SoK: Hardware Defenses Against Speculative Execution Attacks

Guangyuan Hu
Princeton University
gh9@princeton.edu

Zecheng He
Princeton University
zechengh@princeton.edu

Ruby B. Lee
Princeton University
rblee@princeton.edu

Abstract—Speculative execution attacks leverage the speculative and out-of-order execution features in modern computer processors to access secret data or execute code that should not be executed. Secret information can then be leaked through a covert channel. While software patches can be installed for mitigation on existing hardware, these solutions can incur big performance overhead. Hardware mitigation is being studied extensively by the computer architecture community. It has the benefit of preserving software compatibility and the potential for much smaller performance overhead than software solutions.

This paper presents a systematization of the hardware defenses against speculative execution attacks that have been proposed. We show that speculative execution attacks consist of 6 critical attack steps. We propose defense strategies, each of which prevents a critical attack step from happening, thus preventing the attack from succeeding. We then summarize 20 hardware defenses and overhead-reducing features that have been proposed. We show that each defense proposed can be classified under one of our defense strategies, which also explains why it can thwart the attack from succeeding. We discuss the scope of the defenses, their performance overhead, and the security-performance trade-offs that can be made.

Defense and Overhead-reducing Feature	Conference	Year
InvisiSpec [14]	MICRO	2018
DAWG [15]	MICRO	2018
CondSpec [16]	HPCA	2019
Context-sensitive fencing (CSF) [17]	ASPLOS	2019
SpectreGuard [18]	DAC	2019
SafeSpec [19]	DAC	2019
EfficientSpec [20]	ISCA	2019
SpecShield [21]	PACT	2019
STT [22]	MICRO	2019
NDA [23]	MICRO	2019
CleanupSpec [24]	MICRO	2019
MI6 [25]	MICRO	2019
IRONHIDE [26]	HPCA	2020
ConTExT [27]	NDSS	2020
Predictor state encryption [28]	ISCA	2020
MuonTrap [29]	ISCA	2020
Speculative Data-Oblivious Execution (SDO) [30]	ISCA	2020
Clearing the Shadows [31]	PACT	2020
InvarSpec [32]	MICRO	2020
DOLMA [33]	USENIX Security	2021

TABLE I: Hardware defenses and overhead-reducing features against speculative execution attacks published in recent computer architecture and security conferences.

I. INTRODUCTION

Speculative execution attacks, also known as transient execution attacks, are a serious security problem. They exploit performance enhancement features in hardware to access secret data and leak this secret out through microarchitectural covert channels. This negates the confidentiality and integrity protections provided by software isolation, and also by hardware isolation features such as secure enclaves [1], [2].

In particular, Spectre [3], Meltdown [4] and Foreshadow [5] bypass the isolation across processes and privilege levels. The Spectre attack bypasses the memory protection provided by software bounds checking, while the Meltdown attack breaches the memory isolation between the kernel and a user application. Foreshadow [5], and its variants Foreshadow-OS and Foreshadow-VMM [6], breach the Intel SGX enclave isolation, user-to-kernel memory isolation, and virtual-machine-to-hypervisor isolation, respectively.

The severity of these attacks has resulted in many specific fixes for specific attack variants implemented by the computer industry. These include using instructions to serialize execution [7], [8], to flush hardware prediction states [9], to avoid using untrusted predictions [10], and to restrict accesses to secret information [11]–[13]. However, most of these solutions require changes to the existing software. Furthermore,

they also cause significant performance overhead (at least 2X slower [11], sometimes up to 8X). Last but not least, the software countermeasures are usually attack-specific. New patches are required to effectively protect against the emerging attacks, which is neither efficient nor sustainable.

In response to these attacks on hardware microarchitecture performance optimization features, there have been proposals of hardware defenses as well as features that reduce the performance overhead of defenses [14]–[33], which we show in Table I in chronological order. One key advantage is that the hardware solution can monitor the instruction execution status and accurately protect against speculative vulnerabilities. Another advantage of some hardware solutions is their non-intrusive interaction with the existing software, while inducing low performance overhead. These hardware defenses can read the unmodified program but delay or change the execution of secret-leaking instructions so that the information leakage through hardware states is eliminated. Some microarchitectural defenses also allow security-performance trade-offs and overhead-reducing features [30]–[32].

However, the working mechanisms and scope of different hardware defenses have not been systematically described and compared. Hence, our goal in this paper is to systematize

the hardware defenses, to illustrate their key similarities and differences, and to assist future researchers to more easily understand and reason about how an existing defense works. While the goal of this paper is not to describe all the speculative execution attacks in detail, as there are many past work surveying and summarizing these [34]–[37], we analyze the critical attack steps of 23 speculative execution attacks. We then show how the hardware defenses mitigate the attacks by preventing these steps, connecting the attacks and defenses.

Our key contributions are:

- Producing attack taxonomies based on secret access or secret leakage, covering 23 variants of speculative execution attacks.
- Defining new defense strategies based on preventing at least one of the critical attack steps.
- Producing a new taxonomy of 4 hardware defense strategies and lower-level categories of defenses.
- Creating a systematized view and description of 20 representative hardware defenses and overhead-reducing features proposed to date.
- Presenting the performance overhead of the defenses, and illustrating security-performance tradeoffs.

II. MICROARCHITECTURE AND COVERT CHANNEL BACKGROUND

We first describe hardware performance optimization features that can be exploited for speculative execution attacks.

Out-of-Order (OoO) execution. An Out-of-Order (OoO) processor is a microarchitecture performance enhancement feature used to boost the throughput of processors by allowing instructions later in the program order to execute before the previous instructions have completed. For example, an earlier instruction may be waiting for one of its operands, or for a functional unit or memory to free up, or for determining if a branch should be taken or where to branch to. Later instructions in an in-order processor that have no dependencies will have to wait unnecessarily. In contrast, an Out-of-Order processor allows the instructions with no dependencies to execute immediately, as long as they retire in-order.

Fig. 1 shows a generic Out-of-Order (OoO) processor where instructions are fetched in program order but executed Out-of-Order. Instructions are forced to retire in-order to maintain precise exceptions, i.e., if an instruction results in an exception, the following instructions must be "squashed" as if they were never executed. We will use this generic OoO model to explain the defenses in a unified way in the rest of the paper.

Instructions are fetched and decoded to microarchitecture-level operations (denoted μop's) in program order, but after the μop's are dispatched to the execution stage, the hardware scheduler can schedule any ready μop's to different functional units for execution. Thus, the execution of later μop's can complete earlier than those of previous instructions, which also allows the results of these μop's to be used earlier. The result from the execution of a μop is *forwarded* and used by other dependent μop's.

An important microarchitecture structure, that we will refer to in discussing hardware defenses, is the *Re-Order Buffer (ROB)* shown in Fig. 1. The ROB records the instruction's or μop's information as well as its execution status, such as whether the instruction has finished its execution (*Done* = "1" in the figure) and whether the instruction should be squashed (*Squash* = "1"). The ROB guarantees that if an instruction needs to be squashed, all the subsequent instructions are also squashed. The ROB also acts as a FIFO queue to preserve the program order so an instruction can only retire when it reaches the head of the ROB.

Speculative execution. Speculative execution is a further performance enhancement that allows instructions to be tentatively (i.e., speculatively) executed, even when the control flow has not been determined, or the data from memory has not arrived. For instance, when the processor fetches a branch whose operand is not available, e.g., having to be read from memory, the address of the next instruction is predicted and fetched so that the processor does not have to stall its pipeline. If the prediction is found to be correct later, the speculative execution improves the performance by executing code on the correct path in advance. However, if the prediction is found to be incorrect, the processor needs to flush the pipeline so that the results of the speculatively executed instructions are discarded. This is called a *squash*, where the processor restores the architectural state, e.g., the register values visible to the software, as if the mispredicted instructions have not executed.

Hardware predictors. From the microarchitecture perspective, speculative execution happens because hardware predictors are present that allow tentative forward progress even when an instruction has unresolved dependencies. For conditional branch instructions, branch predictors predict whether the branch will be taken or not. For indirect branches, the Branch Target Buffer (BTB) predicts the target address. For return instructions, the Return Stack Buffer (RSB) or Return Address Stack (RAS) predicts the address to return to after a procedure call.

