Exploring speculation barriers for RISC-V selective speculation

Herinomena Andrianatrehina[1], Ronan Lashermes[1 [0000−0002−0309−6533]], Joseph Paturel[1 [0009−0008−6120−2147]], Simon Rokicki[1 [0000−0002−0195−096X]], and Thomas Rubiano[1]

Univ Rennes, Inria, CNRS, IRISA

Abstract. Speculative execution poses significant security risks to modern out-of-order cores, exemplified by attacks such as Spectre. Numerous countermeasures, including selective speculation in both software and hardware, have been proposed. This approach allows enabling or disabling speculative behavior based on circumstances. However, challenges such as evolving attack methods and the complexity of simulating out-of-order cores make these solutions difficult to reproduce and compare.
This paper investigates the use of RISC-V speculation fences to achieve selective speculation in a realistic scenario where the microarchitecture cannot distinguish between confidential and non-confidential data. We examine three aspects: the semantics of speculation fences (ranging from broad to selective constraints), the placement of fences in programs by compilers, and their hardware implementation in a modified NaxRiscv RISC-V out-of-order core. Using a new security metric, we compare configurations within a unified framework.
Our findings highlight that speculative execution of `load` instructions is critical for out-of-order core performance. Furthermore, we demonstrate that selective speculation without confidentiality-tagged data fails to achieve a meaningful security-performance trade-off.

1 Introduction

Modern microarchitectures have undergone many modifications to maximize the utilization of execution units and enhance overall system performance. One such optimization is speculative execution, a predictive technique used to anticipate and execute instructions before their outcomes are definitively known. By speculating on the most probable path based on prior execution history, the processor can continue executing subsequent instructions without waiting for earlier ones to complete. In cases of incorrect predictions, the processor performs a rollback and discards the misspeculated instructions. However, it is challenging to completely revert all microarchitectural states altered during speculative execution,

The ARSENE project was funded by the "France 2030" government investment plan managed by the French National Research Agency, under the reference "ANR-22-PECY-0004"

such as those in cache memories or branch predictors. This inability to clear all speculative traces can be exploited through vulnerabilities like Spectre [19], wherein an attacker influences speculative execution to leak sensitive information by leaving observable states within the microarchitecture.

Since the discovery of the Spectre vulnerability, numerous mitigation techniques have been proposed, as detailed in Section 3. However, these mitigations often produce inconsistent results and are frequently difficult to reproduce, as noted in [26].

This inconsistency can be attributed to several factors. First, the significant impact of test environments on the results of certain mitigation measures, such as the benchmarks used. Second, the difficulty of openly experimenting on speculative microarchitectures, as researchers must choose between using simplified gem5 models of the most complex x86 cores or precise models of simpler RISC-V cores. Understandably, companies with access to RTL models of complex cores do not publish the security assessments of their products. Third, differing threat models and initial assumptions make it difficult to compare solutions designed to address different types of threats and varying levels of knowledge regarding which data requires protection.

Thus, although a variety of mitigation approaches have been proposed to address Spectre, the lack of reproducibility has made it challenging to accurately assess and compare the effectiveness of each solution.

A widely adopted approach is **selective speculation**. The goal is to delay the execution of instructions that potentially access or leak sensitive data, or contribute to leakage, until the processor is certain that it is not a misspeculation. This is achieved through various strategies such as inserting barrier instructions, also called fence instructions, or directly modifying the microarchitecture to handle this functionality in hardware.

In this article, we examine the potential of speculation barriers to achieve efficient selective speculation by providing a thorough and comprehensive analysis of their impact on both performance and security. We aim to address the critical question: **Is selective speculation with dedicated barrier instructions a viable solution to defend against Spectre attacks?**

To investigate this, we created a test environment for exploring and comparing different implementations of the selective speculation approach in a realistic testing environment. We varied the semantics of the barrier instructions, their placement policies, and the hardware implementations.

Our contributions include:

- The definition of different fence instructions for selective speculation, along with their hardware implementation in the open-source NaxRiscv [32] out-of-order processor.
- The modification of an LLVM-based compiler toolchain with several policies for inserting the aforementioned fences.
- The definition of a quantitative security metric used for fair comparison between different Spectre countermeasures, which relies on counting the number of vulnerable instruction sequences in execution traces.

A critical assumption in this work is that there is no way to differentiate a `load` instruction accessing a secret from a `load` instruction accessing innocuous data. As a consequence, our compiler passes cannot rely on security annotations and thus have to protect every `load` instruction, **as in realistic use cases**.

During our experimental study, we have evaluated the trade-offs between security and performance. **Our results indicate that — under our assumptions — there is no viable trade-off regardless of semantics, placement policy, or hardware implementation.** Fundamentally, the advantage of out-of-order cores over in-order cores appears to stem largely from load speculation, which should necessarily be delayed for security reasons.

These findings highlight the critical need for microarchitectures to distinguish between confidential data that must not be speculated on and other data, if we are to reconcile out-of-order execution with robust security.

As discussed, inconsistencies in existing results make a direct comparison between our proposed approach and prior work impossible. Therefore, the comparison cannot rely solely on results from the literature and must be conducted independently. To address this, we implemented the speculative load hardening (SLH) mitigation [9] due to its relatively straightforward integration, allowing us to assess it in our test environment and compare its effectiveness with our own results.

2 Security issues with speculative execution

Speculative execution allows a processor to predictively execute instructions before it is fully certain of their necessity, established at the commit stage. All instructions are executed speculatively since execution is done prior to commit, but most are correctly speculated. Misspeculation means that some instructions have been executed when they should not have been, with the risk of leaving traces in the microarchitectural state that can be exploited. These traces can be observed and used in numerous ways to exfiltrate sensitive data out of its intended environment.

2.1 Covert and side channels

Covert channels and side channels are communication channels where information transfer should not be possible.

They correspond to two different threat models: a covert channel is a communication channel where the attacker controls both the emitter and receiver. In contrast, in a side channel, the attacker only controls the receiver, while the emitter is an innocent victim. Covert channels correspond to a stronger attacker model, capable of actively trying to emit information in the channel. Therefore, this is the threat model that we aim to address in this paper. As detailed in Subsection 2.2, the Spectre attacker manipulates the microarchitecture to emit the target information, making the **covert channel** the primary threat model we aim to address in this paper.

In the context of microarchitecture security, most covert channels exploit timing differences to communicate information. The emitter tries to set up a component, e.g. a cache memory, in a state that can be observed by the receiver through timing variations, e.g. whether a cache line is present or not.

In the microarchitecture, any state can be exploited as a covert channel: cache memories [4, 24], TLB [13], BTB [2], branch predictors [2], prefetchers [31], etc.

