SoK: Practical Foundations for Software Spectre Defenses

Sunjay Cauligi
UC San Diego
MPI Security & Privacy

Craig Disselkoen
UC San Diego

Daniel Moghimi
UC San Diego

Gilles Barthe
MPI Security & Privacy
IMDEA Software Institute

Deian Stefan
UC San Diego

***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 expressiveness of formal frameworks, the complexity 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.

I. Introduction

Spectre attacks have upended the foundations of computer security [44]. With Spectre, attackers can steal secrets across security boundaries—both hardware boundaries provided by the process abstraction [84], and software boundaries provided by memory safe languages and software-based fault isolation (SFI) techniques [79]. 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 can be used to secure programs in the presence of this class of attacks. But Spectre attacks—and speculative execution in general—violate our typical assumptions and abstractions and have proven particularly challenging to reason about and defend against.

Many existing defense mechanisms against Spectre are either incomplete (and thus miss possible attacks) or overly conservative (and thus slow). For example, the MSVC compiler's `/Qspectre` pass—one of the first compiler-based defenses against Spectre [55]—inserts mitigations by finding Spectre gadgets (or patterns). Since these patterns are not based on any rigorous analysis, the compiler misses similarly vulnerable code patterns [60]. As another example, Google Chrome adopted process isolation as its core defense mechanism against Spectre attacks [64]. This is also unsound: Canella et al. [13], for example, show that Spectre attacks can be performed across the process boundary. On the other side of the spectrum, inserting fences at every load or control flow point is sound but prohibitively slow [59].

Language-based security can help us achieve—or at least understand the trade-offs of giving up on—*performance* and *provable security guarantees*. Historically, the security community has turned to language-based security to solidify intricate defense techniques—from SFI enforcement on x86 [58], to information flow control enforcement [66], to eliminating side-channel attacks with constant-time programming [6]. At the core of language-based security are *program semantics*—rigorous models of program behavior which serve as the basis for *formal security policies or foundations*. These policies help us carefully and explicitly spell out our assumptions about the attacker's strength and ensure that our tools are sound with respect to this class of attackers—e.g., that Spectre vulnerability-detection or -mitigation tools find and mitigate the vulnerabilities they claim to find and mitigate.

Formal foundations are key to performance too. Without formalizations, Spectre defenses are usually either overly conservative (which leads to unnecessary and slow mitigations) or crude (and thus vulnerable). For example, speculative load hardening [14] is *safe*—it safely eliminates Spectre-PHT attacks—but is overly conservative and slow: It assumes that *all* array indexing operations must be hardened. In practice, this is not the case [35, 77]. Crude techniques like oo7 [81] are both inefficient *and* unsafe—they impose unnecessary restrictions yet also miss vulnerable code patterns. Foundations allow us to craft defenses that are minimal (e.g., they target the precise array indexes that need hardening [29, 77]) and 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 (Figure 2). 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 in turn determine which *security policies* can be verified or enforced by these tools.

While formal design decisions fundamentally impact the soundness and precision of Spectre analysis and mitigation tools, they have not been systematically explored by the security community. For example, while there are many choices for a leakage model, the constant-time [6] and sandbox isolation [29] models are the most pragmatic; leakage models that only con-

sider the data cache trade off security for no clear benefits (e.g., scalability or precision). As another example, the most practical execution models borrow (again) from work on constant-time: They are detailed enough to capture practical attacks, but abstract across different hardware—and are thus useful for both software-based verification and mitigation techniques. Models which capture microarchitectural details like cache structures make the analysis unnecessarily complicated: They do not fundamentally capture additional attacks and give up on portability.

Contributions. 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. This complements existing, excellent surveys [12, 13, 85] on the low-level details of Spectre attacks and defenses which do not consider foundations or, for example, high-level security policies. Throughout, we discuss the limitations of existing formal frameworks, the defense tools built on top of these foundations, and future directions for research. In summary, we make the following contributions:

- 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 unifying framework, which allows us to understand 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 that could result in tools with stronger security guarantees.

Scope. In this systematization, we focus on software-only defenses against Spectre attacks. We focus on *Spectre* because most other transient attacks (e.g., Meltdown [48], LVI [76], MDS [34], or Foreshadow [75]) can efficiently be addressed in the hardware, through microcode updates or new hardware designs. (This is also the reason existing software-based tools against transient execution attacks focus solely on Spectre, as we discuss in Section IV-D.) We focus on *defenses* because prior work, notably Canella et al. [13], already give an excellent overview of the types of Spectre vulnerabilities and the powerful capabilities they give attackers. And we focus on *software-only* defenses—although proposals for hardware defenses are extremely valuable, hardware design cycles (and hardware upgrade cycles) are very long. Moreover, software foundations are useful for understanding hardware and hardware-software co-designs (e.g., they directly affect execution and leakage models). Having secure software foundations allows us to defend against today's attacks on today's hardware, and tomorrow's as well.

II. 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 < arrALen) {  // mispredicted
  int x = arrA[i];  // x is oob value
  int y = arrB[x];  // leaked via address!
  // ...
```

Fig. 1. Code snippet which an attacker can exploit using Spectre. If an attacker can control `i` and cause the processor to transiently enter the branch, the attacker can load an arbitrary value from memory into `x`, which is then leaked via the following memory access.

A. Spectre vulnerabilities

Spectre [3, 5, 32, 42, 44, 45, 51, 88] is a recently discovered family of vulnerabilities stemming from *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 realizes it has mispredicted, 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 vulnerable function: An attacker can exploit branch misprediction to leak arbitrary memory via the data cache. The attacker first primes the branch to predict that the condition `i < arrALen` is true by causing the code to repeatedly run with appropriate (small) 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 (potentially secret) data into `x`; subsequently, it uses the value `x` as part of the address of a memory read operation. This encodes the value of `x` into the data cache state—depending on the value of `x`, 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 `x`.

B. Breaking cryptography with Spectre

High-assurance cryptography has long relied on *constant-time programming* [6] in order to create software which is secure from timing side-channel attacks. Constant-time programming ensures that program execution does not depend on secrets. It does this via three rules of thumb [6, 8]: control flow (e.g., conditional branches) should not depend on secrets, memory access patterns (e.g., offsets into arrays) should not be influenced by secrets, and secrets should not be used as operands to variable-latency instructions (e.g., floating-point instructions or integer division on many processors). These rules ensure that secrets remain safe from an attacker powerful

enough to perform cache attacks, exfiltrate data via branch predictor state, or snoop data via port contention [10].

In the face of Spectre, constant-time programming is not sufficient. The snippet in Figure 1 is indeed constant-time if `arrA` contains only public data (and `i` and `arrALen` are also public). Yet, a Spectre attack can still abuse this code to leak secrets from anywhere in memory.

Cache-based leaks 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 < arrALen) {
  int x = arrA[i];
  switch(x) {  // leak via branching!
    case 'A': /* ... */
    case 'B': /* ... */
    // ...
```