Microarchitectural state and covert channels. Microarchitectural states are the states of hardware units that are not directly accessible to the programmer or software. Even if invisible from the software's view, these states can impact the execution time of certain programs and the states can be inferred if the processor executes these programs. If one program modifies a certain microarchitectural state with another monitoring it, these two programs form a microarchitectural covert channel in which the former is the sender and the latter is the receiver. Examples include the addresses of cache lines in various cache levels, which we describe in detail below, and the busy status of different hardware resources.

Cache state and covert channel. One critical microarchitectural state is the cache state. A cache has many cache lines corresponding to different addresses. Since cache hits are fast, and cache misses are slow, cache timing attacks are possible, leaking information through observing the cache access time.

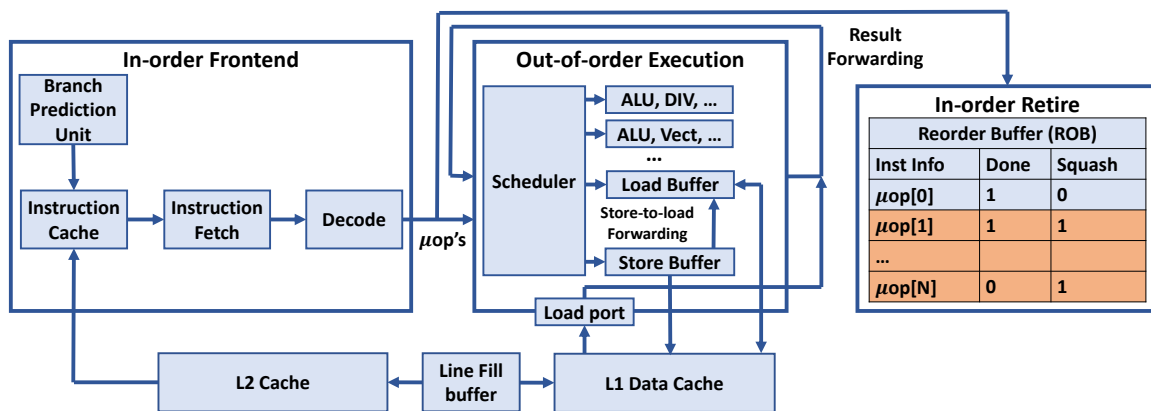One example exploiting a cache covert channel is the *flush-*

Fig. 1: A block diagram of a typical out-of-order processor. The contents of the ROB show a sample situation where instruction μop[0] executed correctly, but μop[1] to μop[N] were executed speculatively and incorrectly and had to be squased.

reload technique [38], where the sender is an insider and the receiver an outsider attacker. During the setup phase of the covert channel, certain cache lines are flushed out of the cache. To send a secret out, the sender accesses a secret-dependent address, which brings back one of the flushed lines. The receiver will later measure the time to reload each cache line and infer whether this cache line is fetched by the sender by observing whether it is a cache hit. The flush-reload cache covert channel is used in most of the speculative execution attacks published.

There are other techniques for covert communication through cache state. In a *prime-probe* [39] covert channel, the receiver first primes the cache to fill the cache with its own cache lines. The sender then accesses certain addresses, evicting some of the receiver's cache lines. The receiver can get to know which cache lines the sender accessed by loading each cache line and observing cache misses. In a *flush-flush* [40] covert channel, the receiver keeps evicting certain addresses by executing the flush instruction. If the sender accesses some of these addresses and brings them into the cache, the time to flush will be longer, so the receiver can infer information from timing the second flush.

Many other types of covert or side channels, not using caches, nor timing, are also possible.

III. Speculative Execution Attacks

We first present some critical attack steps that we have identified in existing speculative execution attacks.

A. Critical Attack Steps

Although the exact workflow of an attack may vary, we observe that they all consist of 6 critical steps. These are shown in the right column of Fig. 2 and described below.

Setup. The *Setup* step sets up the initial hardware state, e.g., the branch predictor state for Spectre v1, so that the processor will enter speculative execution. It also sets up the initial state for the covert channel, e.g., flushing the shared cache lines for a flush-reload channel.

Authorize. The attack starts with the *Authorize* step. The *Authorize* operation performs the authorization required for

accessing a memory location or a protected register. For speculative attacks, the speculative execution window starts when the authorization is delayed.

Access. When the authorization is delayed, the *Access* step in a speculative attack can read a secret from the cache, the memory, a protected register or a microarchitectural buffer that is otherwise not allowed.

Use. The *Use* step uses the secret to generate a secret-dependent operation. Examples are instructions that compute a memory address for a later load operation.

Send. The *Send* step alters the microarchitectural state of the covert channel in a secret-dependent way. Even if the *access*, *use* and *send* operations will all be squashed after the authorization fails, the microarchitectural state change may remain and can be discovered later by the receiver.

Receive. The recovery of the secret from the covert channel by the attacker.

B. A Spectre v1 Attack Example

For concreteness, let us first consider a particular speculative attack, the Spectre v1 attack. In Fig. 2, we show the pseudo-code of the Spectre v1 attack and the RISC assembly instructions executed during speculative execution. Lines 1-3 *set up* the microarchitectural state. The cache lines containing the shared array pointed by SHAREDPtr are flushed from caches as the preparation for the flush-reload cache covert channel which we described in Section II. The size of the private array pointed to by arrayPtr is also flushed so that the load on line 4 will take a long time to finish. Also, the branch predictor is mistrained so that the prediction of the conditional branch in line 5 will be "not taken". The conditional branch, bge, in line 5 performs the *authorization* for the later load byte instruction, lbu, which *accesses* the secret byte in line 7. Since the conditional branch checking is delayed by the previous load instruction in line 4, a branch predictor is invoked. Due to the mistraining, the branch is not taken and the secret is illegally accessed by the lbu instruction. In line 8 and 9, the secret is then *used* to calculate a memory address of the next ld instruction in line 10. This ld instruction is a covert *send*

	Pseudo-code	Critical Steps
	Setup microarchitectural states:	
1	flushArray(SHAREDPtr, 256);	**Setup**
2	flush(&array_size);	
3	mistrainPredictor();	
	Spectre v1 Sender:	
	(r2: SHAREDPtr, r3: offset to a secret, r4: arrayPtr)	
4	ld r5, 0(&array_size)	**(long-latency)**
5	bge r3, r5, outside --> if (x < array_size) {	**Authorize**
6	*add r6, r3, r4*	
7	*lbu r7, 0(r6) --> y = arrayPtr[x];*	*Access*
8	*slli r7, r7, 12*	*Use*
9	*add r8, r2, r7*	
10	*ld r9, 0(r8) --> z = SHAREDPtr[y*4096];*	*Send*
	* }*	
	outside:	
	Receiver:	
11	for i from 0 to 255	
12	t[i] = TimeToReload(SHAREDPtr[i*4096]);	**Receive**
13	Find the minimal t[i]	

Fig. 2: Spectre v1 attack (bypassing array bounds checking). The assembly code and pseudo code of the attack bypass control flow authorization by a conditional branch to access a secret. The attack leaks an 8-bit secret through the most commonly used flush-reload cache covert channel by loading in a cache line in the shared array into the cache. The code in red is the transient execution that will be squashed. The comments after the arrows show the high-level language equivalents of the assembly code.