Countermeasures exist in the form of dedicated instructions to be applied during context switches that reset or partition the microarchitectural state [10, 37]. Unfortunately, perfectly implementing these semantics implies applying formal methods during the hardware design to guarantee the absence of information leakage, a feat that is hardly feasible for the complex speculative cores.

2.2 Spectre

Spectre [19] attacks exploit speculative execution to bypass memory safety boundaries, reading and exfiltrating memory where it should not be possible. They work by tricking the processor into speculatively executing code paths that should not be executed, due to improper control flow or data flow predictions. Spectre attacks are generally categorized by the mechanism used to mislead speculative execution, such as branch prediction or return stack buffer misuse.

One of the most well-known forms of Spectre is Spectre-PHT [19], which involves manipulating the pattern history table (PHT) used for branch direction prediction. In Figure 1, an attacker trains the branch predictor to assume a condition is true, causing speculative execution to proceed down a path where sensitive data is accessed.

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

Fig. 1: Spectre-PHT attacks in C code

Spectre attacks proceed in two key phases:

Predictor Manipulation The attacker manipulates the branch predictor to assume that the branch condition will likely be met (`x` $<$ `array1_size`). This can be done by executing the branch in scenarios where the condition evaluates to true multiple times.

Data Exfiltration The same code is executed again, but now with a condition that should evaluate to false (`x` $\geq$ `array1_size`), where `x` is controlled by the attacker. However, speculative execution still proceeds with executing the branch body due to a now incorrect branch direction prediction. During this phase, a `Spectre gadget` is executed; it built around three elements:

- **Speculation**: an instruction that triggers speculative behavior - the `if` condition.
- **Acquisition**: an instruction that can access a secret - a speculative `load` with the address `array1 + x` reads the secret value s.
- **Disclosure**: an instruction that translates the secret into a microarchitecture state - here, a second `load` exfiltrates the secret s into a tag field in cache memory.

Spectre attacks are especially challenging to mitigate because the mechanisms for speculative execution are integral to modern processor performance. Different variants of Spectre have emerged, each exploiting distinct speculative mechanisms:

- Spectre-PHT uses branch direction prediction via the PHT.
- Spectre-BTB leverages branch destination speculation via a branch target buffer (BTB).
- Spectre-RSB uses speculation from the return stack buffer (RSB).
- Spectre-STL exploits aliasing speculation in the Load Store Queue, known as store-to-load forwarding.

The adaptability of Spectre to exploit different speculative mechanisms makes comprehensive mitigation difficult without negatively impacting processor performance.

In our own nomenclature, the primary difference between microarchitectural data sampling (MDS) attacks [29] and Spectre attacks lies in the threat model. MDS attacks assume that the targeted application legitimately uses a secret value, whereas Spectre attacks assume only that a secret is accessible, even if not directly used by the application.

Within this categorization, Meltdown [21] attacks are considered a subcategory of Spectre attacks, leveraging the speculation of non-occurrence of exceptions during execution.

In these threat models, mitigating MDS attacks is essentially about mitigating covert channels, where Spectre attacks mitigation is more concerned with the speculation behavior. In this paper, we choose to focus on this latter issue.

3 Related works against Spectre attacks

Spectre attacks that allow arbitrary memory reads are dangerous threats, and numerous countermeasures have been proposed to mitigate them. The existing solutions can be categorized into two broad categories: those that only use software structures and those that are based around the modification of the processor hardware.

3.1 Software solutions

The `LFENCE` x86 instruction has seen its semantics changed after the publication of Spectre [19]. Previously a read ordering barrier, it is now effectively a

speculation barrier, preventing all instructions following it from executing, even speculatively, until all earlier instructions have completed.

LLVM SESES (Speculative Execution Side Effect Suppression) [8] is a naive mitigation that prevents all possible Load Value Injection [34] attacks using misspeculated transient execution. This LLVM pass offers the option to add an `LFENCE` instruction before each memory read/write instruction and before the first branch instruction in a group of terminators at the end of a basic block. Benchmarks [25] show how drastically this mitigation affects performance but do not evaluate the actual security benefit it provides. We will discuss and compare these results in later sections.

Retpoline for "return trampoline" prevents Spectre attacks that exploit branch target injection. This variant leverages indirect branch predictions, such as function pointers or virtual function calls, to misdirect the CPU into executing unwanted instructions before the branch prediction is corrected. Retpoline disables speculation on indirect branches by trapping speculative execution in an infinite "safe loop" [1].

SLH is a software mitigation technique designed to protect against the Spectre-PHT variant. The main idea is to mask or "poison" either the pointer or the returned value of a speculated `load` to protect sensitive data. A predicate capturing the speculation status needs to be updated every time a conditional branch is taken. This predicate can also be transferred through function calls using some bits in the stack pointer. SLH is well-documented in LLVM and implemented for x86 [9].

Blade is a software mitigation technique that statically analyzes data flow from *potential sources* of secrets to *potential sinks* and "protects" them using different approaches. Various variants are proposed depending on the target architectures and available tools. Fence or SLH-like masks can be used. In [35], *potential sinks* are formally identified using a static type system that is "transient-aware".

3.2 Hardware solutions

The academic community has been very active on the topic of microarchitecture security, both in terms of attacks and countermeasures. The hardware-based countermeasures can further be categorized into three groups: those that rely on cleanly reversing the changes that misspeculated instructions might have left in the microarchitecture, those that try to detect a Spectre-like leak, and those based on formal methods. It is worth noting that there is no open-source processor with a comparable complexity to Intel, AMD, or Arm cores. The results of the hardware-based solutions must then be extrapolated to these, without certainty about the relevance of this transposition. The literature is too extensive to be described in detail in this document, but pertinent state-of-the-art papers on the topic include [15] and [26].

Clean reversal of misspeculated microarchitectural state Spectre attacks are possible because transient instructions persistently modify the microarchi-

tectural state, even when misspeculated. Therefore, a valid solution would be to revert the microarchitectural state exactly as it was before the misspeculated execution. While the concept is straightforward, implementing it is challenging, as all the possible states must be reverted. That includes caches, branch predictors, finie state machines, Load-Store Queues, etc.

Several works have explored this type of countermeasure, such as InvisiSpec [38], CleanupSpec [27], DAWG [18], and SafeSpec [17]. They all differ in their specific implementations, impacting both performance and security.

