SoK: Practical Foundations for Spectre Defenses

Sunjay Cauligi*† Craig Disselkoen*† Daniel Moghimi† Gilles Barthe♠♣ Deian Stefan†

†UC San Diego, USA ♠MPI for Security and Privacy, Germany ♣IMDEA Software Institute, Spain

ABSTRACT

Spectre vulnerabilities violate our fundamental assumptions about architectural abstractions, allowing attackers to steal sensitive data despite previously state-of-the-art countermeasures. To defend against Spectre, developers of verification tools and compiler-based mitigations are forced to reason about microarchitectural details such as speculative execution. In order to aid developers with these attacks in a principled way, the research community has sought formal foundations for speculative execution upon which to rebuild provable security guarantees.

This paper systematizes the community's current knowledge about software verification and mitigation for Spectre. We study state-of-the-art software defenses, both with and without associated formal models, and use a cohesive framework to compare the security properties each defense provides. We explore a wide variety of tradeoffs in the complexity of formal frameworks, the performance of defense tools, and the resulting security guarantees. As a result of our analysis, we suggest practical choices for developers of analysis and mitigation tools, and we identify several open problems in this area to guide future work on grounded software defenses.

1 INTRODUCTION

Spectre attacks have upended the foundations of computer security [41]—showing that attackers can violate security boundaries and steal secrets despite countermeasures such as *process isolation* [74], *software fault isolation (SFI)* [69], and *control flow integrity (CFI)* [1]. In response, the security community has been working on program analysis tools to both find Spectre vulnerabilities and to guide mitigations (e.g., compiler passes) that make real programs secure in the presence of this class of attacks. However, because Spectre attacks—and speculative execution in general—violate our typical assumptions and abstractions, they have proven particularly challenging to reason about and to defend against.

We have unfortunately gotten off to a rocky start. For example, the MSVC compiler's `/Qspectre` flag—one of the first compiler defenses [49]—inserts mitigations by finding Spectre vulnerability patterns. Since these patterns are not based in any rigorous analysis, the compiler easily misses similarly vulnerable code patterns [54]. As another example, Chrome adopted process isolation as its core defense mechanism against Spectre attacks [56]. This is also unsound: [14] shows that Spectre attacks can be performed across the process boundary, and [57] shows how to read cross-origin data in the browser. More generally, there have been many defense mechanisms against Spectre that turned out to be unsound, incomplete (and thus missed possible attacks), or overly conservative and thus slow.

To thoroughly defend against Spectre, we should take the lessons learned about similar problems in the non-speculative world and transfer those lessons into the new speculative context. One key lesson is that language-based methods provide an effective means to achieve *efficient code* with *provable security guarantees*. For example, the security community turned to language-based security to solidify intricate defense techniques—from SFI enforcement on x86 processors [52], to information flow control [58], to using constant-time programming to eliminate traditional side-channel attacks [7]. Language-based methods are rooted in *program semantics*, which provide rigorous models of program behavior, and serve as the basis for *formal security policies*. These policies help us carefully and explicitly spell out our assumptions about the attacker's strength, and to gain confidence that our tools are sound with respect to this class of attackers—that standalone Spectre vulnerability-detection tools find all of the vulnerabilities they claim to, or that defense techniques (e.g., compiler passes) actually mitigate the attacks they claim.

Formal foundations don't just help us make sure our Spectre defenses are secure, they also help improve the performance of our tools. Without formalizations, Spectre defenses are usually either overly conservative (unnecessarily flagging code as vulnerable, which ultimately leads to unnecessary and slow mitigations) or overly aggressive (and thus vulnerable). For example, speculative load hardening [15] is *safe*—it safely eliminates Spectre-PHT attacks—but is overly conservative: It assumes that *all* array indexing operations must be hardened. Aggressive techniques like oo7 [71] are both inefficient *and* unsafe—they impose unnecessary restrictions yet also miss vulnerable code patterns. Foundations allow us to craft defenses that are, in some sense, minimal (e.g., they target the precise locations where vulnerabilities exist [29, 67]) while still being provably secure.

Alas, not all foundations are equally practical. Since speculative execution breaks common assumptions about program semantics—the cornerstone of language-based methods—existing Spectre foundations explore different design choices, many of which have important ramifications on defense tools and the software produced or analyzed by these tools. For instance, one key choice is the *leakage model* of the semantics, which determines what the attacker is allowed to observe. Another choice is the *execution model*, which simultaneously captures the attacker's strength and which Spectre variants the resulting analysis (or mitigation) tool can reason about. These choices also determine which security *policies* can be captured, and the precision of analysis and mitigation methods.

In this paper, we systematize the community's knowledge on Spectre foundations and identify the different design choices made by existing work and their tradeoffs. For example, while there

*Authors contributed equally to this work.

are many choices for a leakage model, the constant-time [7] and sandbox isolation [29] models are the most pragmatic; leakage models that only consider the data cache trade off security for no clear benefits (e.g., of scalability or precision). As another example, sensible execution models borrow (again) from the constant-time paradigm: They are detailed enough to capture practical attacks, but abstract across different hardware—and are thus useful for both verification and mitigation of software. Other models which capture microarchitectural details like cache structures make the analysis unnecessarily complicated: They do not fundamentally capture additional attacks, and they give up on portability. Throughout, we discuss the limitations of existing frameworks, the defense tools built on top of these foundations, and future directions for research.

Contributions. We systematize knowledge of software Spectre defenses and their associated formalizations, by studying the choices available to developers of Spectre analysis and mitigation tools. Specifically, we:

- Study existing foundations for Spectre analysis in the form of semantics, discuss the different design choices which can be made in a semantics, and describe the tradeoffs of each choice.
- Compare many proposed Spectre defenses—both with and without formal foundations—using a shared framework, and with an emphasis on the differences in the security guarantees they offer.
- Identify open research problems, both for foundations and for Spectre software defenses in general.
- Provide recommendations both for developers and for the research community in order to produce stronger security guarantees.

Scope of systematization. In this systematization, we focus on software-only defenses for Spectre attacks. In particular we focus on *Spectre* and not other transient attacks (such as Meltdown [45], LVI [66], MDS [34], or Foreshadow [65]), in large part because most existing work on software-only defenses also focuses on Spectre attacks—no current tool provides precise detection of any of the other transient attacks at the software level. More fundamentally, many of these other attacks can only be addressed in the hardware, through microcode updates or proprietary changes, making software defenses moot. We focus on *defenses* as other works [14] have already given excellent overviews of the types of Spectre vulnerabilities and the powerful capabilities which they give attackers. And we focus on *software-only* defenses because they allow us to defend against today's attacks on today's hardware. Proposals for hardware defenses are also valuable, but hardware design cycles (and hardware upgrade cycles) are long; software-only defenses will continue to play a valuable role for the foreseeable future.

2 PRELIMINARIES

In this section, we first discuss Spectre attacks and how they violate security in two particular application domains: high-assurance cryptography and isolation of untrusted code. Then, we provide an introduction to formal semantics for security and its relevance to secure speculation in these application domains.

```
if (i < len) {
  int c1 = input[i];
  int c2 = sbox[c1];
  // ...
}
```

Figure 1: Code snippet which an attacker can exploit using Spectre. In this example, out-of-bounds data is leaked via the data cache state.

2.1 Spectre vulnerabilities

Spectre [4, 6, 32, 41, 42, 47, 77] is a recently discovered family of vulnerabilities due to *speculative execution* on modern processors. Spectre allows attackers to learn sensitive information by causing the processor to mispredict the targets of control flow (e.g., conditional jumps or indirect calls) or data flow (e.g., aliasing or value forwarding). When the processor learns that its prediction was wrong, it *rolls back* execution, erasing the programmer-visible effects of the speculation. However, *microarchitectural* state—such as the state of the data cache—is still modified during speculative execution; these changes can be leaked during speculation and can persist even after rollback. As a result, the attacker can recover sensitive information from the microarchitectural state, even if the sensitive information was only speculatively accessed.

Figure 1 gives an example of a function where an attacker can exploit a branch misprediction to leak information via the data cache. The attacker can first prime the branch to predict that the condition `i < len` is true, by causing the code to repeatedly run with appropriate values of `i`. Then, the attacker provides an out-of-bounds value for `i`. The processor (mis)predicts that the condition is still true and speculatively loads out-of-bounds data into `c1`; subsequently, it uses the value `c1` as part of the address of a memory read operation. This encodes the value of `c1` into the data cache state: depending on the value of `c1`, different cache lines will be accessed and cached. Once the processor resolves the misprediction, it rolls back execution, but the data cache state persists. The attacker can later interpret the data cache state in order to infer the value of `c1`.

2.2 Breaking cryptography with Spectre

High-assurance cryptography has long relied on *constant-time programming* [7] in order to create software which is secure from timing side-channel attacks. Constant-time programming dictates that both control flow (e.g., conditional branches) and memory addresses (e.g., offsets into arrays) are not influenced by secret information [7, 9]. These rules ensure that secrets remain safe from an attacker with the ability to perform cache attacks on either the instruction or data cache, exfiltrate data via branch predictor state, or even infer secret data from port contention in the microarchitecture [11]. The rules can also easily be extended to consider timing leaks due to variable-latency instructions (e.g., floating-point, SIMD, or integer division on many processors) [3].

