Efficient Invisible Speculative Execution through Selective Delay and Value Prediction

Christos Sakalis
Uppsala University
Uppsala, Sweden
christos.sakalis@it.uu.se

Stefanos Kaxiras
Uppsala University
Uppsala, Sweden
stefanos.kaxiras@it.uu.se

Alberto Ros
University of Murcia
Murcia, Spain
aros@ditec.um.es

Alexandra Jimborean
Uppsala University
Uppsala, Sweden
alexandra.jimborean@it.uu.se

Magnus Själander
Norwegian University of Science and Technology
Trondheim, Norway
magnus.sjalander@ntnu.no

ABSTRACT

Speculative execution, the base on which modern high-performance general-purpose CPUs are built on, has recently been shown to enable a slew of security attacks. All these attacks are centered around a common set of behaviors: During speculative execution, the architectural state of the system is kept unmodified, until the speculation can be verified. In the event that a misspeculation occurs, then anything that can affect the architectural state is reverted (squashed) and re-executed correctly. However, the same is not true for the microarchitectural state. Normally invisible to the user, changes to the microarchitectural state can be observed through various side-channels, with timing differences caused by the memory hierarchy being one of the most common and easy to exploit. The speculative side-channels can then be exploited to perform attacks that can bypass software and hardware checks in order to leak information. These attacks, out of which the most infamous are perhaps Spectre and Meltdown, have led to a frantic search for solutions.

In this work, we present our own solution for reducing the microarchitectural state-changes caused by speculative execution in the memory hierarchy. It is based on the observation that if we only allow accesses that hit in the L1 data cache to proceed, then we can easily hide any microarchitectural changes until after the speculation has been verified. At the same time, we propose to prevent stalls by value predicting the loads that miss in the L1. Value prediction, though speculative, constitutes an invisible form of speculation, not seen outside the core. We evaluate our solution and show that we can prevent observable microarchitectural changes in the memory hierarchy while keeping the performance and energy costs at 11% and 7%, respectively. In comparison, the current state of the art solution, InvisiSpec, incurs a 46% performance loss and a 51% energy increase.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISCA '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322216

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**.

KEYWORDS

speculative execution, side-channel attacks, caches

ACM Reference Format:
Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient Invisible Speculative Execution through Selective Delay and Value Prediction. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3307650.3322216

1 INTRODUCTION

Side-channel attacks that rely on shared microarchitectural state and behavior to leak information have been known to the architecture and the security communities for years. In particular, side-channel attacks on the cache system have been practically demonstrated in many forms for the level-one cache (L1) (when the attacker can share the same core as the target) [4], the shared last-level cache (LLC) (when the attacker can share the LLC) [33, 60], and the coherence protocol (when the attacker can simply be collocated in the same system, under a single coherence domain, with the target) [17]. However, starting with Spectre [23] and Meltdown [29], a new class of *speculative* side-channel attacks have sent a shockwave through the architecture community. *Speculation,* one of the fundamental techniques we have for achieving high performance, proved to be a significant security hole, leaving the door wide open for side-channel attacks to "see" protected data. As far as the instruction set architecture (ISA) and the target program are concerned, leaking the information across a covert side-channel is not illegal because it does not affect the functional, architectural behavior of the program. The stealthy nature of a speculative side-channel attack is based on microarchitectural state being changed by speculation even when the architectural state is not.

The need to hide speculation is unwaning even in the face of the latest and most sophisticated proposals to counteract side-channel attacks. Side-channel attacks based on the *position* of a cache line in the cache (e.g., Flush+Reload [60] or Prime+Probe [33] attacks)

can be confronted with techniques such as randomization or encryption of addresses. Qureshi recently proposed an efficient and performant solution that can provide strong security guarantees for such attacks [44]. However, even such solutions *can do nothing* for timing attacks such as the invalidation side-channel attack [51]. No matter how addresses are randomized or encrypted (and cache lines shuffled around in the caches), in an invalidation-based coherence protocol, a writer still has to locate and invalidate all shared copies. By timing the difference between writes that invalidate and writes that do not, we can "see" speculative loads that accessed specific addresses. In addition, randomization and cache isolation based solutions do not protect against attacks originating from within the same execution context, such as some variants of Spectre [23]. This makes it a priority to search for an approach to make the effects of speculation invisible without unduly affecting performance or energy consumption.

In a breakthrough paper, Yan et al. were the first to propose an approach, referred to as InvisiSpec, to *hide* changes in microarchitectural state due to speculation [59]. InvisiSpec makes the accesses of a speculative load *invisible* in the cache hierarchy (by not changing cache and coherence state) and subsequently verifies correctness and makes changes in the memory hierarchy with a *visible* access when speculation is validated as correct. If the speculation fails to validate, the invisible access (although it occurred and consumed system resources, e.g., bandwidth) leaves no trace behind.

While Yan et al. provide an elegant solution to overlap the visible accesses of the loads [59], the simple fact that the accesses to the memory hierarchy are effectively doubled carries a non-trivial performance and energy cost [46]. The motivation of our work is to examine if a completely different approach to the same problem, one that still maintains the one-request-per-load execution model, can potentially yield better results.

We present a new solution that not only hides speculation, but recovers a significant portion of the performance and energy cost that double-access proposals such as InvisiSpec are bound to pay. To issue just a single access per load we must ensure that the load is non-speculative, otherwise we risk exposing speculative side-effects. Delaying all loads until they are non-speculative proves to be devastating for performance. However, we do not have to. Our solution is based on two simple observations:

(1) For speculative loads that *hit* in the L1, we can allow them to proceed and use the accessed memory, provided that we do not affect the L1 replacement state or perform any prefetches at that time. This keeps speculative hits invisible.
(2) For speculative loads that *miss* in the L1 we use *value prediction* instead of sending a request deeper in the memory hierarchy. Value prediction is completely invisible to the outside world, thereby enabling the load to proceed with a value while keeping its speculation invisible. When the load is in-the-clear and can no longer be squashed by an older instruction (Section 2), we issue a normal request to the memory system that fetches the actual value. At that point, no matter if the value prediction was correct or not, the access is *non-speculative* and *cannot be squashed*. It is, therefore, safe to modify the memory hierarchy state.

We only focus on loads, because stores are kept hidden in the store buffer until they are committed. In our approach, there is only one single request per load that traverses the memory hierarchy and fetches data. While to validate value prediction we must serialize these non-speculative requests, we show that: *i)* the resulting performance cost is relatively low; and *ii)* value prediction helps reduce the cost of waiting for a load to become non-speculative. We compare against the performance and energy costs of simply delaying loads until they are no longer speculative and delaying L1 misses without performing value prediction. To summarize, we propose and evaluate the following:

- **Delaying Loads**: We delay speculative loads until they are non-speculative, preventing any speculative side-effects from happening. We present two different approaches, one where loads are only executed when they reach the head of the reorder buffer (**naïve delay**), and one where we only delay loads until we know they can no longer be squashed (**eager delay**). The latter is based on the Bell and Lipasti conditions for committing instructions [1], which will be discussed in Section 2. We will also describe a structure for efficiently keeping track of these conditions in Section 4.
- **Delaying Only L1 Misses**: We limit the eager delay solution only to loads that miss in the L1 cache, while allowing hits to execute. We refer to this solution as **delay-on-miss**.
- **Value Predicting Delayed Loads**: We augment the delay-on-miss solution with a value predictor (VP), enabling L1 misses to execute using a predicted value. The value predictor is local to the core and does not leak any information during speculative execution, making the loads completely invisible to others.

We compare our proposed solutions with the current state-of-the-art approach, InvisiSpec [59], which works by executing all speculative loads and hiding their side-effects until the speculation has been verified. Simulation results reveal that our proposed, value prediction based solution can provide secure, memory-side-channel-free speculative execution with a performance cost of 11%, significantly outperforming InvisiSpec.