This code uses `x` as part of a branch condition (in a **`switch`** statement). Just as before, the attacker can speculatively read arbitrary memory into `x`. They can then leak the value of `x` 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 or micro-op cache [65], based on which instructions were (speculatively) accessed; or (4) through port contention [10], branch predictor state [38], or other microarchitectural resources that differ among the branches.

C. Breaking software isolation with Spectre

Spectre attacks also break important guarantees in the domain of *software isolation*. In this domain, a host application executes untrusted code and wants to ensure that the untrusted code cannot access any of the host's data. Common examples of software isolation include JavaScript or WebAssembly runtimes, or even the Linux kernel, through eBPF [23]. Spectre attacks can break the memory safety and isolation mechanisms commonly used in these settings [39, 52, 59, 72].

We demonstrate with a small example:

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

char get_host_val(int idx) {
  if (idx < 100) { // check if within bounds
    return host_arr[idx];
  } else {
    return 0;
} }
```

Here, an attacker-supplied guest function `guest_func` calls the host function `get_host_val` to get values from an array. Although `get_host_val()` implements a bounds check, the attacker can still speculatively access out-of-bounds data by mistraining the branch predictor—breaking any isolation guarantees. Once the attacker (speculatively) obtains an out-of-bounds value of their choosing, they can leak the value (e.g., via data cache, etc.) and recover it after the speculative rollback. In this setting, we need to ensure that, *even speculatively*, untrusted code cannot break isolation.

D. Security properties and execution semantics

Formally, we will 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.

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], writing $[\![\cdot]\!]^{\text{seq}}$ to represent the execution model for standard sequential execution. We notate other execution models similarly; for example, $[\![\cdot]\!]^{\text{pht}}$ models prediction for Spectre-PHT attacks—i.e., conditional branch prediction. 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, they 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 Guarnieri et al. [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, on the other hand, exposes all values loaded from memory in addition to the addresses themselves (or equivalently, it exposes the trace of register values) [29]. Under this model, an attacker is allowed to observe all architectural computation; for a value to remain unobserved, it cannot be accessed at all over the course of execution, adversarial or otherwise. 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 $[\![\cdot]\!]_{\text{ct}}$ (i.e., it models a more powerful attacker). These properties make $[\![\cdot]\!]_{\text{arch}}$ most useful for software isolation, as any out-of-bounds accesses will immediately show up in an $[\![\cdot]\!]_{\text{arch}}$ leakage trace.

Surprisingly, the $[\![\cdot]\!]_{\text{ct}}$ and $[\![\cdot]\!]_{\text{arch}}$ leakage models generalize well to speculative execution—for example, if we want to

[1] Like Guarnieri et al. [29], we omit variable-latency instructions from our formal model for simplicity.

construct a semantics for Spectre-PHT attacks, we need only modify a sequential constant-time semantics to account for branch misprediction. Indeed, the execution model and leakage model of a semantics are orthogonal; we call the combination of the two the *contract* provided by the semantics—a sequential constant-time semantics has the contract $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$, while our hypothetical Spectre-PHT semantics would provide the contract $[\![\cdot]\!]^{\text{pht}}_{\text{ct}}$. Formally, 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 π is defined in terms of 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*: A program satisfies *non-interference* if, for any two π-equivalent initial states for a program p, an attacker cannot distinguish the two resulting leakage traces when executing p. 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 III: Sections III-A and III-B discuss choices in leakage models $[\![\cdot]\!]_\ell$ and security policies π. Sections III-C and III-D discuss tradeoffs for different execution models $[\![\cdot]\!]^\alpha$ and the transition rules in a semantics. In Section III-E, we discuss how the input language of the semantics affects analysis; and finally, in Section III-F, we discuss which microarchitectural features to include in formal models.

III. 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 do not want the security of our code to be dependent on a specific cache replacement policy or branch predictor implementation.

In the non-speculative world, formalisms for constant-time have been successful: 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.

A. 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 II-B, 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) [7, 15, 20, 27, 28, 61, 77]. 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 II-C and II-D describe the $[\![\cdot]\!]_{\text{arch}}$ leakage model, which is a better fit for modeling speculative isolation, e.g., for a WebAssembly runtime executing untrusted code [59] or a kernel defending against memory region probing [26]. Under $[\![\cdot]\!]_{\text{arch}}$, *all* values in the program are observable—this is what lets it easily model properties for software isolation: If we define a policy π where all values and memory regions outside the isolation boundary are secret, then software isolation security (or speculative memory safety) is simply non-interference with respect to $[\![\cdot]\!]_{\text{arch}}$ (and this π).

The $[\![\cdot]\!]_{\text{arch}}$ leakage model appears less frequently than $[\![\cdot]\!]_{\text{ct}}$ in formal models: Only two of the semantics in Figure 2 ([17, 29]) use the $[\![\cdot]\!]_{\text{arch}}$ leakage model. On the other hand, Spectre sandbox isolation frameworks such as Swivel [59], Venkman [72], and ELFbac [39] implicitly use the $[\![\cdot]\!]_{\text{arch}}$ model, as do the detection tools SpecFuzz [60], ASTCVW [43], SpecTaint [63], and the "weak" and "v1.1" modes of oo7 [81]. The three isolation frameworks all explicitly prevent memory reads or writes to any locations outside of the isolation boundary—i.e., enforcing non-interference under $[\![\cdot]\!]_{\text{arch}}$. The detection tools, meanwhile, look for gadgets

