2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-6512-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/EuroSP57164.2023.00044

SoK: Analysis of Root Causes and Defense Strategies for Attacks on Microarchitectural Optimizations

Nadja Ramhöj Holtryd, Madhavan Manivannan and Per Stenström
Chalmers University of Technology, Sweden
Email: {holtryd, madhavan, per.stenstrom}@chalmers.se

***Abstract*—Microarchitectural optimizations are expected to play a crucial role in ensuring performance scalability in the post-Moore era. However, recent attacks have demonstrated that these optimizations, which were assumed to be secure, can be exploited. Moreover, new attacks surface at a rapid pace limiting the scope of existing defenses. These developments prompt the need to review microarchitectural optimizations with an emphasis on security, so as to understand the attack landscape and the potential defense strategies.**

We provide a framework to analyze attacks on a wide range of microarchitectural optimizations and use that to systematize both transient and non-transient attacks and defenses, while highlighting the similarities and differences. We identify four root causes of timing-based side-channel attacks: *determinism*, *sharing*, *access violation* and *information flow*, through our systematic analysis. Leveraging our framework, we systematize existing defenses and show that they target these root causes in the different attack steps. We believe that our framework can assist in understanding the attack and defense landscape and provide guidance for designing secure microarchitectural optimizations.

1. Introduction

Computer architecture is facing a security crisis [77], [78]. Recent attacks [25], [50], [95], [98], [113], [122], [152], [176], [179] have demonstrated that microarchitectural optimizations, which were assumed to be fundamentally secure for a long time, leak information which can be exploited to steal secrets. Furthermore, efficient attacks continuously emerge targeting defenses, thereby limiting their effectiveness or even rendering the defenses moot altogether [17], [19], [24], [26], [51], [57], [139], [140], [142], [164], [182], [190]. Simultaneously, with the slowing down of Moore's Law, microarchitectural optimizations are expected to play an increasingly important role in ensuring performance scalability. Consequently, there is a strong need to be able to leverage microarchitectural optimizations without compromising security.

Microarchitectural optimizations, implemented in commercial processors, like branch predictors [1], [2], [47], [49], [98], caches [66], [138], [194] and prefetchers [37], [41], [111], [161], [181], among others, are prone to attacks. A recent paper [149] demonstrated that several optimizations proposed in literature, but not known to be commercially implemented as yet, such as value prediction [109], are vulnerable. This underscores the importance of conducting a thorough review of microarchitectural optimizations with an emphasis on security.

Prior works have started the important task of analyzing attacks and defenses for different microarchitectural optimizations [30], [32], [34], [53], [76], [80], [117], [166], [191]. However, most of the works focus only on transient attacks and defenses [30], [32], [76], [80], [191], SW-based defenses [34] or cover a limited set of non-transient attacks [53], [166] (see Section 6 for details). Pandora [149] considers a broader set of non-transient microarchitectural optimizations and provides *microarchitectural leakage descriptors* (MLDs) which quantify the information leakage. The MLDs show if a specific optimization can leak and how much information is leaked (1-bit or a few bits). Unfortunately, this information falls short on providing a systematic analysis of the similarities across different microarchitectural optimizations and the underlying root causes which make them vulnerable to attacks. Such an analysis can also help with the categorization of existing defense strategies and with the potential identification of attacks and defenses.

Our goal, in this paper, is to perform a systematic analysis to highlight the common root causes which make microarchitectural optimizations vulnerable to exploits that reveal secrets. In order to enable analysis of a diverse set of microarchitectural optimizations, we present an abstract model of the architecture and the microarchitectural state transitions involved in an attack. Using this model as a framework, we analyze several timing-based side-channel attacks available in the literature on an extensive set of microarchitectural optimizations: cache, prefetching, branch prediction, computational simplification, speculative execution and value prediction. We also analysed several additional microarchitectural optimizations but omit them from the discussion due to space constraints.

Our analysis reveals four root causes which are exploited in order to succeed with attacks targeting the diverse set of microarchitectural optimizations covered. The root causes are **determinism**, **sharing**, **access violation** and **information flow**. Here, determinism causes microarchitectural optimizations to be triggered in the same way under the same pre-conditions, leading to predictable microarchitectural state transitions and timing variations. Sharing of microarchitectural state, which is accessible to both the adversary and the victim, enables the creation of a side-channel. Access violation enables access to a secret outside of the intended protection domain. Finally, information flow refers to exchange of information through microarchitectural state. We note that a subset of these root causes have been identified individually in the context of specific attacks [25], [40], [80], [115], [166]. However, in our analysis we show that a subset of, or all, the root causes are common across attacks on a broad set of microarchitectural optimizations.

We show that the existing defenses that focus on addressing the vulnerabilities in different microarchitectural optimizations can be classified as targeting one or more of the identified root causes. We observe that sim-

DOI 10.1109/EuroSP57164.2023.00044

ilar defense strategy can be/is applied across different microarchitectural optimizations, to target the same root cause vulnerability. For instance, partitioning can thwart attacks using the cache [96], [114], SMT [169] and branch prediction [183], [199], by affecting sharing, information flow and determinism. In addition, the defenses can also be applied to address the applicable root causes in the different steps of the attack. Typically, eliminating the root cause(s) in any of the attack steps can mitigate/protect against an attack that targets a specific resource under a specific threat model. We show the utility of our framework by identifying potential attacks and defenses for value prediction that have not been explored in literature.

Overall, the analysis demonstrates the versatility of our simplistic framework to capture a diverse set of attacks and defense strategies for different microarchitectural optimizations. We expect that our framework can be easily extended to study microarchitectural optimizations we do not explicitly cover in this paper. We also believe that it can assist in understanding the landscape of attacks on a broad range of microarchitectural optimizations, categorizing existing defense strategies proposed to thwart such attacks, and in designing secure microarchitectural optimizations. Interesting avenues for future work leveraging this framework are discussed in Section 5.

In summary, we make the following contributions:

- We provide a framework to analyze both transient and non-transient execution attacks on a broad range of microarchitectural optimizations, highlighting similarities and differences.
- We identify four root causes: **determinism**, **sharing**, **access violation** and **information flow**, that enable timing-based side-channel attacks on a wide range of microarchitectural optimizations.
- We analyze defenses using our framework and make a classification based on the root causes they address. Based on the analysis, we discuss potential attack and defense possibilities for microarchitectural optimizations.

The paper is structured as follows. Section 2 presents our framework and defines the root causes. Section 3 and 4 use the framework to systematize the attacks and defenses, respectively. Section 5 presents general observations and future research directions while Section 6 discusses the closest related works before we conclude in Section 7.

2. Systematization Framework

We first present an abstract architecture model and outline the steps for carrying out attacks based on the model. We then use this model as a framework to identify the root causes that enable attacks. Finally, we present an actual attack in the context of this framework.

2.1. Abstract model and side-channel attack

The architecture model is represented as a finite state machine (FSM) where the *architectural state (AS)*, comprising SW-visible registers and memory, is the externally visible interface, that is accessible to a program. An FSM transition is caused when instruction execution leads to a change in AS.

The microarchitecture represents an implementation of the FSM specification and typically comprises several microarchitectural optimizations, denoted $\{O_1, O_2, \ldots, O_n\}$, to enable an efficient implementation. A microarchitectural optimization uses a set of microarchitectural resources, denoted $R = \{R_1, R_2, \ldots, R_m\}$, to implement the intended functionality. This model permits resources to be shared across different optimizations. We define *microarchitectural state (MS)* as a snapshot of the state of all the m microarchitectural resources in the system at time instance t, denoted $MS = \{state(R_1), state(R_2), \ldots, state(R_m)\}_t$.

It is important to note that while the change in AS caused by an FSM transition remains the same across different implementations of a given FSM specification, the change in *MS* varies depending on the optimizations triggered, resources used and the implementation. Even when considering a specific implementation, there is a one-to-many mapping relationship between AS and *MS*; i.e., a single AS can have several equivalent *MS*. Furthermore, the time it takes for an implementation to make a transition between different *MS* (caused by an action) may vary and this property is typically exploited by attacks. As an example, executing a load instruction will cause a change in the AS while the insertion of a corresponding line in the cache hierarchy as a consequence of executing the load instruction will cause a change in the state of the cache(s) which is a microarchitectural resource. A typical attack exploits information leaked through MS that is not available through AS.

We next consider an abstract model of an attack that shows the different steps involved while leveraging *MS* as the side-channel to communicate the secret from a victim to an adversary. In our model, we define a step as a tuple of current state and action which leads to a new state, $\{MS_{current}, action\} \rightarrow MS_{next}$. Figure 1 shows the different steps listed in this model and is based on the attacks proposed in literature [30], [32], [34], [76], [80], [117], [166], [191]. We assume *MS* is in the initial state (MS_I) before any of the steps in the attack are carried out. When the *setup* step is performed MS_I makes a transition to the primed state (MS_P). The *setup* step ensures that the necessary preconditions are in place to encode the secret into MS_P in the next step of the attack. When the *interact* step is performed the secret is accessed and is encoded in the microarchitectural state (MS_E). The secret is encoded specifically through the state of one or more microarchitectural resources. If the secret is encoded through a microarchitecture resource state, that is accessible to both the victim and the adversary, it can potentially be used as a side-channel to communicate the secret.

Attacks optionally utilize the *transmit* step in case the encoded microarchitectural resource state is not accessible by the adversary or the specific *MS* based side-channel is noisy (i.e. the channel is prone to high error rate and has low channel bandwidth). When the transmit action is performed the secret is usually re-encoded through the state of a different shared microarchitectural resources (MS_T) which can address the aforementioned transmission

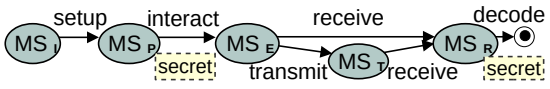

Figure 1. *MS* transitions in different steps of an attack.

limitations. When the *receive* step is performed, the adversary accesses the microarchitectural state of the specific resource(s) and observes timing variations based on the encoded secret while the state transitions to MS_R. Finally, in the decode step, the timing variations observed are used as the basis to infer the secret.

The steps outlined above that cause *MS* transitions and secret information to be leaked can be performed by the adversary, the victim or both, depending on the type of attack. In the abstract model it is required that the state of at least one microarchitectural resource is shared between the victim and an adversary to enable information flow and consequently communicate the secret. The microarchitectural resources that are shared between the victim and the adversary are specific to the implementation and the threat model (see Section 2.3 for details).

Prior works define attack steps differently which leads to fewer/more steps. For example, Xiong et al. [117] defines three attack steps while Hu et al. [80] use six attack steps. In contrast, our attack model include five steps where each step consists of action(s) performed by adversary and/or victim on microarchitectural resource(s) *MS* which leads to a new *MS*. Note that the difference between the *interact* and the *transmit* step is that the former accesses and encodes the secret on *MS* while the latter re-encodes the secret on shared *MS*.

We exemplify the abstract model by describing the steps in the flush+reload [194] attack. This attack uses a single shared microarchitectural resource, a shared cache (*SC*). The goal of the attack is to infer the secret which is revealed through the victim's cache accesses because of data-dependent control flow. A prerequisite for the attack is that the cache lines of interest are mapped to a shared page that is accessible by the adversary as well as the victim. During the *setup* step, the adversary uses the *clflush* instruction to evict the target line(s) belonging to the shared pages from the cache. The fact that the pages are shared allows the adversary to evict data that is accessed by a victim. The state of the cache after this step is $MS_P\{R_{SC}[\{target\}{=}miss]\}$. During the *interact* step the victim executes and interacts with the secret which is encoded in the *SC* state by the presence/absence of the specific target cache line(s). The cache state changes to $MS_E\{R_{SC}[\{target]{=}hit]\}$. Since the cache is shared the adversary can detect the state change that has occurred as a result of the interaction. The adversary, during the *receive* step, accesses the cache line(s) (target) and measures the time. Through timing the adversary deduces which line(s) the victim has inserted and thereby infers the secret.

2.2. Root causes

We define the root causes of an attack in the context of the abstract model and exemplify with an actual attack.

2.2.1. Determinism. We define determinism as the characteristic of a microarchitectural optimization whereby microarchitectural resource(s) used by an optimization, under the same pre-conditions, is/are triggered in the same manner and cause a *predictable* microarchitectural state transition and timing variation. In other words, determinism causes an expected *MS* transition and timing variation upon an action by the adversary and/or the victim. In the abstract model of the attack, determinism enables the adversary to control *MS* transitions from MS_I through to MS_R across multiple steps.