2 SPECULATIVE SHADOWS

For correctness purposes, the architectural side-effects of speculatively executed instructions remain hidden until the speculation can be verified. If a misspeculation is detected, the instructions following the misspeculation point—also referred to as *transient instructions*—are squashed and execution is restarted from the misspeculation point. In practice, on modern out-of-order (OoO) cores, all instructions are considered speculatively executed until they reach the head of the reorder buffer (ROB). This alleviates the need to keep track of which instructions are speculative, reducing the hardware cost and complexity. However, in order to achieve high performance while hiding the *microarchitectural* side-effects of speculatively executed instructions, a more fine-grain approach is required: During dispatch, instructions with potential to cause a misspeculation cast a **speculative shadow** to all the instructions that follow. For example, a branch with an unresolved condition or unknown target address casts a shadow to all instructions that follow it because if the branch is mispredicted, the instructions that

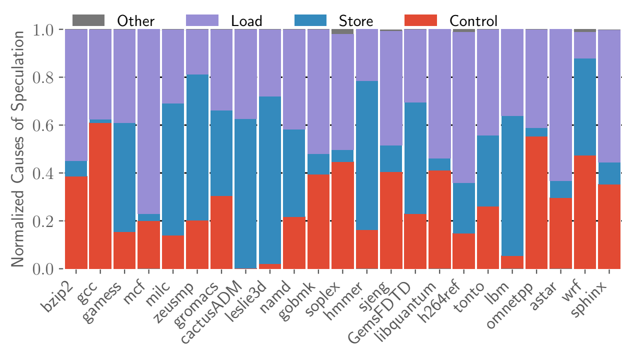

Figure 1: A breakdown of the instructions casting speculative shadows on executed loads.

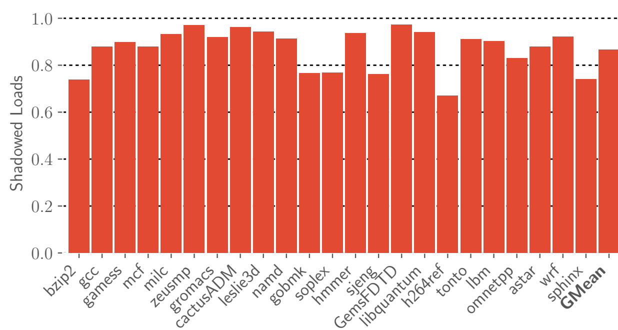

Figure 2: The ratio of loads executed speculatively.

follow it will have to be squashed. We call the instructions under such shadows (speculatively) **shadowed instructions**. As soon as the condition and the branch target are known, the speculation can be verified and the shadow is lifted. We identify the following types of shadows:

- **The E-Shadow**: E-Shadows are cast by memory operations with unknown addresses, arithmetic operations, and any other instructions that can cause an **exception**. The shadow starts when the instruction is dispatched and ends when the absence of an exception can be verified. For memory operations, one can check for exceptions once the address translation has completed and the permissions have been verified. For arithmetic and other instructions that throw exceptions on invalid inputs, checking can be done as soon as the instruction has been executed. Under systems where precise exceptions are not available, this shadow can potentially be ignored, as an exception does not guarantee that the instructions that follow it will not be committed successfully.
- **The C-Shadow**: C-Shadows are cast by **control** instructions, such as branches and jumps, when either the branch condition or the target address are unknown or have been predicted but not yet verified. The shadow starts when the control instruction enters the ROB, and ends when the address and the condition have been verified.
- **The D-Shadow**: D-Shadows are cast by potential memory **dependencies** through stores with unresolved addresses (read-after-write dependencies). Loads can speculatively bypass stores and execute out of order, but they might be squashed if it is later discovered that a store points to the same address as that of a speculatively executed load. The shadow starts when the store enters the ROB and ends when the address is resolved, akin to the E-Shadow for memory operations.
- **The M-Shadow**: Under the total store order (TSO) memory model, or any other model that respects the load-load order, the observable **memory** order of loads needs to be conserved. If the execution of an older load is delayed then younger loads are allowed to execute speculatively but they might be squashed if an invalidation is received, hence causing the M-Shadow. The shadow starts when the older load enters the ROB, and ends when it has finished executing. While there is an overlap with the E-Shadows, the M-Shadow extends further, as the E-Shadow can be lifted after the permission checks. The M-Shadows do not exist under more relaxed memory models, such as release consistency (RC).

In addition to these shadows, interrupts can also halt and possibly divert execution. In this work, we assume that interrupts can be delayed until there are no shadowed loads preceding them. If the opposite is required, e.g. for a real-time system, then all loads would have to remain under a shadow until they reach the head of the ROB.

Figure 1 presents a breakdown of the type of instructions casting shadows over the executed load instructions. The statistics have been gathered over applications from the SPEC CPU 2006 [49] benchmark suite. The hardware parameters of the evaluated system can be found in Table 1. We present the instruction types and not the shadow types, as one instruction can simultaneously cast multiple overlapping shadows. Only the oldest shadow is taken into consideration for these statistics, therefore, eliminating one of the shadow types does not necessarily lead to a proportional decrease in the number of speculatively executed loads. For the E-Shadow, we assume that, other than memory accesses, only integer division operations can throw an exception. Exceptions for the rest of the integer operations are not that common on general-purpose CPUs, so we do not consider them. Additionally, floating point exceptions can usually be configured through special registers or other ISA specific mechanisms. This makes it possible to detect in advance if a floating point operation might throw an exception or not. For our evaluation, we assume that floating point exceptions are disabled. Finally, we assume that the permission bits for memory accesses are verified immediately after the translation has completed. The rest of the causes of exceptions are rare in benchmarks like SPEC CPU, so we can omit them from the simulation without any loss of precision in the evaluation.

Figure 2 displays the ratio of loads executed while under a speculative shadow in each benchmark, based on the conditions discussed previously. The ratio of speculatively executed loads is high for all benchmarks, ranging from 67% (`h264ref`) to 97% (`GemsFDTD`), with a mean of 87%. This strongly indicates that only low-overhead solutions are viable, as the majority of the load operations are affected. A more detailed analysis of the performance implications is provided in the evaluation, Section 5.

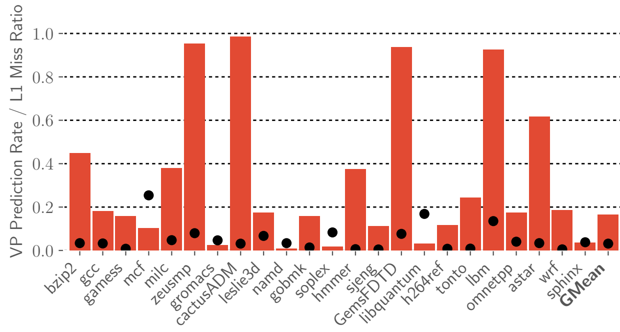

Figure 3: VP prediction rate for shadowed L1 misses (bars), combined with the L1 miss ratio of each benchmark (dots).

3 DELAYING SPECULATIVE LOADS

An intuitive solution for hiding the side-effects of speculative loads is to avoid speculation on loads altogether, i.e., delay the loads until they are no longer speculatively shadowed. We evaluate four different versions of this approach, starting from a very aggressive and then slowly relaxing the delay conditions while still keeping the speculation hidden:

- A **naïve** version where loads are only executed when they reach the head of the ROB.
- An **eager** version where loads are only executed when they are no longer shadowed by another instruction.
- A **delay-on-miss** version where only loads that miss in the L1 cache are delayed. Loads that hit in the L1 cache are satisfied by the cache, and only their side-effects in the memory system (e.g., updating replacement data or prefetching) are delayed.
- A **value predicted** version where, instead of simply delaying them, the delay-on-miss version is extended to provide values for L1 misses through value prediction.

3.1 Naïve Delay

By requiring loads to reach the head of the ROB before they are issued, we can ensure that all loads are executed non-speculatively without the need for additional hardware complexity. On the other hand, this causes all loads to be serialized, thus, making it impossible to take advantage of any memory-level parallelism (MLP) available in the application. Instruction-level parallelism is equally crippled, as loads are necessary in order to feed data to the instructions. In fact, we are only presenting this solution to show that naïvely executing all loads non-speculatively is not viable in OoO CPUs, and to provide a worst-case baseline for the evaluation.

3.2 Eager Delay

Instead of delaying every load until it reaches the head of the ROB, we can take advantage of the Bell and Lipasti conditions [1] and our understanding of the different shadow types (Section 2) to **release** them early. Loads are delayed only until they are no longer shadowed by other instructions, at which point they can be safely issued and executed out-of-order. While this approach still comes with a steep performance cost, it improves significantly over the naïve delay solution (Section 5). Additionally, it performs comparably to InvisiSpec, without the hardware complexity cost of modifying the memory hierarchy or the coherence protocol.