Unfortunately, in the face of Spectre, standard constant-time programming is insufficient: For example, Figure 1 is constant-time if the `input` array only contains public data (and `i` and `len` are also

public). Despite this, a Spectre attack can still abuse this code to leak secret data from elsewhere in memory.

Secret-dependent memory addresses are not the only way for an attacker to learn cryptographic secrets. In the following example, an attacker can again (speculatively) leak out-of-bounds data, but this time the leak is via control flow.

```
if (i < len) {
  int c1 = input[i];
  switch(c1) {
    case 'A': /* ... */
    case 'B': /* ... */
    // ...
} }
```

Instead of using `c1` as part of the address of a memory operation, this code uses `c1` as part of a branch condition (in a `switch` statement). In this case, the attacker could potentially recover the cryptographic secret (speculatively stored in `c1`) in several ways, including: (1) based on the different execution times of the various cases; (2) through the data cache, based on differing (benign) memory accesses performed in the various cases; (3) through the instruction cache, based on which instructions were (speculatively) accessed; or (4) through port contention [11] or other microarchitectural resource pressure, based on which instructions were (speculatively) executed.

By avoiding secret-dependent memory accesses, control flow, and variable-latency instructions—*even speculatively*—we can prevent any secrets from being leaked to the attacker, by any of the above mechanisms. This illustrates why high-assurance cryptographic code must extend the constant-time principle to the domain of speculative execution.

2.3 Breaking software isolation with Spectre

Spectre attacks also break important guarantees in the domain of *software isolation*. In this domain, a host application accepts and executes untrusted code, and wants to ensure that the untrusted code cannot access any of the host's data. Common examples of host applications include JavaScript or WebAssembly runtimes, or even the Linux kernel, through eBPF [24]. Spectre attacks can break the memory safety and isolation mechanisms commonly used in these settings [38, 53, 63].

The following code gives an example of a Spectre vulnerability in an isolation context.

```
int guest_func() {
  get_host_val(1);
  get_host_val(1);
  // ...
  char c = get_host_val(99999);
  // leak c
}

// Guests can call this to get values from the array
// host_arr, which has 100 characters.
char get_host_val(int idx) {
  if (idx < 100) {
    return host_arr[idx];
  } else {
    return 0; // out-of-bounds!
} }
```

Here, an attacker-supplied guest function `guest_func` is allowed to call the host function `get_host_val` to get values from the array `host_arr`, which has 100 characters. Although `get_host_val()` implements a bounds check, the attacker is still able to speculatively access out-of-bounds data by training the bounds check to pass. Once the attacker (speculatively) obtains an out-of-bounds value of their choosing, they can leak the value using any number of mechanisms and then recover it after the speculative rollback, as in the previous examples. In this setting, since the attacker supplies the untrusted guest code, the isolation policy needs to ensure that the guest cannot even speculatively obtain a secret value in a register.

2.4 Security properties and execution semantics

Developers of formal analyses define safety from Spectre attacks as a security property of a *formal (operational) semantics*. The semantics abstractly captures how a processor executes a program as a series of state transitions. The states, which we will write as σ, include any information the developer will need to track for their analysis, such as the current instruction or command and the contents of memory and registers. The developer then defines an *execution model*—a set of transition rules that specify how state changes during execution. For example, in a semantics for a low-level assembly, a rule for a `store` instruction will update the resulting state's memory with a new value.

More importantly, the rules in the execution model determine how and when speculative effects happen. For example, in a sequential semantics, a conditional branch will evaluate its condition then step to the appropriate branch. A semantics that models branch prediction will instead *predict* the condition result and step to the predicted branch. We adapt notation from Guarnieri et al. [29]: The $[\![\cdot]\!]^{\text{seq}}$ model represents standard sequential execution, while $[\![\cdot]\!]^{\text{pht}}$ models prediction on conditional branches. Other execution models are listed in Figure 3.

Next, to precisely specify the attacker model, the developer must define which *leakage observations*—information produced during an execution step—are visible to an attacker. For example, we may decide that rules with memory accesses leak the addresses being accessed. The set of leakage observations in a semantics' rules is its *leakage model*. We again borrow notation from [29], which defines the leakage models $[\![\cdot]\!]_{\text{ct}}$ and $[\![\cdot]\!]_{\text{arch}}$. The $[\![\cdot]\!]_{\text{ct}}$ model exposes leakage observations relevant to constant-time security: The sequence of control flow (the *execution trace*) and the sequence of addresses accessed in memory (the *memory trace*).[1] The $[\![\cdot]\!]_{\text{arch}}$ model exposes the values loaded from memory in addition to the addresses.[2] Since the leakage observations in $[\![\cdot]\!]_{\text{arch}}$ are a strict superset of those in $[\![\cdot]\!]_{\text{ct}}$, we say that $[\![\cdot]\!]_{\text{arch}}$ is *stronger* than $[\![\cdot]\!]_{\text{ct}}$, meaning that it models a more powerful attacker. In fact, with $[\![\cdot]\!]_{\text{arch}}$, effectively all computation is leaked to the attacker. Thus, $[\![\cdot]\!]_{\text{arch}}$ is most useful for software isolation: A sandbox verifier, for example, will want to prove that a guest sandbox cannot retrieve any data at all from outside its sandbox boundary.

[1] Like [29], we omit variable-latency instructions from our formal model for simplicity.
[2] Equivalently, it exposes the trace of register values [29].

The execution model and leakage model of a semantics together form its *contract*. For example, a semantics for sequential constant-time has the contract $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$. The contract governs the attacker-visible information produced when executing a program: Given a program p, a semantics with contract $[\![\cdot]\!]^{\alpha}_{\ell}$, and an initial state σ, we write $[\![p]\!]^{\alpha}_{\ell}(\sigma)$ for the sequence (or *trace*) of leakage observations the semantics produces when executing p.

After determining a proper contract, the developer must finally define the *policy* that their security property enforces: Precisely which data can and cannot be leaked to the attacker. Formally, a policy is given as an equivalence relation $\simeq_\pi$ over states, where $\sigma_1 \simeq_\pi \sigma_2$ iff σ_1 and σ_2 agree on all values that are public (but may differ on sensitive values).

Armed with these definitions, we can state security as a *non-interference property* over execution states. Informally, if a program satisfies non-interference, then an attacker cannot determine anything about any secret values even if they have all leakage observations.

Definition 1 (Non-interference)**.** Program p satisfies *non-interference* with respect to a given contract $[\![\cdot]\!]$ and policy π if, for all pairs of π-equivalent initial states σ and σ', executing p with each initial state produces the same trace. That is, $p \vdash NI(\pi, [\![\cdot]\!])$ is defined as

$$\forall \sigma, \sigma' : \sigma \simeq_\pi \sigma' \Rightarrow [\![p]\!](\sigma) = [\![p]\!](\sigma').$$

A developer has several choices when crafting a suitable semantics and security policy; these choices greatly influence how easy or difficult it is to detect or mitigate Spectre vulnerabilities. We cover these choices in detail in Section 3. In Sections 3.1 and 3.2, we discuss choices in leakage models $[\![\cdot]\!]_\ell$ and security policies π. In Sections 3.3 and 3.4, we discuss tradeoffs for different execution models $[\![\cdot]\!]^\alpha$ and the nature of the transition rules in a semantics. In Section 3.5, we discuss how the input language of the semantics affects analysis; and finally, in Section 3.6, we discuss which microarchitectural features to include in formal models.

3 CHOICES IN SEMANTICS

The foundation of a well-designed Spectre analysis tool is a carefully constructed formal semantics. Developers face a wide variety of choices when designing their semantics—choices which heavily depend on the attacker model (and thus the intended application area) as well as specifics about the tool they want to develop. Cryptographic code requires different security properties, and therefore different semantics and tools, than in-process isolation. Many of these choices also look different for *detection* tools, focused only on finding Spectre vulnerabilities, vs. *mitigation* tools, which transform programs to be secure. In this section, we describe the important choices about semantics that developers face, and explain those choices' consequences for Spectre analysis tools and for their associated security guarantees. We also point out a number of open problems to guide future work in this area.

What makes a practical semantics? A practical semantics should make an appropriate tradeoff between *detail* and *abstraction*: It should be detailed enough to capture the microarchitectural behaviors which we're interested in, but it should also be abstract enough that it applies to all (reasonable) hardware. For example, we don't want the security of our code to be dependent on a specific cache replacement policy or branch predictor implementation.

In this respect, formalisms for constant-time have been successful in the non-speculative world: The principles of constant-time programming—no secrets for branches, no secrets for addresses—create secure code without introducing processor-specific abstractions. Speculative semantics should follow this trend, producing portable tools which can defend against powerful attackers on today's (and tomorrow's) microarchitectures.

3.1 Leakage models

Any semantics intended to model side-channel attacks needs to precisely define its attacker model. An important part of the attacker model for a semantics is the *leakage model*—that is, what information does the attacker get to observe? Leakage models intended to support sound mitigation schemes should be *strong*—modeling a powerful attacker—and *hardware-agnostic*, so that security guarantees are portable. That said, the best choice for a leakage model depends in large part on the intended application domain.