2.2.2. Sharing. We define sharing as the characteristic of a microarchitectural optimization whereby the state of the microarchitectural resource(s) used by an optimization is/are shared between a victim and an adversary. In the abstract model sharing allows for the creation of a side-channel between the victim and the adversary's protection domain through *MS*.

2.2.3. Access violation. We define access violation as the characteristic of a microarchitectural optimization whereby one/many microarchitectural resource(s) permit(s) access to secret data at the microarchitectural level which is outside the protection domain of the program. This consequently enables information to flow outside the intended protection domain and occurs either in the *interact* or the *receive* step of the attack which causes secret information to be encoded into *MS*.

2.2.4. Information flow. We define information flow as the characteristic of a microarchitectural optimization to exchange information through the state of one or many microarchitectural resource(s). Information flow enables the adversary to infer the secret by observing the state change of microarchitectural resource(s).

The flush+reload attack, discussed earlier, exploits determinism, sharing and information flow in each of the steps of the attack. Determinism guarantees that the three state transitions occur in the attack; firstly, the eviction of the target line(s) from the cache in *setup*, followed by insertion of a cache line in *interact*. Finally, timing differences are observed based on the presence and/or absence of specific cache lines in *receive*. Likewise, information flow and sharing guarantee that the secret is encoded and communicated, through *MS* of the shared *SC*, from the victim to the adversary, across the different steps. Access violation is not exploited in this attack since the *interact* step, executed by the victim, does not lead to an access outside its own protection domain, i.e., there is no access violation on the microarchitectural level. In general, attacks can exploit a subset or all of the root causes as we will show in Section 2.4 and 3.

2.3. Threat Model

We consider four types of threat models in our classification. Across the different threat models, the secret, that the adversary attempts to steal, resides in a different protection domain from that of the adversary.

An adversary can execute on a separate core from the victim, referred to as *CrossCore*; be time-multiplexed on the same core as the victim process, referred to as *SameThread*; run on distinct SMT threads executing on the same core, referred to as *SMT* or run in isolation, referred to as *Solo*. In the *Solo* threat model the adversary only needs to have a pointer to the location of the victims data (kernel memory). The threat model determines which set of microarchitectural resources are shared or private in the attack setting on a given machine. A *CrossCore* threat model leads to a scenario where fewer microarchitectural resources are shared. In contrast, assuming the

SameThread or the *SMT* threat model leads to potentially more microarchitectural resources being shared between victim and adversary, leading to a broader attack surface.

Another dimension of the threat model is based on whether the adversary or the victim performs the different actions in an attack. In a typical attack the adversary performs one or more steps. However, it has been shown that an adversary can manipulate the victim to perform some of the required actions through the use of specific gadgets. This is especially useful in scenarios where the adversary does not have access to a shared microarchitectural resource state to facilitate the *MS* transitions. This strategy increase the scope of possible attacks even in cases where the threat models limit the attack surface.

One example is the Spectre v2 attack [98] which requires training the Branch Target Buffer (*BTB*) as part of the *setup* step. Without gadgets such attacks would only be possible with the *SMT*/*SameThread* threat models since the *BTB* is not shared between cores. However, when the victim can be manipulated to perform the training, a *CrossCore* threat model can be used. The manipulation from the adversary can be performed by calling a function in the victims code with a controlled input, i.e., the action of triggering a gadget. The gadget can be constructed using Return Oriented Programming (ROP) [158] where code snippets ending with a return instruction are used by changing the return address and thereby chaining the different snippets together. However, these attack scenarios depend on the availability of gadgets and/or vulnerabilities, such as buffer overflows, and most have not been demonstrated outside of specific environments [32].

For some optimizations and attack scenarios the side-channel can be noisy and/or obscured by other optimizations, thereby making it difficult to decode the secrets based on *MS* transition and the consequent timing variations. Amplification gadget(s) can be used by the adversary to enhance the timing differences and ease the decoding of the secret. A simple example is on the cache side-channel, where prefetching can obscure the secret-related accesses. This can be circumvented by using a linked list [174] or by spreading accesses across pages [98] since most prefetchers only target linear or strided access patterns and do not prefetch across page boundaries.

2.4. Case Study

Next, we will describe an actual attack, Spectre v1, using the abstract model, the root causes we have identified and the threat model, as a framework.

2.4.1. Spectre v1. Spectre v1 [98] leverages three different optimizations: branch prediction, speculative execution and a shared cache. The example code for the attack is shown in Figure 2, where *x* is controlled by the adversary and is used to represent the address delta between the base address of *array1* and the secret's location. The attack involves speculatively executing the if-clause code block, by training the branch predictor to predict taken. When the taken code block is speculatively executed the adversary can cause speculative access to *array2* indexed using the adversary controlled *x*. Even when the speculatively

```
if(x < array1_size){y = array2[array1[x]*4096;}
```

Figure 2. Example code for Spectre v1 attack [98].

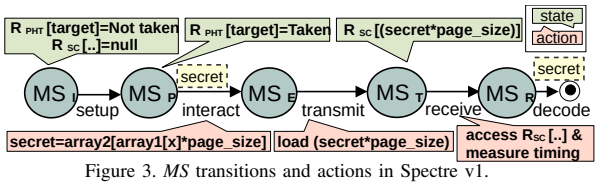

Figure 3. *MS* transitions and actions in Spectre v1.

executed instructions are eventually rolled back this still leaves a trace in the cache, as a consequence of the access to *array2*, which is then used to infer the secret.

Figure 3 show an overview of the attack steps and MS transitions. The *setup* step consists of (mis)training the branch predictor Pattern History Table (*PHT*) by adversary/victim to change the prediction for the targeted conditional branch from $MS_I\{R_{PHT}[[target]=Not\ taken]\}$ to $MS_P\{R_{PHT}[[target]=Taken]\}$. The necessary (mis)training can either be performed by the adversary, restricting the threat model to SMT/SameThread, or be performed by the victim. To make the victim perform the (mis)training of the specific conditional branch a gadget must be located in the victim binary containing instructions which execute and train the target *PHT* entry to mispredict on the conditional branch. Setup of the *SC*, as in flush+reload, is also needed since it will be used later in the *transmit* step.

In the *interact* step the victim executes speculatively and accesses the secret because of the (mis)prediction. Speculative execution allows the CPU to temporarily violate program semantics by transiently executing code that accesses the secret and leave a trace in the *MS*. In the *transmit* step the secret is re-encoded in the state of another microarchitectural resource through a secret dependent access to the *SC* (*load (secret*page_size)*). This access causes the *SC* to transition to $MS_T\{R_{SC}[\{(secret*page\_size)\}=hit]\}$. The *MS* transition occurs before the processor detects that the speculative execution was erroneous and rolls back the register state, leaving a trace in the state of the *SC*. In the *receive* step the adversary accesses the cache, measures the time and infers the secret based on timing variations for cache lines.

The root causes which enable this attack are determinism, sharing, access violation and information flow. Determinism guarantees that, the adversary can cause a *PHT* state update in an intended entry in the *setup* step, which is then used in the *interact* step. In addition, determinism ensures that the *SC* becomes primed, as a result of the flush, in *setup* and that the secret-dependent loads will result in timing variations corresponding to the presence/absence of lines observed in the *receive* step. Access violation enables access to the secret which resides in a different protection domain, through speculative execution. Sharing is exploited in both the *PHT* and the *SC* during the *setup* step and in the cache during the *receive* step. Information flow is allowed through each of the state of the shared resources, across the different attack steps.

3. Systematization of Attacks

In this section we use the abstract model, the root causes and the threat model as a framework to systematize attacks on a broad set of microarchitectural optimizations. We describe a typical attack on each optimization and analyze the necessary root causes exploited in the different steps of the attack. We group attacks wherever possible and also discuss dissimilarities between the attacks on the

same microarchitecture optimization. Note that the goal of this analysis is not to exhaustively discuss all the attacks proposed in literature. Rather, through the discussion, our aim is to highlight commonalities and differences across different attacks that target a microarchitectural optimization by addressing the following questions:

1) Which microarchitectural resource(s) are exploited in an attack?
2) Which root cause(s) are necessary to enable the attack and in which attack step(s)?
3) Under which threat model(s) is/are the attacks possible?

To simplify the discussion, we categorize the microarchitectural optimizations into two broad groups – non-transient and transient optimizations – and analyze attacks on each of them. The systematization of attacks is presented in Table 1. Finally, we also analyse security vulnerabilities in value prediction to demonstrate that our framework can identify new attacks.

3.1. Non-transient attacks

3.1.1. Cache. The last-level cache is typically shared among cores to improve cache utilization and to reduce costly off-chip accesses. The *SC* is vulnerable to side-channel attacks and is an attractive attack surface because of the channel characteristics (i.e., low noise and high attack bandwidth [163]). There exist three high-level attack categories: i) reuse-based [66], [70], [87], [194] where data is shared between adversary and victim allowing both to access it, ii) conflict-based [86], [91], [112], [133], [134], [138], where an adversary creates conflicts to evict target lines belonging to the victim, and iii) observation-based [67], [162], i.e., brute-force conflicts. In observation-based attacks the conflicts and observed behaviour relates to the cache as a whole and is not restricted to a selected target set/line, as in conflict-based attacks. Furthermore, in our classification, attacks that rely on port contention [20], [195] fall in the conflict-based attack category. We note that these categories are common across attacks on other shared resources and have implications for the defense strategies (see Section 4).

Prime+probe [138], a typical conflict-based attack [86], [91], [112], [133], [134], [138], is used in the absence of data sharing between an adversary and a victim. The threat model is *CrossCore*, since the *SC* is shared between cores. In the *setup* step, the adversary first finds the set of cache lines (eviction-set) which will create conflicts in the targeted index shared with cache lines belonging to the victim and evict them from the cache. The state changes to $MS_P\{R_{SC}[[target]{=}A_{adversary}]\}$ for all the lines in the target index. Next, in the *interact* step, the victim accesses the secret during execution which is in turn encoded in the *SC* state through the presence/absence of the specific cache line(s). The state can change to $MS_E\{R_{SC}[[target]{=}B_{victim}]\}$ for at least one line. In the *receive* step the adversary re-accesses the primed cache line(s) and based on the timing variations infers the secret.

The root causes determinism, sharing and information flow enable the attack, as shown in Table 1. Determinism provides different guarantees in the three steps of the attack. First, in the *setup* step it ensures that conflicts can be created at a specific index in the cache which causes an eviction of the targeted cache line. Second, in the *interact* step it ensures the secret is encoded through the *MS* of the *SC*. Third, it also causes a timing variation corresponding to the presence/absence of the cache line. Sharing and information flow ensures that the secret is encoded and communicated, through *MS* of the *SC*, from the victim to the adversary, across the different steps.

Observation-based attacks [67], [162] work by observing set conflicts in all the sets in the cache instead of actively causing them in a few sets like prime+probe. These attacks leverage the observation that the same conflict patterns will re-occur because the cache behavior of a program is deterministic. An alternative *setup* step is used, without relying on using *clflush* or a specific eviction-set, which involves accessing a large buffer [124], [154], [162] to flush all cache lines in the *SC* and creating conflicts across all the sets in the cache. This approach, however, has the drawback of a lower bandwidth since filling the *SC* is time consuming. These observation-based attacks are challenging to defend against since they rely on determinism and the shared *MS* of the *SC* in the *interact* and *receive* steps. This limits defenses that can be used to avoid the attack (for details see Section 4.1.1).

A number of attacks show that other state can also be used for attacks, such as the state of replacement metadata [96], cache coherence [72], [173], way prediction [112], interconnect [135], [184], cache banks [195], a translation-lookaside buffer (*TLB*) [60], [171] or a memory management unit (*MMU*) [178]. Using other microarchitecture resource state (related to *SC*) makes it possible to circumvent defenses targetting *SC*. Van Schaik et al. [178] propose a prime+probe-like attack where an eviction-set is found and used in the *MMU* instead of the *SC* to overcome defenses targeting conflict-based attacks in the *SC*. Wan et al. [184] show an attack using temporal contention on the mesh interconnect, while Paccagnella et al. [135] show an attack on the ring interconnect. These attacks exploit determinism, sharing and information flow as previously discussed in the *SC* attacks. However, sharing in other microarchitectural resources such as the *MMU* or the interconnect are exploited, instead of the *SC*. We do not exhaustively cover all *SC* related attacks since our goal is to discuss a representative set to demonstrate the applicability of our framework.