Delays based on Speculative Secret Tracking Another strategy is to identify the occurrence of a Spectre gadget in order to trigger the suspension of speculative behavior. The idea is to identify the risky behavior corresponding to the three elements of a Spectre gadget: speculation, acquisition, and disclosure: Taint tracking is necessary between acquisition and disclosure to detect if secret data is likely to leak. It is possible to follow these steps and act at each of them, either by preventing any speculation, forbidding speculative `load` instructions, or preventing the covert channel. The earlier the intervention, the safer but more costly the solution is in terms of performance.

Many proposals using this principle have been made, all differing in their implementations: NDA [36], STT [40], SpecShield [3], Efficient InvisiSpec [28], SpectreGuard [11], CondSpec [20], ConTExT [30], InvarSpec [41], Speculative Data-Oblivious Execution (SDO) [39], DOLMA [22], and Speculative Privacy Tracking (SPT) [6].

SpecTerminator [16] is the modern synthesis of this line of techniques that delay unsafe speculative execution with a hardware tainting mechanism and a way to delay some operations. Yet the reported performance penalty of +6% has not been reproduced, as reported in [26].

To enhance flexibility, software can be given greater control through dedicated speculation barriers. Our own implementations are described in this paper, but it was not the first speculation barrier implementation.

In Context-Sensitive Fencing [33], the authors propose automatically injecting speculation barriers in the micro-ops generated by the frontend, depending on pre-specified security policies. An example policy is to inject a barrier between control flow instructions and loads. Speculation barriers have also been proposed by established vendors: Intel's `LFENCE` or ARM's `SB`; ARM has patented this latter speculation barrier. Unfortunately, as far as we know, there is no proper public analysis on placement policies and associated trade-offs between performance and security.

While these solutions are interesting, they suffer from two major issues. The first one is that they are hard to reproduce and therefore to compare. For example, InvisiSpec [38] self-reports a performance penalty of +72%, Efficient InvisiSpec [28] tries to reproduce this first paper and measures a +50% performance penalty, NDA [36] does the same and reports +32%. The papers also differ on what they consider secure: some target specific variants, some all variants known at the time, etc. In practice, they cannot be compared with respect to security.

The second issue is that they are not all relying on realistic assumptions. For example, ConTExT [30] requires annotating secret data by marking with a dedicated bit in each page table entry. This is actually a strong assumption, since we do not have the infrastructure today to do that at scale.

Finally, most (but not all) of these previous works build demonstrators using the gem5 simulator with x86. This makes it actually hard to evaluate whether the corresponding implementations are realistic.

Use of Formal Methods Given the complexity of modern microarchitectures, how can we ensure that secrets are never speculatively accessed or leaked to a covert channel?

Some approaches tackle this challenge by establishing hardware-software contracts for speculative behavior. The work of [14] formalizes the interaction between information leakage models and speculative behavior, demonstrating that simply delaying speculative `load` instructions is insufficient; other speculative behaviors must also be controlled. Speculative taint tracking is shown to be an effective method for ensuring security.

ProSpeCT [7] implements this speculative behavior contract in hardware. It ensures that secret values cannot be speculatively leaked by enforcing constant-time execution. This approach has been implemented on the out-of-order core Proteus, with performance overheads ranging from 0% to 45%, depending on the frequency of secret-related operations and the program's instruction-level parallelism (ILP).

4 Selective speculation semantics

In this work, we assume that the microarchitecture has no way to determine whether data in memory or in a register is secret or public. In Spectre attacks, the microarchitectural control flow is arbitrary since speculative, possibly influenced by the attacker. Therefore, all `load` instructions are at risk of loading secret data, even if the program normally forbids it. As a countermeasure, we propose inserting speculation barriers, also called speculation fences, which are specific instructions that act on the speculative behavior of the microarchitecture. If `load` instructions are not speculative, the invariants of the program remain enforced, and secrets are only handled when allowed by the application. Speculation is controlled through the insertion of these fences by the compiler.

4.1 Fences Instructions semantics

In this paper, we distinguish between speculation fences, serialization fences, and conditional (speculation) fences.

Serialization fences: A register-based serialization fence `fence.ser rd, rs1` has the following semantics:

1. **Predecessor Dependencies**: this fence instruction depends on register `rs1` to be executed. It cannot be executed if `rs1` is not available. If `rs1` = `x0`, then the instruction depends on all architectural registers (`x1`-`x31`).
2. **Successor Dependencies**: subsequent instructions that use `rd` depend on this fence instruction. If `rd` = `x0`, then the core should assume that the fence instruction touched all architectural registers (`x1`-`x31`), even if no values were modified.

There is no functionality associated with a serialization fence. Architecturally, it is a `nop` (no-operation). It limits the possibility to reorder instructions past the fence, hence its name.

Therefore, the serialization fence reduces the possible divergences between the architectural and the microarchitectural control flows. It reduces but does not eliminate all possible divergences:

- **Reordering with respect to source-only or destination-only instructions**: an instruction that has only sources or destinations (such as branches notably) can be reordered since there is no dependency relation. A `fence.ser` following a `branch` can be executed before it.
- **Delta between register availability and execution**: the fact that registers are available, necessary to resolve a dependency, does not imply that execution is not speculative. For example, a branch instruction can have both its registers available, but the condition is not yet computed. In this case, speculative execution based on this branch condition is possible, even for a limited speculation window.

In this paper, we only consider register-based serialization instructions. Another possibility is to have an instruction-based serialization fence, where dependencies are established purely from program order: the fence depends on all previous instructions, and all later instructions depend on the fence. It is worth noting that in modern x86 cores, the `LFENCE` instruction is an instruction-based serialization fence, in addition to being a speculation fence.

Register-based ordering gives more possibilities for the microarchitecture to reorder instructions, by giving finer-grained constraints on instruction dependencies, thus increasing the performances of the solution.

Speculation fences: A speculation fence is a serialization fence that completes execution only in a non-speculative state, i.e. when the core is certain that it will commit.

We define it as the instruction `fence.spec rd, rs1` with the following semantics in addition to the rules given for serialization fences (**1** and **2** are the same as for `fence.ser`).

3. **Non-speculative Execution**: The execution of `fence.spec` can only terminate in a non-speculative state, meaning that the core is certain that the instruction will eventually commit (taking into account exceptions, interrupts, etc.).

Conditional Speculation Fences: A conditional speculation fence, defined as the instruction `fence.cond rs1`, is designed to prevent the Spectre-PHT attack, which exploits a conditional branch as the source of speculation. This version of the fence follows a conditional branch and uses a predicate stored in `rs1` to determine whether the branch results in misspeculation. Hence, additional computation is required to establish a correct predicate value based on the condition of the branch, which is inserted by the compiler. The conditional speculation fence must implement the following behaviors:

1. **Terminate Speculative Window**: If the predicate evaluates to a value other than 0, the instruction stalls its execution until a non-speculative state is achieved.
2. **Successor Dependencies**: All subsequent instructions with a source register depend on the conditional speculation fence instruction (equivalent to previous cases where `rd` = `x0`).

