NDA: Preventing Speculative Execution Attacks at Their Source

Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas Wenisch, Baris Kasikci
University of Michigan

ABSTRACT

Speculative execution attacks like Meltdown and Spectre work by accessing secret data in wrong-path execution. Secrets are then transmitted and recovered by the attacker via a covert channel. Existing mitigations either require code modifications, address only specific exploit techniques, or block only the cache covert channel. Rather than battling exploit techniques and covert channels one by one, we seek to close off speculative execution attacks at their source. Our key observation is that these attacks require a chain of dependent wrong-path instructions to access and transmit secret data. We propose *NDA*, a technique to restrict speculative data propagation. *NDA* breaks the attacks' wrong-path dependence chains while still allowing speculation and dynamic scheduling. We describe a design space of *NDA* variants that differ in the constraints they place on dynamic scheduling and the classes of speculative execution attacks they prevent. *NDA* preserves much of the performance advantage of out-of-order execution: on SPEC CPU 2017, *NDA* variants close 68-96% of the performance gap between in-order and unconstrained (insecure) out-of-order execution.

1. INTRODUCTION

Speculative execution attacks [1–16] exploit micro-architectural behavior and side channels to exfiltrate sensitive information from a system. Unlike classical software exploits that modify and observe only architectural state (such as registers and memory), speculative execution attacks have demonstrated that attackers can retrieve secrets by controlling and observing micro-architectural state (e.g., the cache) during speculative wrong-path execution.

Speculative execution attacks can be classified into two main categories. One class (e.g., Spectre [2], Spectre 1.1 [11], and others [12–16]) allows malicious code to mis-steer a victim program's control flow (e.g., by carefully mis-training branch predictors) to execute specific instructions on the speculative wrong path. Although wrong-path instructions are ultimately squashed (with no effect on architectural state), the victim program is coerced into leaking its own memory contents through a micro-architectural channel. For instance, Chen et al. [16] show how control-flow in an SGX secure enclave [17] can be steered to leak its own protected memory. We classify these attacks as *control-steering* attacks (Fig. 1a).

Another class of attacks [1, 4–10] enables unprivileged attacker code to access privileged memory that is temporarily exposed during wrong-path execution. For instance, Meltdown [1] allows reading kernel memory; Foreshadow [4–6] allows reading hypervisor, OS, SMM, or SGX memory; and LazyFP [7] allows reading AES keys from AVX registers used by another process. MDS attacks [8–10] allow reading recently accessed memory belonging to other processes. Since the attacker generates the code, they can select arbitrary instruction sequences in both correct-path and wrong-path execution. We classify these attacks as *chosen-code* attacks (Fig. 1b). These two classes of attacks are fundamentally different and therefore require different approaches for mitigation.

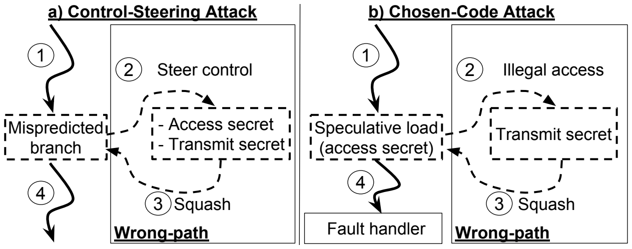


Figure 1: Control-steering vs. chosen-code attacks.

Existing software defenses against speculative execution attacks work by modifying a program's source code to block attack-specific mechanisms. Current software defenses for control-steering attacks—such as Retpoline [18, 19], IBPB [20], and improved `lfence` [21] instructions—focus on preventing the attacker from steering the execution of victim code. Unfortunately, these defenses are not immediately applicable to existing binaries. Specifically, software mitigations against chosen-code attacks involve modifying the OS, hypervisor, and SMM code [6, 22–24]. A recent study by Google [25] discusses why software approaches are an insufficient mitigation for speculative attacks. Even if code modifications are made, these defenses can be bypassed. For instance, attackers can redirect control flow to evade fence instructions (e.g., by mis-training the branch target buffer (BTB) [2, 3] or the return stack buffer (RSB) [11, 13, 14]).

Hardware defenses, on the other hand, have the potential to obviate the need to modify existing software [20, 26–31]. The first disclosed speculative execution attacks [1–3] use caches as a covert channel to leak data from wrong-path execution. Consequently, initial hardware defenses—such as InvisiSpec [26], SafeSpec [27], and others [28–31]—seek to prevent wrong-path execution from leaving secrets in the cache that can later be recovered. Taram et al. [31] suggests a hardware modification to automatically insert `lfence` micro-ops where needed. However, the authors claim to mainly address Spectre v1 attacks that use the data cache as a covert channel.

While these techniques are effective, a recent study [32] noted that closing only the cache covert channel is insufficient to stop speculative execution attacks, since the cache is only one of many potential covert channels. Netspectre [12] and SMoTher Spectre [15] have already shown that secrets can be transmitted via the FPU or via port contention [33]. In §3, we further show how to transmit secrets via the BTB.

Rather than isolating predictive structures [20] or sealing individual covert channels [26, 27]—a ceaseless arms race—we instead seek to close off speculative execution attacks at their source. Our philosophy is to treat potentially wrong-path values as secret and prevent these secret values from propagating through the micro-architecture. Our key observation is that speculative execution attacks require a *chain of dependent wrong-path instructions* to access and transmit data into a covert channel. By preventing potentially wrong-path values from propagating, we break these dependency chains, thwarting the code sequences required to mount attacks.

We propose *NDA*—Non-speculative Data Access—a technique to restrict speculative data propagation in out-of-order (OoO) processors. *NDA* only allows instruction outputs to flow to dependents if the source instruction is considered *safe*. *NDA* restricts data propagation by preventing tag broadcast for unsafe instructions, delaying wake-up of their dependants in the issue queue until the source instruction becomes safe.

We present a taxonomy of the building blocks of speculative execution attacks, show how each class of attack depends upon data propagation in wrong-path execution, and demonstrate how we can define *safe* vs. *unsafe* to prevent the data flow required by the attack. By composing various restrictions on when an instruction becomes safe, we create a design space of *NDA* variants. The variants differ in (1) the constraints they place on the dynamic execution schedule (and therefore, performance), (2) the locations from which secret data might be extracted (e.g., whether general purpose registers are protected), and (3) the kind of speculation attacks they prevent (e.g., control-steering vs. chosen-code).

NDA defeats all 25 documented [15, 32] speculative execution attacks without the need to modify any existing code. Importantly, however, *NDA* does not preclude all speculation or OoO execution. For example, one *NDA* policy treats all instructions after an unresolved branch as unsafe. These instructions may still execute speculatively OoO, but they are restricted from propagating their output to dependents until all preceding branches resolve. As our evaluation demonstrates, despite delayed wake-ups, the vast majority of the performance gap between in-order (the only other model known to eliminate all known speculative execution attacks) and unconstrained OoO execution is recovered.

We simulate *NDA* designs on the SPEC CPU 2017 benchmark suite and compare its performance to InvisiSpec [26] on the same setup. InvisiSpec blocks cache-based attacks and introduces 36.7-94.1% overhead in our setup. In comparison, *NDA* blocks *all* covert channels. We show that an *NDA* policy that mitigates control-steering vulnerabilities, which are fundamental to unconstrained OoO execution, slows execution by only 19.9% and is 3.8× faster than in-order. If we also preclude Meltdown-like hardware implementation flaws, *NDA*'s strictest policy slows down execution by 113% compared to an insecure OoO processor and is 2.1× better than in-order execution

In short, we make the following contributions:

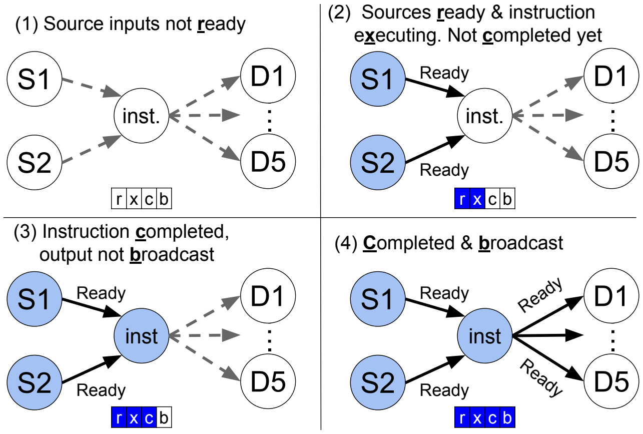

(1) Source inputs not ready
S1
S2
inst.
D1
D5
r x c b
(2) Sources ready & instruction executing. Not completed yet
Ready
Ready
(3) Instruction completed, output not broadcast
inst
(4) Completed & broadcast
Ready
Ready

Figure 2: Life-cycle of instructions in OoO processors.

- We introduce a speculative-execution-attack taxonomy based on how attacks induce wrong-path execution.
- We design *NDA*, a new technique to control speculative data propagation in out-of-order processors to defeat speculative execution attacks. *NDA* offers multiple variants with differing security/performance tradeoffs.
- We evaluate six *NDA* variants on SPEC 2017 and show they are effective and efficient.