3.1.2. Prefetching. Prefetchers predict addresses that will be used by a program and proactively fetches them to help hide memory access latency. The attacks exploiting prefetching can be broadly grouped into two fundamentally different categories, 1) SW-based and 2) HW-based. Attacks that belong to the latter category exploit the availability of a HW prefetcher (microarchitectural resource) and specifically utilize the *MS* of the prefetch tables and/or the *SC* as the side channel while attacks that belong to the former category exploit different prefetch instructions directly which exhibit different timing depending on the state of the *TLB* and/or the *SC*. The attacks exploiting SW-based and HW-based prefetching mechanisms can be grouped into two categories, 1A and 2A) which exploit the lack of permissions check and 1B and 2B) which exploit a secret/state dependent variations.

Microarchitectural optimization	Attack(s)	Resource(s)	Attack steps				Threat model
			A_S	A_I	A_T	A_R	
Shared cache (SC)	flush+flush [66], flush+reload [70], [87], [194]	R_{SC}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	1,2,3
	prime+probe [86], [91], [133], [134], [138]	R_{SC}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	1,2,3
	observation [162], C5 [124]	R_{SC}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	1
	collide+probe [112]	R_{SC}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	2
	load+reload [112]	R_{SC}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	3
	xlate+probe/abort [178]	R_{SC}, R_{MMU}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	1,2,3
	TLBleed [60], [171]	R_{SC}, R_{TLB}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	2,3
	CacheBleed [195]	$R_{SC^T_{bank}}$	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	3
	MemJam [127]	R_{SC}	D/S/I [A]	D/S/A/I [V]	-	D/S/I [A]	2,3
	LoR [135]	R_{ring^T}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	1
	MeshUp [184] MeshAround [42]	R_{mesh^T}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	1
	CacheTiming [20], [175]	R_{SC}	- [A]	D/I [V*]	-	D/I [A]	1/-[2]
Prefetching (P)	Prefetch SCAs [65], [111]	R_{SC},R_{TLB}	D/S/I [A]	D/S/A/I [V]	-	D/S/I [A]	2,3
	prefetch+reload, prefetch+prefetch [72]	R_{SC}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	1
	LeakingControlFlow [37]	R_P,R_{SC},R_{TLB}	D/S/I [A]	D/S/A/I [V]	D/S/I [V]	D/S/I [A]	2,3
	DMP [149], [181]	R_P,R_{SC}	D/S/I [V*/A]	D/S/A/I [V*/A]	D/S/I [V*/A]	D/S/I [A]	2
	Unveiling [161]	R_P,R_{SC}	D/S/I [A]	D/S/I [V]	D/S/I [V]	D/I [A]	1,2,3
	FetchingTale [41]	R_P,R_{SC}	D/S/I [A]	D/S/I [V]	D/S/I [V]	D/S/I [A]	2,3
Branch prediction (BP)	JumpOverASLR [47]	R_{BTB}	D/S/I [A]	D/S/A/I [V]	-	D/S/I [A]	2,3
	PredictingKeys [1]–[3], [47]	R_{BTB}	D/S/I [A]	D/S/I [V]	-	D/S/I [A]	2,3
	BranchScope [49]	R_{PHT}	D/S/I [A]	D/S/A/I [V]	-	D/S/I [A]	2,3
	BranchShadowing [107]	R_{PHT},R_{LRB}	D/S/I [A]	D/S/A/I [V]	-	D/S/I [A]	2,3
Computational simplification (CS)	Subnormal FP [13], [100]	R_{FPU}	(D/S/I)[1] [A]	D (S/I)[1] [V]	-	(D/S/I)[1] [A]	-
	Early termination [62]	R_{MUL}	- [A]	D [V*]	-	- [A]	2,3
Transient attacks: Speculation-based	Spectre v1 [98], [173], v1.1 [97]	R_{SC},R_{PHT},R_{BHB}	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	1,2
	Spectre v2 [17], [36], [98], [190]	R_{SC},R_{BTB},R_{BHB}	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	1,2
	Spectre v4 [125], LVI [177]	R_{SC},R_{STL}	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	2
	Spectre v5 (ret2spec) [102], [121]	R_{SC},R_{RSB},R_{BTB}	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	1,2
	BranchSpec [88]	R_{PHT}	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	2,3
	NetSpectre [154]	R_{PHT},R_{AVX2}/R_{SC}	D/S/I [V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	-[2]
	CROSSTALK [143]	R_{SC},R_{LFB},$R_{staging\_buf.}$	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	1
	SMoTherSpectre [23]	R_{SC},R_{PHT},R_{ports^T}	D/S/I [A]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	3
	SpectreRewind [51]	R_{SC},R_{BTB},R_{ports^T}	D/S/I [A]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	2,3
	Speculative interference [19]	R_{SC},R_{BTB},R_{MSHR},R_{RS},R_{EU^T}	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	1,2,3
	ROB cont. [4]	R_{PHT},R_{ROB}	D/S/I [A/V*]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	1,2,3
Transient attacks: Exception-based	Meltdown [113], [173]	R_{SC},R_{BTB} [PF-US]	D/I [A]	D/S/A/I [A]	D/I [A]	D/I [A]	4
	Foreshadow [176], [188]	R_{SC},R_{TLB} [PF-P]	D/I [A]	D/S/A/I [A]	D/I [A]	D/I [A]	4
	Spectre1.2 [97]	R_{SC},R_{TLB} [PF-RW]	D/I [A]	D/S/A/I [A]	D/I [A]	D/I [A]	4
	LazyFP [165]	R_{SC},R_{FPU},R_{SIMD} [#NM]	D/S/I [V]	D/S/A/I [A]	D/I [A]	D/I [A]	2,3
	Fallout [126]	R_{SC},R_{SB}	D/S/I [A]	D/S/A/I [A&V]	D/I [A]	D/I [A]	2,3
	RIDL [179] ZombieLoad [152]	R_{SC},R_{LFB}	D/S/I [A]	D/S/A/I [A&V]	D/I [A]	D/I [A]	2,3
	LVI [177]	R_{SC},R_{FPU},R_{SB},R_{LFB} [PF]	D/S/I [A]	D/S/I [V*]	D/S/A/I [V*]	D/S/I [A]	2,3,4

TABLE 1. ATTACK SYSTEMATIZATION. ROOT CAUSES: DETERMINISM (D), SHARING (S), ACCESS VIOLATION (A), INFORMATION FLOW (I). ATTACK STEPS: SETUP (A_S), INTERACT (A_I), TRANSMIT (A_T), RECEIVE (A_R), PERFORMED BY ADVERSARY [A] OR VICTIM [V]. GADGET*. THREAT MODELS: 1) CROSSCORE, 2) SAMETHREAD, 3) SMT, 4) SOLO.[1]IN SW.[2]REMOTE.[T]TEMPORAL RESOURCE.

One typical attack from category 1A, a SW-based attack that exploits the lack of permissions check, is the address-translation attack by Gruss et al. [65] where the goal of the attack is to translate between virtual and physical addresses from unprivileged user-space and overcome the protection provided by user-space and kernel-space Address Space Layout Randomization (ASLR). We focus on the first phase of the attack where the adversary searches through possible addresses and tests if two virtual addresses, a and a', map to the same physical address by performing the attack. Here, address a' can be a kernel address or a non-mapped address and not be directly accessible to the adversary. In the *setup* step, the adversary flushes the candidate collision address, a. The state changes to $MS_P\{R_{SC}[\{a\}{=}miss]\}$. In the *interact* step, the adversary prefetches the address a', and performs an access violation. The access violation is due to speculative dereferencing of kernel-space registers from user-space [155], and not because of the prefetch instruction as suggested by Gruss et al. [65]. The state can change to $MS_E\{R_{SC}[\{a'\}{=}hit]\}$. In the *receive* step, the adversary accesses address a and based on the timing variations infers if there is a match between a and a'.

The root causes exploited by the attack are determinism, sharing, access violation and information flow, as shown in Table 1. Determinism enables all the three steps of the attack. First, in *setup* step, it causes the cache line corresponding to a to be evicted. Second, in the *interact* step, it causes the prefetch of a' to be encoded through the *MS* of the *SC*. Third, it makes timing variation to be observed corresponding to the presence/absence of the cache line. Access violation enables the attack by permitting the adversary to prefetch inaccessible address(es). Sharing and information flow guarantees that the secret is encoded and communicated, through *MS* of the *SC*.

In the second category in SW-based attacks, 1B, the attacks [72] use a SW-controlled prefetch instruction (*PREFETCHW*), in the *setup* and the *receive*, to reveal cryptographic keys through the (coherence) state-dependent timing variation for prefetch instructions.

The attacks in category 2A, HW-based without permissions check, use HW prefetchers to prefetch addresses outside of a sand-box [181] or in kernel-space [37]. The prefetcher is trained in the *setup* step, in order to issue a prefetch to the target address in the *interact* step. Chen et al. [37] show that an adversary trained prefetcher can prefetch kernel addresses. In [181] a data memory-dependent prefetcher (DMP) is trained to perform out-of-bounds reads on pointers, since the prefetcher is allowed to use memory content to prefetch irregular address patterns. The root causes exploited by the attacks are determinism, sharing, access violation and information flow. In both attacks, the state of the prefetcher and *SC* are primed to access secrets in the *interact* step and transmit them

through *SC* state leading to timing difference in the *receive* step. Access violation is exploited in the *interact* step since the prefetch is issued without permission checks.

Lastly, the attacks [37], [161] in category 2B, leak secrets through secret data-dependent prefetching pattern(s). Both conflict-based [41] and reuse-based attacks [37] have been explored using prefetch tables entries in this category. Similar to the previously discussed attacks using SW-based prefetching, which also leak secrets through secret/state dependent prefetch access patterns, the root causes are determinism, sharing and information flow.

3.1.3. Branch prediction. Branch predictors record history of branch outcomes in order to predict the direction of control flow after a branch instruction, to improve instruction flow. There are two high-level strategies for attacks using the branch predictor, reuse-based [49], [98], [107] where entries set by one process may influence the other and conflict-based [1]–[3], [47] where contention is used to evict the entry inserted by the other process.

A typical conflict-based attack is JumpOverASLR [47] where the goal of the adversary is to determine the position of a code block in the address space of a victim. Knowing the position of a code block can help break the protection provided by ASLR since the randomization is based on an offset. The attack is launched multiple times using different index values, searching for a collision in the *BTB*. A collision in the *BTB* can be used to infer the address used by the victim and the offset used by ASLR. In the *setup* step, the adversary inserts an entry in the *BTB* which might create a collision with the entry later inserted by the victim code. This changes the *MS* to primed $MS_P\{R_{BTB}[[index_I]{=}addr.\ A]\}$. Next, in the *interact* step, the victim can insert a different target address at the same *BTB* position, creating a collision. The state transitions to $MS_E\{R_{BTB}[[index_I]{=}addr.\ B]\}$. In the *receive* step the adversary executes code which will trigger the *BTB* entry at $index_I$ and measures the execution time. If the *BTB* entry was changed by the victim it would result in a longer execution time since the target address is incorrect (B instead of A).

The root causes which enable this attack are determinism, sharing and information flow, as shown in Table 1. Determinism guarantees that, the adversary can cause a *BTB* state update in an intended entry, induce a conflict on the same entry when the victim executes and measure timing variations due to the conflict. Likewise, information flow and sharing guarantee that the secret is encoded and communicated, through *MS* of the *BTB*, from the victim to the adversary, across the different steps. Note that this attack is restricted to the *SameThread* threat model (although it can be applied in *SMT*) and does not extend to *CrossCore* because *BTB* state is not shared across cores.

There also exist conflict-based attacks which exploit the observation that branch predictions can reveal data-dependent control flow [1]–[3]. For example in [2] secrets are inferred based on the predictions made in the *BTB*.

Reuse-based attacks mostly use the *PHT* instead of the *BTB* [49]. In BranchScope [49], the branch predictor is manipulated into using only the directional branch predictor, *PHT*, where the directional prediction inserted by the adversary is changed by the victim which reveal the direction of conditional branches. The attack can also be used against SGX enclaves, since the *PHT* is shared between processes executing in SGX and outside. The same root causes as in JumpOverASLR enable the attack.

3.1.4. Computational simplification. Computational simplification comprises techniques which eliminate or simplify instruction execution. One example is the zero-skip multiplier and the same principle can be applied on different instruction types such as square root, AND/OR and to different pipeline stages. Attacks on this type of optimizations have been studied [13], [40], [62]. In [13] an attack is described using subnormals in a floating-point division unit, to create visible timing differences. Großschädl et al. [62] describe an attack using early-termination of multiplication where the multiplication is terminated when all remaining digits are zero, creating observable timing differences. The goal of the attack is to leak secret keys from cryptographic SW such as RSA. There are two prerequisites of the attack: Firstly, that the adversary is able to control the plaintext which will be encrypted and, secondly, that the timing can be observed on a side-channel. In the *setup* the adversary calls the cryptographic function on the victim with a plaintext. In the *interact* step the victim encrypts the plaintext and will experience different timings depending on the values of the key. Großschädl et al. do not describe which side-channel could be used in order to allow the adversary to observe the timing difference. We note that either the *MS* of the *SC* or execution unit contention could be used. A gadget is likely needed at the victim for re-encoding of the secret to the side-channel.