Attack	CVE	Description	Date
Spectre v1 [3]	2017-5753	Speculative boundary check bypass for read	2018.1
Spectre v1.1 [41]	2018-3693	Speculative boundary check bypass for write	2018.7
NetSpectre [42]	2017-5753	Remote attack performing a bounds check bypass	2018.1
Spectre v2 [3]	2017-5715	Branch target misprediction	2018.1
Spectre RSB [43], [44]	2018-15572	Return target misprediction	2018.8
Spectre SSB [45]	2018-3639	Speculative store bypass, read stale data in memory	2018.5
Meltdown-Reg (Spectre v3a) [46]	2018-3640	System register value leakage to unprivileged attacker	2018.5
Lazy FP [47]	2018-3665	Leak of FPU state	2018.6
Meltdown (Spectre v3) [4]	2017-5754	Kernel content leakage to unprivileged attacker	2018.1
Foreshadow (L1 Terminal Fault) [5]	2018-3615	SGX enclave memory leakage	2018.8
Foreshadow-OS [6]	2018-3620	OS memory leakage	2018.8
Foreshadow-VMM [6]	2018-3646	VMM memory leakage	2018.8
Spectre v1.2 [41]	N/A	Speculative write to read-only memory	2018.7
RIDL/MLPDS [48], [49]	2018-12127	MDS leakage from load port	2019.5
RIDL/ZombieLoad/ MFBDS [48]–[50]	2018-12130	MDS leakage from line fill buffer	2019.5
Fallout/MSBDS [49], [51]	2018-12126	MDS leakage from store buffer	2019.5
TAA [52]	2019-11135	TSX Asynchronous Abort	2019.11
RIDL/MDSUM [48], [49]	2019-11091	MDS leakage from uncacheable memory	2019.5
VRS [53]	2020-0548	Vector Register Sampling	2020.1
CacheOut/L1DES [54], [55]	2020-0549	L1D Eviction Sampling	2020.1
CROSSTALK/ SRBDS [56], [57]	2020-0543	Special Register Buffer Data Sampling	2020.6
LVI [58]	2020-0551	Load Value Injection causing memory disclosure	2020.3
Speculative Interference [59]	N/A	Speculative interference on non-speculative instructions	2020.9

TABLE II: The Speculative (Transient) Execution Attack variants. *Date* is year.month of publication.

instruction that leaks out the secret through the cache covert channel. In line 11-13, the receiver measures the latency to access the shared array to find out which memory address in the shared array was accessed by the sender. The memory address that hits in the cache leaks the secret.

C. Other Attacks

Table II gives a listing of the speculative attacks published to date [3]–[6], [41]–[59]. We show their Common Vulnerabilities and Exposures (CVE) numbers, description and publication date. All the attack variants in Table II, except for the last speculative interference attack, introduce a new way to bypass authorization to access the secret. The speculative interference attack introduces a new way to change the timing of non-speculative instructions, which adds a new dimension to the covert *Send* operation.

Hardware features for malicious speculative execution. In Fig. 3, we show the hardware features that can be exploited to launch malicious speculative execution attacks, especially to access a secret.

The first major category of features causing misprediction include the conditional branch prediction, the prediction for branch target address and the memory disambiguation. Spectre v1 [3] attack mistrains the conditional branch for bounds checking to read an out-of-bounds secret. Spectre v1.1 [41] also uses misprediction for conditional branch to bypass bounds checking but performs an out-of-bounds write during speculative execution. Even if the write to memory will not become visible, the write may change a jump target, e.g., the return address, and execute an *Access-Use-Send* gadget (i.e.,

a code snippet) as we show in lines 7-10 in Fig. 2 to read and leak a secret. NetSpectre [42] shows that the mistraining of the conditional branch predictor can be performed remotely.

Another control-flow misprediction based attack is the Spectre v2 attack [3], which injects a malicious target into the branch target buffer (BTB) for indirect branches. Similarly, the Spectre RSB attack [43], [44] injects wrong return addresses into the return stack buffer (RSB) for function returns. Both can cause information leakage by directing the control flow to an *Access-Use-Send* gadget.

Memory disambiguation checks whether the value written by a previous store instruction, which has not yet been written back to the cache-memory system, should be forwarded to a later load instruction that reads from the same address. In the Speculative Store Bypass (Spectre SSB) attack, if the store address has not been computed and the processor predicts that the addresses of the current load and a previous store are different, then stale data, which can be a secret, can be loaded from the memory system to the processor and get leaked out.

The second major category of hardware features exploited consists of an illegal access that reads a secret and forwards it to dependent instructions before it is squashed. We call these "faulty access and aggressive forwarding" attacks. The first type of attacks transiently bypasses permission checks of special registers and delays the exception handling. Meltdown-Reg [46] can read the system parameter stored in a system register while LazyFP [47] leaks the stale floating-point unit (FPU) state of a previous domain that is not cleared until first used in a new context.

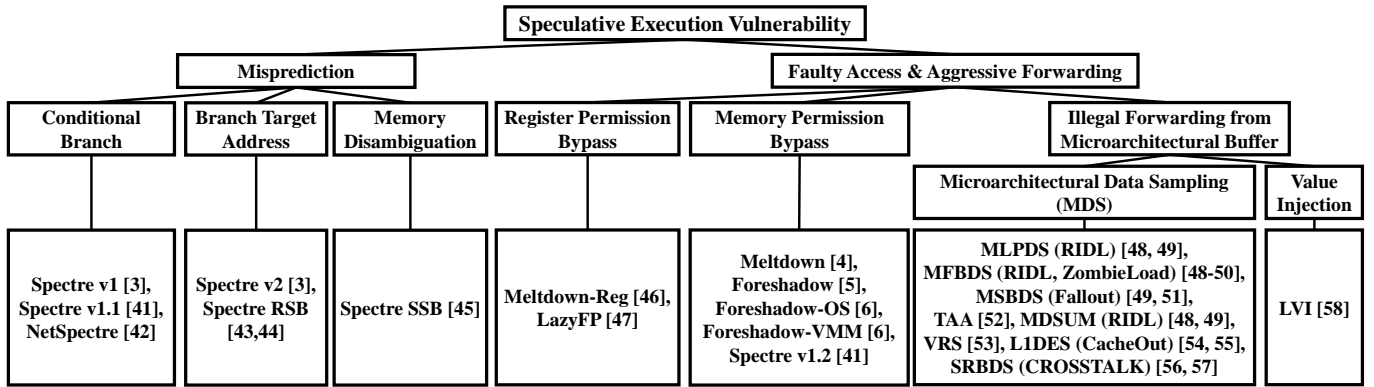The second type of faulty access attacks transiently violate memory access permission checking and reads illegal data

Speculative Execution Vulnerability

Misprediction | **Faulty Access & Aggressive Forwarding**

Conditional Branch	**Branch Target Address**	**Memory Disambiguation**	**Register Permission Bypass**	**Memory Permission Bypass**	**Illegal Forwarding from Microarchitectural Buffer**	
					Microarchitectural Data Sampling (MDS)	**Value Injection**
Spectre v1 [3], Spectre v1.1 [41], NetSpectre [42]	Spectre v2 [3], Spectre RSB [43,44]	Spectre SSB [45]	Meltdown-Reg [46], LazyFP [47]	Meltdown [4], Foreshadow [5], Foreshadow-OS [6], Foreshadow-VMM [6], Spectre v1.2 [41]	MLPDS (RIDL) [48, 49], MFBDS (RIDL, ZombieLoad) [48-50], MSBDS (Fallout) [49, 51], TAA [52], MDSUM (RIDL) [48, 49], VRS [53], L1DES (CacheOut) [54, 55], SRBDS (CROSSTALK) [56, 57]	LVI [58]

Fig. 3: Taxonomy of secret access (bypassed authorization and secret access steps). The third and fourth rows show the hardware mechanisms used to trigger the transient execution. They correspond to delayed *Authorize* operations that are temporarily bypassed. The last row shows the attacks that exploit these hardware features. These are listed in the same order as in Table I, from left to right.

with a memory access instruction. Meltdown [4] reads and leaks kernel data before the execution is squashed due to the failed supervisor permission check of the secret access. The Foreshadow (L1 terminal fault) attack variants [5], [6] exploit loads which do not have a valid virtual address to physical address mapping. The address translation will abort prematurely by returning a partially translated address. If a secret at this incorrect address is present in the L1 cache, it can be speculatively accessed and leaked out. The leaked data can be a secret in an SGX enclave (Foreshadow), in the kernel space (Foreshadow-OS) or in the virtual machine monitor space (Foreshadow-VMM). Spectre v1.2 attack [41] transiently bypasses the read/write permission and writes to a read-only address. The illegal write can trigger an *Access-Use-Send* gadget to leak a secret if it is a branch target.

The more recent type of attacks (in 2019 and 2020) exploit the hardware vulnerability that some stale data, which is stored in microarchitectural buffers can be read by a load that will cause a fault or invoke a microcode assist [49]. The data can belong to another security domain and can be at a different address from the address the faulting load is accessing. This type of attack is called a microarchitectural data sampling (MDS) attack. In an MDS attack, the victim program first executes and accesses a secret. The secret can be temporarily stored in a microarchitectural buffer when it is in-flight. However, the stale secret value can be forwarded to a faulting or microcode-assisted load issued by the MDS attacker which then sends it out through a covert channel.

