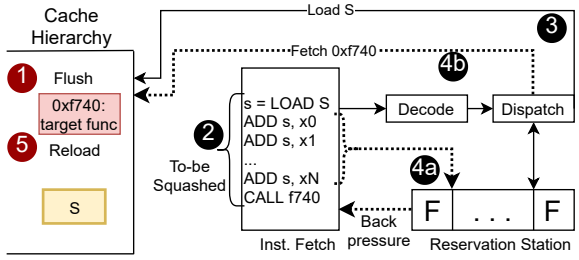


Figure 10: Steps involved in performing an I-Cache attack. Dotted lines are conditional events depending on Block S's presence in cache. Red represents attacker actions and black represents victim actions.

4.3.2 Attack Details. Here we refer to the pseudocode in Figure 5 and describe the attack sequence. Item numbers refer to steps in Figure 10.
① Attacker first primes the cache hierarchy by flushing out the target address (shared function pointer) from the I-Cache.
② When the victim runs next, it mis-speculates on a branch (line 1 in Figure 5) that prompts the frontend to fetch transient (bound-to-squash) instructions, which dispatches a secret-dependent load instruction at address S, followed by a large number of arithmetic instructions dependent on the loaded value s.
③ The next steps proceed similarly to what we describe in the § 3.1.1 G^{I}_{RS} paragraph. The victim dispatches the secret-dependent load (Load S in Figure 10, line 4 in Figure 5), which is setup to hit or miss in the cache hierarchy based on a secret value S.
④a Miss: Load S misses, creating a frontend stall due to dependent ADD instructions. Hence, the target instruction (Call `0xf740` in Figure 10) is not fetched. When the branch resolves, execution continues but because the target instruction was on the mis-speculated path, the target address was never fetched into the cache.
④b Hit: Load S returns quickly and no RS congestion occurs. The target instruction is executed and hence the target address cache line is fetched into the I-Cache. Since the branch resolves after the target instruction is executed, the fetched line leaves a persistent change in cache state after the mis-speculated instructions are squashed.
⑤ After waiting for the victim to run, the attacker performs a standard probe step, re-loading the shared function pointer, either via a function call (I-Cache) or by loading the contents of the function pointer (D-Cache).

4.4 Attack Evaluation

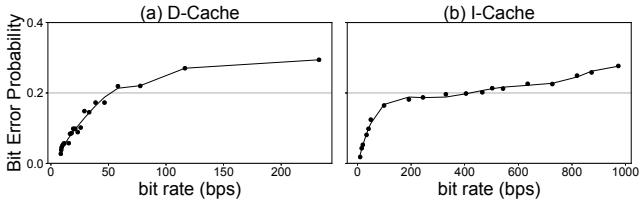


Figure 11: Attack PoC channel error vs. bit rate. (a) DCU PoC (§ 4.2), (b) ICU PoC (§ 4.3).

We run the PoCs in a cross-core setting and evaluate the end-to-end covert channel error rate vs. throughput in Figure 11 (a) and (b) for the D-Cache and I-Cache PoCs, respectively. Throughput is defined as the number of secret bits transmitted per unit time. It is represented as bits per second (bps) and evaluated by measuring the CPU cycles required to leak 1 bit. Error rate is defined as the number of incorrectly inferred bits over the total number of bits transmitted. We can trade-off error rate and bit rate by changing PoC parameters, e.g., the number of times the PoC is run to leak each bit, or the amount of time spent trying to mistrain the branch predictor. We note the I-Cache PoC has a higher transmission rate than the D-Cache PoC because reordering loads in the D-Cache PoC has a higher chance to fail due to noise in the system.

5 DEFENSES

We now discuss various approaches for invisible speculation designs to block speculative interference attacks. To this end, we first propose a formal definition of what it means to block all cache covert channels (§ 5.1). We describe two designs that achieve this goal. The first design (§ 5.2) is straightforward, but imposes significant performance overhead, unlike current designs [5, 39]. We

thus propose a high-level approach for a more efficient solution (§ 5.3), whose exploration we leave to future work.

5.1 Ideal Invisible Speculation

We define an *ideal invisible speculation* security property, which formally models the security goal of eliminating all speculative execution-induced cache covert channels. Informally, ideal invisible speculation requires that the system's cache state is invariant of speculative execution.