5 Hardware implementation of fences

This section details the modifications applied to a modern RISC-V core to implement the different behaviors of the fence instructions previously described.

The target of our work is the NaxRiscv core [32], designed using SpinalHDL. It is an out-of-order, dual-issue superscalar processor with BTB + GSHARE + RSB branch predictors. The NaxRiscv project has a decentralized hardware design that simplifies the integration of new features using a plugin system.

First, some background on how each relevant component of the NaxRiscv operates will be provided. Then, we will detail the modifications needed to implement the fence semantics described in Subsection 4.1.

5.1 NaxRiscv description

To keep track of which instruction is currently being executed, the processor core uses a circular buffer called a reorder buffer (ROB). Each entry in the ROB is uniquely identified by a `RobId` and contains all the relevant information about the corresponding instruction. Internally, the ROB uses two pointers to manage the flow of instructions: The `push` pointer indicates the next available entry in the ROB, used to store the next instruction information. Since the processor fetches and decodes instructions in order, they are also inserted into the ROB in program order. The other pointer is the `pop` pointer, which locates the next instruction to be committed in the ROB. An instruction can be committed when it has been fully executed and pointed to by the `pop` pointer. The ROB ensures that instructions are committed in program order, even though they may be executed out-of-order. This maintains the architectural state consistency, as any exceptions or interruptions can be handled correctly without compromising the final program state.

To enable out-of-order execution of instructions, the processor needs to maintain a list of the dependencies between the instructions that are being processed.

This is done using the Issue Queue and the Dispatch Unit. The Issue Queue stores the decoded instructions prior to their execution; it also keeps a representation of the dependencies between all the instructions it stores. The Dispatch Unit controls which instructions enter and leave the queue. When inserting new instructions, the Dispatch Unit retrieves the `RobId`s of all instructions that write to the source registers of the instructions being pushed and then computes the dependencies for the new instruction. When instructions are ready and the corresponding execution units are available, the Dispatch Unit pops these instructions from the Issue Queue and sends them to the adequate execution unit. The number of instructions that can be popped from the Issue Queue each cycle depends on the number of available execution units.

Note that in the ROB and Issue Queue, the registers are not the architectural registers (`x0`, ..., `x31`) - which correspond to the programmer-visible registers defined by the instruction-set architecture (ISA) - but physical registers. Physical registers are part of the microarchitecture and are mapped to architectural registers in the Rename stage. They are used to avoid data hazards during out-of-order execution.

5.2 Fences Implementations

The implementation of the proposed fences requires modifications to various elements of the microarchitecture. We will discuss these modifications for each fence type in the NaxRiscv core.

Serialization fences: To satisfy the predecessor dependency semantic, the Dispatch Unit monitors whether `rs1` = `x0` when inserting a `fence.ser` instruction. If this is the case, it reads all entries in the Issue Queue and creates a dependency with all valid instructions that have a destination register. Similarly, the Issue Queue manages the successor dependencies with respect to `rd`. If a newly pushed instruction is a fence and has `rd` = `x0`, this means that all architectural registers (`x1` - `x31`) depend on it. To represent this, we add a new flag called `isFullFence` to the associated instruction slots in the Issue Queue and set it to True. Any incoming instruction will automatically depend on any instruction in the Issue Queue that has the `isFullFence` flag set to True. Our `fence.ser` implementation uses the physical register of `rs2` to temporarily store the value of the *old physical register* of `rd`. During the execution stage, this value is written to the new physical register of `rd`. This ensures that the destination architectural register remains unchanged, even after the Rename stage, and the normal program behavior is not altered. The use of `rs2` is not visible at the architectural level.

Speculation fences: Speculation fences operate in a similar fashion to the serialization fences. The key difference is that speculation fences cannot complete their execution while in a speculative state, causing them to stall at a specific stage in the pipeline. As the NaxRiscv CPU does not differentiate between

speculative and non-speculative execution (i.e. all instructions are executed speculatively), a **speculation detector** module has been implemented to identify whether an instruction is considered speculative (when we cannot be sure that it will be committed or not). It uses the ROB to determine if an older instruction capable of triggering speculation has not yet been committed. The **speculation detector** takes the `RobId` of an instruction as an input and returns a boolean value indicating whether the instruction is speculative.

We have implemented three different approaches to implement speculation fences, depending on how they stall when considered speculative.

Execute-Stall Fence Speculation fences cannot exit the execution stage of the pipeline while in speculative mode, thanks to the speculation detector. This approach minimizes performance loss by delaying the stall until the last possible stage before execution. However, this method has a significant drawback: it can deadlock the pipeline under certain conditions. If a speculation fence instruction `B` is speculatively executed before an older one `A`:

- Instruction `B` stalls on its execution unit until it is no longer in a speculative mode.
- Instruction `A` waits for the execution unit to become available as it is currently in use by the instruction `B`.
- Instruction `A` enforces the execution order, ensuring that the instruction triggering the speculation executes only after it completes.

To avoid this effect, the Dispatch Unit adds dependencies between each fence instruction to enforce in-order execution of fence instructions and avoid the deadlock.

Dispatch-Stall fence The stall can be applied at the dispatch stage, before the fences enter the execution stage. This enforces in-order execution of the fence instructions. The Dispatch Unit holds the instruction until the speculation detector confirms a non-speculative state.

Operand-Stall fence Another implementation, called the Operand-Stall Fence, deviates from the original semantics of speculative fences described in Subsection 4.1. Nevertheless, this mechanism may offer an alternative perspective on speculative fences. In this implementation, the fences can be dispatched to an execution unit as soon as the source register `rs1` is committed. Instead of confirming the execution states of the instructions, this implementation prioritizes the architectural correctness of source registers. In other words, it ensures the convergence state of the source register before allowing the execution of the instructions. Thus, this implementation allows the instructions to execute speculatively, which diverges from the original semantics.

Conditional speculation fences: Conditional Fences (`fence.cond`) are unique among the added fence instructions, as they use only a source register `rs1`. During the dependency computation in the Dispatch Unit, each subsequent register

automatically depends on all conditional fences present in the Issue Queue (similar to the previous `rd` = `x0` case). The execution units read the predicate from `rs1`, as the behaviour of a conditional fence depends on its value. If `rs1` $\neq$ 0 (signaling a misspeculation), the execute unit stalls until the `pop` pointer of ROB reaches the conditional fence's `RobId`, forcing the previous conditional branch to commit and trigger a rescheduling first. Unlike SLH [9], this technique halts speculative execution entirely, ensuring no speculative traces are left on the microarchitecture.