Microarchitectural buffers that have been shown to store stale secret values include the load port, the line fill buffer and the store buffer, which we show in Fig. 1. The load port temporarily stores the data when it is read by a load operation and being written into a register. The line fill buffer stores a memory line that missed in the L1 data cache and is being returned from the L2 cache [49]. The store buffer stores the data and addresses of store operations to be written to the L1 data cache. RIDL [48] leaks the secret stored in the load port called Microarchitectural Load Port Data Sampling (MLPDS) [49] and the line fill buffer called Microarchitectural Fill Buffer Data Sampling (MFBDS) [49].

ZombieLoad [50] demonstrates more variants of the line fill buffer leakage (MFBDS), whose secret access is triggered by a microcode assist. Fallout [51] leaks the secret stored in the store buffer called Microarchitectural Store Buffer Data Sampling (MSBDS) [49].

A vulerability similar to MDS is the TSX Asynchronous Abort (TAA) [52] in Intel processors. If the Intel TSX atomic execution is aborted, uncompleted loads in the transaction may also read a secret from the microarchitectural buffers exploited by MDS and leak it through a covert channel.

The MDS and TAA techniques give rise to more attacks. Uncacheable memory accesses [48], [49] can bring data into the buffers mentioned above, which can be accessed using MDS or TAA technqiues and cause the Microarchitectural Data Sampling Uncacheable Memory (MDSUM) attack. The Vector register sampling (VRS) vulnerability [53] allows part of the previously accessed vector register values to be sent to the store buffer and get leaked by an MSBDS-type attacker. The CacheOut [54] or L1D eviction sampling (L1DES) vulnerability [55] shows that the modified data recently evicted from the L1 data cache can be kept in the line fill buffer, which gives an MFBDS-type attacker the chance to read and leak it. In the CrossTalk [56] or special register buffer data sampling (SRBDS) attack [57], the secret value read from certain special registers can be stored in shared buffers and later propagated to the line fill buffer. The secret can be leaked to an MFBDS-type attacker who can even be from a different core. We refer to all the above MDS-related attacks as MDS attacks in Fig. 3.

The other type of microarchitectural buffer related attack, i.e., the load value injection (LVI) attacks [58], explore injecting values to the victim domain to trigger speculation. The attacker first places his malicious data in the microarchitectural buffers and lets the victim access the malicious value through the MDS vulnerabilities. If the malicious value is used by the victim as an address to read a secret or a jump address to an *Access-Use-Send* gadget, the secret can be leaked.

D. Covert Channels for Send Operation

Microarchitectural covert channels are used to transmit the secret that has been illegally accessed. In Fig. 4, we show three
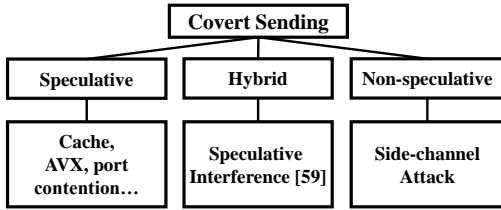
Fig. 4: Different ways to leak a secret through a *Send* operation. The speculative interference attack [59] achieves the final covert *Send* through a non-speculative instruction.

types of covert *Send* operations. These are through speculative instructions, both speculative and non-speculative instructions (hybrid), and only non-speculative instructions. Most of the existing speculative execution attacks are in the first category, executing a speculative *Send* operation to cause a secret-dependent state change in the covert channel that can be recovered later by the receiver. The cache covert channel is the most commonly used channel. Examples of other covert channels include the execution time of AVX instructions [42], port contention [60] and the cache way predictor [61].

The recently discovered speculative interference attack [59] leaks the secret through non-speculative instructions by changing the timing of non-speculative instructions with the speculatively executed instructions. In Fig. 4, we characterize it as doing a hybrid two-step covert sending. In the first step, the speculative execution causes a secret-dependent hardware unit usage, affecting the timing of non-speculative instructions. In the second step, the timing information of non-speculative instructions can leak the secret. Examples include using the speculative 1) miss status handling register (MSHR) or 2) execution unit contention (first step) to change the timing of a non-speculative load (second step). Essentially, the two examples exploit two different covert channels in the first step, rather than the commonly used flush-reload cache channel.

If the *Send* operation is purely non-speculative as shown in the last case of Fig. 4, the attack becomes a side-channel attack, especially when both *Access* and *Send* operations are also non-speculative. This means the program has side-channel vulnerability that allows the secret access and the operation causing a secret-dependent microarchitectural state change, which is beyond the scope of speculative execution attacks.

Takeaway from attack analysis. The important observation we make is that the critical attack steps in Section III-A hold for all speculative execution attacks, not just for the Spectre v1 attack. Moreover, **any valid combination of delayed authorization, speculative secret access and a covert channel can form a new attack variant.** Based on this characterization of speculative attacks, we propose four defense strategies that prevent these speculative execution attacks from succeeding.

IV. DEFENSE STRATEGIES

We propose a taxonomy of defenses depending on the attack step prevented, shown in Fig. 5. We identify four defense strategies, each based on a security policy:

- *No Setup* (Section IV-A): *Setup* is prevented so that either the malicious speculative execution cannot start or the covert channel state cannot be initialized.
- *No Access without Authorization* (Section IV-B): *Access* cannot execute before the authorization is completed.
- *No Use without Authorization* (Section IV-C): *Access* can execute but *Use* of a secret is blocked before the authorization is completed.
- *No Send without Authorization* (Section IV-D): Both *Access* and *Use* can execute but no secret can be sent, before the secret access is authorized.

The insight about *No Access without Authorization* is that while *Authorize* and *Access* may not have any data dependencies, they have a *security dependency* [37] since an access should not be allowed until it is authorized. Hence the *No Access without Authorization* security policy prevents the security breach. Given that *Access*, *Use* and *Send* are a chain of 3 data-dependent instructions, *No Use without Authorization* and *No Send without Authorization* defense strategies can be understood as enforcing the protection at a later stage to try to reduce the performance overhead.

We will describe representative defense proposals for each of these defense strategies.
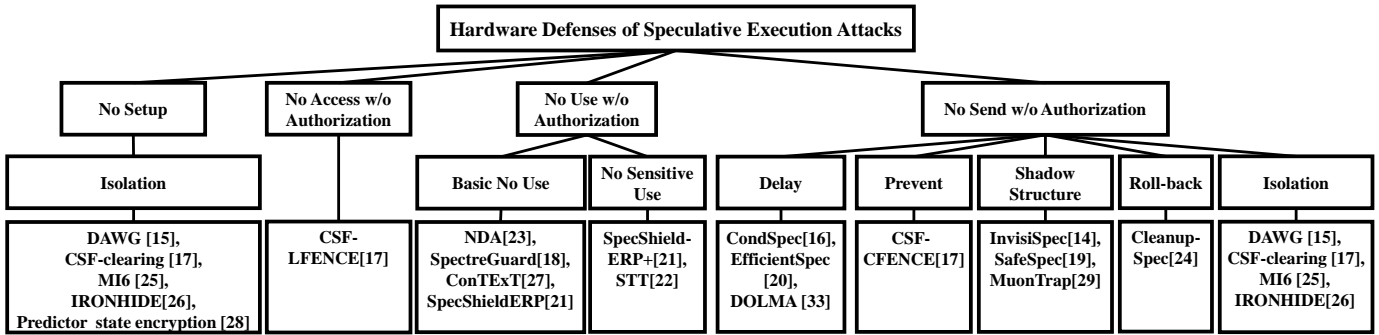
A. No Setup

There are two ways to prevent the *Setup* step. A defense can prevent either the preparation of the covert channel state or the trigger for speculative execution. Both can be achieved with an isolation-based method shown in Fig. 5.

The isolation method requires partitioning of otherwise shared hardware resources or flushing of a hardware resource if it is time-multiplexed. DAWG [15] partitions the cache lines using the domain_id's and guarantees no interference through the cache replacement state. Context-sensitive fencing [17] implements a new micro-op to flush the branch target buffers (BTB) or return stack buffer (RSB) state when entering a different protection domain. MI6 [25] partitions the shared DRAM and last-level cache (LLC) resources between trusted enclaves and untrusted software and enables clearing any per-core states such as branch predictors, L1 caches and TLBs, with a new instruction. IRONHIDE [26] implements a similar partitioning of LLC and memory resources and also a core-level partitioning by reserving certain cores for a security-critical program to reduce the cost of clearing per-core states.