More formally: We assume a multi-core system with private L1 I- and D-Caches and a shared L2 cache (or L2). In an invisible speculation design, the L2 can receive visible and invisible accesses. A *visible* access corresponds to a standard cache fill or writeback, causing changes in both the L1 and L2. An *invisible* request is a request type added by the defense; it does not cause state changes in the L2 and its response does not change state in the L1. We assume that the attacker sees the sequence (without timing information) of visible L2 accesses. We call this the *L2 access pattern.*

We formulate the security goal of the L2 access pattern being invariant of speculation as follows. Given an execution E of the microarchitecture, define $C(E)$ as the L2 access pattern in E. Define $NoSpec(E)$ as the execution that would have occurred if E had no mis-speculations. Then *ideal invisible speculation* is the following property, akin to non-interference [19]: For any execution E: $C(E)$= $C(NoSpec(E))$.

5.2 Basic Defense Design

Here, we present a simple solution that can provide ideal invisible speculation. The idea is that, when instructions that might cause a mis-speculation are inserted in the ROB, the hardware automatically inserts a special type of fence. The fence allows subsequent instructions to be inserted into the ROB but prevents them from being issued until the instruction before the fence becomes non-speculative.

To achieve ideal invisible speculation, fences must be inserted after any instruction that may cause a squash. This threat model (considering all forms of speculation) is sometimes referred to as the *Futuristic* model [56]. The design can be tuned to consider only control-flow speculation (the *Spectre* model [56]) by placing fences only after branches. This requires adjusting the security property to consider only control-flow speculation.

Evaluation of the Basic Defense. We evaluate the performance of the basic defense design on the Gem5 [9] simulator. We model a high performance multi-core system (8-issue OoO 2 GHz cores with 32 KB L1 D-Cache and 2 MB per-core shared L2 banks). This configuration is similar to systems used in previous invisible speculation work (e.g., [5, 39, 56]). We use the SPEC CPU2017 [23] benchmarks with reference input size and Simpoints [46] to identify around 10 representative execution regions per benchmark and run 10 million instructions per simpoint.

Figure 12 shows the performance overhead of the basic defense design over the unsafe, unmodified processor, under both Spectre and Futuristic threat models. When adding the basic defense scheme, the execution time becomes on average 1.58× the execution time of the unsafe baseline for Spectre threat model. In terms of the Futuristic threat model, the execution time of the basic defense

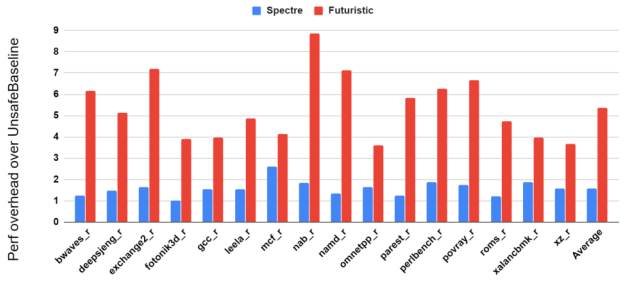


Figure 12: Performance of basic defense on SPEC2017 benchmarks.

scheme is on average 5.38× the execution time of the unsafe baseline. In comparison, prior works report overheads of < 5% and < 20% in the Spectre and Futuristic models, respectively [5, 39, 56]. Therefore, while the simple solution can achieve ideal invisible speculation, it does so at a dramatic performance cost.

5.3 Discussion: Potential Advanced Defense

A high level principle for achieving ideal invisible speculation is: *a speculative instruction must not influence the execution of a non-speculative instruction.* This principle does not preclude an instruction from speeding up younger instructions, which is the basis for invisible speculation's performance benefits. It does imply the microarchitectural rule of blocking loads from changing cache state while speculative, as that can affect non-speculative instructions. As we have shown, however, more microarchitectural rules are required to realize this principle, creating a design space to explore.

One possible approach is based on the following two rules: (1) no instruction ever influences the execution time of an older instruction, and (2) any resources allocated to an instruction at the interface of the frontend and the execution engine are not deallocated until the instruction becomes non-speculative. Rule (1) is straightforward to obtain if every microarchitectural resource is perfectly pipelined but requires further research to implement otherwise. Rule (2) makes instruction fetch rate invariant of speculation, which guarantees that speculation cannot influence when instructions that eventually retire begin executing.

Not Influencing Older Instructions. This rule can be implemented by assigning a priority tag to each instruction, based on its age in the ROB. The general strategy follows three steps. First, when two instructions with priorities i and j are about to use any shared resource, the hardware gives precedence to the instruction with higher priority. Second, to prevent counter wrap-around problems, we maintain two priority tags (head and tail) and use logic akin to FIFO full/empty to check priority tag values on instructions. Third, when a branch is squashed, the priority tag is reset to the correct value.