6 Security policies to insert fences

The proposed fence instructions enable the evaluation of various strategies at the compilation level. As noted in 3.1, prior work on compiler-based Spectre mitigations has focused mainly on x86 and ARM architectures [26]. Some mitigations are straightforward enough to adapt to `RISC-V` architectures and serve as our initial performance references.

Since memory instructions are the primary source of leakage in most Spectre attacks, we implemented an LLVM pass similar to SESES [8] that inserts `fence.spec` instructions near memory operations. However, we must ask ourselves: if only load instructions must be hardened, do stores also need to be protected? Furthermore, should fences be placed before or after the target instructions? Also, could fences be inserted around end-of-branch blocks, calls, or indirect jumps? Each stage of a Spectre gadget plays a crucial role in executing a Spectre attack. The speculation and disclosure stages can be carried out using various instructions and may vary significantly between different microarchitectures. However, the acquisition stage can only be performed using an instruction capable of loading the secret into a register - a `load` instruction.

These considerations result in numerous policies based on the combinations of options, including the various fence semantics introduced earlier. An exhaustive list of these policies can be found in Table 1.

To replicate SESES behavior in the LLVM `RISC-V` back-end, we replaced `LFENCE` with our new instructions, leveraging their similar semantics. For instance, `specall_before_load` uses `fence.spec.all` (`fence.spec x0, x0`), while policies prefixed with `ser` replace `fence.spec` with `fence.ser`. We then modified this pass to handle the cases where `rd` and `rs1` are not equal to `x0`.

To measure performance costs, we implemented a `nop` policy inserting `nop`s (a pseudo-instruction for `addi x0, x0, 0`), which advances the program counter without architectural impact. Despite its simplicity, it affects both security and performance, serving as a baseline.

We implemented `spec_after_load` for comparing the placement of fences after memory instructions with respect to placing them before. Additionally, the `dependency` approach adds fences with operands based on branch conditions, such as `fence.spec rd, rs1` (`spec_dep_load`) or `fence.ser rd, rs1` (`ser_dep_load`). Basic optimizations prevent redundant protection of non-redefined registers within the same basic block.

SLH, introduced in 3.1, can also be implemented on `RISC-V` architectures. On `x86`, "misspeculation predicates" are updated with `cmov` instructions, which are immune to prediction. Since `cmov` is absent in base `RISC-V`, it can be emulated using bitwise operations like `slt`, setting `rd` to 0 or 1 based on comparisons. Avoiding conditional branches prevents new speculation points, though this requires additional instructions compared to `cmov`.

SLH was previously translated for `RISC-V` by Moein Ghaniyoun [12] in the LLVM back-end. We modified their implementation to create `slh` and `slh-ip` policies, where the `-ip` suffix denotes *Inter-Procedural* predicate transfer via the stack pointer.

A conditional fence can be used to enhance this strategy: instead of poisoning the load's value or address, a `fence.cond` at the start of each block uses the SLH predicate (if a load exists) to prevent speculative execution of the entire block, ensuring no transient microarchitectural states. This can also be extended inter-procedurally, forming the `spec_cond_ip` policy.

Gadget counts in Table 1 are the same ones as used to generate Figure 3 but different than in Figure 2. Indeed, in Figure 2 two gadgets occurring at the same acquisition address are counted twice if they have different speculation sources, but are counted once in Table 1 and Figure 3.

Performance is measured as the geometric mean of the number of hot cycles (after the warmup) for the benchmark suite.

7 Benchmark results

To evaluate the effectiveness of the proposed fence semantics and insertion policies, we first need hardware that implements the former. The three different types of stalling mechanisms (execute-, dispatch- and operand-stall) were added in the Naxriscv core, making three different processors. We also need a set of benchmark programs to evaluate the potential security and performance implications of our methods. We have selected the Embench suite [5] for such purposes, as it is open-source and exercises a wide range of applications. Each program in the benchmark suite was evaluated using each processor and each fence insertion policy previously described. A "hardware-policy" configuration therefore corresponds to the specific benchmark binaries compiled with said policy and run on the simulator corresponding to said hardware. The simulations were carried out using the Verilator simulator, and execution traces were collected in a format closely resembling O3PipeView from Gem5. From these traces, different analysis passes were run to extract metrics: instruction mixes, performance, security, misspeculation windows, etc. These traces also enable the extraction of the microarchitectural state in case of any detected vulnerability.

Since execution traces only represent executed code, our results only apply to these executions. In particular, the absence of any Spectre gadget in the traces is no proof that no gadgets could appear, only that they have a low probability of doing so.

7.1 Security metrics

Before analyzing the traces, we must define a security metric. Most of the previous work considers security as a binary value; the execution is either safe or unsafe with respect to a set of exploits (Spectre-PHT, Spectre-BTB, etc). We propose a different strategy that is not tied to any exploit. We identify a Spectre gadget as the combination of 1) a misspeculation trigger, an instruction that initiates a misspeculation window, 2) a secret acquisition via a `load` instruction, and 3) the secret disclosure, where a value depending on the secret is leaked. In our case, a disclosed value is any value used as an issued `load` address, an issued `store` address or value, or an issued `branch` operand. During trace analysis, taint tracking is used for all potential secrets resulting from a `load` up to a disclosing instruction. To accurately assess policies that poison addresses (e.g. `slh`), we detect the cases where `load` instructions target the null address and rule out these gadget cases.

We consider that any such gadget *could* lead to an exploit and must be prevented.

It is possible to count the number of these Spectre gadgets in the execution traces for all benchmarks for each configuration. A gadget is identified by the address of the acquisition instruction, meaning that if two gadgets found in the execution trace share the same acquisition instruction, they are only counted once. This is required since all our benchmarks consist of a loop performing some operation: the same gadget may be found many times in the traces for each iteration of the loop. It would be a measure of the benchmark loop count, not a measure of security, if we counted repeating gadgets.

For any hardware-policy configuration, the security metric is the sum of Spectre gadgets detected across all benchmarks for this configuration.

Figure 2 depicts the proportions of speculation sources for the Spectre gadgets that were detected for several hardware-policy configurations.

We can observe in Figure 2 that most of the gadgets found start from a misspeculated branch direction (*PHT*). It is also worth noting that some implementations are able to mitigate all the gadgets while others focus on disabling only some kinds of gadgets. For instance, `serall_*` policies will serialize store and load instructions, preventing any *STL* gadgets.