The root cause exploited is determinism which enables the data-dependent behaviour of the early-termination optimization and the timing variability. In addition, the side-channel, which enables the adversary to observe the timing differences, exploits sharing and information flow.

3.2. Transient attacks

We finally discuss transient execution attacks which exploit speculative out-of-order (OoO) execution to execute code transiently (i.e., executed but never committed). We use the categorization provided in related works [32], which divide the transient attacks broadly into two groups, speculation-based [4], [19], [23], [36], [51], [88], [98], [121], [125], [143], [154] and exception-based [97], [113], [126], [165], [173], [176], [177], [179] attacks.

3.2.1. Speculation-based attacks. The attacks that fall in this category exploit transient execution, due to branch prediction and/or address/value speculation, to access the secret. An overview of the attacks is shown in Table 1. Spectre v2 [98] is a typical speculation-based attack. The attack exploits an indirect branch to execute a gadget which interacts with the secret in the victims protection domain, leaving a trace in the *MS*. The prerequisites are an indirect branch that can be (mis)trained and a known gadget in the victim's binary that can be manipulated to interact with the secret. The threat models are *SameThread* and *SMT*, since *BTB* is a resource private to a core. However, *CrossCore* can be used if a gadget is used to make the victim perform the (mis)training. In the *setup* step

the adversary/victim (mis)trains the *BTB* to insert a new entry containing the address of the gadget for the indirect branch. The state changes to $MS_P\{R_{BTB}[[index_{target}]$=addr. $A_{gadget}]\}$. The root causes in the *setup* step are determinism, sharing and information flow. Determinism guarantees that the adversary can cause a *BTB* state update in an intended entry, while sharing and information flow enables the state change caused in the *BTB* to be observed by the victim. Note that setup of the *SC* is also performed, i.e., *clflush*, since it will be used later in the *transmit* step.

Next, in the *interact* step, the victim executes the gadget speculatively, accesses the secret and changes the *MS*. The root causes are determinism, access violation and information flow. Determinism guarantees that the (mis)trained *BTB* entry is used. Access violation enables access to the secret through execution of the gadget which results in temporary violation of program semantics, i.e., instructions that access the secret are executed and are later squashed. In the *transmit* step, the secret is re-encoded in the state of the *SC*, by issuing load(s) to the target address(es) by the victim. The root causes exploited are determinism and sharing and information flow, since the *SC* contains the cache line(s) and the *MS* of *SC* is shared between the adversary and the victim. Finally, in the *receive* step the adversary accesses the target cache line(s) and based on the timing variations infers the secret. The root causes are the same as in the previous step, with the difference that determinism ensures observable timing variations based on the state of the *SC*, i.e., the presence/absence of the target cache line(s).

In contrast to Spectre v2, which uses the *BTB*, other Spectre variants exploit different microarchitectural resources to manipulate the control flow, e.g., *PHT* [98], the Return Stack Buffer (*RSB*) [102], [121]. In addition, address speculation has been targeted for manipulating Store-To-Load (*STL*) forwarding that happens in the Load-Store Queue (*LSQ*) [125]. Many of these attack variants use *SC* as the side-channel for transmission.

Next, we discuss attacks which are more restrictive since other microarchitectural resource(s) (not *SC*) is/are used for the transmission of the secret. One example is BranchSpec [88] where the *PHT* is used in the *transmit* and *receive* step. The root causes exploited by the attack are the same as in Spectre v2, while the threat model is more restrictive since the *PHT*, used as the side-channel, is not shared between cores. Another attack, SMoTherSpectre [23], uses port contention to encode the secret and transmit to a co-running SMT thread. Likewise, temporal contention in the floating-point division unit is exploited in SpectreRewind [51]. Like SpectreRewind, the attack proposed by Behnia et al. [19] shows that the secret can be encoded by affecting the timing and order of older instructions, which are issued before the secret dependent instruction(s) in program order. This is in contrast to prior works [93], [108], [192] that focus on studying the secret-dependent effect on younger instructions, issued after the secret dependent instruction(s). In the attack, the non-speculative instructions timing is affected either through the miss status handling register (*MSHR*) or execution unit contention. As an example, let's consider the attack using the *MSHR* described by Behnia et al. [19]. In the *setup* step the adversary evicts a number of cache lines Y. In the *interact* step, a gadget, depending on the secret value, either issues independent loads to the cache lines Y (filling up the *MSHR* entries) or issues loads to the same cache line (using one entry in the *MSHR*). When the target victim load occurs it will experience different timing depending on the *MS* of the *MSHR*.

3.2.2. Exception-based attacks. The attacks that fall in this category exploit transient execution, due to delayed exception handling to access the secret. Meltdown [113] is a typical attack from this category where the adversary exploits transient execution due to delayed exception handling, to read arbitrary kernel memory. In the *setup* step, the adversary causes an exception by accessing a kernel address that resides in a kernel memory page without suitable permissions causing a page fault, e.g., PF-US. Because of the deferred exception handling the execution continues transiently. In the *interact* step, the adversary executes code that uses the loaded value from the faulting kernel address. Suppressing the exception enables the transient execution to continue [113]. In the *transmit* step, the secret is encoded in the *MS* of the *SC*, through a load to the data buffer, in order for the adversary to retain the information after the transient execution is rolled back after exception handling. In the *receive* step, the secret is inferred from the state of the *SC*, through the presence/absence of cache line(s). Note that all the steps of the attack are performed by the adversary.

The root causes enabling the attack are determinism, sharing, access violation and information flow (Table 1). Determinism ensures transient execution due to delayed exception handling in the *setup* step and that the secret is encoded in the state of the *SC* in the *interact* step. Sharing enables the access from the adversary to the victim kernel address in the *interact* step i.e., the adversary has access to a pointer that points to the location of the (kernel) data. Access violation enables the adversary, in the *interact* step, to access kernel data which it does not have the right privileges to access. Information flow enables the transiently accessed secrets to be communicated to the non-transient execution, through the *MS* of the *SC*.

Attacks have shown that different types of exceptions, i.e., page fault (PF), can be used in the *setup* step. For instance, Foreshadow uses PF-P [176], [188], Spectre v1.2 [97] uses PF-RW while LazyFP [165] uses #NM (device not available). Other types of page fault exceptions can also be used as shown by Canella et al. [32]. Furthermore, in addition to reading from kernel memory, attacks have also exploited delayed exception handling to leak data across addresses spaces, virtual machines and even from secure enclaves [176], [188].

Another group of attacks, referred to as the microarchitectural data sampling (MDS) attacks, exploits the state of internal buffers in the CPU, as the Line Fill Buffers (*LFBs*) [152], [179] or the Store Buffer (*SB*) [126], in conjunction with delayed exception handling. Specifically, the attacks leverage the observation that values from these buffers can be leaked as a consequence of accesses that trigger an exception. In the ZombieLoad v1 attack [152] the adversary uses the kernel virtual address (k) corresponding to the user-space address of the victim (u) where the secret resides. Both virtual addresses k and u map to the same physical address s. In the *setup* step, the adversary monitors the victim by performing repeated

flush+reload attacks on the address corresponding to the instruction just before the loading of the secret. This enables the attacker to synchronize with the victim and determine when it can start accessing the state of the buffers to retrieve the secret. In addition, the contents of the data buffer are also flushed from the *SC*. Next, in the *interact* step, the victim performs the load of the secret key, *load u*. This load operation will cause the secret to be inserted in the *LFB* (on a cache miss). The adversary performs a faulting load i.e. the ZombieLoad, to the kernel address of the secret (*load k*), which causes the adversary to retrieve the secret from the *LFB*. In the *transmit* step, the adversary uses the secret as an index to a data buffer to encode the secret into the *MS* of the *SC*, before the transient execution is rolled back. In the *receive* step, the adversary accesses the data buffer entries to infer the secret based on the timing variations arising due to *SC* state. In contrast to Meltdown, victim's accesses cause the secret to be inserted in the internal buffers which are then leaked by loads that trigger exceptions. All four root causes enable this attack. Determinism enables all the steps of the attack while sharing the *MS* of the *LFB* allows for information flow between the victim and the adversary. Access violation occurs during transient execution when the adversary is allowed to read stale data from the *LFB*. Unlike the MDS attacks discussed previously, Load Value Injection (LVI) [177] uses the different types of exception-based vulnerabilities to inject data/code and control victim's execution by controlling the values in the internal CPU buffers. This attack exploits the same root causes as the MDS attacks discussed previously.

3.3. Vulnerable optimization

We discuss potential attacks on value prediction which have not been explored in literature [43], [149].

Value prediction (VP) is a speculative optimization that aims to increase instruction-level parallelism (ILP) and hide memory access latency by predicting values for load misses and consequently breaking instruction dependencies [109], [110], [137], [157], [159]. Accurate predictions can improve ILP by increasing the overlap between memory access(es) and useful computation(s) while mispredictions lead to pipeline squashes and re-execution of instruction(s). In a nutshell, VP is implemented using table-based structures (microarchitectural resource) which sample history to enable prediction.

Reuse-based and conflict-based attacks have been recently demonstrated [43], [149] on VP. Reuse attack variants exploit the reuse behavior whereby the victim trains the predictor leading to the secret being encoded in the predictor state, which is then accessed by the adversary. In conflict attack variants, the victim induces conflicts in the predictor state tables through secret dependent usage, which is monitored by the adversary to infer secrets. The attack steps and the *MS* transitions are similar to those described for the attacks on the branch predictor (albeit on a different resource).

A new possible class of attacks on VP, akin to injection attacks such as LVI [177], involves letting the adversary (mis)train the predictor while simultaneously causing the victim to use the (mis)trained value. This can induce the victim to access a secret transiently (cause an access violation) through a gadget , which can then be leaked to the adversary through a side-channel, such as the *SC*. The root causes exploited by the aforementioned attack are determinism, sharing, access violation and information flow. Determinism permits the *MS* of the prediction table to be accessed and manipulated by the attacker and accessed by the victim. Sharing permits the *MS* of the prediction tables to be accessible to both the adversary and the victim while information flow is enabled through shared *MS* between adversary and victim. Access violation is exploited since the predictions cause the adversary and/or the victim to access data from outside the intended protection domain. This class of attacks on the VP can be used to achieve a similar effect as Spectre v2 and Spectre v4 variants since predicted load values can be used to manipulate control flow and induce secret dependent accesses. Furthermore, defenses tailored for the *BTB* and *STL* forwarding may not be effective at thwarting attacks using the VP. We discuss potential defenses for these attacks in Section 5.

4. Systematization of Defenses

We present a systematization of defenses against attacks targeting different transient and non-transient microarchitectural optimizations. We discuss optimizations for which several attacks and defenses exist in literature: cache, prefetching, branch prediction, computational simplification and transient execution attacks. For each of the defenses, we discuss which root cause(s) and attack step(s) the defenses target. Table 1 show the different root causes for each attack, which can be targeted by a defense. We also categorize defenses into groups, wherever possible, in case there are similarities. In addition, we also describe the protection level offered against the discussed attacks using the resources and the threat model(s) targeted by the defense. Typically, eliminating any of the root causes, exploited in an attack, in any of the attack steps can provide protection only for specific microarchitectural resource(s) targeted under specific threat model(s). Note that a defense can provide protection against an attack while still leaking (some) information. We therefore classify a defense as fully protected only when there is no leakage through the resource(s) targeted by an attack. Our goal is not to exhaustively cover defenses against all possible attacks targeting a microarchitectural optimization. Rather, it is to explore broad defense strategies, in which attack step and root cause they can be applied and their limitations.

4.1. Defenses against non-transient attacks

4.1.1. Cache. Several defenses have been proposed for securing the shared cache against side-channel attacks. We have classified defenses for the *SC* based on the root cause(s) and attack step(s) they target, into different categories, see Table 2. Each row in the table groups defenses based on the root causes restricted in the different attack step(s) and the targeted microarchitectural resource(s).

Disabling *clflush* [193], [194] only addresses reuse-based attacks which typically uses the instruction in the *setup* step. This affects the three root causes determinism, sharing and information flow primarily in the *setup* step.