This design is straightforward when every single resource is perfectly pipelined. This is currently the case for pipelined EUs, cache ports and banks, and writeback links. However, for resources that are not perfectly pipelined (e.g., non-pipelined EUs), three different choices are available.

One approach is to make them fully pipelined (at the expense of longer latency). A second approach is to make the corresponding resource scheduler smarter, so that it "looks ahead in time" and anticipates if, by assigning the resource to a low-priority instruction now, one may have to stall a higher-priority instruction later. In this case, the low priority instruction is stalled until the higher priority one uses the resource. This strategy may not be possible all the time (or be quite expensive/slow). The third approach is to design the EU to be "squashable". This means that it can be freed-up on demand if a higher-priority instruction requests the EU. This complicates the design, as it requires that the instruction currently using the resource be "re-issuable" (i.e., the hardware holds its state and can reuse it to relaunch the operation).

Not Deallocating Resources Early. This rule requires that a speculative instruction releases the hardware resources it uses only when it becomes non-speculative or gets squashed. Examples of such resources are reservation stations and execution units. This rule makes the duration of time that the instruction occupies any resource independent of the instruction's operands. The trade-off is that the instruction may hold on to the resource for longer than in the baseline design.

Takeaway. Extending the invisible speculation approach to block speculative interference attacks appears to involve significant complexity and efficiency costs. Whether defenses focusing on only cache attacks—but *fully* blocking them— can be simpler or more efficient than defenses with more comprehensive threat models [7, 53, 61, 62] is an interesting question for future work.

6 RELATED WORK

Most speculative execution attacks that have been presented to date build on cache-based covert channels to leak data, being inspired by either Spectre [12, 24, 31, 32, 36, 52] or Meltdown [11, 34, 41, 49, 50, 54]. To our knowledge, only SMoTherSpectre [8] and NetSpectre [44] make use of alternative covert channels, such as port contention, for speculative execution attacks.

We provide background on *invisible speculation* schemes [5, 28, 33, 39, 56] in § 2. CleanupSpec [38] targets the invisible speculation goal of blocking only cache covert channels but uses a unique approach of (1) undoing cache occupancy changes upon a squash and (2) using randomized replacement to block replacement-related leakage. CleanupSpec does not block speculative interference but makes its exploitation more challenging. CleanupSpec relies on DoM or InvisiSpec to protect the I-Cache. Thus, a speculative interference attack that delays branch resolution based on a secret can change the order of victim I-Cache accesses and attacker D-Cache accesses and yield an attack (as in V^I-A^D ordering, § 3.2.1). For D-Cache protection, CleanupSpec uses randomized/encrypted address caches, which do not hide whether a previously un-cached line was accessed or not. Thus, a similar attack as in the I-Cache case can go through by delaying a branch resolution, whereby the attacker can change how many D-Cache accesses are made on the mis-speculated path (V^I-V^D ordering, § 3.2.1). We leave this as future work.

Beyond invisible speculation, there are several other hardware mechanisms designed to block speculative execution attacks [7, 18, 29, 43, 47, 53, 60–62]. Data-oblivious ISA extensions [60], SpectreGuard [18], ConTExT [43], DAWG [29] and CSF [47] require some degree of software support (e.g., setting up cache partitions, users annotating what data is secret) which severely constrains adoption. STT [61, 62], NDA [53] and SpecShield [7] are software-transparent hardware mechanisms that propose selective speculation, allowing certain instructions to execute speculatively while delaying (or executing in a data-oblivious fashion [61]) others. These schemes, in principle, can block speculative interference attacks by delaying speculative instructions beyond loads. Yet, none can comprehensively defeat all speculative interference attacks. For example, while STT soundly blocks speculative interference attacks that leak transiently accessed data, it offers no protection against attacks that leak non-transiently accessed (bound-to-retire) data.

Concurrent to this work, Fustos et al. [17] also observed that younger speculative instructions can influence the timing of older bound-to-retire instructions. Yet, their SpectreRewind attack is a traditional contention attack (§ 1) and explicitly outside of the scope of invisible speculation schemes.