7.2 Security-performance trade-off

Figure 3 represents where the different hardware-policy configurations stand in terms of gadget-count to performance loss ratio, compared to an unmodified processor. It is a graphical representation of the data contained in Table 1. We can observe a well-defined Pareto front that spans from the best-performing, unprotected configuration to the worst-performing, most-protected configuration. The latter is equivalent to in-order execution, according to McFarlin et al. [23].

The `slh` policy, which does not use speculation barriers, is on the Pareto front. But this policy is not exempt from gadgets, an expected result since it

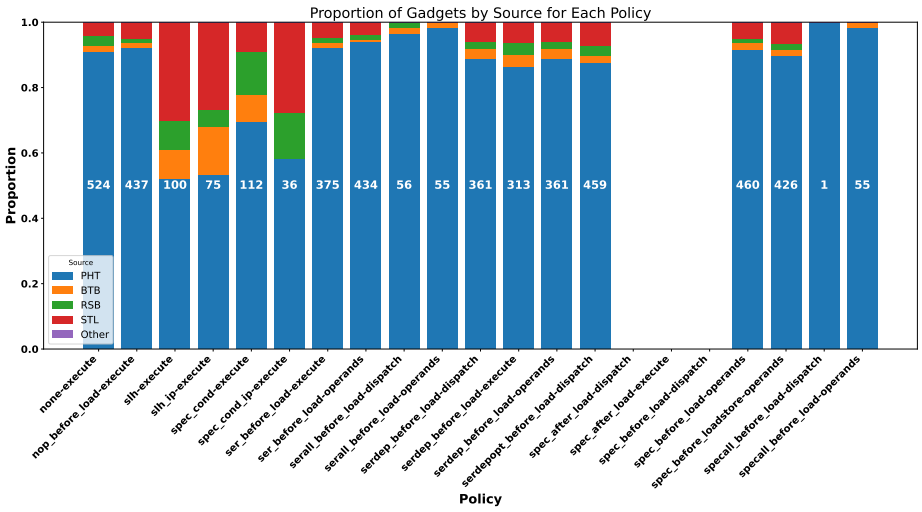

Fig. 2: Gadget speculation sources, the center number is the gadgets count.

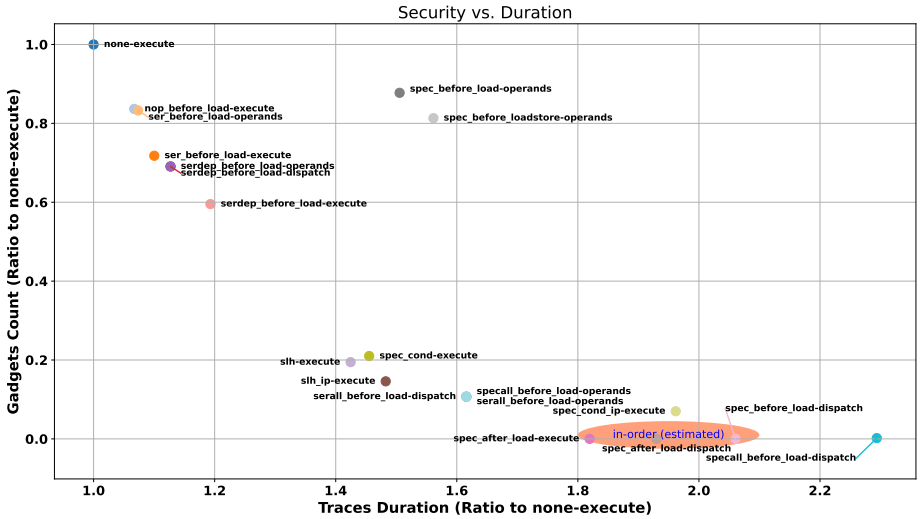

Fig. 3: The hardware-policy configurations according to both their security and performance metrics.

Table 1: List of evaluated policies and results

		Gadget counts			Benchmark duration geomean (hot cycles)		
Policy name	**Description**	`execute`	`dispatch`	`operands`	`execute`	`dispatch`	`operands`
`none`	No policy applied (baseline).	514			3.6 M		
`nop_before_load`	Insert `nop` instruction *before* each `load`.	430			3.8 M		
`ser_before_load`	Insert `fence.ser rA, rA` instruction *before* each `load rB, offset(rA)`.	369	428	428	3.9 M	3.8 M	3.8 M
`serall_before_load`	Insert `fence.ser x0, x0` *before* each `load`.	16	55	55	7.0 M	5.8 M	5.8 M
`serdep_before_load`	Insert `fence.ser rB, rA` with `rB` used as address by the following `load` and `rA` register from the "most dominant branching instruction operands". It is a naive approach to add serialization between `load`s and most dominant branching instruction if there is one.	306	355	355	4.3 M	4.0 M	4.0 M
`slh`	SLH implementation for RISCV.	100			5.1 M		
`slh_ip`	SLH implementation with inter-procedural predicate transfer via a stack pointer.	75			5.3 M		
`spec_after_load`	Insert `fence.spec rB, rB` instruction *after* each `load rB, offset(rA)`.	0	0	0	6.5 M	6.9 M	6.6 M
`spec_before_load`	Insert `fence.spec rA, rA` instruction *before* each `load rB, offset(rA)`.	0	0	451	7.4 M	7.4 M	5.4 M
`spec_before_loadstore`	Insert `fence.spec rA, rA` instruction *before* each `load rB, offset(rA)` or `store rB, offset(rA)`.	3	3	418	8.1 M	8.0 M	5.6 M
`spec_cond`	Insert `fence.cond rA` at the beginning of each basic block that contains `load`. It takes as operand `rA`, an always updated predicate computed as in `slh` policy.	108			5.2 M		
`spec_cond_ip`	As `spec_cond` with inter-procedural predicate transfer.	36			7.0 M		
`specall_before_load`	Insert `fence.spec x0, x0` *before* each `load`.	0	1	55	8.4 M	8.2 M	5.8 M

Gadget counts in Table 1 are the same ones as used to generate Figure 3 but different than in Figure 2. Indeed, in Figure 2 two gadgets occurring at the same acquisition address are counted twice if they have different speculation sources, but are counted once in Table 1 and Figure 3.
Performance is measured as the geometric mean of the number of hot cycles (after the warmup) for the benchmark suite.

targets the PHT variants only. Less expected, we can see in Figure 2 that numerous PHT-variant gadgets are still present. Upon closer inspection, permitted by our tooling, it appears that most of these gadgets are due to the `slh` interprocedural predicate being loaded from the stack. Furthermore, `slh` is done before register allocation with spilling that introduces new unmitigated `load`s. Our tests correctly determine that these `load`s could potentially be hijacked by an attacker, thus forming a Spectre gadget. More generally, proving a correct compiler mitigation pass can never be a perfect solution: the arbitrary control flow in the microarchitecture does not care about the proofs on the binary.