Leakage models for cryptography. As we saw in Section 2.2, high-assurance cryptography implementations have long relied on the constant-time programming model; thus, semantics intended for cryptographic programs naturally choose the $[\![\cdot]\!]_{\text{ct}}$ leakage model. Like the constant-time programming model in the non-speculative world, the $[\![\cdot]\!]_{\text{ct}}$ leakage model is strong and hardware-agnostic, making it a solid foundation for security guarantees. The $[\![\cdot]\!]_{\text{ct}}$ leakage model is a popular choice among existing formalizations: As we highlight in Figure 2, over half of the formal semantics for Spectre use the $[\![\cdot]\!]_{\text{ct}}$ leakage model (or an equivalent) [8, 16, 21, 27, 28, 55, 67]. Guarnieri et al. [29] leave the leakage model abstract, allowing the semantics to be used with several different leakage models, including $[\![\cdot]\!]_{\text{ct}}$.

Leakage models for isolation. Sections 2.3 and 2.4 described the $[\![\cdot]\!]_{\text{arch}}$ leakage model more suited to speculative isolation—e.g., JavaScript or WebAssembly runtimes which execute untrusted code. Like $[\![\cdot]\!]_{\text{ct}}$, this is a hardware-agnostic leakage model, well-suited to providing solid security guarantees. The $[\![\cdot]\!]_{\text{arch}}$ leakage model is, however, less-frequently considered in formal models: Only two of the semantics in Figure 2 ([18, 29]) use the $[\![\cdot]\!]_{\text{arch}}$ leakage model. Spectre sandbox isolation frameworks such as Swivel [53], Venkman [63], and ELFbac [38] appear to use the $[\![\cdot]\!]_{\text{arch}}$ model as a guide, as do SpecFuzz [54] and certain modes of oo7 [71]. Unfortunately, many of these tools are not formalized, so their leakage models are not explicit (and clear).

Weaker leakage models. The remaining semantics and tools in Figure 2 consider only the memory trace of a program, but not its execution trace. The $[\![\cdot]\!]_{\text{mem}}$ leakage model, like $[\![\cdot]\!]_{\text{ct}}$, allows an attacker to observe the sequence of memory accesses during the execution of the program. The $[\![\cdot]\!]_{\text{cache}}$ leakage model instead tracks (an abstraction of) cache state. The attacker in this model can only observe cached addresses at the granularity of cache lines. A few tools have leakage models even weaker than these—for instance, oo7 only emits leakages that it considers can be influenced by malicious input (see Section 3.3), and KLEESpectre (with cache modeling enabled) only allows the attacker to observe the final state of the cache once the program terminates.

Semantics or tool name	Level	Leakage		Variants	Nondet.	Fence	OOO	Win.	Tool	Impl.
Cauligi et al. [16] (Pitchfork)	Low	$[\![\cdot]\!]_{ct}$	P,B,M	P,B,R,S	Directives	✓	✓	✓	Det*	Taint
Cheang et al. [18]	Low	$[\![\cdot]\!]_{arch}$	P,M,S,R	P	Oracle	✓	×	✓	Det/Mit	SelfC+
Daniel et al. [21] (Binsec/Haunted)	Low	$[\![\cdot]\!]_{ct}$	P,M	P,S	Mispredict	×	×	✓	Det	SelfC
Guanciale et al. [27] (InSpectre)	Low	$[\![\cdot]\!]_{ct}$	P,M	P,B,R,S	—	✓	✓	×	—	—
Guarnieri et al. [28] (Spectector)	Low	$[\![\cdot]\!]_{ct}$	P,B,M	P	Oracle	✓	×	✓	Det	SelfC+
Guarnieri et al. [29]	Low	(parametrized)		P[1]	Oracle	✓	✓	✓	Det	SelfC+
Mcilroy et al. [48]	Low	$[\![\cdot]\!]_{cache}$	T	P[2]	Oracle	⌣	×	✓	Mit*	Manual
Barthe et al. [8] (Jasmin)	Medium	$[\![\cdot]\!]_{ct}$	P,B,M	P,S	Directives	✓	×	×	Det	Safety
Patrignani and Guarnieri [55]	Medium	$[\![\cdot]\!]_{ct}$	P,B,M,L[3]	P[1]	Mispredict	✓	×	✓	—	—
Vassena et al. [67] (Blade)	Medium	$[\![\cdot]\!]_{ct}$	B,M	P	Directives	✓	✓	×	Mit	Flow
Colvin and Winter [19]	High	$[\![\cdot]\!]_{mem}$	M	P	Weak-mem	✓	✓	×	Val	Val
Disselkoen et al. [22]	High	$[\![\cdot]\!]_{mem}$	M	P	Weak-mem	✓	✓	×	—	—
AISE [73]	—	$[\![\cdot]\!]_{cache}$	C	P	Mispredict	×	×	✓	Det	Cache+
ELFbac [38]	—	$[\![\cdot]\!]_{arch}$	L	P	—	×[4]	×	×	Mit	Struct
KLEESpectre [70] (w/ cache)	—	$[\![\cdot]\!]_{cache}$	C	P	Mispredict	✓	×	✓	Det	Cache
KLEESpectre [70] (w/o cache)	—	$[\![\cdot]\!]_{mem}$	M	P	Mispredict	✓	×	✓	Det	Taint
oo7 [71] (v1 pattern)	—	$[\![\cdot]\!]_{mem}$	M	P	—	⌣	×	✓	Det/Mit	Flow
oo7 [71] ("weak" and v1.1 patterns)	—	$[\![\cdot]\!]_{arch}$	L	P	—	⌣	×	✓	Det/Mit	Flow
SpecFuzz [54]	—	$[\![\cdot]\!]_{arch}$	L	P	Mispredict	—	—	—	Det	Fuzz
SpecuSym [30]	—	$[\![\cdot]\!]_{cache}$	C	P	Mispredict	×	×	✓	Det	SelfC+
Swivel [53] (poisoning protection)	—	$[\![\cdot]\!]_{mem}$	M	P,B,R	—	⌣[5]	×	×	Mit	Struct
Swivel [53] (breakout protection)	—	$[\![\cdot]\!]_{arch}$	L	P,B,R	—	⌣[5]	×	×	Mit	Struct
Venkman [63]	—	$[\![\cdot]\!]_{arch}$	L	P,B,R	—	⌣	×	×	Mit	Struct

Level – How abstract is the semantics? (Section 3.5)

Low	Assembly-style, with branch instructions
Medium	Structured control flow such as if-then-else
High	In the style of weak memory models
—	The work has no associated formal semantics

Leakage – What can the attacker observe? (Section 3.1)

P	– Path / instructions executed	L	– Values loaded from memory
B	– Speculation rollbacks	R	– Values in registers
M	– Addresses of memory operations	S	– Branch predictor state
C	– Cached lines / cache state	T	– Step counter / timer

Variants (Section 3.3)

P	– Spectre-PHT
B	– Spectre-BTB
R	– Spectre-RSB
S	– Spectre-STL

Fence – Does it reason about speculation fences?

✓	Fully reasons about fences in the target/input code
⌣	The mitigation tool inserts fences, but the analysis doesn't reason about fences in the target/input code (and thus can't verify the mitigated code as secure)
×	Does not reason about, or insert, fences

Nondet. – How is nondeterminism handled? (Section 3.4)

OOO – Models out-of-order execution? (Section 3.6)

Win. – Can reason about speculation windows? (Section 3.3)

Tool – Does the paper include a tool?

Det	Includes a tool that can be used to detect insecure programs or verify secure programs
Mit	Includes a tool that can be used to modify programs to ensure they are secure
Val	Includes a tool that is only used to validate the semantics and doesn't automatically perform any security analysis
—	Does not include a tool
*	The tool's connection to the semantics is incomplete or unclear (e.g., tool does not implement the full semantics)

Implementation – How does the tool detect or mitigate vulnerabilities? (Section 3.4)

Taint	Taint tracking (abstract execution)	Manual	Manual effort
Safety	Memory safety (abstract execution)	Fuzz	Fuzzing
SelfC	Self composition (abstract execution)	Flow	Data flow analysis
Cache	Cache must-hit analysis (abstract execution)	Struct	Structured compilation
+	Includes additional work or constraints to remove sequential trace (Section 3.2)		

Figure 2: Comparison of various semantics and tools. Semantics are sorted by *Level*, then alphabetically; works without semantics are ordered last. [1]Extension to other variants is discussed, but not performed. [2]Semantics includes indirect jumps and rules to update the indirect branch predictor state, but can't mispredict indirect jump targets. [3]"Weak" variants of semantics leak loaded values during non-speculative execution. [4]ELFbac mitigates Spectre-PHT without inserting fences. [5]Swivel operates on WebAssembly, which does not have fences. However, Swivel can insert fences in its assembly backend.

All of these models, including $[\![\cdot]\!]_{\text{mem}}$ and $[\![\cdot]\!]_{\text{cache}}$, are weaker than $[\![\cdot]\!]_{\text{ct}}$—they model less powerful attackers who cannot observe control flow. As a result, they miss attacks which leak via the instruction cache or which otherwise exploit timing differences in the execution of the program. They even miss some attacks that exploit the data cache: If a sensitive value influences a branch, an attacker could infer the sensitive value through the data cache based on differing (benign) memory access patterns on the two sides of the branch, even if no sensitive value influences a memory address. For instance, in the following code, even though `cond` doesn't directly influence a memory address, an attacker could infer the value of `cond` based on whether `arr[a]` is cached or not:

```
if (cond) {
  b = arr[a];
} else {
  b = 0;
}
```

Because the $[\![\cdot]\!]_{\text{mem}}$ and $[\![\cdot]\!]_{\text{cache}}$ leakage models miss these attacks, they cannot provide the strong guarantees necessary for secure cryptography or software isolation. Tools which want to provide sound verification or mitigation should choose a strong leakage model appropriate for their application domain, such as $[\![\cdot]\!]_{\text{ct}}$ or $[\![\cdot]\!]_{\text{arch}}$.