Semantics or tool name	Level	Leakage	Variants	Nondet.	Fence	OOO	Win.	Hij.	Tool	Impl.
Cauligi et al. [15] (Pitchfork)	Low	$[\![\cdot]\!]_{ct}$ P,B,M	P,B,R,S	Directives	✓	✓	✓	✓	Det*	Taint
Cheang et al. [17]	Low	$[\![\cdot]\!]_{arch}$ P,M,S,R	P	Oracle	✓	×	✓	×	Det/Mit	SelfC+
Daniel et al. [20] (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. [54]	Low	$[\![\cdot]\!]_{cache}$ T	P[2]	Oracle	⤻	×	✓	→	Mit*	Manual
Barthe et al. [7] (Jasmin)	Medium	$[\![\cdot]\!]_{ct}$ P,B,M	P,S	Directives	✓	×	×	×	Det	Safety
Patrignani and Guarnieri [61]	Medium	$[\![\cdot]\!]_{ct}$ P,B,M,L[3]	P[1]	Mispredict	✓	×	✓	×	—	—
Vassena et al. [77] (Blade)	Medium	$[\![\cdot]\!]_{ct}$ B,M	P	Directives	✓	✓	×	×	Mit	Flow
Colvin and Winter [18]	High	$[\![\cdot]\!]_{mem}$ M	P	Weak-mem	✓	✓	×	×	Val	Model
Disselkoen et al. [21]	High	$[\![\cdot]\!]_{mem}$ M	P	Weak-mem	✓	✓	×	×	—	—
P. de León and Kinder [62] (Kaibyo)	High	$[\![\cdot]\!]_{mem}$ M	P,S	Weak-mem	✓	✓	✓	×	Det	Model
AISE [83]	—	$[\![\cdot]\!]_{cache}$ C	P	Mispredict	×	×	✓	×	Det	Cache+
ASTCVW [43]	—	$[\![\cdot]\!]_{arch}$ L	P[4]	—	×	×	×	×	Det	Taint
ELFbac [39]	—	$[\![\cdot]\!]_{arch}$ L	P	—	×[5]	×	×	✓	Mit	Struct
KLEESpectre [80] (w/ cache)	—	$[\![\cdot]\!]_{cache}$ C	P	Mispredict	✓	×	✓	×	Det	Cache
KLEESpectre [80] (w/o cache)	—	$[\![\cdot]\!]_{mem}$ M	P	Mispredict	✓	×	✓	×	Det	Taint
oo7 [81] (v1 pattern)	—	$[\![\cdot]\!]_{mem}$ M	P	—	⤻	×	✓	×	Det/Mit	Flow
oo7 [81] ("weak" and v1.1 patterns)	—	$[\![\cdot]\!]_{arch}$ L	P	—	⤻	×	✓	⤻	Det/Mit	Flow
Specfuscator [71]	—	—[6] —	P,B,R	—	×[5]	×	×	✓	Mit	Struct
SpecFuzz [60]	—	$[\![\cdot]\!]_{arch}$ L	P	Mispredict	—	—	—	✓	Det	Fuzz
SpecTaint [63]	—	$[\![\cdot]\!]_{mem}$[7] M	P	Mispredict	✓	×	✓	⤻	Det	Taint
SpecuSym [30]	—	$[\![\cdot]\!]_{cache}$ C	P	Mispredict	×	×	✓	×	Det	SelfC+
Swivel [59] (poisoning protection)	—	$[\![\cdot]\!]_{mem}$ M	P,B,R	—	⤻[8]	×	×	✓	Mit	Struct
Swivel [59] (breakout protection)	—	$[\![\cdot]\!]_{arch}$ L	P,B,R	—	⤻[8]	×	×	✓	Mit	Struct
Venkman [72]	—	$[\![\cdot]\!]_{arch}$ L	P,B,R	—	⤻	×	×	✓	Mit	Struct

Level – How abstract is the semantics? (Section III-E)

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 III-A)

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

Variants (Section III-C)

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 does not reason about fences in the target/input code (and thus cannot verify the mitigated code as secure)
×	Does not reason about, or insert, fences

Hijack – Can it model or mitigate speculative hijack?

✓	Models/mitigates speculative hijack attacks
→	Models/mitigates forward-edge (ijmp) hijack only
⤻	Models/mitigates hijack only via speculative stores
×	Does not model/mitigate speculative hijack attacks

Nondet. – How is nondeterminism handled? (Section III-D)
OOO – Models out-of-order execution? (Section III-F)
Win. – Can reason about speculation windows? (Section III-C)

Tool – Does the paper include a tool?

Det	Tool detects insecure programs or verifies secure programs
Mit	Tool modifies programs to ensure they are secure
Val	Tool is only used to validate the semantics, does not automatically perform any security analysis
—	Does not include a tool
*	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 III-D)

Taint	Taint tracking (abstract execution)
Safety	Memory safety (abstract execution)
SelfC	Self composition (abstract execution)
Cache	Cache must-hit analysis (abstract execution)
Model	Model checking over the whole program
Manual	Manual effort
Fuzz	Fuzzing
Flow	Data flow analysis
Struct	Structured compilation

\+ Includes additional work or constraints to remove sequential trace (Section III-B)

Fig. 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 cannot mispredict indirect jump targets. [3]"Weak" variants of semantics leak loaded values during non-speculative execution. [4]Detects only "speculative type confusion vulnerabilities", a specific subset of Spectre-PHT. [5]Mitigates Spectre-PHT without inserting fences. [6]Defends by effectively preventing speculation, so leakage model is irrelevant. [7]Effectively $[\![\cdot]\!]_{mem}$ for loads, but detects any speculative store to an attacker-controlled address, which is more similar to $[\![\cdot]\!]_{arch}$ for stores. [8]Swivel operates on WebAssembly, which does not have fences. However, Swivel can insert fences in its assembly backend.

that can speculatively access *arbitrary* (or attacker-controlled) memory locations—i.e., breaking speculative memory safety. Unfortunately, these tools are not formalized, so their leakage models are not made explicit (nor 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 only 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 even weaker leakage models—for instance, oo7 only emits leakages that can be influenced by malicious input (see Section III-C) and KLEESpectre (with cache modeling enabled) only allows the attacker to observe the final state of the cache upon termination.

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 directly influences a memory address. For instance, in the following code, even though `cond` is not used to calculate the memory address, an attacker can infer the value of `cond` based on whether `arr[a]` gets 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 instead 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 certain vulnerability classes 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, detection tools can 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, 80] 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 proposed by Colvin and Winter [18], Disselkoen et al. [21], and Ponce de León and Kinder [62] have a more structured view of program execution (for instance, using dependency analysis or pomsets [25]). These semantics define leakage 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.

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 [50] to language-based isolation techniques like information flow control [66]. 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?

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

Definition 1 (Direct non-interference)**.** Program p satisfies *direct 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

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

We elide writing π for brevity—e.g., $NI([\![\cdot]\!]^{\text{pht}}_{\text{ct}})$ expresses constant-time security under Spectre-PHT semantics.

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 π-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

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