Our policy `spec_cond-execute` implements the same predicate technique but, using our new instruction `fence.cond`, has a security-performance tradeoff close to `slh`. However, instead of poisoning load pointers, the `fence.cond` disables any speculative execution, incurring a slight overhead compared to `slh`.

Since widely different hardening techniques, compiler- or hardware-based, are aligned on Figure 3, it is clear that the ability to execute memory requests speculatively is the main performance driver for the NaxRiscv core. In our case, configurations that achieve 0 gadgets have performance levels equivalent to those of an in-order core. **The indiscriminate use of speculation fences, which is the only viable strategy without microarchitectural knowledge of which data is confidential, does not allow for secure execution at out-of-order performance levels.**

Since microarchitectural control flow can be arbitrary, and thus any fence might be speculatively bypassed, one might expect that no policy could prevent all gadgets. Yet some policies, such as `spec_after_load-execute`, have successfully prevented all gadgets in our benchmarks. Therefore, even if no speculation barrier-based policy can guarantee the complete absence of gadgets, it may still offer strong security in practice.

7.3 Area comparison

Since all the changes that were brought to the Naxriscv core are synthesizable, we can also measure the hardware costs of the proposed mechanisms. As a baseline, we considered the core configured as dual-issue with a 32-level deep Issue Queue, with support for single and double precision floating point operations, compressed instructions, and four execution units in total. A Xilinx XC7K325T was used as the target field-programmable gate array (FPGA), and synthesis was done using Xilinx Vivado 2023.1. In its base configuration, the NaxRiscv uses $\approx$ 26k look-up tables (LUTs) and $\approx$ 15k flip-flops (FFs), and the maximum operating frequency reported by Vivado is 51.6 MHz. As shown in Table 2, the overhead is relatively significant with +6.5k LUTs or +25% at best for the `execute-stall` hardware variant. The costs include the addition of an execution unit dedicated to our fences and the speculation detector ($\approx$ 0.9k LUTs).

When analysing the critical paths of our modified cores, we can observe that the modifications of the Dispatch Unit and the addition of a speculation detector play a major role in the operating frequency degradation. The speculation

detector has to perform chained comparisons of all the possible `RobId`s, adding a significant amount of logic on the critical path.

Table 2: FPGA Resources Comparison for Fence Implementations

Fence Implementation	LUTs	FFs	FMAX
Baseline	25811	15105	51.6 MHz
`execute-stall`	32375 (+25%)	16241 (+8%)	38.1 MHz
`dispatch-stall`	29076 (+13%)	15437 (+2%)	50.0 MHz
`operand-stall`	31024 (+20%)	15318 (+1%)	32.9 MHz

7.4 Limitations

Given the significant effort required to experiment with both custom out-of-order cores and custom compilers, some of our choices have been dictated by technical feasibility and may limit the scope of our results.

First, we must note that, to achieve a secure implementation, the number of gadgets must be 0. Any other value implies that an attacker could potentially exploit the specific conditions that result in a gadget. Our security metric serves merely as an indication that the implementation is secure, not as a definitive proof.

Also, it is possible that a bigger Naxriscv configuration, with a wider issue width and a larger Issue Queue, could lead to better performance for the same level of security by enabling more reordering possibilities. However, in our hardware implementations that strictly adhere to the `fence.spec` semantics, fences cannot be reordered relative to one another, imposing a strict limit on reordering possibilities overall.

Finally, a known mitigation is `delay-on-miss`, where the core delays the execution of speculated `load`s only if the target is not in cache memory. This countermeasure cannot be efficiently evaluated in our settings: the Embench benchmarks focus on hot execution paths, where instructions and data are already in cache if possible. But the `delay-on-miss` mitigation has a different threat model: that data in cache memories is public. In our own threat model where data in cache could be confidential, the `delay-on-miss` therefore appears as a new point that coincides with `none-execute`: fast and non-secure, which has been confirmed experimentally.

8 Conclusion

Despite the strong understanding of how Spectre leaks data, research into mitigation continues to face difficulties in providing complete protection against this vulnerability while maintaining a reasonable performance level. In this evaluation of selective speculation through speculation barriers, every `load` instruction

is assumed to potentially access sensitive data and requires protection. However, results reveal that this approach is not efficient in such an attack model. By providing speculation barriers, the ISA shifts the responsibility for security to the developer and compiler, which are tasked with using them correctly. However, our results suggest that there is no way to use them correctly and efficiently, which significantly hinders their practical adoption. We recommend that any ISA specification for speculation barriers be accompanied by a thorough evaluation of their security and performance implications. Besides, the performance impact is so significant that adopting an in-order architecture may be a more viable solution for complete protection against Spectre. The main challenge of selective speculation is identifying which instruction or data must be restricted from speculative execution to avoid data leak.

Lack of insight into the program's secrets forces the hardware to overprotect itself, resulting in unnecessary performance losses. Yet, could this trade-off between performance and security be overcome if the architecture had precise knowledge of sensitive data?

References

1. Abdul Kadir, M.F., Wong, J.K., Ab Wahab, F., Abidin Bharun, A.F.A., Mohamed, M.A., Zakaria, A.H.: Retpoline technique for mitigating spectre attack. In: 2019 6th International Conference on Electrical and Electronics Engineering (ICEEE). pp. 96–101 (2019)
2. Aciiçmez, O., Koç, Ç.K., Seifert, J.: Predicting secret keys via branch prediction. In: Abe, M. (ed.) Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4377, pp. 225–242. Springer (2007)
3. Barber, K., Bacha, A., Zhou, L., Zhang, Y., Teodorescu, R.: Specshield: Shielding speculative data from microarchitectural covert channels. In: 28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019. pp. 151–164. IEEE (2019)
4. Bernstein, D.J.: Cache-timing attacks on aes (2005)
5. Burgess, A., Whetter, A., Field, G., Markall, G., Oosenbrug, H., Pallister, J., Bennett, J., Grech, N., Langlois, P., Cook, S.: Embench iot: Open benchmarks for embedded platforms. https://github.com/embench/embench-iot (2020), gitHub repository
6. Choudhary, R., Yu, J., Fletcher, C.W., Morrison, A.: Speculative privacy tracking (SPT): leaking information from speculative execution without compromising privacy. In: MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021. pp. 607–622. ACM (2021)
7. Daniel, L., Bognar, M., Noorman, J., Bardin, S., Rezk, T., Piessens, F.: Prospect: Provably secure speculation for the constant-time policy. In: Calandrino, J.A., Troncoso, C. (eds.) 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023. pp. 7161–7178. USENIX Association (2023)