3.3 Delay-on-Miss

Overall, we propose delaying speculative loads to prevent any side-effects from appearing in the memory hierarchy. In comparison, other solutions, such as InvisiSpec, execute the loads and instead try to hide the side-effects until the speculation has been verified. While such an approach works, it requires modifications to the whole cache hierarchy and it incurs a significant performance cost. We observe, however, that we can reduce this cost by reducing the scope of the side-effects that need to be hidden. By delaying L1 misses (but not hits), we eliminate the need for an additional data structure that hides fetched cache lines until they can be installed in the cache. Additionally, since the cache line already exists in the L1 on a hit, no additional coherence mechanisms are required. Instead, we only need to delay the side-effects of the hits, such as updating the replacement data and notifying the prefetcher, which exist outside the critical path of the cache access. The L1 behavior for shadowed loads is as follows:

- In case of an L1 hit, the cache responds to the request but delays any operations that might cause visible side-effects, such as updating the replacement state. The CPU will signal the cache to perform these operations after the speculation has been successfully verified.
- In the case of an L1 miss, the request will simply be dropped. We refer to these L1 misses caused by shadowed loads as **shadowed L1 misses**.

If a load has received data from the L1 while under a speculative shadow, and after it has left that shadow, it will send a release request to the cache, signaling that the cache can now perform any side-effect causing operations it might have delayed. On the other hand, if a load has not received any data after executing under a speculative shadow, it will simply repeat the initial memory request, this time triggering the normal miss mechanisms in the cache.

While in this work we focus on the data caches, the translation lookaside buffers (TLB) can also be utilized as a potential side-channel in an attack. In the SPEC CPU benchmarks that we use for the evaluation, TLB misses are rare, with the majority of the benchmarks having a miss ratio of less than 1%. Therefore, for the remainder of this paper, we will assume that TLB misses are handled with the delay-on-miss mechanism that has been described in this section.

3.4 Value Predicting Delayed Loads

The L1 miss ratio of the applications used to evaluate the solution proposed in this paper ranges from less than 1% to 25% (Figure 3). While the delay-on-miss proposal reduces the number of delayed loads significantly, it still incurs stalls when encountering a shadowed L1 miss. To solve this problem, we propose to continue the execution by predicting the value of L1 misses instead of delaying them.

Value prediction is not a new idea [28], and it has been used, among other things, to help improve the number of instructions-per-cycle (IPC) [27], to handle L2 misses [6], to design a new architecture

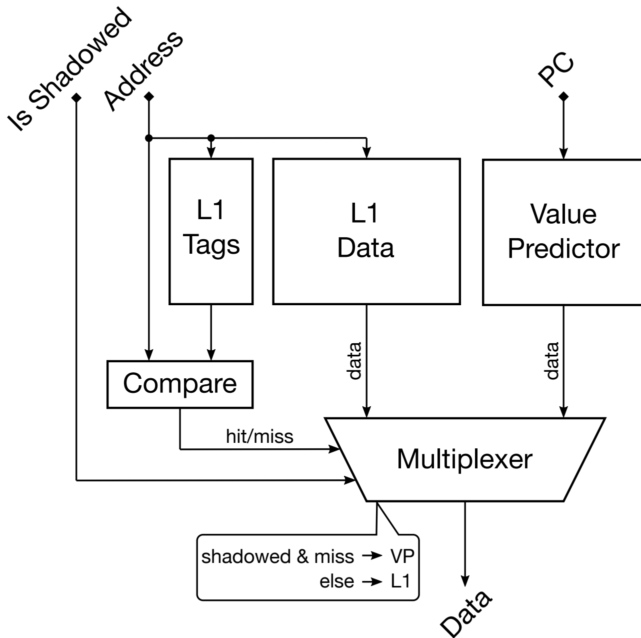

Figure 4: Overview of the interaction between the L1 and the VP, simplified for brevity.

that can bypass the expensive OoO pipeline [40], and even to secure processors from fault attacks [5]. We are only interested in using value prediction to predict loads, and specifically L1 misses. In this context, value predictors have two interesting properties:

(1) The predictor can be local to the core, isolated from other cores or even other execution contexts. The visible state of the predictor is only updated after the prediction is validated.
(2) Because the predicted value needs to be validated with a normal memory access, the predictor can be incorporated into an OoO pipeline with only small modifications to the L1 cache and no modifications to the remaining memory hierarchy and the coherence protocol [34].

The first property implies that the value predictor itself can be used during speculative execution without any visible side-effects. By delaying the memory access used to perform the validation of the prediction, we can value predict loads that miss in the L1 while remaining completely invisible to the rest of the system. In addition, we do not need to introduce any modifications to the cache coherence protocol, as the validation of the predicted values happens only after the speculation shadows have been resolved and it can be issued as a normal load. Figure 4 presents an overview of how this can be achieved. The value predictor is added as part of the L1 cache and is queried in parallel. In case of an L1 miss on a shadowed load, the value is returned by the value predictor instead, if possible. Regardless if the access was resolved from the cache or the value predictor, the cache controller delays any side-effects until the CPU indicates that the speculation has been verified. Since the value predictor is accessed in parallel with the L1, and multiplexing is already necessary for associative caches, we do not introduce any additional delay in the critical path of the cache.

To the best of our knowledge, this is the first solution that takes advantage of the aforementioned properties of value predictors to hide the side-effects of speculative execution in the memory hierarchy and the cache coherence protocol with only small modifications to the former and no modifications to the latter.

With value prediction in place, a new type of speculative shadow is introduced, the **VP-Shadow**. All value-predicted loads cast this shadow until the predicted value can be validated. Currently, because our implementation assumes a system implementing the TSO memory model, the VP-Shadows are completely covered by the M-Shadows. If the M-Shadows were to be eliminated, either by relaxing the memory model or through some other approach [45], then the VP-Shadows could be tracked to a finer granularity and selective replay can be used instead of fully squashing in case of a value misprediction [20, 52]. The VP-Shadows could then be restricted only to the instructions that would have to be selectively squashed and replayed, instead of all younger instructions. With our current proposal, value predicted loads cast a shadow on all instructions until they are validated, and since a validation can only be issued when the load is no longer under a speculative shadow, only one validation at a time can be issued. This restriction can be lifted if the M-Shadow is eliminated and the VP-Shadows are tracked at a finer granularity but evaluating the potential of such a solution is left for future work.

Figure 3 contains the prediction rate for a 13-component VTAGE predictor [41], with 128 entries per component. While we use all loads for training the predictor, it is only queried for shadowed loads that miss in the L1 cache. Other types of value predictors also exist [6, 27, 36, 42], but a thorough evaluation of value prediction mechanisms is beyond the scope of this paper. With the VTAGE predictor we have implemented, the prediction rate varies significantly between applications, ranging from less than 1% to more than 98%, with a mean value of 16%. Our data indicate that *i)* the prediction rate for the shadowed L1 misses is lower than the potential prediction rate of all loads (40%) and that *ii)* delaying the validation of the predictions negatively affects the prediction rate. However, we will see in the evaluation (Section 5) that even with a prediction rate of 16%, significant gains can be achieved. Improving value prediction algorithms is beyond the scope of this work; instead, in order to explore the benefits provided by improving the predictor, we also present results with an oracle predictor that can achieve prediction rates of 50% and 100%.

4 TRACKING SPECULATIVE SHADOWS

To efficiently delay loads or know when a value predicted load can be validated it is necessary to know when the load is no longer covered by a speculative shadow. We make the observation that to answer this question it is enough to know if the oldest shadow casting instruction is older than the load in question. We leverage this to track shadows in a structure similar to a reorder buffer (ROB) but that is much smaller as it only needs to track a single bit per instruction. We call this structure the shadow buffer or SB for short[1]. Shadow-casting instructions (Section 2) are entered into the shadow buffer in the order they are dispatched. Once an instruction no longer causes a shadow, e.g., once a branch has been

[1]Not to be confused with the Speculation Buffer utilized in InvisiSpec.

