Hardware-Software Defense Idea: Data-flow Assertion Instructions

Guillaume Didier[0009−0007−9076−7318]

Universität des Saarlandes, Saarland Informatics Campus, Germany
guillaume.didier@uni-saarland.de

Abstract. Speculative execution in modern processors primarily affects control flow. In contrast, along a given (potentially speculative) control-flow path, the data flow of the program is largely preserved. This observation motivates a software-level approach to mitigating speculative-execution attacks: encoding control-flow invariants within a program's data-flow dependencies. Speculative Load Hardening (SLH) is one instance of this class. To support efficient implementations of such defenses, we propose *data-flow assertion instructions*. Such instructions combine two functionalities: 1. Like a conditional move, they transfer data from one register to another when a condition holds. 2. Acting as an assertion, they raise a fault if that condition is violated in sequential execution. We specify two variants of these instructions as RISC-V extensions, discuss several ways to employ them in Spectre defenses, and include an evaluation plan for this hardware-software co-design approach.

Keywords: Micro-architectural Security · Transient Execution Attacks · Hardware-Software co-design

1 Introduction

Due to their speculative out-of-order designs, modern high-performance processors are vulnerable to a variety of transient execution attacks [7, 11, 12]. While attacks arising from transient execution beyond faults, such as Meltdown [12], have mostly been fixed in hardware in subsequent generations, proposed hardware fixes for transient execution attacks arising from other sources of speculation, such as Spectre [11], have a high overhead and have not been implemented. Indeed, such speculation is needed to fill in the out-order execution window. Speculative Load Hardening techniques [3, 4, 5, 14, 15] are pure software mitigations that protect against several of those attacks in a reliable way, by encoding those invariants into the data flow, using straight-line code, such as conditional moves or masking. Triosecuris [4] is the latest refinement of this approach and covers all Spectre variants that arise from front-end predictions (branch target, branch direction, indirect branch target and return address). For example, for a bound check, the misspeculation flag can be updated using the comparison result, and used to conditionally replace the load address with a null pointer, or mask the loaded value. However, manufacturing safe values can incur a significant overhead, and the data-dependency in SLH can still result in a small transient window of

execution on the wrong-path. The masking is required to ensure safety during the race between the execution of the instruction using the masked value and the wrong-path squashing. We seek to guarantee that those subsequent instructions cannot even be dispatched.

We propose replacing the safe value manufacturing with a conditional data-flow fence. In the transient domain, instead of forwarding a safe value, a violation of the condition results in *a lack of value*, stalling all dependent instructions. Moreover, in correct programs, such a conditional speculation fence should never retire if the condition is not verified, as such we instead propose that its sequential behavior be that of an assertion. If the condition is violated sequentially, our assertion instruction results in a fault, expanding its applicability to other cases, e.g., defensive programming situations where violation of a branch condition would reflect a bug. This sequential behavior is only made possible by the conditional aspect of the instruction. We summarize our contributions as follows:

- We propose *data-flow assertion instructions* as a new defense primitive against micro-architectural attacks.
- We show how these can be used to encode invariants from the sequential control flow that are security-relevant to the transient data flow.
- We show how these can be used to simplify defensive programming patterns.
- We specify how to add these extensions to the RISC-V ISA.
- We propose a strategy for the evaluation of our mitigation.

The remainder of the paper is organized as follows: Section 2 introduces relevant background and related works. Section 3 exposes the reasoning on how to transfer the security invariants to the data flow, then leading to Section 4 detailing the instruction specification, as well as constraints on the implementation, and Section 5 shows how it can then be applied to improve mitigations. Section 6 then offers two solutions to extend the RISC-V ISA to include such instructions. Section 7 discusses directions to extend this to support speculation originating in the execution backend rather than the frontend. Section 8 details the strategy to evaluate this mitigation proposal, before concluding in Section 9.

2 Background and Related Work

Modern processors' performance arises from the extraction of instruction level parallelism (ILP) using speculative-out-of-order execution. The front-end stages, fetch and decode, rely on branch prediction techniques to linearize the control-flow graph into a window of instructions to be executed. These instructions are then executed by the CPU backend out of the sequential (or program) order, according to their data dependencies, *i.e.*, following the data-flow partial order.