2. BACKGROUND

Data Propagation in OoO Processors. Fig. 2 illustrates conceptual steps in an instruction's life-cycle in a modern OoO processor. Upon dispatch into the reorder buffer (ROB), an instruction is not ready to execute until all of its source operands—coming from instructions S1 and S2 in Fig. 2—are ready (step 1). Once all source operands are ready, the instruction issues and enters the execution pipeline (step 2). When execution completes (step 3), the instruction wakes its dependents (D1-D5) by broadcasting a tag corresponding to its destination physical register to waiting instructions (step 4), marking those instructions ready.

The essence of the *NDA* technique is to *delay tag broadcast*, i.e., transition from step 3 to step 4. Rather than waking dependent instructions when their input operands become *ready*, *NDA* wakes them up when their input operands are *safe*. We expand on this basic concept in §5.

Speculative Execution Attacks. Speculative execution attacks exploit side-effects of wrong-path execution, which are typically left undefined by processor vendors. While the contents of architectural registers and memory are guaranteed to reflect precise state of only committed instructions, wrong-path execution affects micro-architectural structures. For instance, a wrong-path cache access may allocate new lines or modify the cache replacement order; these changes are not reverted when wrong-path instructions are squashed. A variety of other micro-architectural structures are also not reverted during squash, for example, branch direction predictors (e.g., pattern history table), pre-decoded micro-op/trace caches, memory dependence predictors, prefetchers, TLBs, fine-grain power management state (e.g., for FPU/AVX units), and performance counters. State changes in these micro-architectural structures can create side channels, where the state can be inferred, for example, based on timing particular execution sequences. We refer to a side channel that is used to intentionally transmit data as a *covert channel*. Attackers can use *wrong-path* execution to transmit data, via a covert channel, that is later inferred by *correct-path* execution and hence leaks that data into architectural state.

3. PROBLEM ANALYSIS

We next classify speculative execution attacks based on three fundamental attack phases that exist in all known attacks. We then describe the existing mitigation techniques, how they block the attacks, and their shortcomings. Lastly, to demonstrate that closing specific side channels is insufficient, we show an attack via a new covert channel—the BTB.

3.1 Classifying Attacks

Attack Phases. All speculative execution attacks of which we are aware comprise three key phases—*access, transmit,* and *recover*—shown in Fig. 3. In the *Access Phase* (①), secret data is loaded into a temporary register. During the *Transmit Phase* (②) the secret data is covertly transmitted using micro-architectural side effects that are not reverted when wrong-path instructions are squashed. Finally, in the *Recover Phase* (③), the transmitted secret is recovered to non-speculative state (e.g., by observing the memory access latency). Whereas the instructions involved in phases ① and ② are speculatively executed and eventually squashed, the phase ③ results are committed to the architectural state. Wrong-path execution is essential to these attacks, as it evades software and hardware protection mechanisms that prevent the secret data from leaking through architectural state.

Control-Steering and Chosen-Code Attacks. We classify attacks based on their methodology for performing the *Access Phase* (①) and the *Transmit Phase* (②). We divide attacks based on their *Access Phase* into two categories, which correspond to different attacker threat models. We further subdivide these two attack classes according to the covert channel exploited in the *Transmit Phase*. Table 1 illustrates this taxonomy for currently-known attacks.

In *control-steering* attacks, the attacker subverts a victim program's control flow to speculatively execute instructions that, as a side-effect, leak data into a covert channel. This attack class leaks data to which the victim application has hardware access privileges, but are intended to be secret and might be protected (e.g., by permission or bounds checks) in software. For example, SGXPectre [16] coerces a secure SGX [17] enclave to access and leak its encrypted memory. We illustrate control-steering attacks in Fig. 1a.

Unlike a classical security vulnerability, wherein the attacker directly hijacks the program counter (e.g., a stack-smashing attack that overwrites a return address), speculative control-steering attacks only misdirect wrong-path execution, for example, by mis-training branch predictors to direct instruction fetch to an attacker-selected target. Hence, they leave no trace in the committed instruction sequence, but still leak data into a covert channel. Several approaches that use control-steering have been demonstrated [2, 11, 13, 14].

In control-steering attacks, the attacker does not typically introduce new instructions into the victim binary, rather, the attacker composes a series of gadgets from existing code, similar to Return Oriented Programming (ROP [34–36]).

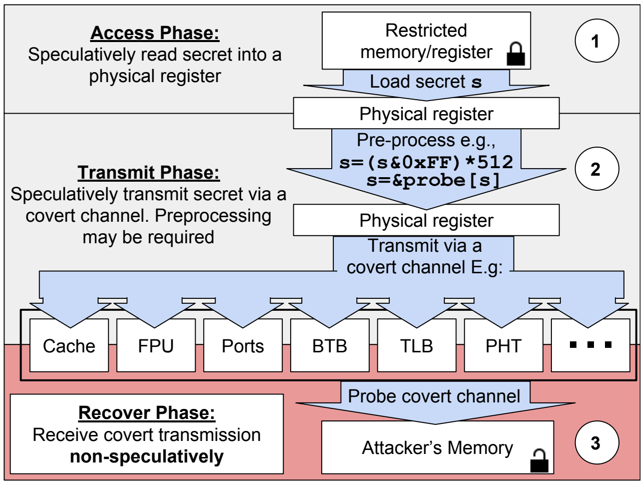

Access Phase:
Speculatively read secret into a physical register
Restricted memory/register
Load secret s
1
Physical register
Transmit Phase:
Speculatively transmit secret via a covert channel. Preprocessing may be required
Pre-process e.g.,
s=(s&0xFF)*512
s=&probe[s]
2
Physical register
Transmit via a covert channel E.g:
Cache
FPU
Ports
BTB
TLB
PHT
...
Recover Phase:
Receive covert transmission non-speculatively
Probe covert channel
Attacker's Memory
3

Figure 3: Three phases of speculative execution attacks.

```
 1 for (i=0; i < 256; i++) // init channel
 2   clflush(probeArray[i*512]);
 3 // Phase ① - access secret data:
 4 // The attacker mis-trains the branch:
 5 if (x < array_size) { // predicted taken
 6   // wrong-path, x >= array_size
 7   secret = array[x];
 8 // Phase ② - covertly transmit secret:
 9   t = probeArray[secret * 512];
10 }
11 // ... somewhere else in attacker's code
12 // Phase ③ - recover secret:
13 for (guess = 0; guess < 256; guess++) {
14   addr = &probeArray[guess*512];
15   t1 = rdtscp(); // read timer
16   temp = *addr;  // access probing array
17   t2 = rdtscp(); // read timer
18   if (t2-t1 <= CACHE_HIT_THRESHOLD)
19     results[guess] += 1;
20 }
```

Listing 1: Exfiltrating secret data using the Spectre v1 *control-steering* and the cache covert channel.

By contrast, in *chosen-code* attacks—our second *Access Phase* category—we consider an adversary who can generate and execute arbitrary code sequences to mount the attack. Such an adversary already has access to its own registers and memory; these attacks instead seek to circumvent *hardware* protections that preclude the attacker from accessing secret data in correct-path code. For instance, Meltdown [1] accesses kernel memory; Foreshadow [4–6] accesses SGX and hypervisor memory; and LazyFP [7] accesses AVX registers used by another process. These attacks exploit implementation flaws in the relative timing of hardware protection checks and data flow between wrong-path instructions—the secret data propagates among instructions and can be leaked into a covert channel before protection checks squash the wrong-path execution. We show chosen-code attacks in Fig. 1b.

Sample Attack Code. Listing 1 illustrates these phases for the Spectre v1 [2] bounds check bypass attack [3], which is a control-steering attack. In this attack, the victim code includes instructions that access `array` at a given index `x` (Line 7). Before accessing `array`, the victim code performs

Phase ①	Attack / Phase ②	*Cache*	*FPU*	*Ports*	*BTB*	*Other?*
Control steering	Spectre v1 [2]	✓			★	
	Spectre v1.1 [11]	✓				
	Spectre v2 [2]	✓				
	Ret2spec [13,14]	✓				
	NetSpectre [12]	▨	✓			
	SMoTher Spectre [15]	▨		✓		
	SSB (Spectre v4) [3]	✓				
	<future attacks>	▨				
Chosen code	Meltdown (v3 / v3a) [1,3]	✓				
	LazyFP[7]	✓				
	Foreshadow/NG (L1TF) [4-6]	✓				
	MDS attacks [8-10]	✓				
	<future attacks>	▨				

✓- demonstrated in prior work; ★- demonstrated in this work

▨ - Cache-based attacks are defeated by prior work [26,27,28,31]

Table 1: Taxonomy of attacks based on secret data access method ① and covert channel ②. *NDA* blocks all existing attacks regardless of the covert channel they use.

a bounds check on x (Line 5). To circumvent the bounds check, the attacker mis-trains the branch direction-predictor by invoking the victim code repeatedly with a valid x.