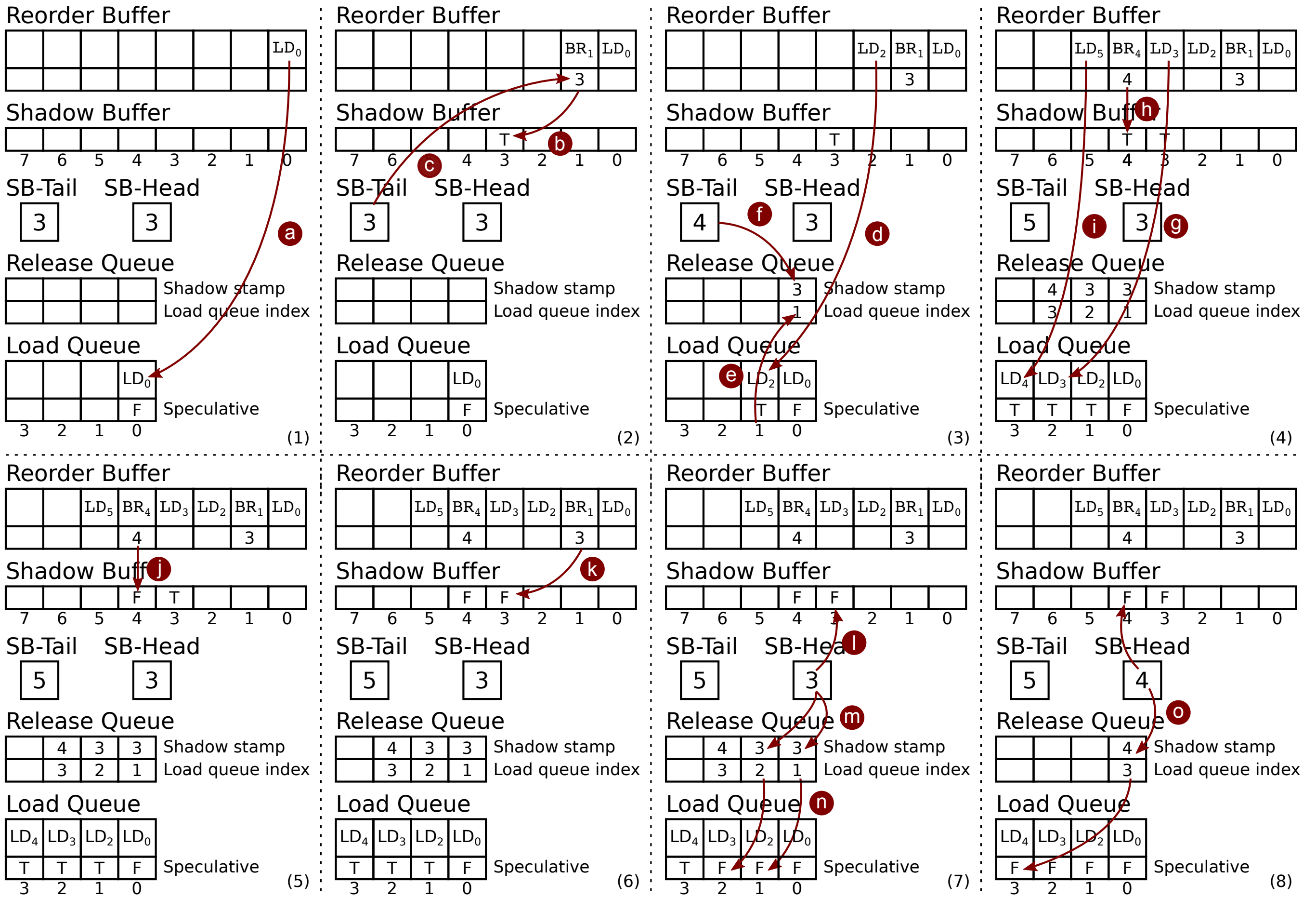

Figure 5: Example of tracking speculative shadows. The values "T" and "F" indicate "true" and "false", respectively.

resolved, then the SB entry is updated. Only when an instruction reaches the head of the SB and no longer casts a shadow does it get removed from the SB. This assures that the oldest shadow-casting instruction is always at the head of the SB and that they exit the SB in program order, similar to how the ROB assures that instructions are committed in order. To determine when a load is no longer covered by a shadow it is enough to *i)* know which was the youngest shadow-casting instruction when the load entered the ROB and *ii)* check for when this shadow-casting instruction has exited the SB. Assume the following code:

```
LD0:  ld r2 [r1]
BR1:  br r2 #0024
LD2:  ld r6 [r3]
LD3:  ld r3 [r3]
BR4:  br r4 #0096
LD5:  ld r1 [r5]
```

Figure 5 shows an example of how the shadows are tracked when executing the code above. The shadow buffer (SB) is implemented as a circular buffer; to avoid using index 0 and 1 for all the structures in the example it is assumed that previous executions have left the SB empty with its head and tail set at 3. The ROB, release queue, and load queue are also assumed to be implemented as circular buffers such that an entry of these structures can always be identified by its index[2]. The SB has as many entries as the ROB and the release queue has as many entries as the load queue to avoid any structural hazards. It might be possible to reduce the area overhead by optimizing these based on the likelihood of how many instructions cause shadows and how many shadowed loads exist. While the example shows a number of distinct steps to clearly illustrate the functionality, in reality many of these can be performed in parallel, similar to how multiple instructions can be committed from the ROB in a single cycle. To further simplify the example, we assume that only branches cast shadows, i.e., we are not considering the E- and M-Shadows of the loads. The example develops as a sequence of nine (one not shown) steps:

(1) When the first load (LD_0) enters the ROB the SB is empty (SB-Head==SB-Tail), indicating that there are no shadow-casting instructions and that the load can be normally executed ⓐ.
(2) The branch (BR_1) causes a shadow until its condition and target have been resolved and is, therefore, inserted into

[2]The head and tail pointers of the reorder buffer, release queue, and load queue are not shown in Figure 5 to simplify the illustration.

the SB ⓑ. The ROB entry of the branch is marked with the index of the SB entry, i.e., the SB-Tail ⓒ. The SB-Tail is then incremented.

(3) When the second load (LD_2) enters the ROB the SB is no longer empty and the load therefore shadowed. It is still entered into the load queue but it is marked as speculative ⓓ, which means that it also needs to be entered into the release queue. There, its index in the load queue ⓔ and the youngest shadow-casting instruction (identified by SB-Tail minus one) ⓕ are marked.

(4) The steps just described are repeated for the following load ⓖ, branch ⓗ, and final load ⓘ.

(5) Once a shadow-casting instruction, in this example the second branch instruction (BR_4), stops casting a shadow, it updates the SB index that is indicated by its ROB entry ⓙ. Nothing happens at this point of time, since the SB-Head is still pointing to an older shadow-casting instruction that has not been executed.

(6) Once the first branch (BR_1) is resolved the SB entry is updated ⓚ.

(7) This triggers a number of events since it is the oldest shadow-casting instruction, as indicated by the SB-Head ⓛ. Before the SB-Head is incremented to indicate that this is no longer the oldest shadow-casting instruction, it is compared with the first entries of the release queue ⓜ. When these match, the load queue entry are updated to indicate that the loads are no longer under a speculative shadow ⓝ. This repeats for every entry in the release queue until a shadow stamp that does not match the SB-Head is encountered.

(8) Once the SB-Head is incremented the entry in the SB is checked, causing the events in step (7) to be repeated and the final load to be marked as non-speculative ⓞ.

(9) Finally, the SB-Head is incremented once more, leaving the head equal to the tail, indicating that the SB is empty and that there are no shadow casting instructions in the ROB (not shown in the illustration).

One big advantage of tracking shadows this way is that it avoids the use of content-addressable memories (CAMs), which are both complex and costly to implement.

5 EVALUATION

We start by explaining our evaluation methodology and then proceed to discuss the memory behavior, performance, and energy usage characteristics of the evaluated solutions. We end by discussing the implications of improving the value-predictor's prediction rate.

5.1 Methodology

We have implemented our proposal and also the current state-of-the-art solution, InvisiSpec, on Gem5 [3], combined with McPAT and CACTI [25, 26] for the energy evaluation. We use the SPEC CPU 2006 benchmark suite [49], with six of the applications excluded due to baseline simulation issues. We model the speculative buffers from InvisiSpec as small caches on McPAT, and the value predictor as a large branch target buffer (BTB). Both are sized appropriately, taking into consideration the amount of storage required both for data and for metadata. The VTAGE predictor is sized pessimistically, assuming that a proper program counter and branch history are stored for each entry. In practice, the storage overhead for metadata can be reduced without compromising the prediction rate [48]. For DRAM, we use the power model built into Gem5, as McPAT does not provide one. We perform the simulations by first skipping one billion instructions in atomic mode and then simulating in detail for another three billion instructions. The characteristics of the simulated system can be found in Table 1. We simulate a system with a private L1 and a shared L2 cache, without L3. The reason why we chose this unusual configuration is to increase the cost of misses in the L2 cache, since the SPEC CPU 2006 workloads are otherwise not memory intensive enough to properly evaluate the cost of the proposed solutions. As the baseline we use a large, unmodified OoO CPU.

Table 1: The simulation parameters used for the evaluation.

Parameter	Value
Technology node	22nm
Processor type	out-of-order x86 CPU
Processor frequency	3.4GHz
Address size	64bits
Issue width	8
Cache line size	64 bytes
L1 private cache size	32KiB, 8-way
L1 access latency	2 cycles
L2 shared cache size	1MiB, 16-way
L2 access latency	20 cycles
Value predictor	VTAGE
Value predictor size	13 components × 128 entries

5.2 Memory Behavior

Since the memory behavior plays a significant role in the performance and energy usage of the application, we begin the evaluation with analyzing the memory behavior of the benchmarks.