Encryption can be applied to hardware states to implement an obfuscation-based isolation defense. Predictor state encryption [28] encrypts the BTB or RAS state with a context-specific secret when storing a new target address and decrypts it for usage. This prevents the attacker in another process from injecting malicious jump/return targets, without requiring the clearing of microarchitectural states. Such context-specific encryption can also be considered a form of isolation.

However, note that these *No Setup* defenses usually require that the victim and the attacker come from different security domains, as the isolation-based method uses the domain information to allocate resources and enforce access control and

Hardware Defenses of Speculative Execution Attacks								
No Setup	**No Access w/o Authorization**	**No Use w/o Authorization**		**No Send w/o Authorization**				
Isolation		**Basic No Use**	**No Sensitive Use**	**Delay**	**Prevent**	**Shadow Structure**	**Roll-back**	**Isolation**
DAWG [15], CSF-clearing [17], MI6 [25], IRONHIDE[26], Predictor state encryption [28]	**CSF-LFENCE[17]**	**NDA[23], SpectreGuard[18], ConTExT[27], SpecShieldERP[21]**	**SpecShield-ERP+[21], STT[22]**	**CondSpec[16], EfficientSpec [20], DOLMA [33]**	**CSF-CFENCE[17]**	**InvisiSpec[14], SafeSpec[19], MuonTrap[29]**	**Cleanup-Spec[24]**	**DAWG [15], CSF-clearing [17], MI6 [25], IRONHIDE[26]**

Performance-enhancing features for hardware defenses: SDO [30], Clearing the Shadows [31], InvarSpec [32]

Fig. 5: Taxonomy of hardware defenses. The second row shows the 4 defense strategies. The third row shows the child defense categories under each strategy. The fourth row shows the proposed hardware defenses belonging to each defense category.
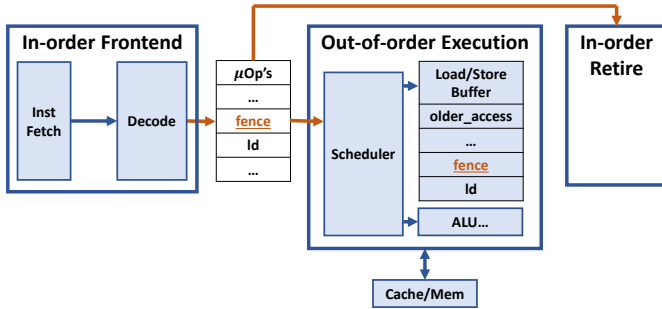

Fig. 6: Inserting fences to stall the speculative execution of loads.

Fig. 7: Hardware modification to support *No Use without Authorization*.

the encryption-based method uses the same key for a certain domain. The same-domain attack, e.g., NetSpectre [42], cannot be mitigated with these techniques.

B. No Access Without Authorization

To prevent a security breach, we should prevent the secret *Access* before the authorization is completed. Software solutions can insert memory barriers such as the *lfence* in the x86 ISA to defeat speculative attacks, but they require re-compilation or post-processing of the binary [62]. Also, significant performance overhead is incurred with these software fences. A hardware defense can also prevent the secret access by automatically inserting a fence micro-op. Hardware-inserted fences have the advantage of non-intrusive protection and much lower overhead.

The Context-Sensitive Fencing (CSF) defense proposed in [17] is shown in Fig. 6. It uses customizable decoding from software instructions to hardware micro-operations to insert hardware fences after a conditional branch instruction before a subsequent load instruction. To defeat the Spectre v1 attack, CSF-LFENCE can place a fence between these two instructions. As no secret data is accessed in the first place, the *No Access without Authorization* defense provides strong protection that is independent of the type of covert channel used to exfiltrate the data.

C. No Use without Authorization

Hardware defenses can allow the secret access but prevent its usage in subsequent execution. This improves performance
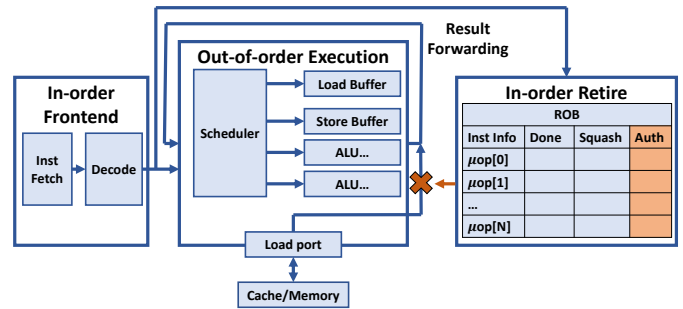
but still blocks the *Use* step in a speculative attack. We call it the *No Use without Authorization* defense strategy.

This strategy requires modifying the feed-forward logic which forwards the result of a producer instruction to dependent instructions so that forwarding is allowed to later operations only when the producer instruction is completed *and* authorized. This can be achieved when both the *Done* and the new *Auth* bits are set in the ROB in Fig. 7.

There are two subclasses of defenses in this category. The "Basic No Use" defenses simply prevent the data forwarding to any dependent instructions. The "No Sensitive Use" defenses improve the performance by only preventing the data forwarding to sensitive instruction types such as memory load instructions, which can be used to send cache covert channel signals, or for other known covert channels.

Basic no use. An example of the "Basic No Use" defense strategy is the NDA (Non-speculative Data Access) defense proposal [23]. This has many variants, based on which authorization checks and *Access* operations are considered. NDA-Permissive checks the resolution of conditional branch conditions and indirect branch addresses (first 2 columns in Fig. 3). NDA-Permissive-BR (Bypass Restriction) checks these and also checks memory address disambiguation (the third column in Fig. 3). These two NDA-Permissive variants protect accesses from the cache and memory and from special registers like control registers.

There are also two NDA-Strict variants: NDA-Strict and NDA-Strict-BR. These are like their NDA-Permissive counterparts, except that they also prevent accesses of secrets that

are already in the general-purpose registers.

The NDA-Load variant further adds hardware to prevent the data forwarding from an *Access* operation until the instruction is retired, i.e., the instruction is at the head of the ROB queue and has its Authorization completed. This covers the first 5 columns in Fig. 3. Since NDA was proposed before the last two columns in Fig. 3, it is not known if it covers the attacks that do illegal forwarding from microarchitectural buffers. NDA-Full is the most secure variant, combining NDA-Strict-BR with NDA-Load.

SpectreGuard [18] is another example of a "Basic No Use" defense. While it only discusses Spectre v1, its key contribution is providing the Linux OS interface to identify sensitive memory pages and mark these as non-speculative. Only data accessed from sensitive pages will not be forwarded during speculative execution, reducing the performance overhead. ConTExT [27] implements similar software support to mark secret data, which should not be used in speculative execution, as non-transient. In addition, ConTExT allows taint propagation in the processor to also taint the values derived from non-transient values. These tainted values cannot be used in speculative execution that happens in the future.

SpecShield [21] also implements a "Basic No Use" defense. It protects any secret in the memory which can be read by load operations. SpecShieldERP prevents data forwarding until the authorization of control flow, memory disambiguation and memory-related permission checking is completed and no violation is found.

No sensitive use. Another variant in [21], SpecshieldERP+, implements a "No Sensitive Use" defense policy by considering the same authorization of control-flow, memory disambiguation authorization and memory permission checking as SpecshieldERP, but only preventing the data forwarding to sensitive instructions like loads and branches.

Speculative Taint Tracking (STT) [22] is another example of the "No Sensitive Use" policy. STT further considers the covert channels due to implicit information flows and marks loads, branches, stores and data-dependent arithmetic instructions as being sensitive. To improve the performance, STT implements an efficient taint tracking mechanism to untaint authorized operations. STT has two variants, STT-Spectre and STT-Future. STT-Spectre considers only the authorization of control flow while STT-future tries to include potential future speculative attacks by deeming a load operation safe only when it reaches the head of the ROB or cannot be squashed.

D. No Send without Authorization

The *No Send without Authorization* defenses prevent sending a signal on a covert channel so that the secret cannot be recovered by the attacker, who is the receiver of the covert channel. This signal is sent by changing the microarchitectural state. The defenses under this strategy are usually specific to one or multiple covert channels. Below, we describe five ways to achieve this goal. Although related defense proposals have considered different sets of covert channels, the cache covert channel is the main target that is addressed by all defenses.

Hence, we consider specifically the memory load instructions, which change the cache state, to explain these covert channels.