7 CONCLUSION

This paper presented speculative interference attacks, which show that invisible speculation schemes are not immune to cache attacks. The broader implication of our work is to demonstrate the security pitfalls of a well-studied approach to building secure processors, namely to ignore "bandwidth" or "contention" or "intermittent" covert channels and solely focus on cache-based channels. Specifically, we show how an attacker can *convert* timing changes into persistent-state changes. Long term, we hope our work helps set a research agenda towards more comprehensive security definitions and more secure, efficient invisible speculation mechanisms.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Dmitry Ponomarev for their valuable feedback. This work was funded in part by NSF under grants 1942888 and 1954521, ISF under grant 2005/17, Blavatnik ICRC at TAU, and by an Intel Strategic Research Alliance (ISRA) grant.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact contains two attack PoCs, the (`dcache_poc`) and (`icache_poc`) as described in § 4. The artifact evaluates the entire attack idea described in the paper in a systematic manner. It contains makefiles, run scripts and detailed README to as a walkthrough for each step and to reproduce the result figures described in the main paper. The artifact has been validated on Intel CPUs (i7-7700 and i7-9700) running Ubuntu OS (18.04). The PoC's require a relatively controlled multi-core environment, with two cores dedicated to running attacker and victim snippets; a several-core machine is preferred. Also, as a step in verifying the replacement state policy, there is a requirement of installing kernel modules related to the nanoBench [3] workflow.

A.2 Artifact Check-List (Meta-information)

- **Algorithm:** Using existing methods to create eviction sets, but implementing specific PoCs developed in the paper
- **Program:** D-Cache and I-Cache PoCs
- **Compilation:** gcc
- **Run-time environment:** Ubuntu 18.04
- **Hardware:** Intel i7-7700
- **Run-time state:** Sensitive to noisy run-time state, ideally requires low traffic/evictions from the LLC
- **Execution:** Pre-profiling for replacement state policy
- **Output:** Figures 7 and 11
- **Experiments:** Timing difference distributions and leak accuracy rates
- **Publicly available:** Yes
- **Code licenses:** Apache-2.0 License
- **Data licenses:** None

A.3 Description

A.3.1 How to Access. The PoC's can be accessed from Github: https://github.com/FPSG-UIUC/speculative_interference

A.3.2 Hardware Dependencies. The proposed attack PoC relies on a proper contention induced by the gadget instructions (Figure 7) and the specific replacement algorithm in the LLC (QLRU_H11_M1_R0_U0). We have tested it on Intel i7-7700 CPU. We expect any processor having the above mentioned replacement state should run the experiment successfully.

A.3.3 Software Dependencies. We have tested our PoCs on Ubuntu 18.04 and Ubuntu 20.04. We have seen issues with some of the gcc default compiler options (position independent code) while using Ubuntu 16.04 with the I-Cache PoC. We do not have any specific kernel or package dependencies. For our runs, we used gcc 9.3 with make 4.2.1 and python 2.7.12 for the included scripts.

A.4 Installation

No specific installations required, but we do require the use of a run script to set the environment which is provided as `run-first.sh`.

A.5 Experiment Workflow

D-Cache PoC. Initially we confirm the robustness of the interference gadget used in the PoC. The desired result is a clean difference in the timing distributions of the interference target in presence/absence of the gadget. Then testing is done to verify the correct replacement state policy (QLRU_H11_M1_R0_U0) for which the PoC is developed. This was achieved using the nanoBench test suite to converge on the given replacement policy. If an alternative replacement policy is detected, the prime and probe sequences may need to be adjusted. Once the replacement policy is confirmed and before the PoC can be run, a contiguous buffer in memory is allocated using shm-crea.c to be used for eviction-set development. The page ids output from shm-crea are stored in a file named "pageids.txt" and are read from the dcache_poc. The PoC itself runs in a series of steps, beginning with eviction set and flush set establishment. Once these have been initialized, a succession of experiments are ran leading up to the multi-core PoC. Further details of running the tests are specified in the README.

I-Cache PoC. The I-Cache PoC has two variants where the attack PoC can be run as a single core or cross core multi-threaded attack. The attack makes use of specific assembly code and jump addresses that needs to be set post compilation into the attack code. The exact steps to achieve this are enumerated in the README file provided. The attack can be run on same core or cross core using `S` or `C` keywords respectively while running the multiThreadIC binary. As with the dcache_poc, the exact build and run steps are provided in the README under icache_poc/multiThread/ directory.

A.6 Evaluation and Expected Results

The PoC evaluates the attack as described in the paper. The scripts included help recreate Figures 7 and 11. Figure 7 shows measurements observed in the presence and absence of the attacker (interference gadget). In Figure 11, we show the bit error rate vs. bit rate (throughput). Figure 11 is created by running the experiment multiple times for a set number of bits and then taking the average time spent on each run versus the number of bits correctly leaked.