Figure 6 presents the L1 data cache miss ratio. Overall, the miss ratio is small, with a baseline mean value of 4%. This is a strong argument for our delay-on-miss solution: While almost all instructions are shadowed when they are issued (Figure 2), only a small percentage of them will have to be delayed or value predicted. On average, the miss ratio of the applications is not affected negatively by any of the evaluated solutions, and it is even improved by the more aggressive delay versions. This is due to the reduced amount of misspeculated memory accesses, which would bring in unneeded data, and also due to the slower execution of the program allowing time for the memory system to respond. InvisiSpec on the other hand incurs a large increase in the miss ratio in some applications, especially in the case of `libquantum`. The increase is caused by the fact that `libquantum` is a streaming application that benefits greatly from prefetching, which is disrupted by InvisiSpec. However, these results can be misleading, as when it comes to the actual number of misses and overall memory accesses happening in the application, the absolute number might change while the ratio remains the same. Instead, Figure 7 features the number of DRAM reads for every application. We can now observe that InvisiSpec incurs a 32% increase in the number of DRAM reads. The effect is particularly

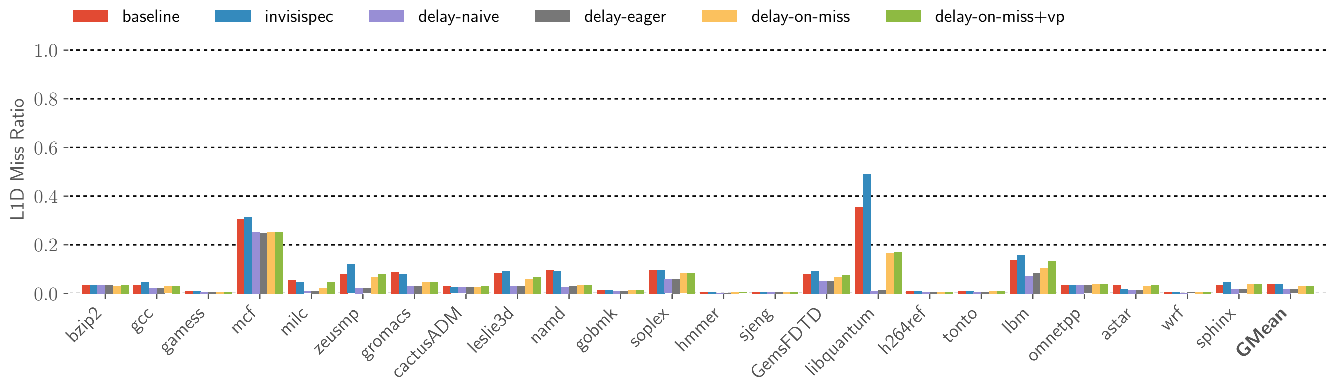

Figure 6: L1D cache miss ratio.

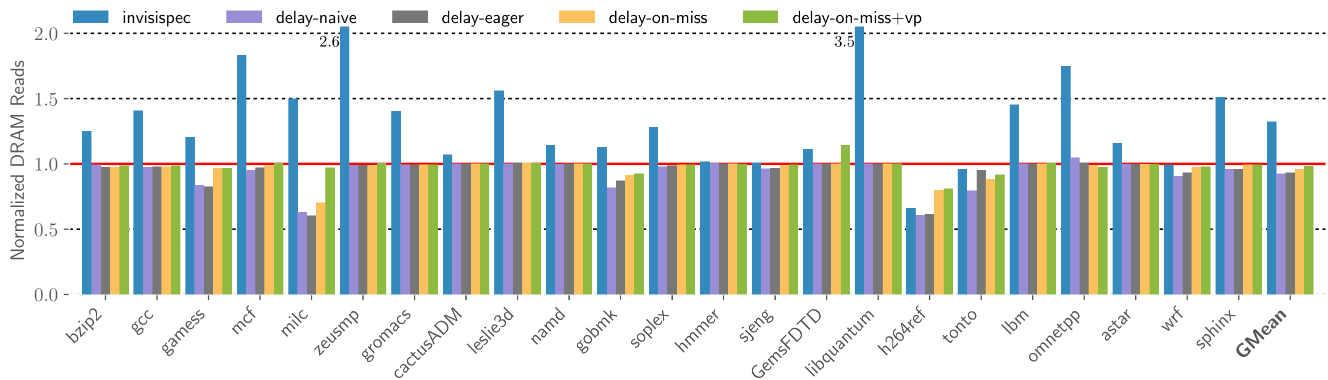

Figure 7: Number of DRAM reads, normalized to the baseline.

prominent in mcf, zeusmp, and as one might expect, libquantum. In contrast, the rest of the evaluated solutions do not have that problem, only marginally exceeding the baseline in one application, GemsFDTD. There is only one case where InvisiSpec behaves better than the delay-on-miss solutions, h264ref, where all approaches reduce the number of DRAM reads below the baseline.

In addition to the potential performance and energy cost, DRAM accesses can also be utilized as a side-channel for attacks [43]. Such attacks are outside the scope of InvisiSpec, but they are covered by our delay-based solutions.

5.3 Performance and Energy

Figures 8 and 9 present the instructions per cycle (IPC) and the energy usage of the benchmarks, respectively. Both are normalized to the baseline. In Figure 9, the bottom shaded part represents the leakage energy of the system, while the top lighter part represents the dynamic energy. The first obvious observation we can make is that the naïve delay version is not a viable solution for high performance CPUs. With a mean performance loss of 74% and an energy increase of 2.4×, it consistently performs worse than any of the other solutions we have evaluated.

For the eager delay case, we can see a significant improvement in performance, with a mean loss of 50%, as well as in the energy usage, with an mean increase of 49% over the baseline. While still far from the baseline, the eager delay version already performs similarly to InvisiSpec, which exhibits a mean performance loss and energy increase of 46% and 51%, respectively. Both solutions have benchmarks in which one outperforms the other, but the eager delay solution is simpler and more secure than InvisiSpec.

Introducing the delay-on-miss optimization, we see a performance improvement of 31 percentage points over the eager delay, reducing the mean performance loss to 19%. Energy usage is also improved, to a 13% cost over the baseline. At this point, delay-on-miss, even without value prediction, is already consistently better than both eager delay and InvisiSpec, with the exception of mcf and GemsFDTD.

Finally, delay-on-miss combined with value prediction performs the best out of all the evaluated solutions. With a mean performance loss of only 11% (8 percentage points better than delay-on-miss), and a mean energy usage increase of 7%, it outperforms all the other evaluated solutions. Part of this energy cost is due to the value predictor and the need to validate the predicted results, as well as the need to update the cache metadata after the speculation has been verified.

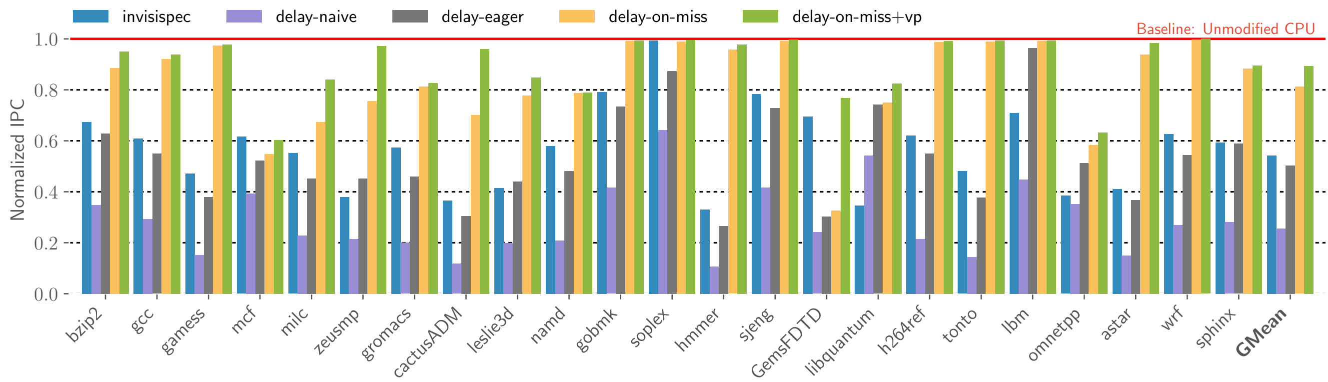

Figure 8: IPC, normalized to the baseline.