2.1 Spectre violates invariants encoded in the control flow

Setting aside Store-to-Load speculation [1, 8, 9, 10], Spectre attacks occur as part of transient execution following incorrect predictions from the frontend. As a consequence, Spectre attacks transiently violate security invariants that are

Listing 1: Illustration of a Spectre-PHT (branch-direction) attack and defenses, as well as defensive bound checks

(a) Spectre-PHT attack

```
1 if (i < size) {
2     char a = array[i];
3     char b = array_2[a * 64];
4 }
```

(b) SLH applied to Spectre-PHT

```
1 /* previous misspeculation flag */
2 bool msf = ... ;
3 if (i < size) {
4     msf |= !(i < size)
5     char a = array[i];
6     int safe_a = msf? 0 : a;
7     char b = array_2[safe_a * 64];
8 }
```

(c) Example defensive bound checking

```
1 if (i >= N) abort();
2 char a = array[i];
```

(d) Same SLH with data-flow assertion

```
1 if (i < size) {
2     msf |= !(i < size)
3     char a = array[i];
4     int safe_a = dassert(!msf, a);
5     char b = array_2[safe_a * 64];
6 }
```

(e) Data-flow assertion for bounds checking

```
1 int safe_i = dassert(i < N, i);
2 char a = array[safe_i];
```

(f) Optimized bounds checking, with assertion+SLH

```
1 int safe_i = dassert(!msf && i<N, i);
2 prefetch(&array[i]);
3 char a = array[safe_i];
```

encoded in the control flow. In the classical Spectre-PHT (Pattern History Table) speculative bound bypass (Listing 1a), involving an incorrect branch direction prediction, the sequential semantics guarantee that accesses are in-bound via the control flow, as the load instruction is only executed if the bound check condition is satisfied, encoding the invariant: *arrays shall not be accessed out of bounds.*

However, such invariants are not preserved by transient execution. The branch predictors predict the direction of conditional branches based on the past, and can be misled into predicting an incorrect direction, e.g., when the attacker has caused the branch in Listing 1a to be executed with an in-bound index a large number of times, the predictor will assume the next index is in-bound, which can result in a transient out-of-bounds access. More precisely, line 1 enforces the security invariant — the array access shall be in-bounds — in the control flow. Under attack, the branch is mispredicted, and the following instructions are speculatively executed, with an invalid array index. On line 2, the memory access reads out-of-bound data, for instance a secret, and forwards it to dependent instructions. On line 3, the leaked value is then encoded on a micro-architectural covert channel, here using a classical cache channel. The attacker can then later determine which address was accessed by line 3 and deduce the leaked value.

2.2 Pure-software injection of invariants into the data flow

The family of Speculative Load Hardening mitigations [3, 4, 5, 14, 15] are a pure software mitigation that detect misspeculation with branchless code, and when misspeculation is detected, replace unsafe operands with safe values. For instance, replacing the memory load address with a null pointer, or `or`-ing the value with an all-ones mask, as illustrated in Listing 1b, using a conditional move or an `or` instruction. If the load instruction used a complex addressing mode, this requires

computing the address before masking it, increasing the number of instructions. On x86, the presence of the flag register may make the `or`-ing more expensive if the flag register is live at that point of the program. Depending on the mitigation variant, either load addresses or load values are protected, and sometimes inputs to other instructions with operand-dependent timings, e.g., divisions.

SLH [3, 5, 14, 15] initially concerned itself exclusively with branch direction (Spectre-PHT), however, Triosecuris [4] has recently shown it is possible to defend against all control-flow speculation, provided some reasonable guarantees on the speculative control-flow. They first require that a CPU only predicts returns via the RSB (and does not fall-back to the BTB on RSB underflow), thus ensuring a return can only return to an instruction immediately after a call, but possibly the wrong one. They also require on the CPU enforcing in the transient domain that any other jump leads to a branch target (e.g., `endbr64` or `endbr32` on x86), when using branch target protections such as Intel's Indirect Branch Tracking (IBT) or ARM Branch Target Identification (BTI). These requirements basically prevent basic blocks from being split in the transient domain, and make multi-instruction approaches possible. This thus showed SLH can be extended to all forms of control flow speculation, including branch direction, direct and indirect branch or jump targets as well as call and returns.

SLH does operate with the speculative values in the data flow, but as they chain the detection of misspeculation on all branches, and exclude other forms of backend speculation, this guarantees the manipulated values are correct. For instance, in a bound check, the array size cannot be incorrect without a misspeculation being already detected.

2.3 Hardware-software solutions based on new fence instructions

Several hardware-software solutions rely on new fence designs, less heavy than a full serializing fence (waiting until all earlier instructions have retired before letting any subsequent instruction execute). These solutions then have different strategies around fence placement. Andrianatrehina et al. [2] made a survey of such solutions, evaluating their performance, and introduced the following fences:

- `fence.ser rd, rs1`, a serializing fence, which introduces a dependency between subsequent instructions using `rd` and the previous instruction writing `rs1` (`x0` implies all registers)
- `fence.spec rd, rs1` a register-register speculation fence, whose execution can only terminate in a non-speculative state.
- `fence.cond rs1` a *conditional speculation fence*, whose execution stalls until non-speculative when the predicate is non-zero, and is depended on by all instructions with a source register, behaving as `fence.spec` with `rd == x0`.