That said, weaker leakage models are still useful in certain settings: Tools which are interested in only a certain vulnerability class can use these weaker models to reduce the number of false positives in their analysis or reduce the complexity of their mitigation. Even though these models may miss some Spectre attacks—even some data cache leakage, as discussed above—some detection tools still use the $[\![\cdot]\!]_{\text{cache}}$ or $[\![\cdot]\!]_{\text{mem}}$ models to find Spectre vulnerabilities in real codebases. Using a leakage model which ignores control flow leakage may help the detection tool scale to larger codebases.

Some tools [30, 70] also provide the ability to reason about what attacks are possible with particular cache configurations—e.g., with a particular associativity, cache size, or line size. This is a valuable capability for a detection tool: It helps an attacker zero in on vulnerabilities which are more easily exploitable on a particular target machine. However, security guarantees based on this kind of analysis are not portable, as executing a program on a different machine with a different cache model invalidates the security analysis. Tools that instead want to make guarantees for all possible architectures, such as verifiers or compilers, will need more conservative leakage models—models that assume the entire memory trace (and execution trace) is always leaked.

Open problems: Leakage models for weak-memory-style semantics. We have described leakage models only in terms of observations of execution traces; this is a natural way to define leakage for *operational semantics*, where execution is modeled simply as a set of program traces. However, the weak-memory-style speculative semantics of [19, 22] have a more structured view of program execution, for instance using pomsets [26]. Both of these semantics define leakages in a way equivalent to the $[\![\cdot]\!]_{\text{mem}}$ leakage model; it remains an open problem to explore how to define $[\![\cdot]\!]_{\text{ct}}$ or $[\![\cdot]\!]_{\text{arch}}$ leakage in this more structured execution model—in particular, what it means for such a semantics to allow an attacker to observe control-flow leakage.

Open problems: Leakage models for language-based isolation. As with most work on Spectre foundations, we focus on cryptography and software-based isolation. Spectre, though, can be used to break most other software abstractions as well—from module systems [31] and object capabilities [46] to language-based isolation techniques like information flow control [58]. How do we adopt these abstractions in the presence of speculative execution? What formal security property should we prove? And what leakage model should be used?

3.2 Non-interference and policies

After the leakage model, we must determine what *secrecy policy* we consider for our attacker model—i.e., which values can and cannot be leaked. Domains such as cryptography and isolation already have defined policies for sequential security properties. For cryptography, memory that contains secret data (e.g., encryption keys) is considered sensitive. Isolation simply declares that all memory outside the program's assigned sandbox region should not be leaked.

The straightforward extension of sequential non-interference to speculative execution is to simply enforce the same leakage model with the same policy (e.g., $NI(\pi, [\![\cdot]\!]^{\text{pht}}_{\text{ct}})$)—no secrets should be leaked whether in normal or speculative execution. We refer to this straightforward extension as a *direct* non-interference property, or direct NI.

Alternatively, we may instead want to assert that the speculative trace of a program has no *new* sensitive leaks as compared to its sequential trace. This is a useful property for compilers and mitigation tools that may not know the secrecy policy of an input program, but want to ensure the resulting program does not leak any additional information. We term this a *relative* non-interference property, or relative NI; a program that satisfies relative NI is no less secure than its sequential execution.

Definition 2 (Relative non-interference)**.** Program p satisfies *relative non-interference* from contract $[\![\cdot]\!]^{\text{seq}}_{\text{a}}$ to $[\![\cdot]\!]^{\beta}_{\text{b}}$ and with policy π if: For all pairs of low-equivalent initial states σ and σ', if executing p under $[\![\cdot]\!]^{\text{seq}}_{\text{a}}$ produces equal traces, then executing p under $[\![\cdot]\!]^{\beta}_{\text{b}}$ produces equal traces. That is, $p \vdash NI(\pi, [\![\cdot]\!]^{\text{seq}}_{\text{a}} \Rightarrow [\![\cdot]\!]^{\beta}_{\text{b}})$ is defined as

$$\forall \sigma, \sigma' : \sigma \simeq_\pi \sigma' \wedge [\![p]\!]^{\text{seq}}_{\text{a}}(\sigma) = [\![p]\!]^{\text{seq}}_{\text{a}}(\sigma') \implies [\![p]\!]^{\beta}_{\text{b}}(\sigma) = [\![p]\!]^{\beta}_{\text{b}}(\sigma').$$

As before, we may elide π for brevity.

Interestingly, any relative non-interference property $NI(\pi, [\![\cdot]\!]^{\text{seq}}_{\text{a}} \Rightarrow [\![\cdot]\!]^{\beta}_{\text{b}})$ for a program p can be expressed equivalently as a direct property $NI(\pi', [\![\cdot]\!]^{\beta}_{\text{b}})$, where $\pi' = \pi \setminus \mathit{canLeak}(p, [\![\cdot]\!]^{\text{seq}}_{\text{a}})$. That is, we treat anything that could possibly leak under contract $[\![\cdot]\!]^{\text{seq}}_{\text{a}}$ as public. Relative NI is thus a weaker property than direct NI, as it implicitly declassifies anything that might leak during sequential execution.

However, relative NI is a stronger property than a conventional implication. For example, the property $NI([\![\cdot]\!]^{\text{seq}}_{\text{ct}}) \Rightarrow NI([\![\cdot]\!]^{\text{pht}}_{\text{ct}})$ makes no guarantees at all about a program that is not sequentially constant-time. Conversely, the relative NI property $NI([\![\cdot]\!]^{\text{seq}}_{\text{ct}} \Rightarrow [\![\cdot]\!]^{\text{pht}}_{\text{ct}})$

guarantees that even if a program is not sequentially constant-time, the sensitive information an attacker can learn during the program's speculative execution is limited to what it already might leak sequentially.

In Figure 3, we classify speculative security properties of different works by which direct or relative NI properties they verify or enforce. We find that tools focused on verifying cryptography or memory isolation verify direct NI properties, whereas frameworks concerned with compilation or inserting Spectre mitigations for general programs tend towards relative NI.

Verifying programs. Direct NI unconditionally guarantees that sensitive data is not leaked, whether executing sequentially or speculatively. This makes it ideal for domains that already have clear policies about what data is sensitive, such as cryptography (e.g., secret keys) or software isolation (e.g., memory outside the sandbox). Indeed, tools that target cryptographic applications ([8, 16, 21, 67]) all verify that programs satisfy the direct *speculative constant-time* (SCT) property.

Additionally, we find that current tools which verify relative NI (e.g., [18, 28]) are indeed capable of verifying direct NI, but intentionally add constraints to their respective checkers to "remove" sequential leaks from their speculative traces. Although this is just as precise, it is an open problem whether tools can verify relative NI for programs without relying on a direct NI analysis.

Verifying compilers. On the other hand, compilers and mitigation tools are better suited to verify or enforce relative NI properties: The compiler guarantees that its output program contains *no new* leakages as compared to its input program. This way, developers can reason about their programs assuming a sequential model, and the compiler will mitigate any speculative effects. For instance, if a program p is already *sequentially* constant-time $NI(\llbracket \cdot \rrbracket_{\text{ct}}^{\text{seq}})$, then a compiler that enforces $NI(\llbracket \cdot \rrbracket_{\text{ct}}^{\text{seq}} \Rightarrow \llbracket \cdot \rrbracket_{\text{ct}}^{\text{pht}})$ will compile p to a program that is *speculatively* constant-time $NI(\llbracket \cdot \rrbracket_{\text{ct}}^{\text{pht}})$. Similarly, if a program is properly sandboxed under sequential execution $NI(\llbracket \cdot \rrbracket_{\text{arch}}^{\text{seq}})$, and is compiled with a compiler that introduces no new *arch* leakage, the resulting program will remain sandboxed even speculatively. Indeed, these propositions are proven in [29].

Patrignani and Guarnieri [55] specifically explore compiler preservation, defining a property they term *Robust SNI Preservation* (RSNIP): A compiler in their framework satisfies RSNIP if, for all programs p in their (sequential) source language semantics that satisfy RSNI (see Figure 3), the translation of p to the (speculative) hardware semantics also satisfies RSNI. However, since all programs in their source language trivially satisfy RSNI, and since their source and target language differ only in the speculative execution (or lack thereof), a compiler that satisfies RSNIP is a therefore a compiler that enforces RSNI on all programs it compiles.

3.3 Execution models

To reason about Spectre attacks, a semantics must be able to reason about the leakage of sensitive data in a speculative *execution model*. A speculative execution model is what differentiates a speculative semantics from standard sequential analysis, and determines what speculation the abstract processor can perform. For developers, choosing a proper execution model is a tradeoff: On the one hand, the choice of behaviors their model allows—i.e., which microarchitectural predictors they include—determines which Spectre variants their tools can capture. On the other hand, considering additional kinds of mispredictions inevitably makes their analysis more complex.