To mount the attack, the attacker now calls the victim code with an illegal value of x. The attacker chooses x such that array[x] will refer to a location in the victim's memory containing a secret. The direction predictor mis-predicts the branch on Line 5 as taken, executing Lines 7–9 on the wrong path. During wrong-path execution, the code *accesses* (①) the secret on Line 7. It then *transmits* (②) the secret (still in wrong-path) on Line 9. Later, in correct-path execution, the attacker executes Lines 13–20 to *recover* (③) the secret from the cache side-channel. The timing for each access to probeArray on Line 16 will vary based on whether or not the corresponding cache line was loaded on Line 9. In our evaluation, we illustrate the difference in access timing (blue squares in Fig. 5), which reveals the secret data.

Listing 2 depicts an example of a chosen-code attack—a simplified Meltdown exploit. Whereas the illegal load on Line 2 will eventually fault, the instruction on Line 6—which executes on the wrong path—will leave evidence in the cache from which the attacker can recover the secret. The *recover* phase is identical to that in Listing 1. To avoid trapping into the fault handler, the attacker may use control-steering techniques to ensure the faulting load executes under a mispredicted branch [1]. Nevertheless, we classify the attack as chosen-code since the attacker controls the executed binary.

3.2 Limitations of Existing Defenses

Current Mitigations. Hardware defenses mitigating control-steering attacks try to prevent the attacker from mistraining branch predictors (IBRS and STIBP [20]) or use a barrier instruction to prevent speculation after a branch or context switch (lfence/IBPB [20]). Unfortunately, recent attacks [11, 13, 14] reveal techniques to overcome these mitigations. SSBD [20, 37] disables Speculative Store Bypass to prevent attackers from reading data that was overwritten [3, 38]. However, SSBD only blocks Spectre v4. and introduces up to 8% overhead [39].

```
1 // Phase ① - access secret:
2 secret = *kernel_addr; // will fault
3 // Phase ② - covertly transmit secret:
4 // Executed in wrong-path
5 // before fault is fired:
6 t = probeArray[secret * 512];
7 // Phase ③ - recover secret:
8 // see Listing 1
```

Listing 2: Exfiltrating secret data using the Meltdown *chosen-code* attack and a cache side-channel.

Software defenses, such as Retpoline [19] and RSB stuffing [18], protect call and ret instructions from missteering. Other compiler approaches [40, 41] create a data dependency between a branch condition and code that follows the branch, disabling speculation. However, these compiler approaches can only defeat Spectre v1 [2] attacks. A recent study suggested a compiler modification that also block Spectre v2 attacks [42]. Unfortunately, this approach can only defeat cache-based attacks with 68-247% overhead.

Chosen-code attacks are mitigated by preventing speculative loads from accessing restricted memory. For instance, Kernel Address Space Layout Randomization (KASLR [24]) and Kernel Page Table Isolation (KPTI [22, 23]) prevent Meltdown attacks from reading privileged kernel memory. KASLR [24] randomizes the kernel address space similar to how ASLR is used to protect user-space processes for strong isolation. KPTI manages separate page tables for the kernel and user-space processes, preventing user code from issuing even illegal loads to kernel memory. KPTI swaps page tables on every transfer between CPU privilege levels. Mitigating Foreshadow [4–6] requires modifications to the OS, hypervisor, and SMM code, such as modifying page-table management, altering virtual machine scheduling, and adding L1 cache flushes when switching security domains [5, 6].

Unfortunately, all these defense mechanisms block only specific exploit techniques. Therefore, one must deploy a myriad software and hardware defenses to be resilient against *all* control-steering and chosen-code attacks.

Recent work suggests preventing both control-steering and chosen-code attacks by blocking the cache side channel [26–28], thus interdicting the *transmit* phase. However, given the abundant supply of covert channels (see Fig. 3), defeating speculative attacks by closing each channel individually might prove challenging. Exploits have already been demonstrated for other channels. Netspectre [12] demonstrated that the power state of the FPU is a viable speculative covert channel. SMoTher Spectre [15] showed how to transmit data via port contention [33]. We next show an attack via the BTB.

The BTB Covert Channel. We demonstrate a new covert channel that can be exploited even when the cache covert channel is not available—the BTB. The BTB stores a mapping between branch instructions' addresses and the associated target addresses. For example, a call instruction located at address A to a function located at address B installs the mapping A => B in the BTB. The next time the processor fetches the call instruction at address A, the processor's front-end will speculatively redirect fetch to address B.

If the BTB predicts correctly (Fig. 4a), the speculatively-

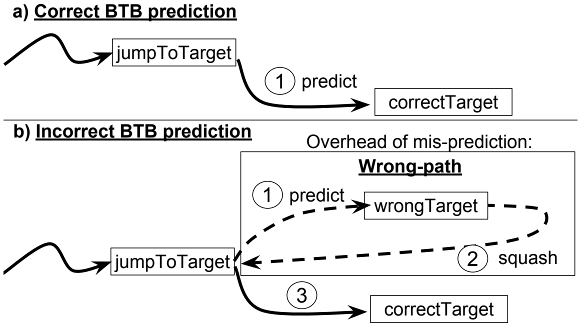


Figure 4: The BTB covert channel. The attacker can observe if the BTB prediction was correct.

fetched instructions are eventually retired. However, if the prediction is wrong, the processor will squash the wrong-path execution, starting at the mispredicted instruction at address B, before executing the correct path. This recovery process is illustrated in Fig. 4b. In our experiments on the *gem5* [43] simulator, we observe that it takes ~16 cycles for the BTB miss to resolve, wrong-path execution to be squashed, and execution to resume at the correct target ($1+2$ in Fig. 4b). Crucially, updates to the BTB during speculation are not reverted by the squash, making it an effective covert channel. Note that (as with caches) in the absence of security concerns, filling the BTB (and updating its replacement policy) during speculation may be advantageous to avoid future BTB misses.

To demonstrate the BTB covert channel, we construct a variant of Spectre v1 [2] that leaks a secret byte through a speculative BTB update, as illustrated in Listing 3. To leak a single byte, our covert channel comprises 256 distinct functions (`targets` in Line 2). During both the *Transmit Phase* and *Recover Phase*, we invoke `targets` only from a single call site, `jumpToTarget` (Line 6), ensuring that BTB entries mapping to `targets` all originate from the same PC and therefore conflict in the BTB.

When the branch on Line 10 is mispredicted, the attacker can *access* any value from the process' address space, depending on the value of `x`. The attacker then *transmits* the secret by speculatively calling `jumpToTarget` with the secret value in Line 13. If the speculation window is large enough, the processor updates the BTB entry for the `call` instruction in Line 6 based on `secret`.

The *access* phase must be repeated for every guess (Line 19) since the *recover* phase is destructive. To confirm that the BTB acts as the covert channel in our attack, it is important to validate that execution time differences do *not* arise from i-cache or d-cache hit or miss latency; no change to the cache hierarchy during the attack may depend upon the secret value. To validate our attack, we ensure the `targets` array in Line 2 and all 256 target functions are cached during access, transmission, and recovery.

We report the effectiveness of the BTB covert channel on *gem5* via the orange circles in Fig. 5. During the *Recover Phase*, in lines 17-24, all the *wrong* guesses will incur the 16-cycle prediction and squashing delay, as depicted in Fig. 4b. The *correct* guess will execute faster, as depicted in Fig. 4a.

On real hardware, it is difficult to isolate the BTB side channel from the cache side channel. The BTB covert channel is one of many potential machine-specific transmission channels. We use our BTB channel PoC to demonstrate that *NDA* is agnostic to any specific transmission channel (§6).

```
 1 // array of 256 unique target functions
 2 void (*targets[256])(void);
 3 // all jumps are from the same location,
 4 // hence the same BTB entry is consulted
 5 void jumpToTarget(int index)
 6  { targets[index](); }
 7 void victim_function(x) {
 8   // Phase ① - access secret data:
 9   // The attacker mis-trains the branch:
10   if (x < array_size) {    // predicted taken
11     secret = array[x];     // wrong path
12   // Phase ② - covertly transmit secret:
13     jumpToTarget(secret); // updates BTB
14 } }
15 // ... somewhere else in attacker's code
16 // Phase ③ - recover secret:
17 for (guess = 0; guess < 256; guess++) {
18   // Induce victim to leak secret value
19   victim_function(x);
20   t1 = rdtscp();             // read timer
21   jumpToTarget(guess);       // BTB prediction
22   t2 = rdtscp();             // read timer
23   if (t2-t1 <= CORRECT_PATH_THRESHOLD)
24     results[guess] += 1;
25 }
```

Listing 3: Exfiltrating secret data using the Spectre v1 *control-steering* attack and the BTB side-channel.

4. THREAT MODELS

NDA design variants address four different threat models, which vary in the locations from which secret data is stolen and whether the attacker may mount control-steering or chosen-code attacks. *NDA*'s goal is to eliminate side-channels created in *wrong-path* execution. Correct-path side channels have been studied in prior work [30, 44, 45].

All threat models are agnostic to the covert channel used in the attacks. For control-steering attacks, we consider two threat models, based on where secrets reside. The first model considers attacks against secrets stored in memory or special registers, as is the case for all currently-known control-steering attacks. Our second control-steering threat model additionally considers hypothetical attacks that leak secrets residing in general-purpose registers (GPRs). In our third threat model, for chosen-code attacks, we consider only threats against secrets in privileged memory and registers, since chosen-code attacks presuppose attacker-controlled GPRs. Lastly, our fourth threat model comprises the union of these threats, considering both control-steering and chosen-code attacks for secrets in memory, special-registers, and GPRs.