For non-terminating programs, we can compare finite prefixes of $[\![p]\!]^{\beta}$ against their sequential projections to $[\![p]\!]^{\text{seq}}$—since speculative execution must preserve sequential semantics, there will always be a valid sequential projection. 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 $\pi' = \pi \setminus canLeak(p, [\![\cdot]\!]^{\text{seq}}_{\text{a}})$. That is, we treat anything that could possibly leak under contract $[\![\cdot]\!]^{\text{seq}}_{\text{a}}$ as public. Relative NI is thus a (semantically) weaker property than direct NI, as it implicitly declassifies anything that might leak during sequential execution.

However, relative NI is still 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 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 ([7, 15, 20, 77]) all verify that programs satisfy the direct *speculative constant-time* (SCT) property.

Additionally, we find that current tools that verify relative NI [17, 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 $NI([\![\cdot]\!]^{\text{seq}}_{\text{ct}})$, then a compiler that enforces $NI([\![\cdot]\!]^{\text{seq}}_{\text{ct}} \Rightarrow [\![\cdot]\!]^{\text{pht}}_{\text{ct}})$ will compile p to a program that is *speculatively* constant-time $NI([\![\cdot]\!]^{\text{pht}}_{\text{ct}})$. Similarly, if a program is properly sandboxed under sequential execution $NI([\![\cdot]\!]^{\text{seq}}_{\text{arch}})$ and is compiled with a compiler that introduces no new *arch* leakage, the resulting program will remain sandboxed even under speculative execution [29].

Similarly, Patrignani and Guarnieri [61] explore whether compilers preserve *robust* non-interference properties. A security property is *robust* if a program remains secure even when linked against adversarial code (i.e., if the program is called with arbitrary or adversarial inputs). A compiler *preserves* a non-interference property if, after compilation from a source to a target language, the property still holds. In Patrignani and Guarnieri's framework, the source language describes sequential execution while the target language has speculative semantics, making their notion of compiler preservation very similar to enforcing relative NI.

C. 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 [7], Binsec/Haunted [20], Pitchfork [15], and Kaibyo [62] 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

Property or tool name	Non-interference prop.	Precision
Mcilroy et al. [54]	$\approx NI(\llbracket\cdot\rrbracket^{\text{pht}}_{\text{ct}})$	hyper
oo7 [81] $\Phi_{spectre}$	$\approx NI(\llbracket\cdot\rrbracket^{\text{pht}}_{\text{mem}})$	taint[1]
oo7 [81] $\Phi^{weak}_{spectre}$, $\Phi^{v1.1}_{spectre}$	$\approx NI(\llbracket\cdot\rrbracket^{\text{pht}}_{\text{arch}})$	taint[1]
Cache analysis [30, 83]	$NI(\llbracket\cdot\rrbracket^{\text{pht}}_{\text{cache}})$	hyper
Cache analysis [80]	$NI(\llbracket\cdot\rrbracket^{\text{pht}}_{\text{cache}})$	taint
Weak memory modeling [18, 21]	$NI(\llbracket\cdot\rrbracket^{\text{pht}}_{\text{mem}})$	hyper
Weak memory modeling [62]	$NI(\llbracket\cdot\rrbracket^{\text{pht-stl}}_{\text{mem}})$	hyper
Speculative constant-time (SCT)[2] [77]	$NI(\llbracket\cdot\rrbracket^{\text{pht}}_{\text{ct}})$	taint
Speculative constant-time (SCT)[2] [7, 20]	$NI(\llbracket\cdot\rrbracket^{\text{pht-stl}}_{\text{ct}})$	hyper
Speculative constant-time (SCT)[2] [15]	$NI(\llbracket\cdot\rrbracket^{\text{pbrs}}_{\text{ct}})$[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) [61]	$NI(\llbracket\cdot\rrbracket^{\text{seq}}_{\text{ct}} \Rightarrow \llbracket\cdot\rrbracket^{\text{pht}}_{\text{ct}})$	hyper
Robust speculative safety (RSS) [61]	$NI(\llbracket\cdot\rrbracket^{\text{seq}}_{\text{ct}} \Rightarrow \llbracket\cdot\rrbracket^{\text{pht}}_{\text{ct}})$	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$^-$) [61]	$NI(\llbracket\cdot\rrbracket^{\text{seq}}_{\text{arch}} \Rightarrow \llbracket\cdot\rrbracket^{\text{pht}}_{\text{ct}})$	hyper
Trace property-dependent observational determinism (TPOD) [17]	$NI(\llbracket\cdot\rrbracket^{\text{seq}}_{\text{arch}} \Rightarrow \llbracket\cdot\rrbracket^{\text{pht}}_{\text{ct}})$	hyper
Weak robust speculative safety (RSS$^-$) [61]	$NI(\llbracket\cdot\rrbracket^{\text{seq}}_{\text{arch}} \Rightarrow \llbracket\cdot\rrbracket^{\text{pht}}_{\text{ct}})$	taint

Execution models (Section III-C)	
$\llbracket\cdot\rrbracket^{\text{seq}}$	Sequential execution
$\llbracket\cdot\rrbracket^{\text{pht}}$	Captures Spectre-PHT
$\llbracket\cdot\rrbracket^{\text{pht-stl}}$	Captures Spectre-PHT/-STL
$\llbracket\cdot\rrbracket^{\text{pbrs}}$	Captures Spectre-PHT/-BTB/-RSB/-STL

Precision of the defined security property	
hyper	Non-interference hyperproperty, requires two π-equivalent executions
taint	Sound approximation using taint tracking, requires only one execution

Fig. 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][81] tracks taint of *attacker influence* rather than value sensitivity. [2]These works all derive their property from the definition given in [15] and share the same property name despite differences in execution mode. [3]The analysis tool of [15], 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 π-equivalent.

absence of additional constraints, they allow for a load to draw data from any prior store to the same address.

Other control-flow mechanisms are significantly more complex: Return instructions and indirect jumps can be *speculatively hijacked* to send execution to arbitrary (attacker-controlled) points in the program.[2] An attacker can trivially hijack a victim program if they can control (mis)prediction of the RSB (for returns) [45, 51] or BTB (for indirect jumps) [44]. Even without this ability, an attacker can hijack control-flow if they speculatively overwrite the target address of a return or jump (e.g., by exploiting a prior PHT misprediction) [42, 53, 73]. Formally, these attacks still fit within our non-interference framework—if a program can be arbitrarily hijacked, then it will be unable to satisfy any non-interference property. However, to formally verify that this is the case, a semantics must model these behaviors.

Although capturing all speculative behaviors in a semantics is possible, the resulting analysis is neither practical nor useful; in practice, developers need to make tradeoffs. For example, the semantics proposed by Cauligi et al. [15] can simulate all of the aforementioned speculative attacks, but their analysis tool Pitchfork only detects PHT- and STL-based vulnerabilities. On the other hand, tools like oo7 (with the "v1.1" pattern) [81] and SpecTaint [63] conservatively assume that writes to transient addresses can overwrite *anything*, and thus immediately flag this behavior as vulnerable.