Delay state change. The processor can delay the execution of a load when it needs to modify the cache state. An example is the Conditional Speculation (CondSpec) defense [16], where an unauthorized memory load that hits in the cache can read the data and complete its execution. However, a load that has a cache miss is held up to be re-issued later.

The Efficient Invisible Speculative execution (EfficientSpec) defense [20] also implements this **"delay on miss"** mechanism while adding a value predictor to provide a predicted value upon a cache miss. This is compared with the real value after the authorization is completed.

The DOLMA defense [33] addresses a broader scope of covert channels including not only data caches but also TLBs, instruction caches and hardware predictor state covert channels. It delays both explicit state changes and the changes caused by implicit secret-dependent execution flow and by resource contention. DOLMA considers stores as well as loads as the *Send* operation.

Prevent state change. The hardware can allow a speculative load to read the data but prevent the cache state change by making the load uncacheable.

Context-sensitive fencing [17], with some variants implementing *No Access without Authorization* (Section IV-B), also provides a new type of fence, CFENCE, to implement *No Send without Authorization*. A load can execute before a previous CFENCE but it will be converted to a non-cacheable load when it causes a cache miss. This allows the data to be read while preventing the cache state change. The defense variant placing a CFENCE before every load is denoted by "CSF-CFENCE" in Fig. 5.

Store speculative state in shadow structures. Visible cache state can be changed only on a successful authorization, by adding a shadow structure to hold the speculatively accessed cache lines.

InvisiSpec [14] prevents the modification of the cache state, including the cache coherence state in the multiprocessor system, by extending the processor with a speculative buffer to store the speculatively accessed data. If the authorization is completed and verified, each speculative load will issue a second access to the same address and cause safe cache state change. If the authorization is completed but rejected, the load is squashed, and no modification is made to the cache state. One InvisiSpec variant, InvisiSpec-Spectre, deems a load unauthorized until all the control-flow predictions are verified. The other variant, InvisiSpec-Futuristic, deems a load unauthorized until it reaches the head of the reorder buffer (ROB) or it cannot be squashed.

The SafeSpec defense [19] implements a similar shadow buffer to prevent the modification of both cache and translation lookaside buffer (TLB) states. The cache coherence state is not protected by SafeSpec.

MuonTrap [29] adds the filter caches as the shadow buffers for I-cache, D-cache and TLB. The speculatively accessed

Feature	Enhanced Defense	Category	Benchmark	Overhead	
				Before	After
SDO [30]	STT [22]	No Use	SPEC2017	About 22%	10.05%
ClearShadow [31]	Delay on miss [20]	No Send	SPEC2006	9% faster than basic delay-on-miss	
InvarSpec [32]	fence [14]	No Access	SPEC2006	199.3%	101.9%
			SPEC2017	195.3%	108.2%
	Delay on miss [16], [20]	No Send	SPEC2006	46.1%	22.3%
			SPEC2017	39.5%	24.4%
	InvisiSpec [14]	No Send	SPEC2006	18.0%	9.6%
			SPEC2017	15.4%	10.9%

TABLE III: The improvement in performance overhead by applying SDO, ClearShadow and InvarSpec to existing defenses.

entries are only stored in these and get cleared upon security domain switches. A key difference from previous work is that MuonTrap allows non-sensitive modification to the cache coherence state. In a MESI protocol, a speculative access can only be fetched in shared state and any sensitive action changing another cache line from M or E state to S or I state is delayed until it is authorized.

Restore state change (Roll-back). The hardware can allow the cache state change during speculative execution but restore the old cache state if the authorization fails.

CleanupSpec [24] prevents a speculative execution attack from modifying the cache state by restoring the cache state when the speculation is found to be wrong. Before the authorization is completed, CleanupSpec allows bringing new cache lines into the cache during speculative execution, but extends each memory request with its side-effect fields to track which cache line is fetched into the cache and which cache line is evicted from the L1 data cache, due to this unauthorized request. If a memory request needs to be squashed, a request is sent to invalidate any new cache line fetched during speculative execution, and bring back any cache line evicted speculatively from the L1 data cache. The L2 and last-level caches in CleanupSpec implement address encryption [63] to prevent eviction-based information leakage.

Isolation of states between security domains. Assuming that the sender and the receiver are from different security domains, some isolation-based defenses that prevent *Setup* can also prevent the attacker from receiving the covert signaling. For example, the clearing of the branch predictor state can prevent mistraining in the *Setup* phase and also prevent the leakage through covert sending [64]. Hence, a defense can prevent two steps as a *No Setup* defense and a *No Send without Authorization* defense.

E. Reducing Overhead of Defenses

Techniques have been proposed to reduce the performance overhead of defenses described earlier. Table III shows the performance improvements they achieve.

Speculative Data-Oblivious Execution (SDO) [30] allows an instruction, which may depend on a secret, to execute. For instance, a speculative load can access certain cache levels without making any state changes and the performance is improved if the data is found. SDO can be integrated with STT [22].

Clearing the Shadows (ClearShadow) [31] improves the performance by accelerating the computation of branch conditions

Strategy	Defense	Platform	Performance Overhead (%)
No Setup & No Send	DAWG [15]	Zsim [65]	0 ~15
	MI6 [25]	RiscyOO [66]	16.4
	IRONHIDE [26]	Tilera Tile-Gx72 processor [67]	-20 (Compared to an SGX-like baseline)
No Access	CSF-LFENCE [17]	GEM5 [68]	48
No Use	NDA [23]	GEM5 [68]	10.7 ~125
	SpectreGuard [18]	GEM5 [68]	8, 20
	ConTExT [27]	Software approximation on Intel processor	0.1 ~71.1
	SpecShieldERP(+) [21]	GEM5 [68]	10, 21
	STT [22]	GEM5 [68]	8.5, 14.5, 24, 27
No Send	CondSpec [16]	GEM5 [68]	6.8, 12.8, 53.6
	EfficientSpec [20]	GEM5 [68]	11 (IPC loss)
	DOLMA [33]	GEM5 [68]	10.2 ~42.2
	CSF-CFENCE [17]	GEM5 [68]	7.7, 21
	InvisiSpec [14], [69]	GEM5 [68]	5, 17
	SafeSpec [19]	MARSSx86 [70]	-3
	MuonTrap [29]	GEM5 [68]	-5, 4
	CleanupSpec [24]	GEM5 [68]	5.1

TABLE IV: Performance numbers reported by existing work. The numbers may not be directly comparable as they are measured in different configurations. Numbers separated by commas are for different defense variants or benchmarks.

and memory addresses so that *Authorize* can finish earlier. ClearShadow moves the instructions that *Authorize* depends on to the front to shorten or remove the speculation window. ClearShadow has been used to improve a "delay-on-miss" defense [20].

InvarSpec [32] allows some sensitive instructions to execute earlier without protection. InvarSpec software identifies the safe set (SS) of an instruction *I* which contains instructions that are older than *I* but do not affect *I*'s input and execution. InvarSpec hardware extension reads the SS and allows *I* to be issued even if some SS instructions are not resolved. InvarSpec can be applied to the fence-based defense, the delay-on-miss defense and the InvisiSpec defense as we show in Table III.

F. Software-hardware Co-design

Some hardware defenses require software support. One way is changing the application software as described above for ClearShadow [31] and InvarSpec [32]. Another way is modifying the system software. DAWG [15] needs the system software to assign a proper domain ID to the protected program so that the domain ID is not shared with any potential attackers. Context-sensitive fencing [17] has a set of model-specific registers (MSRs) to specify the fence type and the insertion strategy. SpectreGuard [18] and ConTExT [27] enable marking secret data as non-transient by using a bit in the page table entry, which requires both compiler and OS software modifications.

V. UNDERSTANDING PERFORMANCE OVERHEAD

A. Performance Overhead Reported by Defense Papers

TABLE IV shows the performance overhead reported by some hardware defenses, listed according to the hardware defense taxonomy we presented in Fig. 5. The same gem5 cycle-accurate processor simulator [68] is used by most of the hardware defense papers. The overhead of isolation-based defenses to prevent cross-domain *Setup* and *Send* is mainly due to the clearing of microarchitectural states and the partitioning of hardware resources. CSF-LFENCE [17]

HW Defense	No Use w/o Auth.					
	NDA					
	Permissive	Permissive+BR	Strict	Strict+BR	Load	Full
Perf. Overhead	10.7%	22.3%	36.1%	45%	100%	125%

(a) Performance overhead with different Authorization and Access types

HW Defense	No Send w/o Auth.	
	InvisiSpec	
	Spectre	Futuristic
Perf. Overhead	5%	17%

(b) Different Authorization types

HW Defense	No Use w/o Auth.	
	SpecShield	
	SpecShieldERP	SpecShieldERP+
Perf. Overhead	21%	10%

(c) Restricting potential covert channels

TABLE V: Security-performance trade-offs of different variants within the same work.