8. Developers, L.: Llvm seses - speculative execution side effect suppression. https://groups.google.com/g/llvm-dev/c/EL8rUhvRCgo, accessed: 2024-10-25
9. Developers, L.: Speculative load hardening. https://llvm.org/docs/SpeculativeLoadHardening.html, accessed: 2024-10-25
10. Escouteloup, M., Lashermes, R., Fournier, J., Lanet, J.: Under the dome: Preventing hardware timing information leakage. In: Grosso, V., Pöppelmann, T. (eds.) Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13173, pp. 233–253. Springer (2021)
11. Fustos, J., Farshchi, F., Yun, H.: Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In: Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019. p. 61. ACM (2019)
12. Ghaniyoun, M.: Moein ghaniyoun's website. https://moeinghaniyoun.github.io/, accessed: 2024-10-25
13. Gras, B., Razavi, K., Bos, H., Giuffrida, C.: TLBleed: When Protecting Your CPU Caches is not Enough. In: Black Hat USA (Aug 2018)
14. Guarnieri, M., Köpf, B., Reineke, J., Vila, P.: Hardware-software contracts for secure speculation. In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. pp. 1868–1883. IEEE (2021)
15. Hu, G., He, Z., Lee, R.B.: Sok: Hardware defenses against speculative execution attacks. In: 2021 International Symposium on Secure and Private Execution Environment Design (SEED), Washington, DC, USA, September 20-21, 2021. pp. 108–120. IEEE (2021)
16. Jin, H., He, Z., Qiang, W.: Specterminator: Blocking speculative side channels based on instruction classes on RISC-V. ACM Trans. Archit. Code Optim. **20**(1), 15:1–15:26 (2023)
17. Khasawneh, K.N., Koruyeh, E.M., Song, C., Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.B.: Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In: Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019. p. 60. ACM (2019)
18. Kiriansky, V., Lebedev, I.A., Amarasinghe, S.P., Devadas, S., Emer, J.S.: DAWG: A defense against cache timing attacks in speculative execution processors. In: 51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018. pp. 974–987. IEEE Computer Society (2018)
19. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 1–19 (2019)
20. Li, P., Zhao, L., Hou, R., Zhang, L., Meng, D.: Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks. In: 25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019. pp. 264–276. IEEE (2019)
21. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: 27th USENIX Security Symposium (USENIX Security 18) (2018)

22. Loughlin, K., Neal, I., Ma, J., Tsai, E., Weisse, O., Narayanasamy, S., Kasikci, B.: DOLMA: securing speculation with the principle of transient non-observability. In: Bailey, M.D., Greenstadt, R. (eds.) 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021. pp. 1397–1414. USENIX Association (2021)
23. McFarlin, D.S., Tucker, C., Zilles, C.B.: Discerning the dominant out-of-order performance advantage: is it speculation or dynamism? In: Sarkar, V., Bodík, R. (eds.) Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013. pp. 241–252. ACM (2013)
24. Percival, C.: Cache missing for fun and profit (2005)
25. Phoronix: Intel cxl r lvi benchmarking. https://www.phoronix.com/review/intel-cxlr-lvi, accessed: 2024-10-25
26. Randal, A.: This is how you lose the transient execution war. CoRR **abs/2309.03376** (2023)
27. Saileshwar, G., Qureshi, M.K.: Cleanupspec: An "undo" approach to safe speculation. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019. pp. 73–86. ACM (2019)
28. Sakalis, C., Kaxiras, S., Ros, A., Jimborean, A., Själander, M.: Efficient invisible speculative execution through selective delay and value prediction. In: Manne, S.B., Hunter, H.C., Altman, E.R. (eds.) Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 723–735. ACM (2019)
29. van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C.: RIDL: Rogue in-flight data load. In: S&P (May 2019)
30. Schwarz, M., Lipp, M., Canella, C., Schilling, R., Kargl, F., Gruss, D.: Context: A generic approach for mitigating spectre. In: 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020. The Internet Society (2020)
31. Shin, Y., Kim, H.C., Kwon, D., Jeong, J., Hur, J.: Unveiling hardware-based data prefetcher, a hidden source of information leakage. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 131–145. ACM (2018)
32. SpinalHDL: Naxriscv: An out-of-order risc-v cpu core. https://github.com/SpinalHDL/NaxRiscv (2024), accessed: 2024-07-11
33. Taram, M., Venkat, A., Tullsen, D.M.: Mitigating speculative execution attacks via context-sensitive fencing. IEEE Des. Test **39**(4), 49–57 (2022)
34. Van Bulck, J., Moghimi, D., Schwarz, M., Lippi, M., Minkin, M., Genkin, D., Yarom, Y., Sunar, B., Gruss, D., Piessens, F.: Lvi: Hijacking transient execution through microarchitectural load value injection. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 54–72 (2020)
35. Vassena, M., Disselkoen, C., von Gleissenthall, K., Cauligi, S., Kici, R.G., Jhala, R., Tullsen, D.M., Stefan, D.: Automatically eliminating speculative leaks from cryptographic code with blade. Proc. ACM Program. Lang. **5**(POPL), 1–30 (2021)
36. Weisse, O., Neal, I., Loughlin, K., Wenisch, T.F., Kasikci, B.: NDA: preventing speculative execution attacks at their source. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019. pp. 572–586. ACM (2019)
37. Wistoff, N., Schneider, M., Gürkaynak, F.K., Benini, L., Heiser, G.: Microarchitectural timing channels and their prevention on an open-source 64-bit RISC-V core.

In: Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021. pp. 627–632. IEEE (2021)
38. Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C.W., Torrellas, J.: Invisispec: Making speculative execution invisible in the cache hierarchy. In: 51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018. pp. 428–441. IEEE Computer Society (2018)
39. Yu, J., Mantri, N., Torrellas, J., Morrison, A., Fletcher, C.W.: Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In: 47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020. pp. 707–720. IEEE (2020)
40. Yu, J., Yan, M., Khyzha, A., Morrison, A., Torrellas, J., Fletcher, C.W.: Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data. IEEE Micro **40**(3), 81–90 (2020)
41. Zhao, Z.N., Ji, H., Yan, M., Yu, J., Fletcher, C.W., Morrison, A., Marinov, D., Torrellas, J.: Speculation invariance (invarspec): Faster safe execution through program analysis. In: 53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020. pp. 1138–1152. IEEE (2020)