Spectre variants and predictors. Most semantics and tools in Figure 2 only consider the conditional branch predictor, and thus only Spectre-PHT attacks. (Mis)predictions from the conditional branch predictor are constrained—there are only two possible choices for every decision—so the analysis remains fairly tractable. Jasmin [8], Binsec/Haunted [21], and Pitchfork [16] all additionally model *store-to-load* (STL) predictions, where a processor forwards data to a memory load from a prior store to the same address. If there are multiple pending stores to that address, the processor may choose the wrong store to forward the data—this is the root of a Spectre-STL attack. STL predictions are less constrained than predictions from the conditional branch predictor: In the absence of additional constraints, they allow for a load to draw data from any prior store to the same address.

Other prediction mechanisms (e.g., the BTB or RSB) are significantly more complex. An RSB misprediction can cause execution to jump to a prior (and possibly stale) return site [42, 47], while a BTB misprediction can send execution nearly *anywhere* in the program [41].[3] Capturing these behaviors in a semantics is possible, but the resulting analysis is not practical or useful; in practice, developers need to make tradeoffs. For example, the semantics in [16] can express all of the aforementioned variants of Spectre, but their analysis tool Pitchfork only detects PHT- and STL-based vulnerabilities.

The InSpectre semantics [27] goes even further—it allows the processor to mispredict arbitrary values, even the values of constants. InSpectre also allows more out-of-order behavior than most other semantics (see Section 3.6)—in particular, it allows the processor to commit writes to memory out-of-order. As a result, InSpectre is very expressive: It is capable of describing a wide variety of Spectre variants both known and unrealized. But, with such an expressive semantics, InSpectre cannot feasibly be used to verify programs; instead, the authors pose InSpectre as a framework for reasoning about and analyzing microarchitectural features themselves.

Speculation windows. As shown in Figure 2, several semantics and tools limit speculative execution by way of a *speculation window*. This models how hardware has finite resources for speculation, and can only speculate through a certain number of instructions or branches at a time.

Explicitly modeling a speculation window serves two purposes for detection tools. One, it reduces false positives: a mispredicted branch will not lead to a speculative leak thousands of instructions later. And two, it bounds the complexity of the semantics and thus the analysis. Since the abstract processor can only speculate up to a certain depth, an analysis tool need only consider the latest window of instructions under speculative execution. Some semantics refine this idea even further: Binsec/Haunted [21], for example, uses different speculation windows for load-store forwarding than it uses for branch speculation.

[3] Including, on x86-family processors, into the *middle* of an instruction [10].

Property or tool name	Non-interference prop.	Precision
Mcilroy et al. [48]	$\approx NI(\llbracket \cdot \rrbracket^{\text{pht}}_{\text{ct}})$	hyper
oo7 [71] $\Phi_{spectre}$	$\approx NI(\llbracket \cdot \rrbracket^{\text{pht}}_{\text{mem}})$	taint[1]
oo7 [71] $\Phi^{weak}_{spectre}, \Phi^{v1.1}_{spectre}$	$\approx NI(\llbracket \cdot \rrbracket^{\text{pht}}_{\text{arch}})$	taint[1]
Cache analysis [30, 73]	$NI(\llbracket \cdot \rrbracket^{\text{pht}}_{\text{cache}})$	hyper
Cache analysis [70]	$NI(\llbracket \cdot \rrbracket^{\text{pht}}_{\text{cache}})$	taint
Weak memory modeling [19, 22]	$NI(\llbracket \cdot \rrbracket^{\text{pht}}_{\text{mem}})$	hyper
Speculative constant-time (SCT)[2] [67]	$NI(\llbracket \cdot \rrbracket^{\text{pht}}_{\text{ct}})$	taint
Speculative constant-time (SCT)[2] [8, 21]	$NI(\llbracket \cdot \rrbracket^{\text{pht-stl}}_{\text{ct}})$	hyper
Speculative constant-time (SCT)[2] [16]	$NI(\llbracket \cdot \rrbracket^{\text{pbrs}}_{\text{ct}})$[3]	hyper, taint
Speculative non-interference (SNI) [28, 29]	$NI(\llbracket \cdot \rrbracket^{\text{seq}}_{\text{ct}} \Rightarrow \llbracket \cdot \rrbracket^{\text{pht}}_{-})$[4]	hyper
Robust speculative non-interference (RSNI) [55]	$NI(\llbracket \cdot \rrbracket^{\text{seq}}_{\text{ct}} \Rightarrow \llbracket \cdot \rrbracket^{\text{pht}}_{\text{ct}})$	hyper
Robust speculative safety (RSS) [55]	$NI(\llbracket \cdot \rrbracket^{\text{seq}}_{\text{ct}} \Rightarrow \llbracket \cdot \rrbracket^{\text{pht}}_{\text{ct}})$	taint
Conditional noninterference [27]	$NI(\llbracket \cdot \rrbracket^{\text{seq}}_{\text{ct}} \Rightarrow \llbracket \cdot \rrbracket^{\text{pbrs}}_{\text{ct}})$	hyper
Weak speculative non-interference (wSNI) [29]	$NI(\llbracket \cdot \rrbracket^{\text{seq}}_{\text{arch}} \Rightarrow \llbracket \cdot \rrbracket^{\text{pht}}_{-})$[4,5]	hyper
Weak robust speculative non-interference (RSNI$^-$) [55]	$NI(\llbracket \cdot \rrbracket^{\text{seq}}_{\text{arch}} \Rightarrow \llbracket \cdot \rrbracket^{\text{pht}}_{\text{ct}})$	hyper
Trace property-dependent observational determinism (TPOD) [18]	$NI(\llbracket \cdot \rrbracket^{\text{seq}}_{\text{arch}} \Rightarrow \llbracket \cdot \rrbracket^{\text{pht}}_{\text{ct}})$	hyper
Weak robust speculative safety (RSS$^-$) [55]	$NI(\llbracket \cdot \rrbracket^{\text{seq}}_{\text{arch}} \Rightarrow \llbracket \cdot \rrbracket^{\text{pht}}_{\text{ct}})$	taint

Execution models (Section 3.3)		Precision of the defined security property	
$\llbracket \cdot \rrbracket^{\text{seq}}$	Sequential execution	hyper	Non-interference hyperproperty, requires two low-equivalent executions
$\llbracket \cdot \rrbracket^{\text{pht}}$	Captures Spectre-PHT	taint	Sound approximation using taint tracking, requires only one execution
$\llbracket \cdot \rrbracket^{\text{pht-stl}}$	Captures Spectre-PHT/-STL		
$\llbracket \cdot \rrbracket^{\text{pbrs}}$	Captures Spectre-PHT/-BTB/-RSB/-STL		

Figure 3: Speculative security properties in prior works and their equivalent non-interference statements. We write $\approx NI(\cdots)$ for unsound approximations of non-interference properties. [1][71] tracks taint of *attacker influence* rather than value sensitivity. [2]These works all derive their property from the definition given in [16] and share the same property name despite differences in execution mode. [3]The analysis tool of [16], Pitchfork, only verifies the weaker property $NI(\llbracket \cdot \rrbracket^{\text{pht-stl}}_{\text{ct}})$. [4]The definitions of SNI and wSNI are parameterized over the target leakage model. [5]The definition of wSNI in [29] does not require that the initial states be low-equivalent.

Speculation windows are also valuable for mitigation tools: although tools like Blade [67] and Jasmin [8] are able to prove security without reasoning about speculation windows, modeling a speculation window would reduce the number of fences (or other mitigations) these tools need to insert, improving the performance of the compiled code.

Eliminating variants. Instead of modeling all speculative behaviors, compilers and mitigation tools can use clever tricks to sidestep particularly problematic Spectre variants. For example, even though Jasmin [8] does not model the RSB, Jasmin programs do not suffer from Spectre-RSB attacks: The Jasmin compiler inlines all functions, so there are no returns to mispredict. Mitigation tools can also disable certain classes of speculation with hardware flags [33]. After eliminating complex or otherwise troublesome speculative behavior, a tool only needs to consider those that remain.

In-place vs. out-of-place training. Previous systematizations of Spectre attacks [14] differentiate between attacks with *in-place* and *out-of-place* training. In-place attacks are generally simpler to perform, as they rely on repeatedly executing the victim code itself in order to train a microarchitectural predictor. Out-of-place attacks are more powerful, as they allow an attacker to perform the training step on a branch within the attacker's own code rather than the victim code.

Most of the semantics and tools in Figure 2 make no distinction between in-place and out-of-place attacks, as they ignore the mechanics of training and consider all predictions to be potentially malicious. A notable exception is oo7 [71], which explicitly tracks *attacker influence*. Specifically, oo7 only considers mispredictions for conditional branches which can be influenced by attacker input. Thus, oo7 effectively models only in-place training. Unfortunately, as a result, oo7 misses Spectre vulnerabilities in real code, as demonstrated by Wang et al. [70].

3.4 Nondeterminism

Speculative execution is inherently *nondeterministic*: Any given branch in a program may proceed either correctly or incorrectly, regardless of the actual condition value. More generally, predictors such as the BTB can send execution to an entirely indeterminate

location. The semantics in Figure 2 all allow these nondeterministic choices to be actively adversarial—for instance, given by attacker-specified directives [16, 67], or, equivalently, by consulting an abstract oracle [18, 28, 29, 48]. These semantics all (conservatively) assume that the attacker has full control of microarchitectural prediction and scheduling; we explore the different techniques they use to verify or enforce security in the face of adversarial nondeterminism.