Fig. 8: Performance analysis of different speculative execution defense strategies on a fast access (a) and a slow access (b).

inserts lfence for only kernel loads but already incurs an overhead of 48%. The *No Use without Authorization* defenses differ a lot in their performance overhead as they may cover different types of authorization (NDA), protect certain data region (SpectreGuard), be emulated with software (ConTExT), and prevent certain sensitive *Use*'s (SpecShield and STT). The *No Send without Authorization* defenses generally have lower perfromance overhead as they only address certain covert channels, especially the cache covert channel.

We illustrate how some of the defenses trade off security and performance. For increased security, more attacks and vulnerabilities can be covered, and more covert channels mitigated, but at increased performance overhead.

Increased overhead for covering more attacks. Table V(a) shows the increase in performance overhead for the NDA [23] defense variants to prevent more types of attacks. The "Permissive" variant considers the control flow authorization only (the first two columns in Fig. 3). The "Permissive+BR (Bypass Restriction)" variant further considers memory disambiguation authorization (the third column in Fig. 3). "Load" (NDA-Load) considers the first five columns in Fig. 3 by not deeming an *Access* operation authorized until it is retired. A fair comparison of performance overhead is from "Permissive" (10.7%), to "Permissive+BR" (22.3%), then to "Load" (100%), since they all protect secrets in memory and special registers.

InvisiSpec [14] provides two variants: InvisiSpec-Spectre defends against control-flow misprediction based attacks and InvisiSpec-Futuristic tries to defend against future attacks, where any speculative load may pose a threat. The latter one is more secure but has more performance overhead (17% vs. InvisiSpec-Spectre's 5% in Table V (b)) [69].

Access type vs. Performance trade-off. TABLE V(a) also shows that as more types of *Access* are considered, the performance overhead increases. The variant "Strict+BR" considers the accesses to general-purpose registers (GPRs) in addition to special registers and memory, which are considered by "Permissive+BR". The overhead increases from 22.3% (Permissive+BR) to 45% (Strict+BR) due to the GPR access consideration.

Mitigated covert channels vs. Performance trade-off. In Table V (c) which compares two SpecShield [21] variants, SpecShieldERP disallows the forwarding from a speculative load to all instructions while SpecShieldERP+ only disallows the forwarding to sensitive loads and branch instructions,
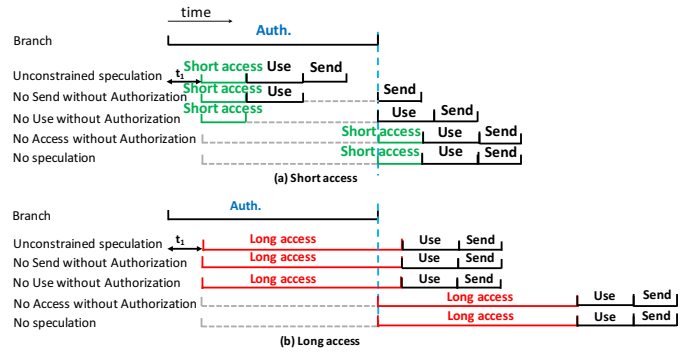
which may be covert *Send*'s. SpecShieldERP+ relaxes some of the security guarantees to reduce the performance impact of SpecShieldERP from 21% to 10%.

B. Security-Performance Tradeoffs Considering Our Defense Strategies

We now consider the theoretical performance overhead reductions that might be expected, as we relax the security policy from *No Access without Authorization* to *No Use without Authorization* to *No Send without Authorization*. These correspond to the three main categories in our defense strategies in Fig. 5. Since speculative attacks are rare, our goal is to compare the impact of these defense strategies on normal (benign) speculative execution. We consider an example benign program containing a branch instruction, a first load instruction, an arithmetic instruction and a second load instruction, that is data dependent on the arithmetic instruction which is data dependent on the first load instruction. While there may be an arbitrary number of instructions between these 3 instructions, we show them as sequential (in Fig. 8), to simplify the discussion. To help correlate this code with a speculative execution attack, e.g., Spectre v1 in Fig. 2, these 4 instructions correspond to the *Authorize*, *Access*, *Use* and *Send* operations.

We illustrate the timelines of the *Authorize*, *Access*, *Use* and *Send* operations in Fig. 8. We consider two scenarios: a fast secret access (Fig. 8(a)), e.g., the first load has a cache hit, and a slow secret access (Fig. 8(b)), e.g., the first load has a cache miss. In each scenario, we present 1) an insecure OoO processor allowing any speculation (*Unconstrained Speculation*); 2) a *No Send without Authorization* defense; 3) a *No Use without Authorization* defense; 4) a *No Access without Authorization* defense; and 5) a processor disabling speculation (*No Speculation*).

The *Access*, *Use* and *Send* are a chain of three data-dependent instructions. Therefore the *Access*, *Use* and *Send* operations cannot run together. In Fig. 8(a)), the *No Send without Authorization*, the *No Use without Authorization*, and the *No Access without Authorization* strategies delay the *Send*, *Use* and *Access* operations, respectively, till after the Authorization is resolved. Hence they have increasing performance overhead, showing the intrinsic performance overhead in these

defense strategies. The slowest but most rigorous security policy, *No Access without Authorization*, is as slow as *No Speculation*, if the first load instruction (*Access*) immediately follows the branch instruction, and there are no other non-dependent instructions that can be executed in the speculation window.

In Fig. 8(b), the slow *Access* (cache miss) may not be fully covered by the delay in the *Authorize* operation. The *No Speculation* defense still causes the longest delay, same as the *No Access without Authorization* defense strategy. The *No Use without Authorization* and the *No Send without Authorization* defense strategies introduce shorter delay, and can achieve the fastest performance like the *Unconstrained Speculation* case.

Takeaway: In general, the *No Speculation* and *Unconstrained Speculation* cases give the upper and lower bounds, respectively, for the total execution time. The overheads of the three defense strategies decrease from the strict security policy of *No Access without Authorization*, to the more relaxed but still secure *No Use without Authorization*, to the *No Send without Authorization* strategies, with not much difference in overhead between the last two strategies.

VI. Problems for Some Defenses

Problem scenarios for isolation-based defenses The isolation-based defenses can be used to either prevent the *Setup* step (Section IV-A) or the covert *Send* step (Section IV-D). These defenses can prevent the attack when the victim and the attacker are from different security domains. However, the mis-training can happen in the same domain [35], [42]. The sender and the receiver of covert channel communication can also be from the same domain. For instance, in a Meltdown attack demo [71] where the secret is in the kernel space, the sender instructions that read the secret and send it out are in a user-level process, which also executes the receiver's code to reveal the secret. These special cases can make the isolation-based defenses ineffective. For instance, the DAWG [15] cache uses the domain ID to parition the cache resouces and therefore, it cannot prevent the same-domain attack where the sender and the receiver are in the same process and have the same domain ID. Other *No Send without Authorization*defenses may also have to be applied, e.g., defenses following *Delay*, *Prevent*, *Shadow Structure* or *Roll-back* in Section IV-D.

Problem with covert channel blocking defenses. The issue with the *No Send without Authorization* defenses is that they only protect against one or a few covert channels. However, they do not restrict how secrets are illegally accessed and can protect against new speculative execution or other attacks - but only for the specific covert channels considered in the defense.

VII. Recommendations and Future Work

Recommendation of defense strategies. From a security perspective, preventing the *Access* and *Use* of a secret is more critical, since all covert channels requiring secret-dependent

usage are eliminated. Between these two strategies, the *No Use without Authorization* has better performance as some long-latency loads can be performed under speculative execution, reducing the performance overhead in benign situations.