4.1 Leaking Memory via Control-Steering

The first step of all known control-steering attacks is to steer wrong-path execution into code accessing a secret in memory or manipulate execution timing to cause a load to observe a stale value. We assume the attacker can steer execution at any branch instruction and manipulate the execution timing of all instructions. Branch instructions include all variants of `jmp`, `call`, and `ret`.

We do not consider *phantom branches*, where the BTB is mis-trained to steer control flow from a program counter value that does not correspond to a branch. The dispatch stage stalls micro-ops whose opcode is unknown. Hence, if the BTB predicts a branch where there is none, dispatch will stall at the phantom branch until its opcode is obtained, which will resolve the misprediction and cause any younger fetched instructions to be discarded before they enter the OoO back-end. Wrong-path instructions that are squashed before dispatch are not a threat.

We also do not consider potentially faulting instructions as steering points in this model. Whereas a fault can result in wrong-path execution, we consider attacks based on faulting instructions (e.g., Meltdown, Foreshadow, LazyFP, MDS, etc.) as part of the threat model for chosen-code attacks.

Speculative Store Bypass. Also known as SSB, or Spectre variant 4 [3], this attack performs the *Access Phase* (① in Fig. 3) by having a malicious speculative load bypass a store whose address is still unresolved. The malicious load then speculatively yields stale (secret) data. Although this attack may not necessarily require misdirected control flow in the *Access Phase*, we consider it a special case of control-steering, since the attacker leverages an existing code snippet. If the attacker could choose the code, they could read the stale data without the need to exploit the speculative store-bypass.

4.2 Leaking GPRs via Control-Steering

All currently-known control-steering attacks extract secrets residing in memory. Nevertheless, we recognize that future attacks might extract secrets residing in the victim's GPRs. So, our second threat model considers the attacker of §4.1 that steers the victim's control flow to leak GPR contents.

In this scenario, the steered victim's code already possesses the secret in a GPR. At this point, the access phase of the control-steering attack (① in Fig. 3) has already (possibly unintentionally) been done by the victim. We therefore focus on hindering the attacker from performing the second phase (② in Fig. 3)—transmitting the GPR-resident secret. All known attacks require data flow between micro-ops during the transmit phase to preprocess the secret (e.g., calculate an offset relative to a base address) before it can be leaked.

We do not prevent an attack that leaks a secret using only a single speculative micro-op. In principle, it may be possible to covertly transmit GPR-based secrets using a single micro-op. For instance, if a GPR contains a secret value that corresponds to a valid virtual memory address, the attacker can speculatively issue a `load` that will fetch this address into the cache hierarchy, thus performing the transmit phase in a single micro-op. However, such a scenario would require (a) a secret value that forms a valid memory address, and (b) victim code that voluntarily loads the secret into a GPR shortly before the vulnerable steering point. No known attacks (cf. Table 1) exploit this behavior, and we were unable to find single micro-op attacks within the x86 instruction set.

4.3 Leaking Memory with Chosen-Code

For chosen-code attacks, we consider attackers that attempt to access secrets residing in memory. Specifically, we consider an attacker who can influence code generation to control both correct-path and wrong-path execution. We treat special-purpose registers, such as AVX (as abused in LazyFP [7]) and Model Specific Registers (MSRs, in Meltdown variant 3a [3]) like memory in crafting our defense—the special instructions (e.g., `rdmsr`) used to access these registers are treated like loads in our solution. In chosen-code attacks, the attacker already controls their own GPRs and we therefore do not consider the contents of any GPR to be secret.

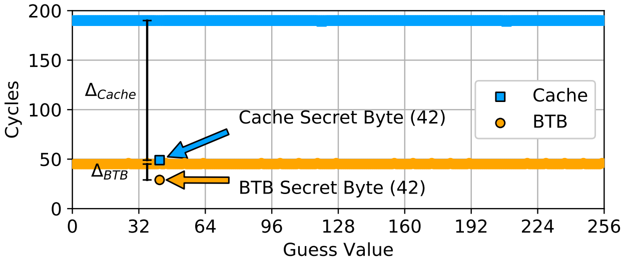


Figure 5: Spectre v1, using either the cache (blue squares) or the BTB (orange circles) as a covert channel. For the cache channel, only the correct guess produces a cache hit, creating the cycle difference Δ_{Cache}. For the BTB channel, only the correct guess successfully predicts the jump target, creating the cycle difference Δ_{BTB}.

Instructions are guaranteed to be correct-path when they retire. At retirement, the head of the ROB satisfies *hardware* permission and memory-ordering checks. Ergo, retired instructions cannot leak secrets accessed from the *wrong-path*.

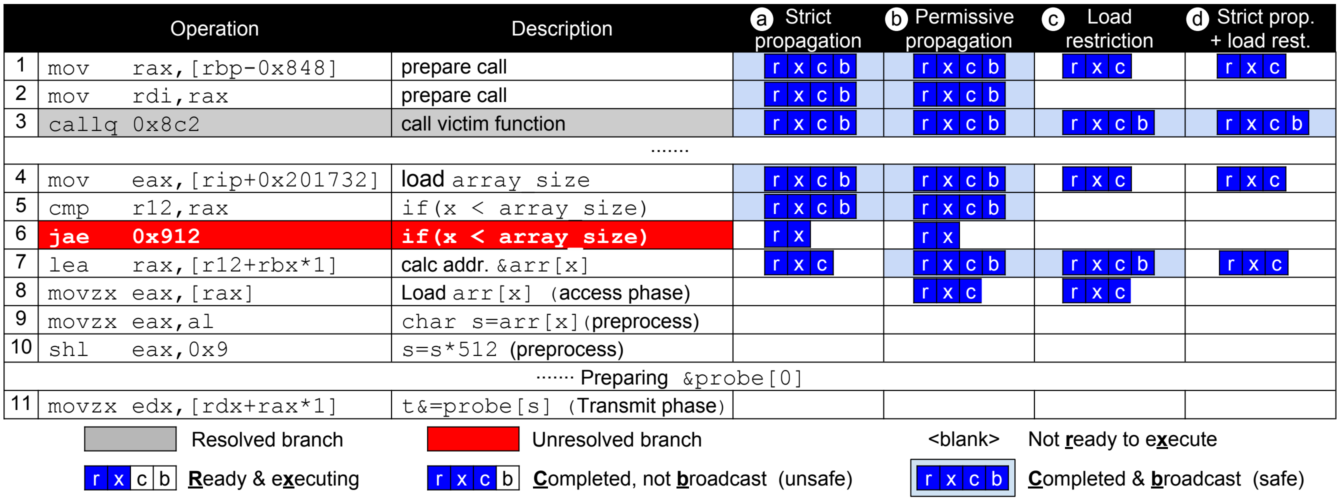
4.4 Combining the Threat Models

Finally, we consider *NDA*'s most conservative threat model—a combination of all threats outlined above. We suppose an attacker that conducts both (a) control-steering attacks to extract secrets from the victim's memory and GPRs, *and* (b) chosen-code attacks to access privileged memory and special registers. This combined threat model is similar to the practical approach taken by Windows and Linux, which deploy mitigations for both classes of attacks [20, 22, 23, 46, 47].

5. DESIGN

NDA's main design goal is to mitigate both control-steering and chosen-code attacks while reaping the benefits of OoO speculative execution as much as possible. We next discuss different variants of *NDA*, which provide different policies for speculative data propagation depending on the threat model. Different *NDA* data propagation policies offer different security guarantees and have corresponding performance implications. We build *NDA* upon a baseline physical register-based OoO micro-architecture [48].

The key insight behind *NDA*'s design is that speculative instructions (either in the *correct* or the *wrong*-path) can safely execute without leaking secrets as long as their inputs are results of *safe* instructions. We define instructions as *safe* with respect to our threat models such that wrong-path execution can not leak any more information into a side channel than a correct-path instruction. Consequently, we eliminate the gap between *speculative* side channel attacks and *non-speculative* side channels, which security-conscious programmers already must reason about. The different *NDA* policies, listed in rows 1-6 of Table 2, define which instructions are considered *safe* such that they may wake-up dependent instructions (allow instructions to advance from step 3 to step 4 in Fig. 2).



	Operation	Description	(a) Strict propagation	(b) Permissive propagation	(c) Load restriction	(d) Strict prop. + load rest.
1	`mov rax,[rbp-0x848]`	prepare call	r x c b	r x c b	r x c	r x c
2	`mov rdi,rax`	prepare call	r x c b	r x c b		
3	`callq 0x8c2`	call victim function	r x c b	r x c b	r x c b	r x c b
		....				
4	`mov eax,[rip+0x201732]`	load `array size`	r x c b	r x c b	r x c	r x c
5	`cmp r12,rax`	`if(x < array size)`	r x c b	r x c b		
6	`jae 0x912`	`if(x < array size)`	r x	r x		
7	`lea rax,[r12+rbx*1]`	calc addr. `&arr[x]`	r x c	r x c b	r x c b	r x c
8	`movzx eax,[rax]`	Load `arr[x]` (access phase)		r x c	r x c	
9	`movzx eax,al`	`char s=arr[x]`(preprocess)				
10	`shl eax,0x9`	`s=s*512` (preprocess)				
		.... Preparing `&probe[0]`				
11	`movzx edx,[rdx+rax*1]`	`t&=probe[s]` (Transmit phase)				

Resolved branch
Unresolved branch
<blank> Not ready to execute
r x c b Ready & executing
r x c b Completed, not broadcast (unsafe)
r x c b Completed & broadcast (safe)

Figure 6: An ROB snapshot during the execution of Spectre v1 (Listing 1), with different *NDA* policies.