Exploring nondeterminism. Several Spectre analysis tools are built on some form of abstract execution: They simulate speculative execution of the program by tracking ranges or properties of different values. By checking these properties throughout the program, they determine if sensitive data can be leaked. Standard tools for (non-speculative) abstract execution are designed only to consider concrete execution paths; they must be adapted to handle the many possible nondeterministic execution paths from speculation. SpecuSym [30], KLEESpectre [70], and AISE [73] handle this nondeterminism by following an *always-mispredict* strategy. When they encounter a conditional branch, they first explore the execution path which mispredicts this branch, up to a given speculation depth. Then, when they exhaust this path, they return to the correct branch. This technique of course only handles the conditional branch predictor; i.e., Spectre-PHT attacks. Pitchfork [16] and Binsec/Haunted [21] adapt the always-mispredict strategy to additionally account for out-of-order execution and Spectre-STL. Although it may not be immediately clear that these always-mispredict strategies are sufficient to prove security, especially when the attacker can make any number of antagonistic prediction choices, these strategies do indeed form a sound analysis [16, 21, 28].

Unfortunately, simulating execution only works for semantics where the nondeterminism is relatively constrained: Conditional branches are a simple boolean choice, and store-to-load predictions are limited to prior memory operations within the speculation window. If we pursue other Spectre variants, we will quickly become overwhelmed—again, an unconstrained BTB can land almost anywhere in a program. The always-mispredict strategy here is nonsensical at best; abstract execution is thus necessarily limited in what it can soundly explore.

Abstracting out nondeterminism. Mitigation tools have more flexibility dealing with nondeterminism: Tools like Blade [67] and oo7 [71] apply dataflow analysis to determine which values may be leaked along *any* path, instead of reasoning about each path individually. Then, these tools insert speculation barriers to preemptively block potential leaks of sensitive data. This style of analysis comes at the cost of some precision: Blade, for example, conservatively treats *all* memory accesses as if they may speculatively load sensitive values, as its analysis cannot reason about the contents of memory. However, Blade (and mitigation tools in general) can afford to be less precise than verification and detection tools—these must maintain higher precision to avoid floods of false positives.

Restricting nondeterminism. Compilers such as Swivel [53], Venkman [63], and ELFbac [38] restructure programs entirely, imposing their own restricted set of speculative behavior at the software layer. ELFbac allocates sensitive data in separate memory regions and uses page permission bits to disallow untrusted code from accessing these regions of memory—regardless of how a program may misspeculate, it will not be able to read (and thus leak) sensitive data. Swivel and Venkman compile code into carefully aligned blocks so that indirect jumps always land at the tops of protected code blocks, even speculatively; Swivel accomplishes this by clearing the BTB state after untrusted execution, while Venkman proposes to recompile all programs on the system. Developers that use these compilers can then reason about their programs much more simply, as the set of speculative behaviors is restricted enough to make the analysis tractable. Of the techniques discussed in this section, this line of work seems the most promising: It produces mitigation tools with strong security guarantees, without relying on an abundance of speculation barriers (as often results from dataflow analysis) or resorting to heavyweight simulation (such as symbolic execution).

3.5 Higher-level abstractions

Spectre attacks—and speculative execution—fundamentally break our intuitive assumptions about how programs should execute. Higher-level guarantees about programs no longer apply: Type systems or module systems are meaningless when even basic control flow can go awry. In order to rebuild higher-level security guarantees, we first need to repair our model of how programs execute, starting from low-level semantics. Once these foundations are firmly in place, only then can we rebuild higher-level abstractions.

Semantics for assembly or IRs. The majority of formal semantics in Figure 2 operate on abstract assembly-like languages, with commands that map to simple architectural instructions. Semantics at this level implement control flow directly in terms of jumps to *program points*—usually indices into memory or an array of program instructions—and treat memory as largely unstructured. Since these low-level semantics closely correspond to the behavior of real hardware, they capture speculative behaviors in a straightforward manner, and provide a foundational model for higher-level reasoning. Similarly, many concrete analysis tools for constant-time or Spectre operate directly on binaries or compiler intermediate representations (IRs) [16, 20, 21, 28, 70]. These tools operate at this lowest level so that their analysis will be valid for the program unaltered—compiler optimizations for higher-level languages can end up transforming programs in insecure ways [9, 20, 21]. As a result however, these tools necessarily lose access to higher-level information such as control flow structure or how variables are mapped in memory.

Semantics for structured languages. The semantics in [8], [55], and [67] build on top of these lower-level ideas to describe what we term "medium-level" languages—those with structured control flow and memory, e.g., explicit loops and arrays. For these medium-level semantics, it is less straightforward to express speculative behavior: For instance, instead of modeling speculation directly, Vassena et al. [67] first translate programs in their source language to lower-level commands, then apply speculative execution at that lower level.

In exchange, the structure in a medium-level semantics lends itself well to program analysis. For example, Vassena et al. are able to use a simple type system to prove security properties about a program. Barthe et al. [8] also take advantage of structured semantics: They prove that if a sequentially constant-time program is *speculatively (memory) safe*—i.e., all memory operations are in-bounds

array accesses—then the program is also speculatively constant-time. Since their source semantics can only access memory through array operations, they can statically verify whether a program is speculatively safe (and thus speculatively secure). An interesting question for future work is whether their concept of speculative (memory) safety can combine with other sequential security properties to give corresponding speculative guarantees, such as for sandboxing, information flow, or rich type systems.

Weak-memory-style semantics. Colvin and Winter [19] and Disselkoen et al. [22] both present a further abstracted semantics in the style of weak memory models. These semantics represent a fundamentally different approach: Rather than creating operational models of speculative hardware, these authors lift the concept of speculative execution directly to a higher level and reason about it there.

These works provide interesting insights about the relation between Spectre attacks and the weak memory models which characterize modern hardware. They also open the door to adapting analysis and design techniques from that community to defend against Spectre attacks in software. However, as these models are abstracted away from microarchitectural details, they are only suited for analyzing particular Spectre variants—both [19] and [22] focus only on Spectre-PHT—and are difficult to adapt to other attacks. In addition, it remains an open problem to translate a semantics of this style into a concrete analysis tool: Neither [19] nor [22] present a tool which can automatically perform a security analysis of a target program.[4] That said, this high-level approach to speculative semantics is certainly underexplored compared to the larger body of work on operational semantics, and is worthy of further investigation.

Compiler mitigations. With adequate foundations in place, one avenue to regaining higher-level abstractions is to modify compilers of higher-level languages to produce speculatively secure low-level programs. Many compilers already include options to conservatively insert speculation barriers or hardening into programs, which (when done properly) provides strong security guarantees. Although some such hardening passes have been verified [55], they are overly conservative and incur a significant performance cost. Other compiler mitigations been shown unsound [54]—or worse, even introduce new Spectre vulnerabilities [21]—further supporting our position that these techniques must be grounded in a formal semantics.

Open problems: Formalization of new compilation techniques. Swivel [53], Venkman [63], and ELFbac [38] show how the structure of code itself can provide security guarantees at a reduced performance cost. For instance, [53, 63] demonstrate that organizing instructions into *bundles* or *linear blocks* respectively can restrict BTB and RSB mispredictions, making these Spectre variants tractable to analyze and mitigate. However, none of these compiler-based approaches are yet grounded in a formal semantics. Formalizing these systems would increase our confidence in the strong guarantees they claim to provide.

Open problems: New languages. Another promising approach is to design new languages which are inherently safe from Spectre attacks. Prior work has produced secure languages like FaCT [17], which is (sequentially) constant-time by construction. An extension of FaCT, or a new language built on its ideas, could prevent Spectre attacks as well. Vassena et al. [67] have already taken a first step in this direction: They construct a simple while-language which is guaranteed safe from Spectre-PHT attacks when compiled with their fence insertion algorithm. It would be valuable to extend this further, both to more realistic (higher-level) languages, and to more Spectre variants. The key question is whether dedicated language support can provide a path to secure code that outperforms the de-facto approach: Compiling standard C code with Spectre mitigations.

3.6 Expressivity and microarchitectural features

One theme of this paper has been that a good (practical) semantics needs to have an appropriate amount of *expressivity*: On one hand, we want a semantics which is *expressive*—able to model a wide range of possible behaviors (e.g., Spectre variants). This allows us to model powerful attackers. On the other hand, a semantics which is too expressive—allows too many possible behaviors—makes many analyses intractable. One fundamental purpose of semantics is to provide a reasonable abstraction (simplification) of hardware to make analysis easier; a semantics which is too expressive simply punts this problem to the analysis writer. Thus, choosing how much expressivity to include in a semantics represents an interesting tradeoff.

By far the most important choice for the expressivity of a semantics is which misprediction behaviors to allow—i.e., which Spectre variants to reason about. We discussed these tradeoffs in Section 3.3. But beyond speculative execution itself, there are many other microarchitectural features which could be relevant for a security analysis, and which have been—or could be—modeled in a speculative semantics. These features also affect the expressivity of the semantics, which means that choosing whether to include them results in similar tradeoffs.