A.7 Experiment Customization

D-Cache PoC. The configurable parts of the D-Cache PoC are specified in the README. Important parameters for the success of the PoC include the fixed delay used to offset the load B, as well as the iterations of the dependency chain of instructions used to issue load A. The interaction between load A and load B needs to be configured to have an ordering influenced by the secret-dependent contention.

I-Cache PoC. The I-Cache PoC can be customized for the number bit-leaks and the variant of the attack we want to run. All the customizations are detailed in the provided README.

REFERENCES

[1] [n.d.]. 8th and 9th Generation Intel® Core™ Processor Families Datasheet, Volume 1 of 2. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/8th-gen-core-family-datasheet-vol-1.pdf.

[2] [n.d.]. Kaby Lake - Microarchitectures - Intel - WikiChip. https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake.

[3] Andreas Abel and Jan Reineke. 2019. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. *arXiv preprint arXiv:1911.03282* (2019).

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

[5] Sam Ainsworth and Timothy M. Jones. 2020. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. In *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[6] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port contention for fun and profit. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[7] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2019. SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[8] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News* 2 (2011), 1–7.

[10] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. 2020. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *Proc. of the USENIX Security Symposium (USENIX)*.

[11] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-Resistant CPUs. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[12] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*.

[13] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Proc. of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[14] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Understanding and Mitigating Covert Channels Through Branch Predictors. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016).

[15] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proc. of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[16] Agner Fog et al. 2011. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering* 93 (2011), 110.

[17] Jacob Fustos, Michael Bechtel, and Heechul Yun. 2020. SpectreRewind: Leaking Secrets to Past Instructions. *arXiv preprint arXiv:2003.12208* (2020).

[18] Jacob Fustos, Farzad Farshchi, and Heechul Yun. 2019. SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks. *Proc. of the Design Automation Conference (DAC)* (2019), 1–6.

[19] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[20] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. 2020. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*.

[21] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2009. Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications. In *Proc. of the International Conference on Information Security and Cryptology (ICISC)*.

[22] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc.

[23] John L Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News* 4 (2006), 1–17.

[24] Jann Horn. 2018. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[25] Intel. 2020. Refined Speculative Execution Terminology. https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology.

[26] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH Computer Architecture News* 38, 3 (2010), 60–71.

[27] Mike Johnson. 1991. *Superscalar Microprocessor Design*. Prentice Hall Englewood Cliffs, New Jersey.

[28] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *Proc. of the Design Automation Conference (DAC)*.

[29] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *Proc. of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[30] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *arXiv preprint arXiv:1807.03757*, Article arXiv:1807.03757 (2018). arXiv:1807.03757 [cs.CR]

[31] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[32] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *Proc. of the USENIX Workshop on Offensive Technologies (WOOT)*.

[33] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. 2019. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *Proc. of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[34] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proc. of the USENIX Security Symposium (USENIX)*.

[35] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. https://doi.org/10.1109/SP.2015.43

[36] Giorgi Maisuradze and Christian Rossow. 2018. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[37] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proc. of the Cryptographers' Track at the RSA Conference (CT-RSA)*.

[38] Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. CleanupSpec: An "Undo" Approach to Safe Speculation. In *Proc. of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[39] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction. In *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[40] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. 2018. Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt.

[41] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[42] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic timers and where to find them: high-resolution microarchitectural attacks in JavaScript. In *Proc. of the International Conference on Financial Cryptography and Data Security (FC)*. Springer.

[43] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. 2019. ConTExT: Leakage-Free Transient Execution. *arXiv e-prints*, Article arXiv:1905.09100 (May 2019). arXiv:1905.09100 [cs.CR]

[44] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*.

[45] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *Proc. of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[46] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proc. of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[47] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-Sensitive Fencing : Securing Speculative Execution via Microcode Customization. In *Proc. of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[48] Robert M Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33.

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

[50] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[51] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. 2020. CacheQuery: Learning Replacement Policies from Hardware Caches. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[52] Jack Wampler, Ian Martiny, and Eric Wustrow. 2019. ExSpectre: Hiding Malware in Speculative Execution.. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*.

[53] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *Proc. of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[54] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report* (2018).

[55] Wenjie Xiong and Jakub Szefer. 2020. Leaking Information Through Cache LRU States. In *Proc. of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[56] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proc. of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[57] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[58] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *Proc. of the USENIX Security Symposium (USENIX)*.

[59] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* 7, 2 (2017), 99–112.

[60] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. 2019. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*. https://eprint.iacr.org/2018/808.

[61] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. 2020. Speculative Data-Oblivious Execution (SDO): Mobilizing Safe Prediction For Safe and Efficient Speculative Execution. In *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[62] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proc. of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*.