To mitigate control-steering attacks, *NDA* restricts data propagation following an unresolved branch or unresolved store address (rows 1-4 in Table 2), depending on where secrets reside and if store-bypass (SSB) is a threat. We consider any instruction following a predicted branch as *unsafe* until the branch target and direction are resolved. We also consider loads that follow a store with an unresolved address as *unsafe* (see Bypass Restriction in §5.2). To mitigate chosen-code attacks, *NDA* introduces a *propagate-on-retire* mechanism (row 5), which defeats all 11 documented chosen-code attack variants [32] and similar future exploits that rely on speculative loads. In this policy, the value returned by *any* load instruction (or other instructions that read sensitive registers, such as `rdmsr` on x86) are considered *unsafe* until the load is ready to retire. Finally, the two mechanisms can be combined to defend against both classes of attacks (row 6).

5.1 Strict Data Propagation

NDA addresses control-steering attacks by defining unresolved branches and unresolved stores—for which predictions may be incorrect—as the borders between safe and unsafe speculation. When a branch micro-op enters the ROB, it is *unresolved*. Since the fetch unit predicts which instructions to fetch following the branch (via the BTB, RSB, etc.), subsequently dispatched micro-ops may be wrong-path. Similarly, when a store micro-op enters the ROB, it is *unresolved* until its address is calculated. If a store's address has not been calculated, loads that follow the store may erroneously access stale data if their addresses overlap. We consider two variants of data propagation restrictions with regards to control-steering attacks: strict and permissive. Both variants leverage a *Bypass Restriction* mechanism to defeat SSB attacks. We now describe strict propagation and then explain permissive propagation and bypass restriction in §5.2.

Strict Propagation (rows 3-4 in Table 2) defends against threat models where secrets may reside in memory, special registers, and GPRs (i.e., the union of the threats described in §4.1 and §4.2). Under this policy, *NDA* marks *all* micro-ops dispatched after an unresolved branch or store as *unsafe*. Unsafe instructions may wake up and compete to issue as in a baseline OoO (i.e., they may issue when their operands become ready). But, when an unsafe micro-op completes execution (step 3 in Fig. 2), it writes back to its destination physical register, but does not broadcast its destination tag to dependent instructions, does not mark its destination register ready, and does not forward its output value on the bypass network. Hence, dependent instructions will not issue and cannot observe the unsafe value.

Managing Value Propagation. When the eldest outstanding micro-op resolves, it marks instructions in the ROB *safe* until the next eldest unresolved branch/store. ROB entries are extended with three bits: `unsafe` tracks if the instruction follows a still-unresolved micro-op, `exec` tracks if the instruction has executed, and `bcast` tracks if the instruction has broadcast its tag to wake dependents. Upon instruction completion, if `unsafe`, tag broadcast is deferred. When a micro-op resolves, the `unsafe` bit for subsequent ROB entries until the next unresolved branch/store are cleared. `!unsafe && exec && !bcast` instructions arbitrate for tag broadcast ports, competing with instructions completing in the current cycle (completing instructions have priority to avoid pipeline stalls); `bcast` is set when broadcasting.

When *safe* instructions broadcast their tags to the issue queue, they mark their destination register(s) ready, waking their dependents (step 4 in Fig. 2). We do not add additional tag broadcast ports to the ROB over baseline OoO; the number of broadcasts is unchanged, broadcasts are time-shifted until preceding micro-ops resolve. For example, assume that the broadcast bandwidth is four and that two instructions completed this cycle. If another three instructions were marked safe, two of these newly-safe instructions can wake dependents; the third waits for the next cycle. In the majority of our evaluation, we assume broadcast and wake-up of newly-*safe* instructions fits within the existing wake-up critical path. In Fig. 9e, we include a sensitivity study that shows the impact of further delay due to critical path constraints; a one-cycle delay reduces CPI by less than 2.5%.

	Mechanism	Control steering (memory)	Control steering (GPRs)	Chosen code	Overhead vs. OoO
1	Perm. propagation	□			13.4%
2	Perm. propagation+BR	🟩			19.9%
3	Strict propagation	□	◇		44.3%
4	Strict propagation+BR	🟩	◇		51.6%
5	Load restriction	🟩		🟩	81.2%
6	Full protection (4+5)	🟩	◇	🟩	113%
7	InvisiSpec-Spectre*	○	○		36.7%
8	InvisiSpec-Future*	○	○	○	94.1%

🟩 Defeats all covert channels ○Defeats cache-based attacks
□ Defeats all covert channels, but does not block SSB
◇ Defeats all covert channels, except single micro-op GPR-attacks
* Our evaluation of InvisiSpec[23] on SPEC 2017 is detailed in §6.1

Table 2: *NDA* propagation policies (rows 1-6) and the attacks they prevent. Bypass Restriction (BR) adds protection against SSB (Spectre v4). Special registers, such as AVX and MSRs (LazyFP [7] and Spectre v3a [3]), are protected by treating their accesses like loads. None of the 25 documented attacks [15, 32] leak data from GPRs nor without at least two dependent micro-ops.

Fig. 6 illustrates an ROB snapshot when executing code akin to Listing 1, depicting various *NDA* data propagation policies. Column ⓐ shows the ROB snapshot under strict propagation. The branch at Line 6 has not resolved, so all following instructions are marked *unsafe*. Whereas the instruction at Line 7 executes to completion, it is *unsafe* and therefore cannot wake the dependent instruction on Line 8.

Branches resolve when the branch micro-op completes execution. Upon a misprediction, all younger micro-ops in the ROB are squashed and renaming tables are recovered, discarding values in physical registers that never became safe, preventing potentially secret data from leaking.

5.2 Permissive Data Propagation

For threat models where *NDA* only protects secrets in memory or special registers, we can safely optimize performance via *permissive propagation* (rows 1-2 in Table 2), which marks only *load* instructions after an unresolved branch/store as *unsafe*. Arithmetic and control instructions are unconditionally marked *safe* at dispatch.

The key intuition for this policy is that only loads can introduce new secret values into the microarchitecture. Loads that precede the eldest unresolved micro-op will commit their value to architectural GPRs, which are not protected under this threat model. Note that wrong-path execution due to exceptions (As in Meltdown or Foreshadow) are also not addressed under this threat model; we address these as chosen-code attacks (§5.3).

For example, consider two dependent instructions i_1 and i_2 fetched after an unresolved branch. If i_1 is an arithmetic instruction (any non-load), it is considered *safe*. Therefore, i_1 can broadcast its output upon completion—allowing i_2 to issue—without waiting for the branch to resolve.

The threat model can include special registers (e.g., AVX or MSRs, see LazyFP [7] and Meltdown v3a [3]). If so, the instructions to read these registers (e.g., `rdmsr`) are also marked *unsafe* when dispatched after an *unresolved* branch.

Lines 7-8 in Fig. 6 illustrates the difference between strict (column ⓐ) and permissive (column ⓑ) propagation. In contrast to strict propagation, the `lea` instruction on Line 7 is marked *safe* since it is not a load operation. Therefore, `lea` wakes its dependent instruction on Line 8 immediately.

Bypass Restriction (BR). To defeat SSB [3] attacks we introduce a new mechanism for safe store bypass, which we use in tandem with both strict and permissive propagation (rows 2,4 in Table 2). In this scheme, unlike Intel's SSBD [20], loads are allowed to execute even if they bypass stores in the Load Store Queue (LSQ). However, loads are marked *unsafe* until all bypassed stores' addresses are resolved. If a bypassed store resolves its address in a way that generates an order violation, the offending load and younger instructions are squashed by the memory dependency unit.

5.3 Load Restriction

NDA protects against chosen-code attacks by blocking data propagation from speculative loads (row 5 in Table 2), such as in Meltdown [1], Foreshadow [4, 5], and LazyFP [7]. These attacks exploit specific flaws in processor implementations where data propagates from a load that will eventually fault. Each of these flaws has been individually patched [6, 20]. However, given the complexity of modern processor implementations, one might expect similar implementation errors in the future. Moreover, in the chosen-code context, there are a myriad of ways to induce wrong-path execution (faulting loads, Intel TSX transaction aborts, interrupt delivery, breakpoint and syscall instructions, performance counter overflow, load replay due to memory-order misspeculation [48, 49], etc.) As prior work [26] suggests, effective defenses must address the common problems underlying chosen-code attacks.