Out-of-order execution. Many speculative semantics simulate a processor feature called *out-of-order execution*: they allow instructions to be executed in any order, as long as those instructions' dependencies (operands) are ready. Out-of-order execution is mostly orthogonal to speculative execution; in fact, out-of-order execution is not required to model Spectre-PHT, -BTB, or -RSB—speculative execution alone is sufficient. However, out-of-order execution is included in most modern processors, and for that reason,[5] many speculative semantics also model out-of-order execution. Modeling out-of-order execution may provide an easier or more elegant way to express a variety of Spectre attacks, as opposed to modeling speculative execution alone. Further, as a result of including out-of-order execution in their respective semantics, [22] and [27] propose to abuse out-of-order execution to conduct (at least theoretical) novel side-channel attacks.[6]

That said, although modeling out-of-order execution might make the semantics simpler, the additional expressivity definitely makes

[4] Colvin and Winter do present a tool, but it is only used to mechanically explore manually translated programs.

[5] Or, perhaps because out-of-order execution is often discussed alongside, or even confused with, speculative execution

[6] Disselkoen et al. [22] propose to abuse compile-time instruction reordering, which is different from microarchitecture-level out-of-order execution, but related.

the resulting analysis more complex. Fully modeling out-of-order execution leads to an explosion in the number of possible executions of a program; naively incorporating out-of-order execution into a detection or mitigation tool results in an intractable analysis. Indeed, while Guarnieri et al. [29] and Colvin and Winter [19] present analysis tools based on their respective out-of-order semantics, they only analyze very simple Spectre gadgets, not code used in real programs. Instead, for analysis tools based on out-of-order semantics to scale to real programs, developers need to use lemmas to reduce the number of possibilities the analysis needs to consider. As one example, Pitchfork [16] operates on a set of "worst-case schedules" which represent a small subset of all possible out-of-order schedules. The developers formally argue that this reduction does not affect the soundness of Pitchfork's analysis.

Caches and TLBs. Some speculative semantics and tools ([30, 48, 70, 73]) include abstract models of caches, where the cache state captures which addresses may be in the cache at a given time. One could imagine also including detailed models of TLBs. As discussed in Section 3.1, modeling caches or TLBs is probably not helpful, at least for mitigation or verification tools—not only does it make the semantics more complicated, but it potentially leads to non-portable guarantees. In particular, including a model of the cache usually leads to the $[\![\cdot]\!]_{\text{cache}}$ leakage model, rather than the $[\![\cdot]\!]_{\text{ct}}$ or $[\![\cdot]\!]_{\text{arch}}$ leakage models which provide stronger defensive guarantees. Following in the tradition of constant-time programming in the non-speculative world, it seems wiser for our analyses and mitigations to be based on microarchitecture-agnostic principles as much as possible, and not depend on details of the cache or TLB structure.

Other leakage channels. There are a variety of specific microarchitectural mechanisms which could result in leakages, beyond the ones we've been focusing on in this paper. For instance, in the presence of multithreading, port contention in the processor's execution units can reveal sensitive information [11]; and many processor instructions, e.g., floating-point or SIMD instructions, can reveal information about their operands through timing side-channels [5]. Most existing semantics do not model these effects. However, the commonly-used $[\![\cdot]\!]_{\text{ct}}$ and $[\![\cdot]\!]_{\text{arch}}$ leakage models are strong enough to already capture leakages from most of these sources: for instance, port contention can only reveal sensitive data if the sensitive data influenced which instructions are being executed—and the $[\![\cdot]\!]_{\text{ct}}$ leakage model would have already considered the sensitive data leaked once it influenced control flow. For variable-time instructions, most works' definitions of $[\![\cdot]\!]_{\text{ct}}$ don't capture this leakage, but extending those definitions to cover it is straightforward [3]. In both of these examples, the $[\![\cdot]\!]_{\text{arch}}$ leakage model would capture all of the leaks, because it (even more conservatively) would already consider the sensitive data leaked once it reached a register, long before it could influence control-flow or be used in a variable-time instruction. Although modeling any of these effects more precisely could increase the precision with which an analysis detects potential vulnerabilities, the tradeoff in analysis complexity is probably not worth it, and for mitigation and verification tools, the $[\![\cdot]\!]_{\text{ct}}$ and $[\![\cdot]\!]_{\text{arch}}$ leakage models provide stronger and more generalizable guarantees.

In a similar vein, most semantics and tools do not explicitly model parallelism or concurrency: They reason only about single-threaded programs and processors. Instead, they abstract away these details by giving attackers broad powers in their models—e.g., complete power over all microarchitectural predictions, and the capability to observe the full cache state after every execution step. The notable exceptions are the weak-memory-style semantics presented by Colvin and Winter [19] and Disselkoen et al. [22]—multiple threads are an inherent feature for this style of semantics. These semantics may be a promising vehicle for further exploring the interaction between speculation and concurrency. For other semantics, adding detailed models of multithreading is probably not worth the increased analysis complexity.

Open problems: Process isolation. In practice, a common response to Spectre attacks has been to move all secret data into a separate process—e.g., Chrome isolates different *sites* in separate processes [56]. This shifts the burden to OS engineers from application and runtime system engineers. Developing Spectre foundations to model the process abstraction would elucidate the security guarantees of such systems. This would be especially useful since there is evidence showing that the process boundary does not keep an attacker from performing out-of-place training of the conditional branch predictor, or from leaking secrets via the cache state [14].

4 RELATED WORK

There has been a lot of interest in Spectre and other transient execution attacks, both in industry and in academia. We discuss other systematization papers that address Spectre attacks and defenses, and we briefly survey related work which otherwise falls outside the scope of this paper.

4.1 Systematization of Spectre attacks and defenses

Canella et al. [14] present a comprehensive systematization and analysis of Spectre and Meltdown attacks and defenses. They first classify transient execution attacks by whether they are a result of misprediction (Spectre) or an execution fault (Meltdown); then they further classify the attacks by their root microarchitectural cause, yielding the nomenclature we use in this paper (e.g., *Spectre-PHT* is named for the pattern history table). They then categorize previously known Spectre attacks, revealing several new variants and exploitation techniques for each. Canella et al. also propose a sequence of "phases" for a successful Spectre or Meltdown attack, and group published defenses by the phase they target. A followup survey by Canella et al. [13] expands on the idea of attack phases, categorizing both hardware and software Spectre defenses according to which attack phase they prevent: preparation, misspeculation, data access, data encoding, leakage, or decoding. In contrast, our systematization focuses on the formal semantics behind Spectre analysis and mitigation tools rather than the specifics of attack variants or types of defenses.

4.2 Hardware-based Spectre defenses

In this paper, we focus only on software-based techniques for existing hardware. The research community has also proposed several hardware-based Spectre defenses based on cache partitioning [40],

cleaning up the cache state after misprediction [59], or making the cache invisible to speculation by incorporating some separate internal state [2, 39, 75]. Unfortunately, attackers can still use side channels other than the cache to exploit speculative execution [11, 62]. NDA [72] and Speculative Taint Tracking (STT) [76] block additional speculative covert channels by analyzing and classifying instructions that can leak information.

Fadiheh et al. [23] define a property for hardware execution that they term UPEC: A hardware that satisfies UPEC will not leak speculatively anything more than it would leak sequentially. I.e., UPEC is equivalent to the relative non-interference property $NI(\pi, [\![\cdot]\!]^{\text{seq}}_{\text{arch}} \Rightarrow [\![\cdot]\!]^{\text{pht}}_{\text{arch}})$.

4.3 Software-hardware co-design

Although hardware-only approaches are promising for future designs, they require significant modifications and introduce non-negligible performance overhead for all workloads. Several works instead propose a software-hardware co-design approach. Taram et al. [64] propose context-sensitive fencing, making various speculative barriers available to the software. Li et al. [44] propose memory instructions with a conditional speculation flag. Context [60] and SpectreGuard [25] allow software to mark secrets in memory. This information is propagated through the microarchitecture to block speculative access to the marked regions. SpecCFI [43] suggests a hardware extension similar to Intel CET [37] that provides target label instructions with speculative guarantees. Finally, several recent proposals allow partitioning branch predictors based on context provided by the software [68, 78]. As these approaches require both software and hardware changes, we will need a formal semantics to apply them correctly. Adapting Spectre semantics to software-hardware co-design would be valuable future work.

4.4 Other transient execution attacks

We focus exclusively on Spectre, as other transient execution attacks are probably better addressed in hardware. For completeness, we briefly discuss these other attacks.

Meltdown variants. The Meltdown attack [45] bypasses implicit memory permission checks within the CPU during transient execution. Unlike Spectre, Meltdown does not rely on executing instructions in the victim domain, so it cannot be mitigated purely by changes to the victim's code. Foreshadow [65] and microarchitectural data sampling (MDS) [12, 34] demonstrate that transient faults and microcode assists can still leak data from other security domains, even on CPUs that are resistant to Meltdown. Researchers have extensively evaluated these Meltdown-style attacks leading to new vulnerabilities [50, 51, 61], but most recent Intel CPUs have hardware-level mitigations for all these vulnerabilities in the form of microcode patches or proprietary hardware fixes [36].

Load value injection. Load value injection (LVI) [66] exploits the same root cause as Meltdown, Foreshadow, and MDS. But LVI reverses these attacks: The attacker induces the transient fault into the victim domain instead of crafting arbitrary gadgets in their own code space. This inverse effect is subject to an exploitation technique similar to Spectre-BTB for transiently hijacking control flow. Although there are software-based mitigations proposed against LVI [35, 66], Intel only suggests applying them to legacy enclave software. Like Meltdown, LVI does not need software-based mitigation on recent Intel CPUs, and our systematization does not apply.