The defenses in the next category, partitioning (part.), target sharing and information flow, in all the attack steps, by providing isolation between processes/threads.

Defense	Res.	Attack step				Threat model	P
		A_S	A_I	A_T	A_R		
disable clflush [193], [194]	R_{SC}	D/S/I	-	-	-	1,2,3	◐
part.: [44], [71], [114], [186]	R_{SC}	D/S/I	D/S/I	-	D/S/I	1,2	◐
[94], [150], [156], [185]	R_{SC}	D/S/I	D/S/I	-	D/S/I	1,2	◐
part.: static [27], [96], [132]	R_{SC}	D/S/I	D/S/I	-	D/S/I	1,2	●
rand. [116], [141], [142], [172]	R_{SC}	D/I	D/I	-	D/I	1,2	◐
[167], [189]	R_{SC}	D/I	D/I	-	D/I	1,2	◐
repl. [45], [90], [92], [115], [146], [186]	R_{SC}	D/I	D/I	-	D/I	1,2	◐
const. time [21], [22], [40]	-	-	D/I	-	-	1,2,3	●

TABLE 2. DEFENSES FOR ATTACKS USING THE *SC*. PROTECTION (P): FULL●/PARTIAL◐.

[27], [71], [73], [94], [96], [114], [131], [132], [136], [150], [156], [185], [186]. Partitioning can, in principle, provide full protection against reuse-based, conflict-based and observation-based attacks where victim and adversary share the *SC* state. However, the defenses provide different levels of protection depending on the level of isolation, i.e., whether all or only some of the data is partitioned. For example, MI6 and IRONHIDE [27], [132] statically partition both *SC* and DRAM and can thereby enable full protection of the *SC*, albeit at a higher performance cost. In contrast, STEALTHMEM [94] only provides partial protection for a limited number of cache lines per core, which are not allowed to be evicted. This will lead to higher performance but also a lower protection level, since the state of the unprotected cache lines are shared.

The next category, randomization (rand.), targets conflict-based attacks and is typically achieved by modifying the mapping of addresses to sets in the *SC* [116], [141], [142], [167], [172], [189]. Randomization target determinism and information flow. The strategy affects determinism by complicating the process of creating an eviction-set needed to evict a target cache line at a specific address. The change in mapping leads to limited information flow. This makes randomization effective especially against conflict-based attacks. However, in spite of defense, the *SC* is still shared and insertions by the adversary can result in evictions for the victim process and vice versa. This limits the effectiveness against observation-based attacks since the working-set size of an application can be observed and can be leveraged by attacks [54].

The category replacement-based defenses (repl.) – insertion and/or eviction – leverage randomization to provide protection [45], [90], [92], [115], [146], [186]. These proposals mainly target determinism and information flow through the *SC* state. Specifically, through randomising insertion and/or eviction, the *SC* do not react in the same way under the same preconditions. Information flow is limited by reducing/avoiding set conflicts, i.e., by preventing an adversary from evicting data inserted by the victim. These defenses offer protection but eventually leak information since determinism, sharing and information flow in the *SC* are not completely eliminated [166].

Lastly, constant-time programming paradigm [21], [22], [40] can be used to avoid data-dependent implementations which affects determinism and information flow. However, this is challenging to utilize in practice since it cannot be generically applied.

4.1.2. Prefetching. Existing defenses to protect against prefetch-based attacks can be categorized broadly into different groups, as shown in Table 3. Disabling the prefetcher [37], [41], [111], [161] impacts determinism, sharing, access violation and information flow, in all the attack steps. This strategy is equally applicable to attacks

Defense	Res.	Attack step				Threat model	P
		A_S	A_I	A_T	A_R		
disable [37], [41], [111], [161]	$R_{\text{Pref.}}$	D/S/I	D/S/A/I	-	D/S/A/I	1,2,3	●
privilege checks [65]	$R_{\text{Pref.}}$	-	A	-	A	1,2,3	◐
kernel/user isol. [64]	R_{TLB}	-	S/A/I	-	-	2	◐
flush [37], [41]	$R_{\text{Pref.}}$	I	I	-	I	2	◐
replicate [37], [41]	$R_{\text{Pref.}}$	S/I	S/I	-	S/I	2	◐
const. time [22], [40], [59], [161]	-	-	D/I	-	-	1,2,3	◐
SC defenses	R_{SC}	D/S/I	-	-	D/S/I	1,2,3	◐

TABLE 3. DEFENSES FOR ATTACKS USING THE PREFETCHER.

using HW- or SW-based prefetching. However, the performance cost can be high since prefetching can provide significant speedups.

The defense in the second group, by Gruss et al. [65], propose introducing privilege checks on prefetch instructions. This would cause a segmentation fault when there is an attempt to prefetch kernel data. This prevents access violation in the *interact* step and affect all SW- and HW-based attacks that exploit the lack of permission checks [37], [65], [111], [149], [181].

Another strategy is to provide stronger isolation between kernel and user-space to protect against attacks that leverage the lack of permission checks [64]. This approach has been adopted in both Linux [56] and Windows [89]. This would provide protection against SW-based attacks using the prefetch instruction [65], [111] and against HW-based attacks [37], [149], [181]. This approach restricts sharing and information flow through the page tables (and TLB), in the *interact* step of the attack.

The next group of defenses targets attacks that exploit the state of HW-based prefetchers (prefetch tables). Specifically, these defenses replicate and flush prefetcher state at context switches [37], [41]. This ensures that state is no longer shared across context switches which restricts sharing and information flow. This only affects the HW-based attacks which rely on *MS* in the prefetcher. Furthermore, the threat model targeted is limited to *SameThread* since the technique does not affect concurrently executing SMT threads sharing prefetcher state.

Another strategy is to change the SW implementation to ensure that any prefetch activity is not dependent on any secret. This would affect the attacks relying on data-dependent execution paths and observing prefetch patterns [37], [72], [161]. This can be achieved using constant-time programming practices [21], [161], for example, rewriting table based look-ups to be immune to prefetches [59], [161]. This strategy affects determinism and information flow and can theoretically protect against attacks that exploit prefetch patterns. However, it is challenging to implement this broadly in practice.

Lastly, it should be noted that some of the attacks rely on the shared state of the *SC* [37], [65], [72], [161], [181]. The defenses proposed for the *SC* could be used to defend against these attacks as well.

4.1.3. Branch prediction. The defenses against branch-prediction based attacks can be grouped into five categories, see Table 4. The first group relies on SW-based techniques, as if-conversion, where the compiler restructures code to avoid conditional branches and use predication instead [38]. This restricts determinism, sharing and information flow since branching is avoided and is akin to disabling the direction prediction. However, the applicability to real-world code with complex control flow is limited [49]. Furthermore, highly predictable branches have been shown to perform poorly when if-converted [38].

Defense	Resource	Attack step				Threat model	P
		A_S	A_I	A_T	A_R		
if-conversion [38], [49]	-	D/S/I	D/S/I	-	D/S/I	2,3	●
enc.: [106]	$R_{\text{BTB}}/R_{\text{PHT}}$	D/I	D/I	-	D/I	2,3	◐
[48], [61], [199]	$R_{\text{BTB}}/R_{\text{PHT}}$	D/I	D/I	-	D/I	2,3	◐
flush [183], [199]	$R_{\text{BTB}}/R_{\text{PHT}}/R_{\text{BHB}}$	I	I	-	I	2	●
part. [183], [199]	$R_{\text{BTB}}/R_{\text{PHT}}$	S/I	S/I	-	S/I	2	◐
rand. [79], [198]	$R_{\text{BTB}}/R_{\text{PHT}}$	D	D	-	D	2,3	◐

TABLE 4. DEFENSES FOR ATTACKS USING THE BRANCH PREDICTOR.

The next category of defenses use randomization to thwart attacks. Specifically, encryption of the *BTB*/*PHT* have been proposed [48], [61], [106] to prevent the adversary from easily manipulating the branch prediction logic. Encryption restricts determinism and information flow, in all the steps of the attack. Determinism is affected in the *setup* step because the target is encrypted which makes manipulating collisions difficult. Information flow is also hindered since a process can only access correct entries in the presence of a valid key. The limitation of these encryption-based solutions is that they cannot guarantee protection against brute-force approaches.

Defenses in the next category flush the state of the branch predictor, i.e., *BTB*/*PHT*/*BHB*, on context switches [183], [199]. This would affect information flow in the context of the *SameThread* threat model. However, the performance overhead is usually high [183] and the effectiveness is restricted to the *SameThread* model.

Partitioning has been shown to thwart attacks on the branch predictor [49], [199]. Partitioning affects sharing and information flow, in all the steps of the attack, since the state of the *BTB*/*PHT* is isolated. HyBP [199] combines isolation and encryption, and selectively replicates parts of the predictor state, while using encryption for the larger tables. The focus of these proposals is to use partitioning/replication to avoid the high performance cost of flushing the entire branch prediction state upon a context switch. Replicating the entire prediction state among SMT threads, although a possibility, is prohibitively expensive.

The last category makes the state transition of the predictor probabilistic, by affecting the saturating counters, as proposed by Zhao et al. [198]. This defense restricts determinism in all the steps of the attack. However, the protection offered by the technique is limited since an adversary, through repeated measurements, can eventually infer the secret from the state of the branch predictor.

4.1.4. Computational simplification. A few strategies have been proposed for protecting against attacks using computational simplification, see Table 5. One strategy is to selectively disable the optimization for parts of the program which accesses sensitive information [40], [62]. Disabling restricts determinism in the *interact* step.

The other strategy is to change the implementation to avoid any data-dependent timing variations, even for computational simplification. This strategy targets determinism in the *interact* step. One way to achieve data-independent implementation is to use constant-time programming practices [13], [21], [22], [40]. In [13] an FP library (LibFTFP) is shown, providing a fixed-point data type with all library operations executing in constant time.

Defense	Resource	Attack step				Threat model	P
		A_S	A_I	A_T	A_R		
disable [40], [62]	$R_{\text{MUL}}/R_{\text{FPU}}$	-	D	-	-	2,3	●
const. time: [14], [40], [62]	$R_{\text{MUL}}/R_{\text{FPU}}$	-	D	-	-	2,3	◐
[13], [21], [22], [144]	$R_{\text{MUL}}/R_{\text{FPU}}$	-	D	-	-	2,3	◐

TABLE 5. DEFENSES FOR ATTACKS USING COMP. SIMPLIFICATION.

This approach may, in some cases, preclude the benefit from the optimization altogether.

Lastly, in the case of the browser-based attack [13] the SW-construct which enables the sharing and information flow, can be disabled i.e. cross-origin SVG-filters [100].

4.2. Defenses against Transient Attacks

4.2.1. Defenses against speculation-based attacks. The defenses against speculation-based attacks can be broadly grouped into four high-level categories based on the defense strategy: localized defenses, disabling defenses, restriction defenses and isolation defenses, see Table 6.

Localized defenses leverage the defenses available for individual optimizations/resources that interact with speculative execution, such as the BP and/or *SC*. Branch prediction can be targeted in the *setup* step to stop the adversary from being able to (mis)train the predictor, see Section 4.1.3. The defense is only applicable for the attacks which uses the specific predictor resource. Another approach is to target the side-channel that permits information flow across protection domain through transient execution. In most attacks the *SC* is used since it can offer the highest bandwidth, which makes the defenses in Section 4.1.1 applicable. However, studies have demonstrated that a wide variety of microarchitectural resources can be used, such as execution units, ports, *PHT*, *MSHR* etc. This is a challenge because multiple resources may need to be protected, since they can all act as potential side-channels. By defending the resource which enables the easily accessible (high bandwidth and low noise) side-channels, the attack bandwidth can be reduced. In addition, limiting the threat model (for instance to *CrossCore*) can restrict the number of resources which can be used as side-channels.

The second category involves disabling the optimization. This approach has been proposed in specific scenarios where other defenses are not applicable. Support for disabling indirect branch predictions using a barrier, Indirect Branch Predictor Barrier (IBPB) [9], [11], has been adopted in commodity HW. Likewise, to thwart Spectre v4 attack, the STL mechanism can also be disabled using Speculative Store Bypass Disable (SSBD) microcode updates from Intel and AMD [9], [10]. These techniques affect the root causes determinism, sharing and information flow in the predictor, in all the steps where it is used. However, the performance cost is potentially high since the predictions are restricted. Another defensive measure is to limit the threat model by disabling SMT [58], [74], [105]. However, disabling SMT comes at a potentially high performance cost.