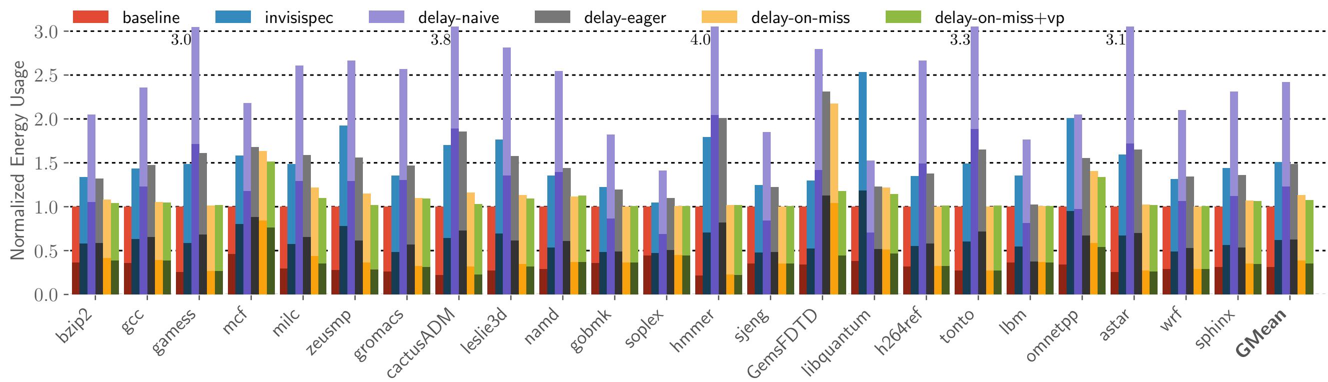

Figure 9: Normalized energy usage. The bottom (shaded) part represents the static (leakage) energy of the system.

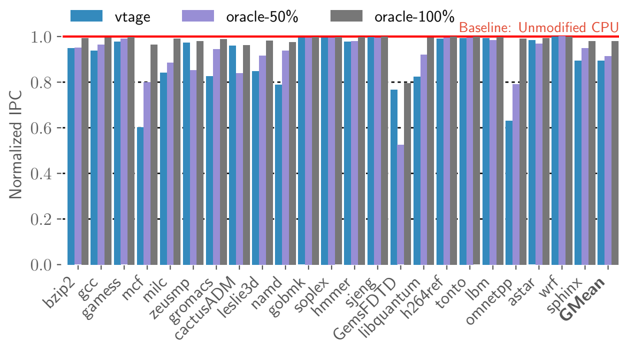

Figure 10: Normalized IPC with two Oracle value predictors, with prediction rates of 50% and 100%.

5.4 Improving the VP Prediction Rate

We have implemented an Oracle predictor with a variable prediction rate to evaluate the effects that an improved value prediction rate would have on the delay-on-miss solution. We evaluate two different rates, 50% of all shadowed L1 misses, and 100%. Note that the Oracle predictor decides randomly every time it is queried, and therefore, the prediction rate is not tied to the PC or any other characteristics of the instruction being predicted. As seen in Figure 10, having perfect value prediction would lead to significant performance improvements, with the majority of the benchmarks approaching or reaching the baseline. There is one exception, `GemsFDTD`, in which the VTAGE predictor already provides almost perfect value prediction, so the Oracle predictor can not provide any significant improvements. The missing performance in all cases can be explained by the overhead introduced due to only allowing one validation at a time.

Overall, we observe performance improvements of 2 and 9 percentage points over the VTAGE predictor, for prediction rates of 50% and 100% respectively. Given that the VTAGE predictor has a mean prediction rate of 16%, an increase of more than 2× in prediction rate yielding only 2 points improvement brings into question whether improving the value predictor is worth the effort and potential hardware cost. However, the 9 points increase provided by the perfect predictor is still encouraging, although achieving perfect value prediction is of course impossible.

```
1   struct program_data {
2       char storage[10];
3       char secret_key; /* = 7 */
4   }
5
6   char access_data(int index) {
7       if (index <= 9)
8           return program_data.storage[index];
9       else
10          return -1;
11  }
```

Listing 1: Victim code for the Spectre attack example.

```
1   void train() {
2       for (i = 0; i < 1000; i++)
3           access_data(0);
4   }
5
6   void probe(char probe_array[]) {
7       for (i = 0; i < 16; i++) {
8           t1 = RDTSCP();
9           x = probe_array[i * CACHE_LINE_SIZE];
10          delay = RDTSCP() - t1;
11      }
12  }
13
14  void attack() {
15      char probe_array[16 * CACHE_LINE_SIZE];
16      train();
17      flush(probe_array);
18      secret = access_data(10);
19      x = probe_array[secret * CACHE_LINE_SIZE];
20      probe(probe_array);
21  }
```

Listing 2: Simplified Spectre attack example.

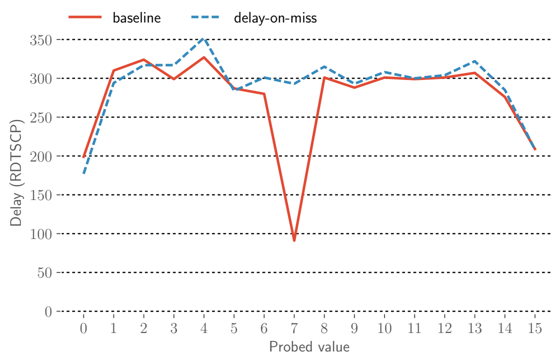

Figure 11: Probe access times (from `rdtscp`) for the Spectre example in Listing 2. The secret value of seven can be easily deduced in the baseline.

6 SECURITY EVALUATION

Listings 1 and 2 contain pseudocode for a Spectre attack example. Listing 1 contains the victim code that is going to be exploited by the attack. It consists of two parts, *i)* the program data, which contains some user accessible storage and a secret key, and *ii)* a function (`access_data`) that provides access to the storage, making sure that no out-of-bounds accesses are allowed. While this is a simplified example, similar patterns can be found, for instance, in web browsers where the JavaScript runtime needs to prevent the code for accessing other parts of the browser's memory. Listing 2 shows the attack code and how it can be used to bypass the victim's checks in five simple steps:

(1) Line 16—The attack starts by training the branch predictor to always expect the `if` statement in the `access_data` function (Listing 1, Line 7) to be true. This is done by simple calling the `access_data` function multiple times in a row with an index value that is within the storage bounds.
(2) Line 17—To prepare for the Flush+Reload side-channel [60], the probe array is flushed from the cache.
(3) Line 18—Now that the system has been set up for the attack, the Spectre code calls the `access_data` function with an out-of-bounds index of "10". This index points to the element directly after the end of the array, which is where the secret key resides in the memory. From a software point of view, we expect the `access_data` function to detect the out-of-bounds access and return a value of "−1" but, due to speculative execution, this is not exactly what happens. Because we have trained the branch predictor to always expect the `if` statement to be true, i.e., to always predict the index to be inside the storage bounds, the OoO CPU will speculatively perform the memory access to `storage[10]` (Listing 1, Line 8), and return the loaded value (i.e., the secret key).
(4) Line 19—The attack code will then use the value returned by the `access_data` function to index into the probe array[3], bringing one of its cache lines into the cache.
(5) Line 20—Eventually, the CPU will realize that the branch was mispredicted and the incorrect execution will be squashed, eventually leading to the attack code receiving the correct value of "−1", but any data brought into the cache in the meantime will remain there. The attack code can then scan through the probe array, measuring the delay (time) of accessing each cache line. As one cache line was brought in during the speculative part of the attack, we can deduce, based on its position in the probe array, what the value at `storage[10]` was, thus deducing the secret key (Figure 11).

Note that both the victim and the attacker are part of the same execution context, which is why cache partitioning or randomization solutions (Section 7) are ineffective against Spectre. In contrast, when delay-on-miss is enabled, the speculative access to the probe array will be prevented from loading any data into the cache and probing the array later will not lead to any measurable differences in execution time (Figure 11).

This variant of Spectre presented here takes advantage of the branch predictor and speculative loads to bypass software-enforced

[3]For simplicity we use a probe array that can encode 16 different values; in practice, if one byte is read during the attack, 256 values would be required.

bound checks and gain access to secret information. However, this is not the only variant that has been discovered. For example, the Spectre v1.1 and v1.2 variants [8, 9] take advantage of speculative stores to redirect the execution stream and to bypass sandboxing. Our solution does not prevent the redirection of the execution stream, after all how the victim is coerced into executing the attack code is beyond the scope of this work, but it does prevent the attack code from leaking any sensitive information. Regardless of how the attack code is executed, it still needs a side-channel to leak any sensitive information it accesses, and our solution blocks such side-channels in the cache and memory system. What our solution does not protect against is side-channels outside of the memory hierarchy. For example, the NetSpectre attack [47] uses the slowdown caused by the AVX engine on some modern Intel CPUs. Since we do not delay computational instructions, only loads, that side-channel is left exposed. Protection against side-channels outside of the cache and memory system is left for future work. Similarly, attacks that require physical access to the machine, such as power analysis attacks, are also outside the scope of this work.