5 CONCLUSION

Spectre attacks break the abstractions afforded to us by conventional execution models, fundamentally changing how we must reason about security. We systematize the community's work towards rebuilding foundations for formal analysis atop the loose earth of speculative execution, evaluating current efforts in a shared formal framework and pointing out open areas for future work in this field.

We find that, as with previous work in the sequential domain, solid foundations for speculative analyses require proper choices for semantics and attacker models. Most importantly, developers must consider leakage models no weaker than $[\![\cdot]\!]_{\text{arch}}$ or $[\![\cdot]\!]_{\text{ct}}$. Weaker models—those that only capture leaks via memory or the data cache—lead to weaker security guarantees with no clear benefit. Next, though many frameworks focus on Spectre-PHT, sound tools must consider all Spectre variants. Although this can increase the complexity of analysis, developers can combine analyses with structured compilation techniques—e.g., to restrict or remove entire categories of Spectre attacks by construction. Finally, we recommend *against* modeling unnecessary (micro)architectural details in favor of the simpler $[\![\cdot]\!]_{\text{arch}}$ and $[\![\cdot]\!]_{\text{ct}}$ models; details like cache structures, port contention, or hyperthreading introduce complexity and give up on portability.

When properly rooted in formal guarantees, software Spectre defenses provide a firm foundation on which to rebuild secure systems. We intend this systematization to serve as a reference and guide for those seeking to build atop formal frameworks and to develop sound Spectre defenses with strong, precise security guarantees.

ACKNOWLEDGMENTS

We thank Matthew Kolosick for helping us understand some of the formal systems and in organizing the paper. This work was supported in part by gifts from Cisco; by the NSF under Grant Number CNS-1514435 and CCF-1918573; and, by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Work by Gilles Barthe was supported by the Office of Naval Research (ONR) under project N00014-15-1-2750.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *TISSEC* (2009).
[2] Sam Ainsworth and Timothy M Jones. 2019. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. *arXiv:1911.08384* (2019).
[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying constant-time implementations. In *USENIX SEC*.
[4] AMD. 2020. SECURITY ANALYSIS OF AMD PREDICTIVE STORE FORWARDING. https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf.
[5] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On subnormal floating point and abnormal timing. In *IEEE S&P*.
[6] ARM. 2020. Straight-line Speculation. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/downloads/straight-

line-speculation.
[7] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level non-interference for constant-time cryptography. In *CCS*.
[8] Gilles Barthe, Sunjay Cauligi, Benjamin Gregoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 2021. High-Assurance Cryptography in the Spectre Era. In *IEEE S&P*.
[9] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In *CSF*.
[10] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. 2020. SpecROP: Speculative Exploitation of ROP Chains. In *RAID*.
[11] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: exploiting speculative execution through port contention. In *CCS*.
[12] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*.
[13] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N Khasawneh. 2020. Evolution of defenses against transient-execution attacks. In *Great Lakes Symposium on VLSI*.
[14] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX SEC*.
[15] Chandler Carruth. 2018. RFC: Speculative Load Hardening (a Spectre variant #1 mitigation). https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html.
[16] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new Spectre era. In *PLDI*.
[17] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: a DSL for timing-sensitive computation. In *PLDI*.
[18] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. In *CSF*. https://doi.org/10.1109/CSF.2019.00027
[19] Robert J Colvin and Kirsten Winter. 2019. An abstract semantics of speculative execution for reasoning about security vulnerabilities. In *FM*.
[20] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *IEEE S&P*.
[21] Lesly-Ann Daniel, Sebastian Bardin, and Tamara Rezk. 2021. Hunting the Haunter — Efficient Relational Symbolic Execution for Spectre with Haunted RelSE. In *NDSS*.
[22] Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. 2019. The code that never ran: Modeling attacks on speculative evaluation. In *IEEE S&P*.
[23] Mohammad Rahmani Fadiheh, Johannes Müller, Raik Brinkmann, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2020. A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors. In *DAC*. https://doi.org/10.1109/DAC18072.2020.9218572
[24] Matt Fleming. 2017. A thorough introduction to eBPF. *Linux Weekly News* (2017).
[25] Jacob Fustos, Farzad Farshchi, and Heechul Yun. 2019. SpectreGuard: An efficient data-centric defense mechanism against Spectre attacks. In *DAC*.
[26] Jay L Gischer. 1988. The equational theory of pomsets. *Theoretical Computer Science* (1988).
[27] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *CCS*.
[28] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. 2020. SPECTECTOR: Principled detection of speculative information flows. In *IEEE S&P*.
[29] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *IEEE S&P*.
[30] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. 2020. SpecuSym: Speculative symbolic execution for cache timing leak detection. In *ICSE*.
[31] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and Jf Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *PLDI*.
[32] Jann Horn. 2018. Speculative execution, variant 4: speculative store bypass.
[33] Intel. 2018. Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115. https://software.intel.com/security-software-guidance/software-guidance/speculative-store-bypass.
[34] Intel. 2019. Deep Dive: Intel Analysis of Microarchitectural Data Sampling.
[35] Intel. 2020. An Optimized Mitigation Approach for Load Value Injection. https://software.intel.com/security-software-guidance/best-practices/optimized-mitigation-approach-load-value-injection.
[36] Intel. 2020. Side Channel Mitigation by Product CPU Model. https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model.
[37] Intel 2021. Intel 64 and IA-32 Architectures Software Developer's Manual.
[38] Ira Ray Jenkins, Prashant Anantharaman, Rebecca Shapiro, J Peter Brady, Sergey Bratus, and Sean W Smith. 2020. Ghostbusting: Mitigating Spectre with intraprocess memory isolation. In *HotSos*.
[39] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. Safespec: Banishing the Spectre of a Meltdown with leakage-free speculation. In *DAC*.
[40] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *MICRO*.
[41] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE S&P*.
[42] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*.
[43] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2020. SPECCFI: Mitigating Spectre Attacks using CFI Informed Speculation. In *IEEE S&P*.
[44] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. 2019. Conditional speculation: An effective approach to safeguard out-of-order execution against Spectre attacks. In *HPCA*.
[45] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX SEC*.
[46] Sergio Maffeis, John C Mitchell, and Ankur Taly. 2010. Object capabilities and isolation of untrusted web applications. In *IEEE S&P*.
[47] G. Maisuradze and C. Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*.
[48] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv:1902.05178* (2019).
[49] Microsoft. 2018. Spectre mitigations in MSVC. https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/.
[50] Daniel Moghimi. 2020. Data Sampling on MDS-resistant 10th Generation Intel Core (Ice Lake). *arXiv:2007.07428* (2020).
[51] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. 2020. Medusa: Microarchitectural data leakage via automated attack synthesis. In *USENIX SEC*.
[52] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: better, faster, stronger SFI for the x86. In *PLDI*.
[53] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. 2021. Swivel: Hardening WebAssembly against Spectre. In *USENIX SEC*.
[54] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2020. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *USENIX SEC*.
[55] Marco Patrignani and Marco Guarnieri. 2020. Exorcising Spectres with Secure Compilers. *arXiv:1910.08607* (2020).
[56] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site isolation: Process separation for web sites within the browser. In *USENIX SEC*.
[57] Stephen Röttger and Artur Janc. 2021. A Spectre proof-of-concept for a Spectre-proof web. https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html.
[58] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *Journal on Selected Areas in Communications* 21, 1 (2003).
[59] Gururaj Saileshwar and Moinuddin K Qureshi. 2019. CleanupSpec: An "Undo" Approach to Safe Speculation. In *MICRO*.
[60] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. ConTExT: A Generic Approach for Mitigating Spectre. In *NDSS*.
[61] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*.
[62] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*.
[63] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. 2019. Restricting Control Flow During Speculative Execution with Venkman. *arXiv:1903.10651* (2019).
[64] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ASPLOS*.
[65] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX SEC*.

[66] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE S&P*.

[67] Marco Vassena, Craig Disselkoen, Klaus V Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. In *POPL*.

[68] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. 2019. BRB: Mitigating Branch Predictor Side-Channels. In *HPCA*.

[69] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *SOSP*.

[70] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. 2020. KLEESpectre: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM TOSEM* (2020).

[71] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2019. oo7: Low-overhead Defense against Spectre attacks via Program Analysis. *IEEE Transactions on Software Engineering* (2019).

[72] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *MICRO*.

[73] Meng Wu and Chao Wang. 2019. Abstract interpretation under speculative execution. In *PLDI*.

[74] Yongzheng Wu, Sai Sathyanarayan, Roland HC Yap, and Zhenkai Liang. 2012. Codejail: Application-transparent isolation of libraries with tight program interactions. In *European Symposium on Research in Computer Security*.

[75] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. Invisispec: Making speculative execution invisible in the cache hierarchy. In *MICRO*.

[76] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *MICRO*.

[77] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. 2020. Exploring Branch Predictors for Constructing Transient Execution Trojans. In *ASPLOS*.

[78] Lutan Zhao, Peinan Li, Rui Hou, Jiazhen Li, Michael C Huang, Lixin Zhang, Xuehai Qian, and Dan Meng. 2020. A lightweight isolation mechanism for secure branch predictors. *arXiv:2005.08183* (2020).