Defense	Resource	Attack step				Threat model	P
		A_S	A_I	A_T	A_R		
local: BPU	R_{BTB}/R_{PHT}	D/S/I	-	-	-	2,3	◐
SC	R_{SC}	D/S/I	-	-	D/S/I	1,2	◐
disable: IBPB [9], sb [15]	R_{BTB}/R_{PHT}	D/S/I	-	-	-	1,2,3	◐
SSBD [9], [10]	R_{STL}	D/S/I	-	-	-	1,2,3	◐
SMT [58], [74], [105]	-	D/S/I	D/S/I	-	D/S/I	3	●
restrict: IBRS,STIBP [9], [11]	R_{BTB}/R_{PHT}	D/S/I	-	-	-	1,2,3	◐
retpoline [12], [85]	R_{RSB}	I	-	-	-	1,2,3	◐
randpoline [28]	R_{RSB}	D	-	-	-	1,2,3	◐
lfence, serial. [9]	-	-	D/I	-	-	1,2,3	◐
fence: [103], [170], [180]	$R_{\mu OP\_Q}$	-	D/I	-	-	1,2,3	◐
[33], [130], [160]	-	-	D/I	-	-	1,2,3	◐
delay: [16], [52], [151], [187]	R_{ROB}, $R_{reg.}$	-	D/I/A	-	-	1,2,3	◐
[29], [196]	R_{ROB}, $R_{reg.}$	-	D/I/A	-	-	1,2,3	◐
[108], [118], [148]	R_{ROB}, $R_{reg.}$	-	-	D/I	-	1,2,3	◐
rollback [119], [147]	R_{SC}	-	-	D/I	D/I	1,2	◐
isolate: [5], [6], [93], [192]	R_{buf}/R_{L0}	-	-	D/S/I	-	2	◐

TABLE 6. CLASSIFICATION OF DEFENSES FOR TRANSIENT ATTACKS.

The third category is restriction-based defenses. Here, the high-level idea is to restrict the speculative execution, selectively, to avoid the attacks. Speculation restriction can be performed in the different steps of the attack and using different mechanisms in HW and/or SW. One HW mechanism, adopted in commodity HW to avoid (mis)training, is Indirect Branch Restricted Speculation (IBRS) [9], [11], which affects the *setup* step. Using IBRS restricts the training of indirect targets inside an enclave. Likewise, using Single Thread Indirect Branch Prediction (STIBP) restricts the use of prediction entries trained in another SMT thread. A recent attack [17] has shown that isolation defenses only affects the *BTB* while the Branch History Buffer (*BHB*) is still shared and exploitable. Speculative execution have also been restricted using microcode updates [84]. SW-based defenses [12], [28], [85] also aim to restrict speculation by avoiding the state transition $MS_I \rightarrow MS_P$, and preventing the speculation triggered by the the branch prediction. Here, the (exploitable) indirect branch is replaced by a retpoline sequence which will cause the code to execute a controlled loop sequence until speculation has been resolved. To lower the performance cost of the technique both a probabilistic variant, randpoline [28], and a HW variant [12] have been proposed. The root cause exploited is information flow. A recent paper, RETBLEED [190], has shown that the retpoline strategy only provides partial protection, since it can be circumvented using manipulation of return instructions.

Another mechanism to restrict speculation is to introduce fences to limit transient execution. Many existing attack mitigations use the serializing *lfence* instruction before sensitive parts of the code. In order to improve the usability and performance cost, defenses have been proposed which selectively and automatically choose when to use fences, either in SW [103], [160], [170], [180] or in HW [170]. For example Shen et al. [160] split code into small blocks and insert fences between the entry point and a potentially leaking memory access to defend against Spectre attacks. The root causes affected are determinism and information flow, depending on the solution. Another example is Context-Sensitive Fencing (CSF) [170] which uses customized decoding from instructions to micro-ops to insert fences after a conditional branch instruction and before a subsequent load instruction. The root causes affected by fences are determinism and information flow, in the *interact* step. However, Ren et al. [145] show that lfence can be circumvented since the younger instructions are fetched and then stalled, leaving a footprint in the micro-op cache. Another way similar to fences is Speculative Load Hardening (SLH) [33] a compiler-level technique where the idea is to introduce a data dependency on the condition, in order to guarantee that the control flow is valid. The technique is supported in LLVM and GCC [46]. Oleksenko et al. [130] restrict speculation by introducing a data dependency in order to guarantee that a load will only start if the comparison is in registers or L1 cache. However, the technique is only effective if the load is performed after the comparison.

Another method to restrict speculations is to wait for authorization or until data is no longer transient [29], [52], [108], [118], [148], [151], [187] which affect the execution in the *interact* and/or *transmit* steps. Here, the defenses delay to avoid the access violation which leads to leakage (in *interact*) or stall the load which would update *MS* of *SC* (in *transmit*). For example NDA (Non-speculative Data Access) [187] provide different policies for controlling control flow and data propagation in *interact*. The root causes affected are determinism, sharing and information flow. SpectreGuard [52] also affects the *interact* step and proposes to mark secret data and selectively restrict speculation only for data from sensitive pages. These defenses affect the root cause determinism, access violation and/or information flow in the *interact* step. CondSpec [108] affects *transmit* by handling loads differently, a load that hits in the *SC* can read the data and complete its execution while a load that experience a cache miss will be stalled and re-issued later. This affects the root cause access violation and/or information flow. CleanUpSpec [147], on the contrary, restricts how speculative updates are encoded in the *MS* of the *SC*. Speculative accesses are allowed to progress and make changes to the *SC* but these are removed in case of miss-speculation [119]. This has been shown to be insufficient in certain conditions [19]. This defense affects the root causes determinism and information flow, in the *transmit* and *receive* step of the attack.

The performance cost of the restriction-based defenses depends on how restrictive the rule set is, i.e., how much execution differs from the unconstrained speculation scenario. Introducing protections at a later attack step generally leads to more flexibility, since more speculation can be allowed, and comes at a lower performance cost [80]. The trade-off is that more possible side-channels can be used for the state transitions $MS_I \rightarrow MS_T$ and $MS_T \rightarrow MS_R$, which makes ensuring full protection challenging.

The last category targets isolation by introducing a shadow structure to hold the speculative *MS* until it is deemed safe [5], [6], [93], [192]. For example, in MounTrap [6] an L0 filter cache is used for speculative data, which is only allowed to propagate to the rest of the cache hierarchy after commit. This allows the speculation and potential access violation to occur but not affect *MS* of the *SC*, for example. This restricts information flow from the transient execution to the non-transient execution.

4.2.2. Defenses against exception-based attacks. Exception-based attacks typically exploit implementation oversight in HW. The Meltdown attack exploits a race condition between authorization and access [75] which enables transient execution to continue with an unauthorized value, leading to access violation. Newer CPUs contain patches whereas existing ones are mostly protected through microcode updates or other workarounds [30]. For instance, the issue has been addressed on newer Intel microarchitectures [9], Whiskey Lake and onward, by returning zero when accessing privileged memory [31]. In the case of the MDS-attacks the leaks are attributed to a use-after-free vulnerability where stale data is read in the internal registers [152], allowing unintended information flow through shared CPU buffers. The issue is solved by flushing the internal buffers to restrict the information flow. Similarly, to defend against LazyFP, FPU registers are flushed on context switches when changing protection domains with SGX, for hypervisors and for logical cores. In addition, since Linux 4.6 eager FPU switching is used by default [120]. This disables the fault since the FPU is always available. Foreshadow has been mitigated on

Intel CPUs through setting a physical page number field of unmapped page-tables to refer to non-existing physical memory [83], [188], thereby restricting access violation. The LVI attacks [177] are no longer possible when the corresponding fault or microcode assist is mitigated.

Another defense strategy is to provide stronger address space isolation [55], [64], [82], [101], which mitigates or limits the possible access violation in the attacks. For example MemoryRanger [101] isolates drivers, kernel and user space into separate address spaces using Extended Page Table (EPT). This defense restricts access violation in the *interact* step, by providing isolation.

4.3. General defense strategies

Broader system-level strategies for providing protection have also been proposed. Leveraging constant-time programming paradigm is one such strategy where the key idea is to rewrite the SW-implementation to avoid timing variability. The challenge with constant-time programming is to provide protection for all types of attacks. Another strategy is to decrease the accuracy of timing measurement [39], [81], [99], [123]. One recent proposal by Cook et al. [39] introduces noise in the timer. This strategy affects determinism in the *receive* step and can be used as a defense against several attacks on different optimizations. The strategy leads to lower attack bandwidth but has been shown to be ineffective in providing complete protection since many attacks can amplify the timing difference [125] or use other timer mechanisms [153]. Another strategy is to leverage detection to identify/document an attack and/or apply defenses selectively [53]. Several mechanisms for detecting attacks on different microarchitectural optimizations have been proposed [168], [193], [197]. Leveraging detection based strategy can lower the performance cost of applying a specific defense since the defense only need to be enabled when an attack is detected. However, prior work have shown that it is very complex to detect all attacks especially as new attacks are discovered and attack behaviour can be tailored to bypass detection [53]. Another approach is to classify sensitive data and only apply the defence selectively [44], [185]. Another strategy is to use formal models [35], [68], [69], [75], [128] in order to automatically synthesize vulnerabilities. This strategy could potentially find all exploitable scenarios in the design phase itself. However, identifying all vulnerabilities complicate the model building process and current proposals therefore only target a limited set. Additionally, leveraging programming language based security [104] is another option wherein defenses are co-designed leveraging programming language annotations to mark sensitive parts of the programs, along with suitable HW primitives. However, a recent analysis [129] found that most programming languages and their execution environments does not have support for Spectre mitigations. This shows the challenge of relying on programming languages and execution environments to provide complete protection. Finally, a promising strategy is to combine defenses with different characteristics and/or explore cross-layer approaches spanning hardware and software support to target multiple root causes as a way to achieve better performance/security trade-off. HyBP [199], for instance, combines partitioning and flushing to secure the branch predictor, by targeting both sharing and information flow while CATalyst [114] uses HW/SW support to isolate sensitive data with low overhead.

Microarchitectural optimizations	Defense	Performance overhead (%)
Cache	part.	DAWG 0-15 [96], MI6 16.4 [27]
	part.	IRONHIDE -20 [132], CATalyst 0.5-0.7 [114]
	rand.	CEASAR 1.1 [141], NewCache -0.5-1 [116]
	repl.	NoMo 0.2-0.8 [45], Mirage 2.2 [146]
Prefetch	flush	0.2 [37]
Branch prediction	part./flush	HyBP 0.2-0.6 [199]
	rand.	17.17 [79], 0.2-2.2 [198]
Comp. simp.		if-conversion 0-24 [40]
Transient execution		SSBD 2-8 [10], retpoline 5-10 [85]
		serial. 62-74.8 [33], SLH 29-36.4 [33]
		NDA 10.7-125 [187], SpecShield 10-20 [16]
		ConTExT 0.1-71.1 [151], STT 8.5-27 [196]
		CondSpec 6.8-53.6 [108], DOLMA 10.2-42.2 [118]
		InvisiSpec 5-17 [192], SafeSpec -3 [93]
		MounTrap -5-4 [6], CleanupSpec 5.1 [147]

TABLE 7. PERFORMANCE OVERHEADS FOR DEFENSES. A NEGATIVE VALUE INDICATES PERFORMANCE IMPROVEMENT.

4.4. Performance overheads

The performance overheads reported by some of the existing work for defenses that target different microarchitectural optimizations are shown in Table 7. We report a range for cases where performance differs between defense variants and/or benchmarks. Note that the numbers cannot be directly compared since they are measured on different configurations, simulators, while running different benchmarks and assuming diverse threat models. In general, the reported performance overhead varies widely across different defenses that attempt to thwart the same attack. Furthermore, the overhead is mostly lower for defenses that focus on a specific microarchitectural resource such as the cache, prefetch and branch prediction (PHT, BTB) whereas the overhead, in general, is larger for defenses against transient execution attacks that severely restrict speculative execution or disable it altogether. Additionally, the overheads for transient execution defenses vary widely, even for the same defense, based on the targeted threat model. We believe that to facilitate a clear comparison of the overheads across defenses and make insights, it is essential to standardize the evaluation methodologies used in the community.

5. Discussion

Commonalities: We have shown that the root causes for attacks are common, across a wide range of microarchitectural optimizations. Furthermore, our analysis of the defenses show that these target one or more root cause, across the different steps of an attack. There are commonalities even in the defense strategies used to protect against attacks on these diverse microarchitectural optimizations. Some of those include disabling the optimization to restrict all the root causes; isolating the state related to the optimization, to restrict sharing; applying randomization and/or restriction to limit information flow and introducing permission checks to limit resources from exposing/accessing state outside of the intended domain.