Mitigating the aggressive forwarding of faulty access. The faulty access attacks can read data that the current program does not have the permission to access. Some defenses prevent [17], [23] these attacks by blocking the execution or completion of an *Access* operation until it is at the head of the reorder buffer (ROB) so that any exception will be immediately handled. We argue that for the faulty access that violates the permission check of memory or special registers, delaying the forwarding to any dependent instructions until its permission check is finished is enough, i.e., a *No Use without Authorization* policy.

For the illegal forwarding from microarchitectural buffers, our suggestion is to disallow the forwarding to any faulting memory accesses, or return a dummy value and disallow its usage. Simply returning a dummy value without preventing its usage is not enough. For instance, returning a dummy value of 0 may cause the leakage of the data at address 0 if the dummy value is speculatively used as an address.

VIII. Conclusions

In this paper, we first show how speculative execution attacks can be classified according to what hardware features are exploited to bypass security checks, that we call *authorizations*. We then show a new attack characterization based on the critical attack steps common to all speculative execution attacks, namely, *Setup*, *Authorize*, *Access*, *Use*, *Send* and *Receive*. We observe that the root cause of the attacks succeeding is the bypassing of the *Authorize* step during speculative execution. This attack characterization enables us to propose the first taxonomy of defense strategies, where each strategy prevents one of the critical attack steps of *Setup*, *Access*, *Use* and *Send*. We show that the 20 defense proposals considered in this paper can be categorized under at least one of these four defense strategies, or as an overhead-reducing feature. We describe important features in these defenses and some of their key hardware modifications. Security-performance tradeoffs are also discussed for defenses that propose multiple variants. We discuss the scope of these hardware defense strategies and show their relative performance overhead.

Future work can consider new attacks and defenses, using and adding to our taxonomies of attacks and defenses. New defenses can be proposed to reduce the performance overhead and/or cover more attack types. For fair comparisons, new defenses should compare their performance with those that target the same set of exploited vulnerabilities, secret accesses and covert channels.

REFERENCES

[1] "Intel SGX," https://software.intel.com/content/www/us/en/develop/documentation/sgx-developer-guide/top.html.

[2] "Arm tee," https://www.arm.com/why-arm/technologies/trustzone-for-cortex-a/tee-reference-documentation.

[3] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[4] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium*, 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[5] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium*, 2018.

[6] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," *Technical report*, 2018, see also USENIX Security paper Foreshadow [5].

[7] "Intel analysis of speculative execution side channels," https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, 2018.

[8] "Arm cache speculation side-channels," https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper, 2018.

[9] "Deep dive: Indirect branch restricted speculation," https://software.intel.com/security-software-guidance/deep-dives/deep-dive-indirect-branch-restricted-speculation, 2018.

[10] "Retpoline: A branch target injection mitigation," https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf?source=techstories.org, 2018.

[11] "A year with spectre: a v8 perspective," https://v8.dev/blog/spectre, 2019.

[12] "What spectre and meltdown mean for webkit," https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/, 2018.

[13] "The chromium projects: Mitigating side-channel attacks," https://www.chromium.org/Home/chromium-security/ssca, 2018.

[14] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *The 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[15] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *The 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[16] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[17] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *The International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019. [Online]. Available: https://doi.org/10.1145/3297858.3304060

[18] J. Fustos, F. Farshchi, and H. Yun, "Spectreguard: An efficient data-centric defense mechanism against spectre attacks," in *The 56th Design Automation Conference (DAC)*, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3317914

[19] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *The 56th Design Automation Conference (DAC)*, 2019.

[20] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *The 46th International Symposium on Computer Architecture (ISCA)*, 2019. [Online]. Available: https://doi.org/10.1145/3307650.3322216

[21] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Specshield: Shielding speculative data from microarchitectural covert channels," in *The 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[22] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data," in *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019. [Online]. Available: https://doi.org/10.1145/3352460.3358274

[23] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "Nda: Preventing speculative execution attacks at their source," in *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019. [Online]. Available: https://doi.org/10.1145/3352460.3358306

[24] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An "undo" approach to safe speculation," in *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019. [Online]. Available: https://doi.org/10.1145/3352460.3358314

[25] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019. [Online]. Available: https://doi.org/10.1145/3352460.3358310

[26] H. Omar and O. Khan, "Ironhide: A secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[27] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "Context: A generic approach for mitigating spectre," in *The 27th Annual Network and Distributed System Security Symposium (NDSS'20), San Diego, CA, USA*, 2020.

[28] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, D. A. Jimnez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, "Evolution of the samsung exynos cpu microarchitecture," in *The ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[29] S. Ainsworth and T. M. Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *The ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[30] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution," in *The ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[31] K.-A. Tran, C. Sakalis, M. Själander, A. Ros, S. Kaxiras, and A. Jimborean, "Clearing the shadows: Recovering lost performance for invisible speculative execution through hw/sw co-design," in *The ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2020. [Online]. Available: https://doi.org/10.1145/3410463.3414640

[32] Z. N. Zhao, H. Ji, M. Yan, J. Yu, C. W. Fletcher, A. Morrison, D. Marinov, and J. Torrellas, "Speculation invariance (invarspec): Faster safe execution through program analysis," in *The 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[33] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, "DOLMA: Securing speculation with the principle of transient non-observability," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/loughlin

[34] "Consumption of speculative data barrier," https://msrc-blog.microsoft.com/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities.

[35] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium*, 2019. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[36] W. Xiong and J. Szefer, "Survey of transient execution attacks and their mitigations," *ACM Computing Surveys*, 2021.

[37] Z. He, G. Hu, and R. Lee, "New models for understanding and reasoning about speculative execution attacks," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[38] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX*

Security Symposium, 2014. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[39] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and counter-measures: The case of aes," in *Topics in Cryptology – CT-RSA 2006*, D. Pointcheval, Ed., 2006.

[40] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., 2016.

[41] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv preprint arXiv:1807.03757*, 2018.

[42] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Net-spectre: Read arbitrary memory over network," in *European Symposium on Research in Computer Security*, 2019.

[43] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[44] G. Maisuradze and C. Rossow, "Ret2spec: Speculative execution using return stack buffers," in *The 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018. [Online]. Available: https://doi.org/10.1145/3243734.3243761

[45] J. Horn, "Speculative execution, variant 4: Speculative store bypass, 2018," *URl: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528*, 2018.

[46] "Spectre v3a (rsre)," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html, 2018.

[47] "Lazy fp," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00145.html, 2018.

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

[49] "Microarchitectural data sampling," https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling.html, 2019.

[50] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *The 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[51] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar *et al.*, "Fallout: Leaking data on meltdown-resistant cpus," in *The 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[52] "TAA," https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/advisory-guidance/intel-tsx-asynchronous-abort.html, 2019.

[53] "VRS," https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/advisory-guidance/vector-register-sampling.html, 2020.

[54] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on intel cpus via cache evictions," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[55] "L1d eviction sampling," https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/advisory-guidance/l1d-eviction-sampling.html, 2020.

[56] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[57] "Special register buffer data sampling," https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/technical-documentation/special-register-buffer-data-sampling.html, 2020.

[58] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "Lvi: Hijacking transient execution through microarchitectural load value injection," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[59] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison *et al.*, "Speculative interference attacks: Breaking invisible speculation schemes," in *The 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[60] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," in *The 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[61] M. Lipp, V. Hažić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a way: Exploring the security implications of amd's cache way predictors," in *The 15th ACM Asia Conference on Computer and Communications Security*, 2020. [Online]. Available: https://doi.org/10.1145/3320269.3384746

[62] "Speculative execution side channel mitigations," https://software.intel.com/content/dam/develop/external/us/en/documents/336996-speculative-execution-side-channel-mitigations.pdf, 2018.

[63] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *The 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[64] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *The Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018. [Online]. Available: https://doi.org/10.1145/3173162.3173204

[65] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *The 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013. [Online]. Available: https://doi.org/10.1145/2485922.2485963

[66] S. Zhang, A. Wright, T. Bourgeat, and A. Arvind, "Composable building blocks to open up processor design," in *The 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[67] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, 2007.

[68] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[69] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "Correction: Invisispec: Making speculative execution invisible in the cache hierarchy," 2019. [Online]. Available: https://iacoma.cs.uiuc.edu/iacoma-papers/corrected_micro18.pdf

[70] A. Patel, F. Afram, and K. Ghose, "Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors," in *1st International Qemu Users' Forum*, 2011.

[71] "Meltdown proof-of-concept," https://github.com/IAIK/meltdown, 2019.