We therefore propose a blanket *NDA* protection policy, *load restriction*, which both blocks all 11 documented [32] chosen-code attacks and offers the potential to prevent future variants. For instance, *NDA*'s load restriction blocks MDS [8–10] attacks, which were discovered after our submission. Under load restriction, loads are considered unsafe until they are the eldest unretired instruction (i.e., at the head of the ROB). With load restriction, the micro-architecture guarantees that a load will wake its dependents if and only if it will immediately retire. Column ⓒ of Fig. 6 illustrates an ROB snapshot when load restriction is used. The `loads` in Lines 1, 4 are independent and can execute concurrently, enabling high Memory & Instruction Level Parallelism MLP & ILP. However, each will wake its dependents (at Lines 2, 5) only when it retires.

5.4 Preventing All Classes of Attacks

To defeat both control-steering and chosen-code attacks, *NDA*'s final policy composes strict propagation and load restriction (row 6 in Table 2). This *NDA* policy is the most defensive, so we call it *full protection*. Column ⓓ in Fig. 6 illustrates an ROB snapshot when the full-protection policy is used. The loads on Lines 1 and 4 are issued and executed to completion, but are not considered *safe*. In contrast to the load-restriction case presented in Column ⓒ, the arithmetic operation on Line 7 is considered *unsafe* in Column ⓓ and

Parameter	Value
Architecture	X86-64 at 2.0 GHz
Core (OoO)	8-issue, no SMT, 32 Load Queue entries, 32 Store Queue entries, 192 ROB entries, 4096 BTB entries, 16 RAS entries
Core (in-order)	TimingSimpleCPU from *gem5*
L1-I/L1-D Cache	32kB, 64B line, 8-way set associative (SA), 4 cycle round-trip (RT) latency, 1 port
L2 Cache	2MB, 64B line, 16-way SA, 40 cycle RT latency
DRAM	50ns response latency

Table 3: Gem5 simulation configuration.

therefore cannot wake the instruction on Line 8. However, parallel execution is still possible (e.g., lines 4 and 7 still execute in parallel) unlike in an in-order processor.

5.5 Security Analysis

Strict Propagation with Bypass Restriction. This policy protects secrets in memory and hinders exfiltration of secrets in GPRs via control-steering attacks. Spectre v1, v1.1, v2, v4 (SSB) [3], and ret2spec [13, 14] are blocked. Most importantly, NetSpectre [12], SMoTher Spectre [15], and our BTB attack (§3)—which are not addressed by prior work [26, 28, 31]—are defeated. For secrets residing in memory, the output of the access phase (① in Fig. 3) cannot be used by the transmit phase ② in the same wrong-path execution window. If the secret data already resides in a GPR prior to the *unresolved* branch, then the transmit phase must comprise only micro-ops that do not depend on one another and may only depend on values from instructions prior to the branch. We note that all existing attacks (cf. Table 1) require more than a single micro-op to transmit secrets. One way to prevent a single micro-op from exfiltrating data is a channel-specific mitigation, such as InvisiSpec [26].

Permissive Propagation with Bypass Restriction. This policy protects secrets in memory but does not protect secrets in GPRs (e.g., `rax`). This level of protection is on par with the threat model presented in recent work [28, 31] with the added benefit of blocking *all* covert channels. All 14 documented control-steering attacks [15, 32], including those listed above, are blocked. Any `load` following an unresolved branch or store is marked unsafe. Therefore, the transmission phase ② will not be able to read the output of the `load`. However, unlike in strict propagation, non-load micro-ops are marked safe. If the secret already resides in a GPR, the attacker can pre-process and transmit the secret using a sequence of wrong-path operations.

Load Restriction. The *load restriction* policy addresses all known chosen-code attacks, including Spectre v3, v3a, v4 [3], LazyFP [7], and Foreshadow/NG [4–6]. In chosen-code threat models, the attacker already controls the executed code, and can thus trivially access the contents of their own GPRs and memory space. Load restriction protects secrets in privileged memory and special registers. Specifically, any micro-op depending on a `load` (or `load`-like instruction) will be ready only after the load retires. Upon retirement, the values returned by `loads` are no longer speculative and are accordingly safe to read.

Load restriction also has the potential to block future chosen-code attacks that access memory and special registers. Additionally, given that none of the 25 existing speculative execution attacks [15, 32] leak secrets from GPRs, the load restriction policy prevents all known control-steering attacks.

Full Protection. Combining load restriction with the strict propagation policy (row 6 in Table 2) offers the most defensive design point of *NDA*. The *full-protection* policy defeats all 25 known control-steering and chosen-code attacks exfiltrating data from memory, special registers, *and* hinders the attacker's ability to transmit contents of GPRs.

6. EVALUATION

We next demonstrate *NDA*'s effectiveness in mitigating speculative execution attacks and evaluate the performance of six different *NDA* policies.

6.1 Experimental Setup & Methodology

We evaluate *NDA* on *gem5* [43] running the SPEC CPU 2017 benchmark suite [50]. Table 3 shows our CPU configuration, which reflects a Haswell-like microarchitecture and matches that used in recent architectural studies of speculative execution attacks [26]. To obtain results that represent SPEC benchmark performance with statistical confidence guarantees, we extend *gem5* to enable a simulation sampling methodology similar to SMARTS [51]. We run SPEC benchmarks on real hardware (Haswell Xeon E5-2699) and dump snapshots of their execution state at fixed intervals using `gdb`. We have developed a new tool to convert these snapshots to *gem5* checkpoints and resume their execution in simulation.

From each checkpoint, we warm simulation state for 5 million instructions and measure performance for 100,000 instructions. We validate that the number of unknown cache references during measurement (references to a cache set for which not all tags are initialized in warmup) is negligible (i.e., the worst-case performance error due to unknown cache references is much smaller than the sampling error). We report 95% confidence intervals of CPI in Fig. 7.

We compare NDA's performance to both variants of InvisiSpec [26] with the same SMARTS methodology and *gem5* configuration, using the source code provided by the authors [52]. NDA's and InvisiSpec's performance for the baseline configuration on SPEC 17 are similar within the confidence interval. Absolute performance numbers for InvisiSpec, depicted in Fig. 7, differ from the original paper due to different benchmarks (SPEC 06 vs. SPEC 17) and sampling methodology (a single billion-instruction segment vs. SMARTS sampling).

6.2 Effectiveness of NDA

We evaluate Spectre v1 [2] (Listing 1 and Listing 3) on unmodified *gem5* without *NDA* protections. As illustrated in Figure 5, both the cache and the BTB covert timing channels clearly leak the secret byte. For the correct guess for the secret byte, the cache covert channel yields a ~140-cycle decrease due to a cache hit. The BTB covert channel similarly yields a ~16-cycle decrease due to the overhead of mis-prediction, as shown in Figure 4. However, when running the Spectre v1 cache and BTB attacks with permissive propagation enabled, *NDA* blocks the speculative data leakage *regardless of the covert channel in use*. As depicted in Figure 8, the correct secret value is indistinguishable from the other 255 candidates.