The InSpectre semantics [27] proceeds in the opposite direction—it allows the processor to predict arbitrary values, even the values of constants. InSpectre also allows more out-of-order behavior than most other semantics (see Section III-F)—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, as a result, 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. Several semantics and tools in Figure 2 limit speculative execution by way of a *speculation window*. This models how hardware has finite resources for speculation,

[2]Including, on x86-family processors, into the *middle* of an instruction [9].

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. 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 [20], for example, uses different speculation windows for load-store forwarding than it uses for branch speculation.

Speculation windows are also valuable for mitigation tools: although tools like Blade [77] and Jasmin [7] are able to prove security without reasoning about speculation windows, modeling a speculation window reduces 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 techniques to sidestep particularly problematic Spectre variants. For example, even though Jasmin [7] 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 need only consider those that remain.

Cross-address-space attacks. Previous systematizations of Spectre attacks [13] differentiate between *same-address-space* and *cross-address-space* attacks. Same-address-space attacks rely on repeatedly causing the victim code to execute in order to train a microarchitectural predictor. Cross-address-space attacks are more powerful, as they allow an attacker to perform the training step on a branch within the attacker's own code.

Most of the semantics and tools in Figure 2 make no distinction between same-address-space and cross-address-space attacks, as they ignore the mechanics of training and consider all predictions to be potentially malicious. A notable exception is oo7 [81], 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 same-address-space attacks. Unfortunately, as a result, oo7 misses Spectre vulnerabilities in real code, as demonstrated by Wang et al. [80].

D. 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, speculative hijack attacks can send execution to entirely indeterminate locations. All of the semantics in Figure 2 allow these nondeterministic choices to be actively adversarial—for instance, given by attacker-specified directives [15, 77] or by consulting an abstract oracle [17, 28, 29, 54]. 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, these tools 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 [80], and AISE [83] 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, though, only handles the conditional branch predictor; i.e., Spectre-PHT attacks. Pitchfork [15] and Binsec/Haunted [20] adapt the *always-mispredict* strategy to account for out-of-order execution and Spectre-STL. Although it may not be immediately clear that *always-mispredict* strategies are sufficient to prove security—especially when the attacker can make any number of antagonistic choices—these strategies do indeed form a sound analysis [15, 20, 28].

Unfortunately, simulating execution only works for semantics where nondeterminism is relatively constrained: Conditional branches are a simple boolean choice, and store-to-load predictions are limited by the speculation window. If we pursue other Spectre variants, we will quickly become overwhelmed—again, an unconstrained hijack gadget can redirect control to 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 [77] and oo7 [81] 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. Similarly, oo7's "v1.1" pattern detection conservatively flags all (attacker-controlled) transient *stores*, as they may lead to speculative hijack. However, Blade and oo7—and mitigation tools in general—can afford to be less precise than verification or detection tools; these, conversely, must maintain higher precision to avoid floods of false positives.

Restricting nondeterminism. Compilers such as Swivel [59], Venkman [72], and ELFbac [39] 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—regardless of how a program may misspeculate, it will not be able to read (and thus cannot leak) sensitive data. Swivel and Venkman compile code into carefully aligned blocks so that control flow always land at the tops of protected code blocks, even speculatively; Swivel accomplishes this by clearing the BTB state after untrusted execution, while Venkman recompiles all programs on the system to mask addresses before jumping. Both systems also enforce speculative control-flow integrity (CFI) checks to prevent speculative hijacking, whether by relying on hardware features [37] or by implementing custom CFI checks with branchless assembly instructions. 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 (e.g., symbolic execution).

Open problems: Rigorous performance comparison. To the best of our knowledge, no work has rigorously compared the performance of all the tools in Figure 2. Perhaps the most complete comparison is by Daniel et al. [20], who compare the detection tools KLEESpectre, Pitchfork, and Binsec/Haunted in terms of the analysis time required to detect known violations in a few chosen targets. A general and objective performance comparison is difficult, if not impossible: The tools in Figure 2 operate on different types of programs (general-purpose, cryptographic, sandboxing) and different languages (x86, LLVM, WebAssembly). They also provide different security guarantees, as we discuss above. An intermediate step towards an expanded performance comparison, which would be a valuable contribution on its own, would be to develop a larger corpus of known attacks on realistic (medium-to-large-size) programs. This corpus would help evaluate both the security and performance of existing or newly-proposed tools.

E. 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) [15, 19, 20, 28, 80]. 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 [8, 19, 20]. 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 proposed by Jasmin [7], Patrignani and Guarnieri [61], and Blade [77] 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. [77] 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. [7] 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 only accesses 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 combines with other sequential security properties to give corresponding guarantees, such as for sandboxing, information flow, or rich type systems.