Using these common strategies together with the root causes enable us to envision new defenses for vulnerable microarchitectural optimizations. For instance, we can apply these common strategies to defend against potential attacks exploiting value prediction (Section 3.3). Possible new hardware defense strategies, that have not been

explored in literature [43], [149], include flushing the table at context switches, isolating and/or partitioning the table to avoid conflict and reuse-based attacks, introducing randomization to limit information flow, selectively disable the optimization when running sensitive parts of the program or combining some of the aforementioned defenses to improve performance/security trade-off.

Observations: Firstly, vulnerabilities are not always due to the fundamental behaviour of the microarchitectural optimization but are rather a result of the design and/or the implementation. This points in the direction of promoting an understanding of the root causes and the potential defense strategies in the design and implementation phases, rather than as an afterthought.

Secondly, the ease of exploiting a microarchitectural optimization and the severity of the leak vary. There exist limiting factors which are not easy to quantify, such as the availability and capabilities of gadgets [32]. The link between attack bandwidth and effectiveness in practical scenarios is also difficult to quantify.

Finally, we think that the simplicity and the abstract nature of the framework can aid designers and security researchers, exploring vulnerabilities, by providing a model to easily reason about attacks and defenses on microarchitectural optimizations.

Future research directions: Our focus has been on understanding the root causes behind vulnerabilities in microarchitectural optimizations that target performance and the defense strategies that can be used to protect against them. There are several interesting avenues for future work leveraging our framework. An important line of future work is to perform implementation specific analysis focusing on specific resources/optimizations in Intel, AMD and ARM architectures. We expect that our framework can help in the process of identifying vulnerabilities in microarchitectural optimizations/resources. Another line of work would be to analyze attacks and defenses on optimizations and resources that we have not explored (such as NoC and DRAM). We believe our framework to be versatile to accommodate such an analysis. Lastly, investigating and/or extending the root cause framework to include microarchitectural optimizations for security, such as Intel SGX, performance degradation attacks [7], [8], [18], [63] and power-based side channel attacks are interesting directions for future work.

6. Related Work

Prior works have analyzed attacks and defenses for different microarchitectural optimizations and can broadly be categorized into two groups. In the first group, focusing on transient attacks [30], [32], [76], [80], [191], Canella et al. [32] provide a taxonomy for classification of transient attacks based on Spectre and Meltdown attack variants, analyse attacks on specific microarchitectural resources across Intel/AMD/ARM architectures and examine gadget prevalence. Xiong et al. [191] also survey transient attacks with an emphasis on whether the adversary/victim is triggering the different attack steps and the threat models used. In addition, they discuss the sources of transient execution and discuss hardware mitigation strategies.

In the second group, considering non-transient attacks [53], [166], Ge et al. [53] focus on threat models and how it impacts microarchitectural resource sharing. This is used as the basis to categorize attacks on different resources and present common defense strategies. Szefer [166] attempts to identify key features of functional units (microarchitectural resources) which makes the side/covert channels possible. In addition, they classify attacks based on whether the attacker/victim is operating in a virtualised setting or not. However, the key features identified are a mix of some root causes, attack steps and threat model. This makes it hard to clearly distinguish the root causes from others and limits the utility of their approach for analysis of diverse optimizations.

We advance the understanding of attacks and defenses by building on related work that analyses attack steps [32], [76], [80], [191], the location of adversary/victim [32], [53], [191], prevalence of gadgets [32], [191], and the importance of sharing [53], [166], [191]. Our contributions are; 1) We provide an abstract model of the architecture and the microarchitectural state transitions involved in the different steps of an attack. 2) Using this as a framework, we identify four root causes which enable timing-based side-channel attacks across diverse microarchitectural optimizations. 3) We show that the existing defenses for different optimizations can be classified as targeting one or more of the identified root causes and observe that similar defense strategies can be applied across optimizations. 4) We review several transient and non-transient attacks and defenses to demonstrate the versatility of our framework. We believe that our framework can aid in the understanding the attack and defense landscape and help with designing secure optimizations.

7. Conclusions

Microarchitectural optimizations will play an increasingly important role as the cadence of technology scaling is expected to slow down. However, recent advancements have demonstrated that they introduce security vulnerabilities that can be exploited by attacks. It is therefore crucial to understand the causes behind why optimizations are vulnerable to attacks and what strategies can be used to defend against them. We provide a simple framework to analyze attacks on a wide range of microarchitectural optimizations and use that to systematize both transient and non-transient attacks and defenses, highlighting the similarities and differences. We identify four root causes — determinism, sharing, access violation and information flow — that enable timing-based side-channel attacks across different microarchitectural optimizations. Based on our analysis we discuss potential attacks and defenses for a vulnerable optimization that have not been explored in literature. We believe that our framework can assist with the understanding of the landscape of attacks and defenses across diverse microarchitectural optimizations and in providing guidance for designing secure microarchitectural optimizations.

8. Acknowledgements

We thank the anonymous reviewers for their insightful comments which helped improve the paper. This research has been funded by the Swedish Research Council (VR) under the PRIME project (registration no. 2019-04929).

References

[1] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, page 312–320, New York, NY, USA, 2007. Association for Computing Machinery.

[2] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'07, page 225–242, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in openssl and necessary software countermeasures. In Steven D. Galbraith, editor, *Cryptography and Coding*, pages 185–203, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[4] Pavlos Aimoniotis, Christos Sakalis, Magnus Själander, and Stefanos Kaxiras. Reorder buffer contention: A forward speculative interference attack for speculation invariant instructions. *IEEE Computer Architecture Letters*, 20(2):162–165, 2021.

[5] Sam Ainsworth. Ghostminion: A strictness-ordered cache system for spectre mitigation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 592–606, New York, NY, USA, 2021. Association for Computing Machinery.

[6] Sam Ainsworth and Timothy M. Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 132–144. IEEE Press, 2020.

[7] Alejandro Cabrera Aldaya and Billy Bob Brumley. Hyperdegrade: From ghz to mhz effective cpu frequencies, 2021.

[8] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 422–435, New York, NY, USA, 2016. Association for Computing Machinery.

[9] AMD. Speculative execution side channel mitigations, 2018, 2018.

[10] AMD. Amd speculative bypass store disable, technical report, 2022.

[11] AMD. Software techniques for managing speculation on amd processors, 2022, 2022.

[12] Nadav Amit, Fred Jacobs, and Michael Wei. {JumpSwitches}: Restoring the performance of indirect branches in the era of spectre. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 285–300, 2019.

[13] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*, pages 623–639, 2015.

[14] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1369–1382, New York, NY, USA, 2018. Association for Computing Machinery.

[15] ARM. Arm architecture reference manual armv8, 2018, 2018.

[16] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. Specshield: Shielding speculative data from microarchitectural covert channels. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 151–164, 2019.

[17] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against Cross-Privilege spectre-v2 attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 971–988, Boston, MA, August 2022. USENIX Association.

[18] Michael G Bechtel and Heechul Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention, 2019.

[19] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. Speculative interference attacks: Breaking invisible speculation schemes. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 1046–1060, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Daniel J. Bernstein. Cache-timing attacks on aes. 2005.

[21] Daniel J. Bernstein. The poly1305-aes message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption*, pages 32–49, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[22] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[23] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 785–800, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Rahul Bodduna, Vinod Ganesan, Patanjali SLPSK, Kamakoti Veezhinathan, and Chester Rebeiro. Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser. *IEEE Computer Architecture Letters*, 19(1):9–12, 2020.

[25] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC leak: Architecturally leaking uninitialized data from the microarchitecture. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3917–3934, Boston, MA, August 2022. USENIX Association.

[26] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. Casa: End-to-end quantitative security analysis of randomly mapped caches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1110–1123, 2020.

[27] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. Mi6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 42–56, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Rodrigo Branco, Kekai Hu, Ke Sun, and Henrique Kawakami. Efficient mitigation of side-channel based attacks against speculative execution processing architectures, September 14 2021. US Patent 11,119,784.

[29] Gianpiero Cabodi, Paolo Camurati, Fabrizio Finocchiaro, and Danilo Vendraminetto. Model checking speculation-dependent security properties: Abstracting and reducing processor models for sound and complete verification. In Claude Carlet, Sylvain Guilley, Abderrahmane Nitaj, and El Mamoun Souidi, editors, *Codes, Cryptology and Information Security*, pages 462–479, Cham, 2019. Springer International Publishing.

[30] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. Evolution of defenses against transient-execution attacks. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, GLSVLSI '20, page 169–174, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. Kaslr: Break it, fix it, repeat. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, page 481–493, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 249–266, USA, 2019. USENIX Association.

[33] Chandler Carruth. Rfc: Speculative load hardening (a spectre variant1 mitigation), 2018, 2018.

[34] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. Sok: Practical foundations for software spectre defenses, 2021.

[35] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 288–28815, 2019.

[36] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 142–157, 2019.

[37] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. Leaking control flow information via the hardware prefetcher, 2021.

[38] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin-Fook Ngai. The impact of if-conversion and branch prediction on program execution on the intel® itanium™ processor. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, page 182–191, USA, 2001. IEEE Computer Society.

[39] Jack Cook, Jules Drean, Jonathan Behrens, and Mengjia Yan. There's always a bigger fish: A clarifying analysis of a machine-learning-assisted side-channel attack. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 204–217, New York, NY, USA, 2022. Association for Computing Machinery.

[40] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE Symposium on Security and Privacy*, pages 45–60, 2009.

[41] Patrick Cronin and Chengmo Yang. A fetching tale: Covert communication with the hardware prefetcher. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 101–110, 2019.

[42] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. Don't mesh around: Side-Channel attacks and mitigations on mesh interconnects. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2857–2874, Boston, MA, August 2022. USENIX Association.

[43] Shuwen Deng and Jakub Szefer. New predictor-based attacks in processors. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 697–702, 2021.

[44] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. *HYBCACHE: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments*. USENIX Association, USA, 2020.

[45] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Archit. Code Optim.*, 8(4), January 2012.

[46] R Earnshaw. Mitigation against unsafe data speculation (cve-2017-5753, 2018.

[47] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[48] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *ACM Trans. Archit. Code Optim.*, 13(1), mar 2016.

[49] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *SIGPLAN Not.*, 53(2):693–707, mar 2018.

[50] Anders Fogh. Negative result: Reading kernel memory from user mode, july 28 2017, 2017.

[51] Jacob Fustos, Michael Bechtel, and Heechul Yun. Spectrerewind: Leaking secrets to past instructions. In *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*, ASHES'20, page 117–126, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Jacob Fustos, Farzad Farshchi, and Heechul Yun. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[53] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.*, 8(1):1–27, 2018.

[54] Daniel Genkin, William Kosasih, Fangfei Liu, Anna Trikalinou, Thomas Unterluggauer, and Yuval Yarom. Cachefx: A framework for evaluating cache security, 2022.

[55] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. Lazarus: Practical side-channel resilient kernel-space randomization. In Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 238–258, Cham, 2017. Springer International Publishing.

[56] Thomas Gleixner. x86/kpti: Kernel page table isolation (was kaiser), 2017, 2017.

[57] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. *Speculative Probing: Hacking Blind in the Spectre Era*, page 1871–1885. Association for Computing Machinery, New York, NY, USA, 2020.

[58] Google. Product status: Microarchitectural data sampling (mds), 2022.

[59] Vinodh Gopal, James Guilford, Erdinç Öztürk, Wajdi Feghali, Gil Wolrich, and Martin Dixon. Fast and constant-time implementation of modular exponentiation. 01 2009.

[60] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, Baltimore, MD, August 2018. USENIX Association.

[61] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A. Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, and Ankit Ghiya. Evolution of the samsung exynos cpu microarchitecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 40–51, 2020.

[62] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In Donghoon Lee and Seokhie Hong, editors, *Information, Security and Cryptology – ICISC 2009*, pages 176–192, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[63] D. Grunwald and S. Ghiasi. Microarchitectural denial of service: insuring microarchitectural fairness. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, pages 409–418, 2002.

[64] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 161–176, Cham, 2017. Springer International Publishing.

[65] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 368–379, New York, NY, USA, 2016. Association for Computing Machinery.

[66] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299, Cham, 2016. Springer International Publishing.

[67] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive Last-Level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., August 2015. USENIX Association.

[68] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: principled detection of speculative information flows. *CoRR*, abs/1812.08639, 2018.

[69] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware/software contracts for secure speculation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, S&P 2021. IEEE, 2021.