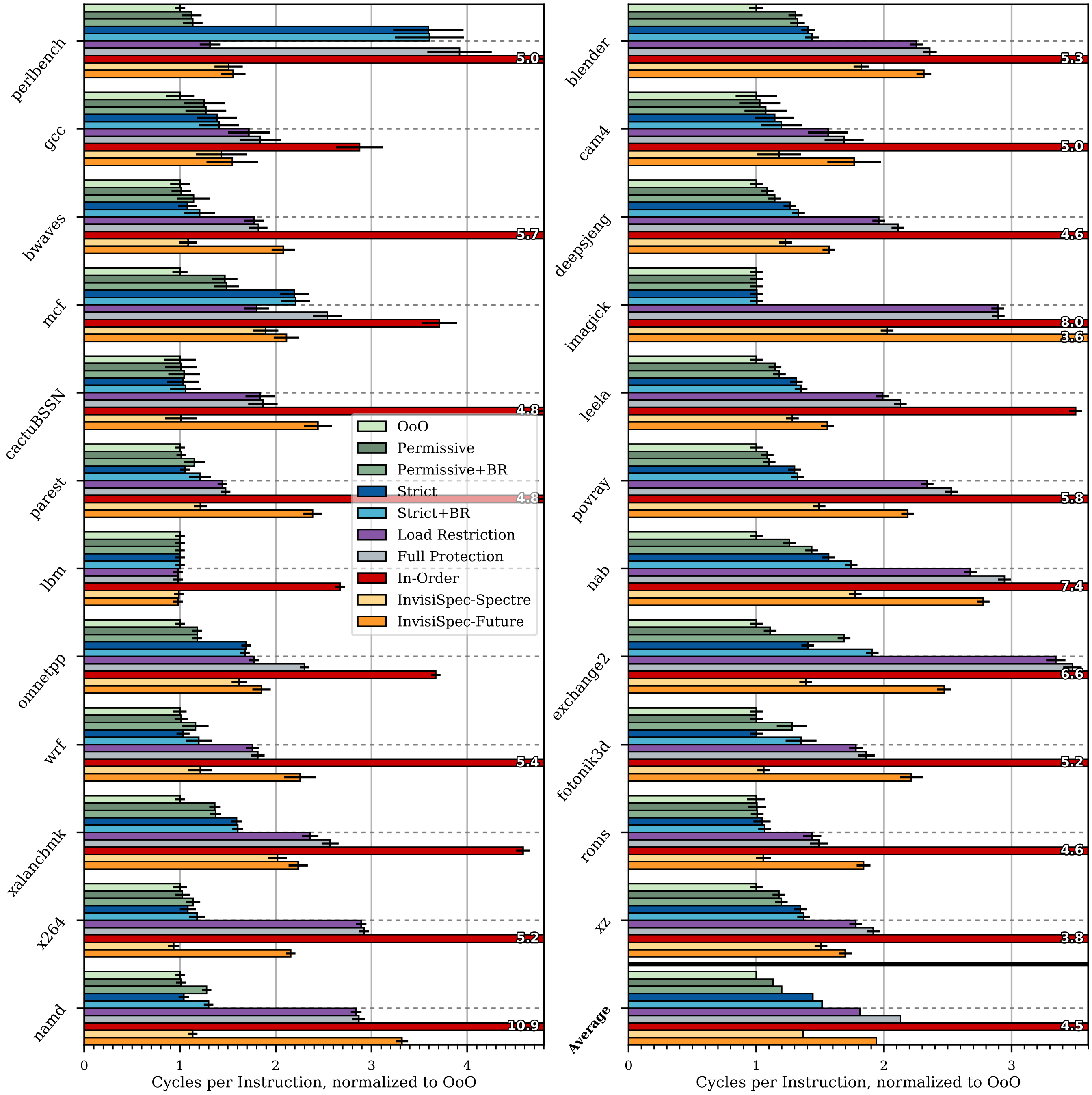


Figure 7: *NDA* and InvisiSpec [26] performance on SPEC 2017. Error bars depict the 95% confidence intervals.

6.3 NDA Performance

We evaluate *NDA*'s performance with ten different configurations; the six *NDA* policies described in §5, two baselines, and two InvisiSpec configurations. The baseline configurations are the in-order and unconstrained OoO processors listed in Table 3. The in-order processor represents the extreme case of no speculation and is thus trivially immune to speculative execution attacks. We note that, besides *NDA*'s *load-restriction* and *full-protection*, the in-order processor is the only other execution model known to defeat all 25 documented speculative execution attacks, regardless of the covert channel they use. The unconstrained OoO processor offers the best performance, but is insecure.

Cycles Per Instruction (CPI). Fig. 7 depicts the CPI of all configurations across all benchmarks (averages at the bottom right). The overheads of different policies are summarized in Table 2. Defeating SSB with Bypass Restriction (BR) adds 6.5-7.3 % overhead. In the case of *permissive propagation*

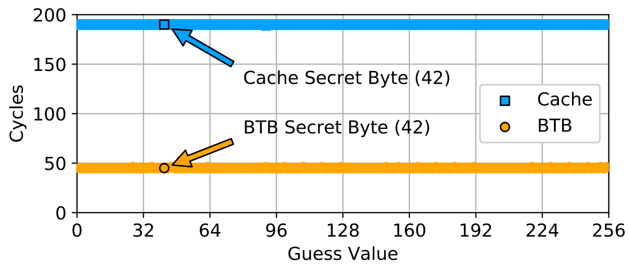


Figure 8: Spectre v1 when using *NDA* permissive propagation policy. The cycle differences in Fig. 5 (Spectre v1 *without NDA*) are eliminated. Thus, *NDA* conceals the secret byte's value, regardless of the covert channel.

with BR (row 2 in Table 2)—our highest performance policy which prevents all 14 control-steering vulnerabilities—the average performance loss relative to the OoO baseline is 19.9%. This policy thwarts all known control-steering attacks and recovers 96% of the performance gap between the OoO and In-Order baselines.

In the case of *full protection* (row 6 in Table 2)—our most secure policy—the average performance loss is 113%. This policy prevents all 25 documented variants of both control-steering and chosen-code attacks while also offering potential protection against future attacks. Despite the restrictions it imposes on the dynamic schedule, full protection still closes 68% of the performance gap between in-order and OoO.

Fig. 9a depicts an average time breakdown for all OoO design variants. The bars are normalized to the baseline OoO design point. *Commit* cycles are cycles in which at least one instruction retires. *Memory stalls* are cycles in which the head of the ROB is an incomplete memory operation. *Back-end stalls* are cycles in which the head of the ROB is a non-memory operation that is not yet ready to retire. *Front-end stalls* are cycles in which the ROB is empty or cycles which are spent squashing wrong-path execution. *NDA* policies restrict data propagation and thereby limit dynamic scheduling. Therefore, on average, fewer instructions are committed in a given cycle, increasing the overall number of *commit* cycles. Since instruction-level parallelism for both memory and non-memory instructions is reduced, more cycles are spent on *memory stalls* and *back-end stalls*. *Front-end stall* cycles generally vary little across designs, on average contributing only 2% of the difference in cycles.

Wake-up Latency. *NDA* introduces a delay between instruction completion and tag broadcast. Whereas broadcast delay does not *directly* affect CPI, the delay propagates to dependent instructions in the ROB by delaying their issue. We measure this effect by measuring the average delay instructions experience from dispatch to wake-up under each design. The average latencies across all benchmarks are shown in Fig. 9d. *NDA* policies add on average 4-39 cycles. This increased latency also manifests in up to 78% increase in cycles spent on *back-end stalls*, shown in Fig. 9a. However, the wake-up latency has a modest impact on overall performance (CPI), as we explained next.

Memory and Instruction Parallelism (MLP/ILP). The favorable performance of *NDA* compared to the in-order processor can be explained by observing the Memory- and Instruction-Level Parallelism (MLP & ILP) of each profile. The geometric means of MLP & ILP across all benchmarks are depicted in Fig. 9b-c. We follow Chou et al [53] and report MLP as the average number of outstanding off-chip misses when at least one is outstanding. Whereas the MLP & ILP in the various *NDA* profiles are at times lower than the OoO baseline by as much as 6% and 44% (respectively) they are better than the in-order baseline processor by 72% and 39%, where MLP & ILP cannot exceed 1.0. These results suggest that *NDA* enables execution parallelism among off-chip misses despite the scheduling restrictions of speculative instructions. Importantly, *NDA* does not typically restrict the issue time of loads, only when they may wake dependents. Ergo, typically only dependent loads are delayed, which do not add to MLP or ILP.

Comparison to InvisiSpec [26]. Since *NDA* and InvisiSpec have different threat models, detailed in Table 2, a direct comparison is not straight forward. In our evaluation, InvisiSpec-Spectre defeats all cache-based control-steering attacks with 36.7% slowdown. In comparison, *NDA* blocks control-steering attacks, regardless of the covert channel they use, with 19.9%-51.6% slowdown, depending on where secrets reside. For futuristic chosen-code attacks, InvisiSpec-Future introduces 94.1% overhead compared to 113% in *NDA*. However, *NDA* blocks all covert channels, including port contention [15], the FPU [12], and the BTB (§3).

7. RELATED WORK

The first micro-architectural side-channel attacks used the cache side channel to infer AES keys from a neighboring process or VM [54–56]. Since then, a myriad of side channel techniques have been developed, such as Flush+Reload [57] and other advanced techniques [58–65]. We refer to these attacks as *classical* cache attacks. These attacks do not leverage speculative wrong-path execution. Other work demonstrates how the cache side channel can be used as a *covert channel* [66–68]. DRAM [69] and issue ports [15, 33] have also been shown to be viable covert channels.

The first speculative execution attacks—Meltdown [1] and Spectre [2]—leveraged prior work on cache covert channels to transmit data obtained from wrong-path execution. Other speculative attacks using various techniques to access secrets or steer execution also leveraged the cache covert channel [3–7, 11, 13, 14, 16, 70]. Since the cache covert channel is widely exploited, InvisiSpec [26], SafeSpec [27], and Conditional-Speculation [28] proposed preventing loads following unresolved branches that result in cache misses from executing and thereby modifying the cache hierarchy. Instead, Netspectre [12] and SMoTher Spectre [15] demonstrated speculative attacks leaking data through covert channels based on FPU activation or port contention [33] Additionally, Conditional-Speculation [28] aims at protecting secrets placed in memory, not in GPRs. In comparison, NDA is agnostic to the channel used in the *Transmit Phase*. NDA therefore defeats NetSpectre and SMotherSpectre attacks, while providing better protection for secrets in registers.