Weak-memory-style semantics. Weak-memory-style semantics present a fundamentally different approach, lifting the concept of speculative execution directly to a higher level. As these models are abstracted away from microarchitectural details, they are well-suited for analyzing Spectre variants in terms of data flow: Indeed, both Colvin and Winter [18] and Disselkoen et al. [21] treat Spectre-PHT as a constrained form of instruction reordering, while Ponce de León and Kinder [62] analyze dependency relations between instructions.

However, it remains challenging to translate a flexible semantics of this style into a concrete analysis tool: Of the three works discussed here, only Ponce de León and Kinder present a tool which can automatically perform a security

analysis of a target program,[3] though even they admit that it is slower than comparative tools based on operational semantics. 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 [61], they are overly conservative and incur a significant performance cost. Other compiler mitigations been shown unsound [60]—or worse, even introduce new Spectre vulnerabilities [20]—further reinforcing that these techniques must be grounded in a formal semantics.

Open problems: Formalization of new compilation techniques. Swivel [59], Venkman [72], and ELFbac [39] show how the structure of code itself can provide security guarantees at a reduced performance cost. For instance, both Venkman and Swivel demonstrate that organizing instructions into *bundles* or *linear blocks* (respectively) can mitigate speculative hijacks, making these transient attacks tractable to analyze and prevent. 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 [16], 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. [77] 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—that is, compiling standard C code and inserting Spectre mitigations.

F. Expressivity and microarchitectural features

One theme of this paper is 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 allows too many possible behaviors makes many analyses intractable. Indeed, a fundamental purpose of semantics is to provide a reasonable abstraction or simplification of hardware to ease analysis; 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 (discussed in Section III-C). But beyond speculative execution itself, there are many other microarchitectural features which are 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,[4] many speculative semantics also model it. 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. Furthermore, Disselkoen et al. [21] and Guanciale et al. [27] demonstrate how to abuse out-of-order execution to conduct (at least theoretical) novel side-channel attacks.[5]

Although modeling out-of-order execution might make a semantics simpler, the additional expressivity makes 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 [18] present analysis tools based on their respective out-of-order semantics, they only analyze very simple Spectre gadgets and 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 [15] operates on a set of "worst-case schedules" which represent a small subset of all possible out-of-order schedules—the developers formally show that this reduction does not affect the soundness of Pitchfork's analysis.

Caches and TLBs. Some speculative semantics and tools [30, 54, 80, 83] include abstract models of caches, tracking which addresses may be in the cache at a given time. One could imagine also including detailed models of TLBs. As discussed in Section III-A, modeling caches or TLBs is probably not helpful, at least for mitigation or verification tools—not only

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

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

[5] Disselkoen et al. [21] propose to abuse compile-time instruction reordering, which is different from microarchitectural out-of-order execution, but related.

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 can result in leakages beyond the ones we directly focus on in this paper. For instance, in the presence of multithreading, port contention in the processor's execution units can reveal sensitive information [10]; and many processor instructions, e.g., floating-point or SIMD instructions, can reveal information about their operands through timing side channels [4]. Most existing semantics do not model these specific effects. However, the commonly-used $[\![\cdot]\!]_{\text{ct}}$ and $[\![\cdot]\!]_{\text{arch}}$ leakage models are already strong enough to 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 already considers the sensitive data to be leaked once it influences control flow. For variable-time instructions, most definitions of $[\![\cdot]\!]_{\text{ct}}$ do not capture this leakage—but extending those definitions is straightforward [2]. In both of these examples, the $[\![\cdot]\!]_{\text{arch}}$ leakage model captures all leaks, as it (even more conservatively) already considers the sensitive data as leaked once it reaches a register—long before the data can influence control-flow or be used in an instruction. Although modeling any of these effects more precisely can 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 [18, 21, 62]—multiple threads are an inherent feature for this style, making them a promising vehicle for further exploring the interaction between speculation and concurrency.

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 [64]. This shifts the burden from application and runtime system engineers to OS engineers. Developing Spectre foundations to model the process abstraction will elucidate the security guarantees of such systems. This is especially useful, as the process boundary does not keep an attacker from performing out-of-place training of the conditional branch predictor, nor from leaking secrets via the cache state [13].

IV. Related Work

Both in industry and in academia, there has been a lot of interest in Spectre and other transient execution attacks. 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.

A. Systematization of Spectre attacks and defenses

Canella et al. [13] 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); and 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. 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. [12] 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. Separately, Xiong et al. [85] also survey transient execution attacks, with a specific focus on the mechanics of exploits for these attacks. 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.

B. 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 [41], cleaning up the cache state after misprediction [67], or making the cache invisible to speculation by incorporating some separate internal state [1, 40, 86]. Unfortunately, attackers can still use side channels other than the cache to exploit speculative execution [10, 70]. NDA [82], DOLMA [49], and Speculative Taint Tracking (STT) [87] block additional speculative covert channels by analyzing and classifying instructions that can leak information.