[70] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505, 2011.

[71] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. *IVcache: Defending Cache Side Channel Attacks via Invisible Accesses*, page 403–408. Association for Computing Machinery, New York, NY, USA, 2021.

[72] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. Adversarial prefetch: New cross-core cache side channel attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1458–1473, 2022.

[73] S. Kariyappa Gururaj Saileshwar and Moinuddin Qureshi. Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning. 2021.

[74] Red Hat. Simultaneous multithreading in red hat enterprise linux, 2022.

[75] Zecheng He, Guangyuan Hu, and Ruby Lee. New models for understanding and reasoning about speculative execution attacks, 2020.

[76] Zecheng He, Guangyuan Hu, and Ruby Lee. New models for understanding and reasoning about speculative execution attacks. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 40–53, 2021.

[77] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, jan 2019.

[78] Mark D. Hill. Technical perspective: Why 'correct' computers can leak your information. *Commun. ACM*, 63(7):92, jun 2020.

[79] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. Mitigating branch-shadowing attacks on intel sgx using control flow randomization. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, SysTEX '18, page 42–47, New York, NY, USA, 2018. Association for Computing Machinery.

[80] Guangyuan Hu, Zecheng He, and Ruby B. Lee. Sok: Hardware defenses against speculative execution attacks. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 108–120, 2021.

[81] W.-M. Hu. Reducing timing channels with fuzzy time. In *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 8–20, 1991.

[82] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. EPTI: Efficient defence against meltdown attack for unpatched VMs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 255–266, Boston, MA, July 2018. USENIX Association.

[83] INTEL. L1 terminal fault, 2018, 2018.

[84] INTEL. Affected processors: Transient execution attacks & related security issues by cpu, 2022.

[85] Intel. Retpoline: A branch target injection mitigation, technical report, 2022.

[86] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes. In *2015 IEEE Symposium on Security and Privacy*, pages 591–604, 2015.

[87] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, pages 299–319, Cham, 2014. Springer International Publishing.

[88] Md Hafizul Islam Chowdhuryy, Hang Liu, and Fan Yao. Branchspec: Information leakage attacks exploiting speculative branch instruction executions. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 529–536, 2020.

[89] Ken Johnson. Kva shadow: Mitigating meltdown on windows, 2018, 2018.

[90] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. Ric: Relaxed inclusion caches for mitigating llc side-channel attacks. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.

[91] Mehmet Kayaalp, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[92] Georgios Keramidas, Alexandros Antonopoulos, Dimitrios N. Serpanos, and Stefanos Kaxiras. Non deterministic caches: a simple and effective defense against side channel attacks. *Des. Autom. Embed. Syst.*, 12(3):221–230, 2008.

[93] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[94] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, Bellevue, WA, August 2012. USENIX Association.

[95] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *SIGARCH Comput. Archit. News*, 42(3):361–372, jun 2014.

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

[97] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses, 2018.

[98] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[99] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 463–480, Austin, TX, August 2016. USENIX Association.

[100] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 69–81, Vancouver, BC, August 2017. USENIX Association.

[101] Igor Korkin. Divide et impera: Memoryranger runs drivers in isolated kernel spaces, 2018.

[102] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *Proceedings of the 12th USENIX Conference on Offensive Technologies*, WOOT'18, page 3, USA, 2018. USENIX Association.

[103] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Speccfi: Mitigating spectre attacks using cfi informed speculation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 39–53. IEEE, 2020.

[104] Dexter Kozen. Language-based security. In *International Symposium on Mathematical Foundations of Computer Science*, pages 284–298. Springer, 1999.

[105] M. Larabel. Openbsd disabling smt/hyper threading due to security concerns, 2022.

[106] Jaekyu Lee, Yasuo Ishii, and Dam Sunwoo. Securing branch predictors with two-level encryption. *ACM Trans. Archit. Code Optim.*, 17(3), aug 2020.

[107] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 557–574, USA, 2017. USENIX Association.

[108] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 264–276, 2019.

[109] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 226–237, 1996.

[110] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, page 138–147, New York, NY, USA, 1996. Association for Computing Machinery.

[111] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD prefetch attacks through power and time. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[112] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of amd's cache way predictors. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, page 813–825, New York, NY, USA, 2020. Association for Computing Machinery.

[113] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. Meltdown: Reading kernel memory from user space. *Commun. ACM*, 63(6):46–56, may 2020.

[114] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, 2016.

[115] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 203–215, 2014.

[116] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, 2016.

[117] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Comput. Surv.*, 54(6), jul 2021.

[118] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. Dolma: Securing speculation with the principle of transient non-observability. In *USENIX Security Symposium*, 2021.

[119] Jason Lowe-Power, Venkatesh Akella, Matthew K. Farrens, Samuel T. King, and Christopher J. Nitta. Position paper: A case for exposing extra-architectural state in the isa. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[120] A. Lutomirski. x86/fpu: Hard-disable lazy fpu mode, june 2018., 2018.

[121] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2109–2122, New York, NY, USA, 2018. Association for Computing Machinery.

[122] Giorgi Maisuradze and Christian Rossow. Speculose: Analyzing the security implications of speculative execution in cpus, 2018.

[123] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 118–129, 2012.

[124] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-cores cache covert channel. In Magnus Almgren, Vincenzo Gulisano, and Federico Maggi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64, Cham, 2015. Springer International Publishing.

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

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

[127] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *Int. J. Parallel Program.*, 47(4):538–570, aug 2019.

[128] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. Axiomatic hardware-software contracts for security. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 72–86, New York, NY, USA, 2022. Association for Computing Machinery.

[129] Amir Naseredini, Stefan Gast, Martin Schwarzl, Pedro Miguel Sousa Bernardo, Amel Smajic, Claudio Canella, Martin Berger, and Daniel Gruss. Systematic analysis of programming languages and their execution environments for spectre attacks, 2021.

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

[131] Hamza Omar, Brandon D'Agostino, and Omer Khan. Optimus: A security-centric dynamic hardware partitioning scheme for processors that prevent microarchitecture state attacks. *IEEE Transactions on Computers*, 69(11):1558–1570, 2020.

[132] Hamza Omar and Omer Khan. Ironhide: A secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 111–122, 2020.

[133] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1406–1418, New York, NY, USA, 2015. Association for Computing Machinery.

[134] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[135] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the cpu on-chip ring interconnect are practical, 2021.

[136] D. Page. Partitioned cache architecture as a side-channel defence mechanism, 2005. page@cs.bris.ac.uk 13017 received 22 Aug 2005.

[137] Arthur Perais and André Seznec. Eole: Combining static and dynamic scheduling through value prediction to reduce complexity and increase performance. *ACM Trans. Comput. Syst.*, 34(2), apr 2016.

[138] C. Percival. Cache missing for fun and profit. In *http://www.daemonology.net/papers/ htt.pdf*, pages 974–987, 2005.

[139] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 987–1002, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[140] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic prime+probe attacks and covert channels in scattercache. *CoRR*, abs/1908.03383, 2019.

[141] Moinuddin K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 775–787, 2018.

[142] Moinuddin K. Qureshi. New attacks and defense for encrypted-address cache. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 360–371, New York, NY, USA, 2019. Association for Computing Machinery.

[143] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1852–1867, 2021.

[144] Ashay Rane, Calvin Lin, and Mohit Tiwari. Secure, precise, and fast Floating-Point operations on x86 processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 71–86, Austin, TX, August 2016. USENIX Association.

[145] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. I see dead µops: Leaking secrets via intel/amd micro-op caches. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 361–374, 2021.

[146] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[147] Gururaj Saileshwar and Moinuddin K. Qureshi. Cleanupspec: An "undo" approach to safe speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 73–86, New York, NY, USA, 2019. Association for Computing Machinery.

[148] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 723–735, New York, NY, USA, 2019. Association for Computing Machinery.

[149] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. Opening pandora's box: A systematic study of new ways microarchitecture can leak private data. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 347–360, 2021.

[150] Sercan Sari, Onur Demir, and Gurhan Kucuk. Fairsdp: Fair and secure dynamic cache partitioning. In *2019 4th International Conference on Computer Science and Engineering (UBMK)*, pages 469–474, 2019.

[151] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. Context: A generic approach for mitigating spectre. In *NDSS*, 2020.

[152] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 753–768, New York, NY, USA, 2019. Association for Computing Machinery.

[153] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security*, pages 247–267, Cham, 2017. Springer International Publishing.

[154] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *Computer Security – ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I*, page 279–299, Berlin, Heidelberg, 2019. Springer-Verlag.

[155] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative dereferencing: Reviving foreshadow. 2021. 25th International Conference on Financial Cryptography and Data Security : FC 2021 ; Conference date: 01-03-2021 Through 05-03-2021.

[156] Brian C. Schwedock and Nathan Beckmann. Jumanji: The case for dynamic nuca in the datacenter. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 665–680, 2020.

[157] André Seznec. Exploring value prediction with the eves predictor. 2018.

[158] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 552–561, New York, NY, USA, 2007. Association for Computing Machinery.

[159] Rami Sheikh, Harold W. Cain, and Raguram Damodaran. Load value prediction via path-based address prediction: Avoiding mispredictions due to conflicting stores. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 423–435, New York, NY, USA, 2017. Association for Computing Machinery.

[160] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. Restricting control flow during speculative execution. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2297–2299, 2018.

[161] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 131–145, New York, NY, USA, 2018. Association for Computing Machinery.

[162] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel.

[163] Olin Sibert, Phillip A. Porras, and Robert Lindell. The intel 80x86 processor architecture: Pitfalls for secure systems, 1995.

[164] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu. Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 955–969, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[165] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels, 2018.

[166] Jakub Szefer. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *J. Hardw. Syst. Secur.*, 3(3):219–234, 2019.

[167] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. Phantomcache: Obfuscating cache conflicts with localized randomization. In *NDSS*, 2020.

[168] Ya Tan, Jizeng Wei, and Wei Guo. The micro-architectural support countermeasures against the branch prediction analysis attack. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 276–283, 2014.

[169] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. SecSMT: Securing SMT processors against Contention-Based covert channels. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[170] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 395–410, New York, NY, USA, 2019. Association for Computing Machinery.

[171] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[172] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, 2018.

[173] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols, 2018.

[174] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *J. Cryptol.*, 23(1):37–71, jan 2010.

[175] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of des implemented on computers with cache. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, pages 62–76, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[176] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 991–1008, USA, 2018. USENIX Association.

[177] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72, 2020.

[178] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 937–954, Baltimore, MD, August 2018. USENIX Association.

[179] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105, 2019.

[180] Marco Vassena, Craig Disselkoen, Klaus V Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically eliminating speculative leaks from cryptographic code with blade. *arXiv preprint arXiv:2005.00294*, 2020.

[181] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. Augury: Using data memory-dependent prefetchers to leak data at rest. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1491–1505, 2022.

[182] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 39–54, 2019.

[183] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M. Al-Hashimi, and Geoff V. Merrett. Brb: Mitigating branch predictor side-channels. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 466–477, 2019.

[184] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. Meshup: Stateless cache side-channel attack on cpu mesh. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1506–1524, 2022.

[185] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Secdcp: Secure dynamic cache partitioning for efficient timing channel protection. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[186] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. *SIGARCH Comput. Archit. News*, 35(2):494–505, jun 2007.

[187] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 572–586, New York, NY, USA, 2019. Association for Computing Machinery.

[188] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018.

[189] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 675–692, Santa Clara, CA, August 2019. USENIX Association.

[190] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3825–3842, Boston, MA, August 2022. USENIX Association.

[191] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Comput. Surv.*, 54(3), may 2021.

[192] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441, 2018.

[193] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 347–360, New York, NY, USA, 2017. Association for Computing Machinery.

[194] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[195] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *J. Cryptogr. Eng.*, 7(2):99–112, 2017.

[196] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 954–968, New York, NY, USA, 2019. Association for Computing Machinery.

[197] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquin Garcia-Alfaro, editors, *Research in Attacks, Intrusions, and Defenses*, pages 118–140, Cham, 2016.

[198] Lu-Tan Zhao, Rui Hou, Kai Wang, Yu-Lan Su, Pei-Nan Li, and Dan Meng. A Novel Probabilistic Saturating Counter Design for Secure Branch Predictor. *J. Comput. Sci. Technol.*, 36(5):1022–1036, 2021.

[199] Lutan Zhao, Peinan Li, Rui Hou, Michael C. Huang, Xuehai Qian, Lixin Zhang, and Dan Meng. Hybp: Hybrid isolation-randomization secure branch predictor. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 346–359, 2022.