Taram et al. [31] suggest Context Sensitive Fencing, a hardware modification to automatically insert `lfence`

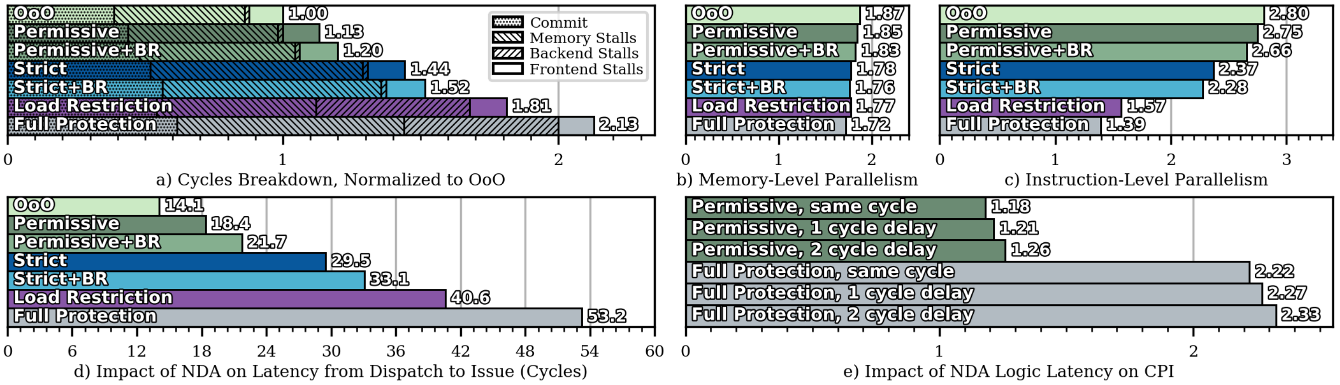


Figure 9: Aggregated statistics over SPEC 17 benchmarks. *(a) NDA* extend the cycles spent on commit and backend stalls. *(b),(c)* MLP & ILP is still high across *NDA* policies. *(d)* As expected, *NDA* causes delays in latency-to-issue. However, overall impact on CPI is substantially smaller. *(e)* The impact of *NDA* logic latency on CPI is relatively small.

```
1 stop_speculative_exec();
2 register long secret = *secret_addr;
3 // ... operate on secret
4 secret = 0; // scrub secret
5 resume_speculative_exec();
```

Listing 4: Closing the registers-to-memory security gap.

micro-ops where needed. However, as stated by the authors, their main goal is to block Spectre v1 attacks leaking data via the data cache. NDA defeats all known variants regardless of the covert channel they use.

Recent work (such as DAWG [29], CEASER [30], and others [44, 45]) hinder the attacker's ability to deterministically cause a cache line collision with another process or VM, thwarting most cache-based side and covert channels. However, these techniques do not mitigate attacks that use non-cache covert channels.

We addressed related work on deployed defense mechanisms for speculative execution attacks in §3.2.

8. DISCUSSION

NDA is capable of defeating both control-steering and chosen-code attacks while performing considerably better than in-order processors. However, even though *NDA* blocks all known attacks, it may still be possible to use a control-steering attack to read general-purpose registers if there exists a single micro-op that can leak the register's contents.

To protect registers, one can introduce an instruction or a processor mode that temporarily disables speculation and out-of-order execution during the window of vulnerability when a secret value is loaded from memory and resides in a register until it is overwritten. We illustrate such a defense in Listing 4. We note this defense would only be effective if used in addition to *NDA*. Without *NDA*, a control-steering attack could simply steer the execution to bypass Line 1 and speculatively execute Lines 2-3 to leak the register's contents.

9. CONCLUSION

Speculative execution attacks are challenging to mitigate. Blocking individual covert channels or specific exploitation techniques is insufficient. To design effective mitigations, we introduced a new classification of speculative execution attacks based on how each attack induces wrong-path execution. Our new technique for controlling speculative data propagation, *NDA*, defeats all known speculative execution attacks and drastically reduces the attack surface for future variants. On SPEC 2017, we show that the four *NDA* design points offer effective and performant mitigations.

References

[1] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[2] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy*, 2019.

[3] Intel. Intel Analysis of Speculative Execution Side Channels. `https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf`.

[4] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium. USENIX Association*, 2018.

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

Technical report, 2018. See also USENIX Security paper Foreshadow [4].

[6] Intel. *Deep Dive: Intel Analysis of L1 Terminal Fault*. `https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault`.

[7] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv preprint arXiv:1806.07480*, 2018.

[8] Stephan van Schaik, Alyssa Milburn, Sebastian ÃŰsterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[9] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Berk Sunar, Frank Piessens, and Yuval Yarom. Fallout: Reading kernel writes from user space. 2019.

[10] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. *arXiv:1905.05726*, 2019.

[11] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[12] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network. *arXiv preprint arXiv:1807.10535*, 2018.

[13] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122. ACM, 2018.

[14] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies, WOOT*, pages 13–14, 2018.

[15] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. *arXiv preprint arXiv:1903.01843*, 2019.

[16] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. *arXiv preprint arXiv:1802.09085*, 2018.

[17] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013.

[18] Intel. *Retpoline: A Branch Target Injection Mitigation*. `https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf`.

[19] Google. *Retpoline: a software construct for preventing branch-target-injection*. `https://support.google.com/faqs/answer/7625886`.

[20] Intel. Speculative Execution Side Channel Mitigations. `https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf`.

[21] Debian. *Debian Bug report logs - #886367 intel-microcode: spectre microcode updates*. `https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=886367`.

[22] LWN. *A page-table isolation update*. `https://lwn.net/Articles/752621/`.

[23] Microsoft. *Mitigating speculative execution side channel hardware vulnerabilities*. `https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/`.

[24] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.

[25] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*, 2019.

[26] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the 51th International Symposium on Microarchitecture (MICRO'18)*, 2018.

[27] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. *arXiv preprint arXiv:1806.05179*, 2018.

[28] Lutan Zhao Peinan Li and CAS) Rui Hou (Institute of Information Engineering, CAS); Lixin Zhang (HXT Semiconductor Co.LTD); Dan Meng (Institute of Information Engineering. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *Proceedings of the 25th IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 2019.

[29] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE, 2018.

[30] Moinuddin K Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *Proceedings of 51th International Symposium on Microarchitecture*, 2019.

[31] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[32] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. *arXiv preprint arXiv:1811.05441*, 2018.

[33] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida Garcia, and Nicola Tuveri. Port contention for fun and profit. Cryptology ePrint Archive, Report 2018/1060, 2018. https://eprint.iacr.org/2018/1060.

[34] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[35] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[36] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[37] AMD. Speculative Store Bypass Disable. https://developer.amd.com/wp-content/resources/124441_AMD64_SpeculativeStoreBypassDisable_Whitepaper_final.pdf.

[38] Project Zero. speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[39] Intel. *Details and Mitigation Information for Variant 4*. https://newsroom.intel.com/editorials/addressing-new-research-for-side-channel-analysis/#gs.4778nz.

[40] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506*, 2018.

[41] Google. *Speculative Load Hardening*. https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT_61e_Ko3TmoCS3uXLcJR0/edit#heading=h.phdehs44eom6.

[42] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. Restricting control flow during speculative execution with venkman. *arXiv preprint arXiv:1903.10651*, 2019.

[43] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[44] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 347–360. IEEE, 2017.

[45] Mengjia Yan, Yasser Shalabi, and Josep Torrellas. ReplayConfusion: detecting cache-based covert channel attacks using record and replay. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 39. IEEE Press, 2016.

[46] Microsoft. *Protect your Windows devices against Spectre and Meltdown*. https://support.microsoft.com/en-us/help/4073757/protect-your-windows-devices-against-spectre-meltdown.

[47] Ubuntu. *Spectre And Meltdown*. https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/SpectreAndMeltdown.

[48] Kenneth C Yeager. The MIPS R10000 superscalar microprocessor. *IEEE micro*, 16(2):28–41, 1996.

[49] Kourosh Gharachorloo, Anoop Gupta, and John L Hennessy. Two techniques to enhance the performance of memory consistency models. 1991. https://courses.engr.illinois.edu/cs533/sp2019/reading_list/gharachorloo91two.pdf.

[50] SPEC. Standard Performance Evaluation Corporation SPEC CPU 2017. https://www.spec.org/cpu2017/.

[51] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ACM SIGARCH Computer Architecture News*, volume 31, pages 84–97. ACM, 2003.

[52] *InvisiSpec-1.0 source code*. https://github.com/mjyan0720/InvisiSpec-1.0.

[53] Yuan Chou, Brian Fahs, and Santosh Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 76–87. IEEE, 2004.

[54] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[55] Daniel J Bernstein. Cache-timing attacks on AES. 2005. http://palms.ee.princeton.edu/system/files/Cache-timing+attacks+on+AES.pdf.

[56] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 201–215. Springer, 2006.

[57] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, pages 719–732, 2014.

[58] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *ACSAC*, pages 422–435. ACM, 2016.

[59] Cesar Pereida Garcia and Billy Bob Brumley. Constant-Time Callees with Variable-Time Callers. In *USENIX Security Symposium*, pages 83–98. USENIX Association, 2017.

[60] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID*, volume 8688 of *Lecture Notes in Computer Science*, pages 299–319. Springer, 2014.

[61] Cesar Pereida Garcia, Billy Bob Brumley, and Yuval Yarom. "Make Sure DSA Signing Exponentiations Really are Constant-Time". In *ACM Conference on Computer and Communications Security*, pages 1639–1650. ACM, 2016.

[62] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To BLISS-B or not to be: Attacking strongSwan's Implementation of Post-Quantum Signatures. In *CCS*, pages 1843–1855. ACM, 2017.

[63] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*, pages 897–912. USENIX Association, 2015.

[64] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *ACM Conference on Computer and Communications Security*, pages 990–1003. ACM, 2014.

[65] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *40th IEEE Symposium on Security and Privacy*, 2019.

[66] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security symposium*, pages 159–173, 2012.

[67] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64. Springer, 2015.

[68] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of l2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 29–40. ACM, 2011.

[69] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, pages 565–581, 2016.

[70] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 693–707. ACM, 2018.