Fadiheh et al. [22] 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. In other words, UPEC is equivalent to the relative non-interference property $NI(\pi, [\![\cdot]\!]^{\text{seq}}_{\text{arch}} \Rightarrow [\![\cdot]\!]^{\text{pht}}_{\text{arch}})$.

The insights and recommendations from our work can guide future hardware mitigations; properties like $[\![\cdot]\!]_{\text{ct}}$ or $[\![\cdot]\!]_{\text{arch}}$ can serve as contracts of what software expects from hardware [29].

C. 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. [74] propose context-sensitive fencing, making various speculative barriers available to software. Li et al. [47] propose memory instructions with a conditional speculation flag. Context [68] and SpectreGuard [24] allow software to mark secrets in memory. This information is propagated through the microarchitecture to block speculative access to the marked regions. SpecCFI [46] 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 [78, 89]. As these approaches require both software and hardware changes, should develop a formal semantics to apply them correctly.

D. Other transient execution attacks

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

Meltdown variants. The Meltdown attack [48] 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 [75] and microarchitectural data sampling (MDS) [11, 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 [56, 57, 69], 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) [76] exploits the same root cause as Meltdown, Foreshadow, and MDS, but 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, 76], Intel only suggests applying them to legacy enclave software. Like Meltdown, LVI does not need software-based mitigation on recent Intel CPUs.

V. 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 increases the complexity of analysis, developers can combine analyses with structured compilation techniques 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 or port contention introduce complexity and reduce 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 or employ formal frameworks and to develop sound Spectre defenses with strong, precise security guarantees.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback. We thank Matthew Kolosick for helping us understand some of the formal systems discussed and in organizing the paper. This work was supported in part by gifts from Intel and Google; by the NSF under Grant Numbers CNS-2120642, CCF-1918573 and CAREER CNS-2048262; by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and, by the Office of Naval Research (ONR) under project N00014-15-1-2750.

References

[1] S. Ainsworth and T. M. Jones. MuonTrap: Preventing cross-domain Spectre-like attacks by capturing speculative state. In *ISCA*, 2020.

[2] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *USENIX SEC*, 2016.

[3] AMD. Security analysis of AMD predictive store forwarding. https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf, 2020.

[4] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In *IEEE S&P*, 2015.

[5] ARM. Straight-line speculation. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/downloads/straight-line-speculation, 2020.

[6] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In *CCS*, 2014.

[7] G. Barthe, S. Cauligi, B. Gregoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe. High-assurance cryptography in the Spectre era. In *IEEE S&P*, 2021.

[8] G. Barthe, B. Grégoire, and V. Laporte. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In *CSF*, 2018.

[9] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. Abu-Ghazaleh, C. Song, and M. Payer. SpecROP: Speculative exploitation of ROP chains. In *RAID*, 2020.

[10] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In *CCS*, 2019.

[11] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *CCS*, 2019.

[12] C. Canella, S. M. Pudukotai Dinakarrao, D. Gruss, and K. N. Khasawneh. Evolution of defenses against transient-execution attacks. In *Great Lakes Symposium on VLSI*, 2020.

[13] C. Canella, J. Van 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 *USENIX SEC*, 2019.

[14] C. Carruth. RFC: Speculative load hardening (a Spectre variant #1 mitigation). https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html, 2018.

[15] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe. Constant-time foundations for the new Spectre era. In *PLDI*, 2020.

[16] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan. FaCT: a DSL for timing-sensitive computation. In *PLDI*, 2019.

[17] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan. A formal approach to secure speculation. In *CSF*, 2019.

[18] R. J. Colvin and K. Winter. An abstract semantics of speculative execution for reasoning about security vulnerabilities. In *FM*, 2019.

[19] L.-A. Daniel, S. Bardin, and T. Rezk. Binsec/Rel: Efficient relational symbolic execution for constant-time at binary-level. In *IEEE S&P*, 2020.

[20] L.-A. Daniel, S. Bardin, and T. Rezk. Hunting the haunter — efficient relational symbolic execution for Spectre with Haunted RelSE. In *NDSS*, 2021.

[21] C. Disselkoen, R. Jagadeesan, A. Jeffrey, and J. Riely. The code that never ran: Modeling attacks on speculative evaluation. In *IEEE S&P*, 2019.

[22] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz. A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors. In *DAC*, 2020.

[23] M. Fleming. A thorough introduction to eBPF. *Linux Weekly News*, 2017.

[24] J. Fustos, F. Farshchi, and H. Yun. SpectreGuard: An efficient data-centric defense mechanism against Spectre attacks. In *DAC*, 2019.

[25] J. L. Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 1988.

[26] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida. Speculative probing: Hacking blind in the Spectre era. In *CCS*, 2020.

[27] R. Guanciale, M. Balliu, and M. Dam. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *CCS*, 2020.

[28] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. SPECTECTOR: Principled detection of speculative information flows. In *IEEE S&P*, 2020.

[29] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila. Hardware-software contracts for secure speculation. In *IEEE S&P*, 2021.

[30] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo. SpecuSym: Speculative symbolic execution for cache timing leak detection. In *ICSE*, 2020.

[31] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with WebAssembly. In *PLDI*, 2017.

[32] J. Horn. Speculative execution, variant 4: speculative store bypass, 2018.