7 RELATED WORK

Cache side-channel attacks [2, 15–17, 37, 55, 60] have been known for years before their speculative variants emerged. Instead of trying to leak arbitrary memory regions of arbitrary applications, they focus on identifying keys by detecting memory access patterns of cryptographic applications, or on covertly transferring information across software and hardware barriers, such as across virtualized containers. Both hardware and software solutions have been proposed [10, 12–14, 18, 21, 22, 24, 30–32, 38, 56, 57, 61], using either cache partitioning, cache locking, or access pattern obfuscation through randomization. These solutions do not work with speculative attacks such as Spectre [23], where the attack can be performed from within the same execution context (Section 6). Additionally, most of these solutions focus on only protecting small amounts of sensitive data, such as cryptographic keys, while the speculative side-channel attacks we are focusing on can attack the whole memory address range.

When it comes to the currently known speculative side-channel attacks, hardware and software [7, 39, 53] vendors have already promised or delivered solutions. These solution are specific to the issue they are trying to fix (e.g., not performing loads before the permissions have been checked, for Meltdown), and do not protect against all existing and potential future speculative side-channel attacks. Our solution instead provides a holistic approach that eliminates the threat of speculative attacks that try to take advantage of the memory hierarchy as a side-channel. It builds on insights provided by our work on Ghost loads [46], which evaluates, in detail, the implications of trying to hide speculative loads in the cache hierarchy.

There currently exist, to the best of our knowledge, two proposed solutions for hiding the side-effects of all speculative memory accesses in the cache hierarchy. The first, InvisiSpec by Yan et al. [59] has already been discussed in more details in Sections 1 and 5, but we will briefly summarize the differences here as well. First, InvisiSpec does not delay any memory accesses, instead it performs them and does not cache the data. The data are kept in buffers ("Speculative Buffers") at the L1 and the LLC, and are released after the speculation has been resolved. To avoid attacks on these buffers, they are not kept coherent; instead a second access ("Validation") is used to validate the loaded value in accordance to the memory model. Given the speculative nature of the loads until they can be validated, only one validation at a time can be issued. Observing that not all loads are actually executed speculatively, they also propose an optimization where loads not executed under a speculative shadow require no validation. Unfortunately, requiring duplicate memory accesses for the majority of the loads in an application leads to significant performance and energy overheads, something that is present both in the original work and in our evaluation. In comparison, our proposal does not allow the memory access in the cases where InvisiSpec would require a validation and also eliminates the need for coherence modifications by only allowing L1 hits. Note that while validations are employed by both InvisiSpec and our proposal, in our case they are the only memory access performed by the load, since the initial value is provided by the value predictor, with the only overhead being an additional access to the L1 cache. Finally, InvisiSpec only considers the cache hierarchy, while our solution prevents speculative side-effect leakage in the whole memory hierarchy.

In addition to InvisiSpec, Khasawneh et al. [19] have been working on a solution similar to InvisiSpec, called SafeSpec. SafeSpec does not consider cache coherence, making it inapplicable to multithreaded and multiprogram workloads. For this reason, we only consider InvisiSpec as a viable alternative to our proposal.

Outside of the cache hierarchy, attacks on the rest of the memory system also exist [43, 55, 58], and so do proposed solutions [11, 35, 62]. Our solution prevents any speculative (shadowed) loads from accessing any part of the system other than the L1, thus making them completely invisible to the rest of the memory system. Attacks outside the memory hierarchy, such as the NetSpectre [47] attack that utilizes the AVX engine on x86 CPUs, are outside the scope of this work, and solutions proposed for these attacks are complementary to ours.

8 FUTURE WORK

While we achieve significant performance improvement over the state-of-the-art solution, we can still not recover all the performance lost from the baseline. One of the issues we have encountered is that the delay introduced into the validation of the value predictions negatively affects the prediction rate of the predictor. We would like to investigate ways of reducing this effect, either by changing the design of the predictor or by introducing a new mechanism for earlier validation of the predictions. Additionally, even with the perfect predictor, there is still some performance missing over the baseline. This performance loss is due to the loss of MLP suffered during the VP validations, as VP-shadowed loads cannot validate until their shadow has been lifted. We would like to further investigate methods for eagerly unshadowing loads, using either more aggressive hardware techniques or compiler information. For example, previous work has investigated a technique for decoupling the calculation of the memory address from the execution of the load [50]. Using such a technique, we can limit the duration of the

E- (and maybe D-) Shadows, as the address can be calculated potentially before the load is even dispatched. Furthermore, predicated execution can, in some cases, replace branches, thus eliminating the C-Shadows. Finally, under TSO, Ros et al. [45] have proposed a non-speculative solution for load-load reordering, referred to as WritersBlock, which can eliminate the M-Shadows. By utilizing WritersBlock and by tracking the VP-Shadows at a fine granularity, we can allow for multiple validations to be issued in parallel, restoring the MLP and the performance.

In addition to improving the performance of our current proposal, there is also more work to be done in further reducing the exposure of the side-effects of speculation in the rest of the system. For example, we only focus on data accesses, but the instruction cache needs to be secured as well. Unlike data caches, accesses to the instruction cache cannot be freely delayed, as the pipeline will stall.

Finally, in this work we assume that all data are equally important and need to be kept secret. In practice, there might be data that do not need to be protected from speculative side-channel attacks, significantly limiting the number of speculative accesses that need to be delayed or predicted. Vijaykumar et al. [54] have already proposed a solution for tagging memory regions that can be used exactly for this purpose, assuming that the sensitive data regions have been identified by the programmer.

9 CONCLUSION

In this work, we propose an efficient solution for preventing visible side-effects in the memory hierarchy during speculative execution. It is based on the observation that hiding the side-effects of executed instructions in the whole memory hierarchy is both expensive, in terms of performance and energy, and complicated. At the same time, delaying all speculative loads is also not a viable solution, since most loads are executed speculatively. Instead, it is easier and cheaper to hide the side-effects of loads that hit in the L1 data cache, and prevent all other loads from executing. To limit the performance deterioration caused by these delayed loads, we augment the L1 data cache with a value predictor that covers the loads that miss in the L1. We observe that one of the limitations that prevents us from reaching the baseline performance, or even exceeding it thanks to the value predictor, is that value predicted loads can only be validated one at a time. However, even with this limitation, we provide secure execution, free of visible side-effects, with a cost of only 11% in performance and 7% in energy.

ACKNOWLEDGMENTS

This work was funded by Vetenskapsrådet project 2015-05159, by the SSF Strategic Mobility 2017 grant SM17-0064, the Spanish MCIU and AEI, as well as European Commission FEDER grant RTI2018-098156-B-C53. The computations were performed on resources provided by SNIC through UPPMAX and by UNINETT Sigma2.

REFERENCES

[1] Gordon B. Bell and Mikko H. Lipasti. 2004. Deconstructing Commit. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, Washington, DC, USA, 68–77. https://doi.org/10.1109/ISPASS.2004.1291357

[2] Daniel J. Bernstein. 2005. Cache-timing attacks on AES.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (Aug. 2011), 1–7. Issue 2. https://doi.org/10.1145/2024716.2024718

[4] Joseph Bonneau and Ilya Mironov. 2006. Cache-Collision Timing Attacks Against AES. In *Proceedings of the Cryptographic Hardware and Embedded Systems*. Springer, Berlin, Heidelberg, 201–215. https://doi.org/10.1007/11894063_16

[5] Rosario Cammarota and Rami Sheikh. 2018. VPsec: Countering Fault Attacks in General Purpose Microprocessors with Value Prediction. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, New York, NY, USA, 191–199. https://doi.org/10.1145/3203217.3203276

[6] Luis Ceze, Karin Strauss, James Tuck, Josep Torrellas, and Jose Renau. 2006. CAVA: Using Checkpoint-assisted Value Prediction to Hide L2 Misses. *ACM Transactions on Architecture and Code Optimization* 3, 2 (June 2006), 182–208. https://doi.org/10.1145/1138035.1138038

[7] Jonathan Corbet. 2017. KAISER: hiding the kernel from user space. https://lwn.net/Articles/738975/.