[33] Intel. Speculative store bypass / CVE-2018-3639 / INTEL-SA-00115. https://software.intel.com/security-software-guidance/software-guidance/speculative-store-bypass, 2018.

[34] Intel. Deep dive: Intel analysis of microarchitectural data sampling, 2019.

[35] Intel. An Optimized Mitigation Approach for Load Value Injection. https://software.intel.com/security-software-guidance/best-practices/optimized-mitigation-approach-load-value-injection, 2020.

[36] Intel. Side channel mitigation by product CPU model. https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model, 2020.

[37] Intel 64 and IA-32 architectures software developer's manual, 2021.

[38] M. H. Islam Chowdhuryy, H. Liu, and F. Yao. BranchSpec: information leakage attacks exploiting speculative branch instruction executions. In *ICCD*, 2020.

[39] I. R. Jenkins, P. Anantharaman, R. Shapiro, J. P. Brady, S. Bratus, and S. W. Smith. Ghostbusting: Mitigating Spectre with intraprocess memory isolation. In *HotSos*, 2020.

[40] 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 *DAC*, 2019.

[41] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *MICRO*, 2018.

[42] V. Kiriansky and C. Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757*, 2018.

[43] O. Kirzner and A. Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *USENIX SEC*, 2021.

[44] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE S&P*, 2019.

[45] E. M. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre returns! Speculation attacks using the return stack buffer. In *WOOT*, 2018.

[46] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. SPECCFI: Mitigating Spectre attacks using CFI informed speculation. In *IEEE S&P*, 2020.

[47] 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 *HPCA*, 2019.

[48] 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 *USENIX SEC*, 2018.

[49] 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 *USENIX SEC*, 2021.

[50] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *IEEE S&P*, 2010.

[51] G. Maisuradze and C. Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*, 2018.

[52] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus. Speculator: a tool to analyze speculative execution attacks and mitigations. In *ACSAC*, 2019.

[53] A. Mambretti, A. Sandulescu, A. Sorniotti, W. Robertson, E. Kirda, and A. Kurmus. Bypassing memory safety mechanisms through speculative control flow hijacks. In *EuroS&P*, 2021.

[54] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest.

Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv:1902.05178*, 2019.
[55] Microsoft. Spectre mitigations in MSVC. https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/, 2018.
[56] D. Moghimi. Data sampling on MDS-resistant 10th Generation Intel Core (Ice Lake). *arXiv:2007.07428*, 2020.
[57] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *USENIX SEC*, 2020.
[58] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: better, faster, stronger SFI for the x86. In *PLDI*, 2012.
[59] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan. Swivel: Hardening WebAssembly against Spectre. In *USENIX SEC*, 2021.
[60] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *USENIX SEC*, 2020.
[61] M. Patrignani and M. Guarnieri. Exorcising Spectres with secure compilers. In *CCS*, 2021.
[62] H. Ponce de León and J. Kinder. Cats vs. Spectre: An axiomatic approach to modeling speculative execution attacks. In *IEEE S&P*, 2022.
[63] Z. Qi, Q. Feng, Y. Cheng, M. Yan, P. Li, H. Yin, and T. Wei. SpecTaint: Speculative taint analysis for discovering Spectre gadgets. In *NDSS*, 2021.
[64] C. Reis, A. Moshchuk, and N. Oskov. Site isolation: Process separation for web sites within the browser. In *USENIX SEC*, 2019.
[65] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat. I see dead μops: Leaking secrets via Intel/AMD micro-op caches. In *ISCA*, 2021.
[66] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Journal on Selected Areas in Communications*, 21(1), 2003.
[67] G. Saileshwar and M. K. Qureshi. CleanupSpec: An "undo" approach to safe speculation. In *MICRO*, 2019.
[68] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss. ConTExT: A generic approach for mitigating Spectre. In *NDSS*, 2020.
[69] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.
[70] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss. NetSpectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*, 2019.
[71] M. Schwarzl, C. Canella, D. Gruss, and M. Schwarz. Specfuscator: Evaluating branch removal as a Spectre mitigation. In *FC*, 2021.
[72] Z. Shen, J. Zhou, D. Ojha, and J. Criswell. Restricting control flow during speculative execution with Venkman. *arXiv:1903.10651*, 2019.
[73] M. Sternberger. Spectre-ng: An avalanche of attacks. In *Wiesbaden Workshop on Advanced Microkernel Operating Systems (WAMOS)*, 2018.
[74] M. Taram, A. Venkat, and D. Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ASPLOS*, 2019.
[75] 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 *USENIX SEC*, 2018.
[76] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *IEEE S&P*, 2020.
[77] M. Vassena, C. Disselkoen, K. V. Gleissenthall, S. Cauligi, R. G. Kici, R. Jhala, D. Tullsen, and D. Stefan. Automatically eliminating speculative leaks from cryptographic code with Blade. In *POPL*, 2021.
[78] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett. BRB: Mitigating branch predictor side-channels. In *HPCA*, 2019.
[79] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.
[80] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury. KLEESpectre: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM TOSEM*, 2020.
[81] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. oo7: Low-overhead defense against Spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 2019.
[82] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci. NDA: Preventing speculative execution attacks at their source. In *MICRO*, 2019.
[83] M. Wu and C. Wang. Abstract interpretation under speculative execution. In *PLDI*, 2019.
[84] Y. Wu, S. Sathyanarayan, R. H. Yap, and Z. Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In *European Symposium on Research in Computer Security*, 2012.
[85] W. Xiong and J. Szefer. Survey of transient execution attacks and their mitigations. *ACM Computing Surveys*, 2021.
[86] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *MICRO*, 2018.
[87] 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 *MICRO*, 2019.
[88] T. Zhang, K. Koltermann, and D. Evtyushkin. Exploring branch predictors for constructing transient execution trojans. In *ASPLOS*, 2020.
[89] L. Zhao, P. Li, R. Hou, J. Li, M. C. Huang, L. Zhang, X. Qian, and D. Meng. A lightweight isolation mechanism for secure branch predictors. In *DAC*, 2021.