[8] National Vulnerability Database. 2017. CVE-2017-5753. Available from MITRE, CVE-ID CVE-2017-5753.. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5753

[9] National Vulnerability Database. 2017. CVE-2018-3693. Available from MITRE, CVE-ID CVE-2018-3693.. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3693

[10] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable Caches: Low-complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization* 8, 4 (Jan. 2012), 35:1–35:21. https://doi.org/10.1145/2086696.2086714

[11] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. 2018. Spectres, Virtual Ghosts, and Hardware Support. In *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, New York, NY, USA, 5:1–5:9. https://doi.org/10.1145/3214292.3214297

[12] Hongyua Fang, Sai Santosh Dayapule, Fan Yao, Miloš Doroslovački, and Guru Venkataramani. 2018. Prefetch-guard: Leveraging hardware prefetches to defend against cache timing channels. In *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust*. IEEE Computer Society, Washington, DC, USA, 187–190. https://doi.org/10.1109/HST.2018.8383912

[13] Adi Fuchs and Ruby B. Lee. 2015. Disruptive Prefetching: Impact on Side-channel Attacks and Cache Designs. In *Proceedings of the ACM International Systems and Storage Conference*. ACM, New York, NY, USA, 14:1–14:12. https://doi.org/10.1145/2757667.2757672

[14] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and efficient cache side-channel protection using hardware transactional memory. In *Proceedings of the USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 217–233.

[15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *Proceedings of the USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 897–912.

[16] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Access-Based Cache Attacks on AES to Practice. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 490–505. https://doi.org/10.1109/SP.2011.22

[17] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross Processor Cache Attacks. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security*. ACM, New York, NY, USA, 353–364. https://doi.org/10.1145/2897845.2897867

[18] Georgios Keramidas, Antonios Antonopoulos, Dimitrios N. Serpanos, and Stefanos Kaxiras. 2008. Non deterministic caches: a simple and effective defense against side channel attacks. *Design Automation for Embedded Systems* 12, 3 (Sept. 2008), 221–230. https://doi.org/10.1007/s10617-008-9018-y

[19] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2018. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. arXiv:1806.05179 http://arxiv.org/abs/1806.05179

[20] Ilhyun Kim and Mikko H. Lipasti. 2004. Understanding scheduling replay schemes. In *Proceedings of the International Symposium High-Performance Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 198–209. https://doi.org/10.1109/HPCA.2004.10011

[21] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. In *Proceedings of the USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 11–11.

[22] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 974–987. https://doi.org/10.1109/MICRO.2018.00083

[23] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: Exploiting speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 19–37. https://doi.org/10.1109/SP.2019.00002

[24] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. 2009. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proceedings of the International Symposium High-Performance Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 393–404. https://doi.org/10.1109/HPCA.2009.4798277

[25] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[26] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. 2011. CACTI-P: Architecture-Level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society, Washington, DC, USA, 694–701. https://doi.org/10.1109/ICCAD.2011.6105405

[27] Mikko H. Lipasti and John Paul Shen. 1996. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 226–237.

[28] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value Locality and Load Value Prediction. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 138–147. https://doi.org/10.1145/237090.237173

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. arXiv:1801.01207 http://arxiv.org/abs/1801.01207

[30] Fangfei Liu and Ruby B. Lee. 2013. Security Testing of a Secure Cache Design. In *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, New York, NY, USA, 3:1–3:8. https://doi.org/10.1145/2487726.2487729

[31] Fangfei Liu and Ruby B. Lee. 2014. Random Fill Cache Architecture. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 203–215. https://doi.org/10.1109/MICRO.2014.28

[32] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. 2016. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. *IEEE Micro* 36, 5 (Sept. 2016), 8–16. https://doi.org/10.1109/MM.2016.85

[33] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 605–622. https://doi.org/10.1109/SP.2015.43

[34] Milo M. K. Martin, Daniel J. Sorin, Harold W. Cain, Mark D. Hill, and Mikko H. Lipasti. 2001. Correctly Implementing Value Prediction in Microprocessors That Support Multithreading or Multiprocessing. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 328–337. https://doi.org/10.1109/SP.2015.43

[35] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *Proceedings of the International Symposium on Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 118–129. https://doi.org/10.1145/2366231.2337173

[36] Lois Orosa, Rodolfo Azevedo, and Onur Mutlu. 2018. AVPP: Address-first Value-next Predictor with Value Prefetching for Improving the Efficiency of Load Value Prediction. *ACM Transactions on Architecture and Code Optimization* 15, 4 (Dec. 2018), 49:1–49:30. https://doi.org/10.1145/3239567

[37] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Proceedings of the RSA Conference*. Springer, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/11605805_1

[38] Dan Page. 2005. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. IACR Cryptology ePrint archive.

[39] Andrew Pardoe. 2018. Spectre mitigations in MSVC. https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/.

[40] Arthur Perais and André Seznec. 2014. EOLE: Paving the Way for an Effective Implementation of Value Prediction. In *Proceedings of the International Symposium on Computer Architecture*. ACM, New York, NY, USA, 481–492. https://doi.org/10.1109/ISCA.2014.6853205

[41] Arthur Perais and André Seznec. 2014. Practical data value speculation for future high-end processors. In *Proceedings of the International Symposium High-Performance Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 428–439. https://doi.org/10.1109/HPCA.2014.6835952

[42] Arthur Perais and André Seznec. 2015. BeBoP: A cost effective predictor infrastructure for superscalar value prediction. In *Proceedings of the International Symposium High-Performance Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 13–25. https://doi.org/10.1109/HPCA.2015.7056018

[43] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks.. In *Proceedings of the USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 565–581.

[44] Moinuddin K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 775–787. https://doi.org/10.1109/MICRO.2018.00068

[45] Alberto Ros, Trevor E. Carlson, Mehdi Alipour, and Stefanos Kaxiras. 2017. Non-Speculative Load-Load Reordering in TSO. In *Proceedings of the International Symposium on Computer Architecture*. ACM, New York, NY, USA, 187–200. https://doi.org/10.1145/3079856.3080220

[46] Christos Sakalis, Mehdi Alipour, Alberto Ros, Alexandra Jimborean, Stefanos Kaxiras, and Själander Magnus. 2019. Ghost Loads: What is the Cost of Invisible Speculation?. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, New York, NY, USA, 153–163. https://doi.org/10.1145/3310273.3321558

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

[48] André Seznec. 2007. A 256 kbits l-TAGE Branch Predictor. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)* 9 (2007), 1–6.

[49] Standard Performance Evaluation Corporation. 2006. SPEC CPU Benchmark Suite. http://www.specbench.org/osg/cpu2006/.

[50] Michael Stokes, Ryan Baird, Zhaoxiang Jin, David Whalley, and Soner Onder. 2018. Decoupling Address Generation from Loads and Stores to Improve Data Access Energy Efficiency. In *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, New York, NY, USA, 65–75. https://doi.org/10.1145/3211332.3211340

[51] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. arXiv:1802.03802 http://arxiv.org/abs/1802.03802

[52] Dean M. Tullsen and John S. Seng. 1999. Storageless Value Prediction Using Prior Register Values. In *Proceedings of the International Symposium on Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 270–279. https://doi.org/10.1145/300979.301002

[53] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886.

[54] Nandita Vijaykumar, Abilasha Jain, Diptesh Majumdar, Keving Hsieh, Gennady Pekhimenko, Eiman Ebrahimi, Nastaran Hajinazar, Philip B. Gibbons, and Onur Mutlu. 2018. A Case for Richer Cross-Layer Abstractions: Bridging the Semantic Gap with Expressive Memory. In *Proceedings of the International Symposium on Computer Architecture*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1109/ISCA.2018.00027

[55] Zenghong Wang and Ruby B. Lee. 2006. Covert and Side Channels Due to Processor Architecture. In *Proceedings of the Annual Computer Security Applications Conference*. IEEE Computer Society, Washington, DC, USA, 473–482. https://doi.org/10.1109/ACSAC.2006.20

[56] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the International Symposium on Computer Architecture*. ACM, New York, NY, USA, 494–505. https://doi.org/10.1145/1250662.1250723

[57] Zhenghong Wang and Ruby B. Lee. 2008. A Novel Cache Architecture with Enhanced Performance and Security. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 83–93. https://doi.org/10.1109/MICRO.2008.4771781

[58] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud.. In *Proceedings of the USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 159–173.

[59] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 428–441. https://doi.org/10.1109/MICRO.2018.00042

[60] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 719–732.

[61] Yinqian Zhang and Michael K. Reiter. 2013. Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*. ACM, New York, NY, USA, 827–838. https://doi.org/10.1145/2508859.2516741

[62] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. 2004. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 72–84. https://doi.org/10.1145/1024393.1024403