Davoli et al. [6] propose `dfence`, an unconditional data-flow fence implemented on the Proteus core, also tracking speculation originating from the backend, and develop mitigations for store-to-load forwarding speculation and value prediction. `dfence` forwards its input to its output once it is no longer speculative.

3 Enforcing program security invariants in the data flow

To ensure the invariant on a specific value (e.g., an array index or a value read using such an index) in the data flow, we wish to create a data dependency from the condition to the possibly unsafe value. As soon as the invariant condition can be verified (e.g., the array index is in-bounds, or possibly, in a safe region containing no secrets), it is safe to forward the previously unsafe value, which is now safe (*i.e.*, forward the array index or the value read using it). If however, the invariant is violated (*i.e.*, in our example, the array index is out-of-bounds), this value must not be used by subsequent instructions. While processors try to squash incorrect branches as fast as possible, this is still a race condition, which can result in the value being used. The SLH solution to this is to manufacture a *safe replacement*, which may be expensive. We propose instead for the instruction to not forward the unsafe value, and thus stall any dependent instruction. Such a behavior can be intuitively understood as combining the previous `fence.spec` and `fence.cond`. If a less restrictive and more easily compute invariant suffices to guarantee confidentiality in the transient domain, the execution stall can be shortened, with subsequent instructions making progress while still speculative[1].

In this specific case, compared with SLH, we reduce the use of execution units by instructions that will get squashed, preventing them from delaying useful instructions, and improving energy-efficiency, by avoiding needless work.

However, if the condition of the fence must be verified in the sequential domain, as a consequence of the control-flow invariants, then such a conditional fence must never retire when its condition is false. As such, if such an instruction were to be sequentially executed with an invariant violating value, this would actually reflect a program bug, or a memory corruption. A fence instruction would not change the semantics of the program, and let the bug propagate. But from a security point of view, we would have then detected an illegal situation in the sequential-domain. We thus advocate for a different behavior: **reporting the violation, and thus causing the program to be interrupted**.

In practice, this can be done by faulting, handing control to the operating system fault handler. It could then, for instance, deliver a signal to the offending process and allow the process to take appropriate actions, e.g., calling `panic!` in a Rust program. We thus call such an instruction a **data flow assertion** instruction. In C/C++, we represent it as `T dassert(bool condition, T value)`.

This comes with an additional for branches used for defense in depth, that e.g., triggers a `panic!` / `abort` when an invariant is violated: If the same invariant check must be enforced in both the transient and sequential domain, then the assertion instruction makes the branch redundant. We can remove the branch, simplifying the control-flow graph, e.g., transforming Listing 1c into 1e or 1f.

[1] If a ring buffer contains no sensitive data, the sequential check on the current dynamic size can be protected in the transient domain with the constant maximal size, as a transient access to a free element of the ring buffer does not breach confidentiality.

4 Requirements for the assertion instruction

This section aims to define the sequential and transient behavior of the data-flow assertion, as well as requirements on the front end.

The data flow assertion has two register inputs, a condition input and a data input, and one register output. Sequentially, if the condition is true, the input is written to the output register. If the condition is false, the assertion instruction results in a fault, and the output register does not get written.

Transiently, the instruction uses the condition and input registers speculatively. If the condition is true, the input is forwarded to the output, which let subsequent instructions dispatch, otherwise no output is produced. The CPU must also not speculate on the condition value.

As in Triosecuris [4], minimal guarantees on the front-end are needed to ensure that sequences of instructions do not get broken up. We require an implementation to ensure that the stream of instructions to the execution backend only includes jumps from decoded control-flow instructions to properly marked branch or jump targets. Including a target instruction for returns would obviate the constraint on the RSB from Triosecuris. Such a requirement can probably be fulfilled by checking these properties in the decode stage at a reasonable cost, detecting blatantly invalid predictions from the fetch-stage after branches get decoded.

5 Applications

5.1 Improved Speculative Load Hardening

This instruction can be used as a drop in replacement of the safe value manufacturing and conditional move, as shown in Listing 1d. This can reduce the register pressure when the safe value manufacturing requires non-trivial computations. If less restrictive conditions are used in the data flow, this also reduces the pressure on execution ressources when misspeculating, as it prevents the execution of wrong-path data-dependent instructions, instead of letting them execute unnecessarily with a safe value.

5.2 Defensive programming assertions

As shown in Listing 1e, it is possible to remove a branch to `abort` from the original code (1c), by using a data-flow assertion, simplifying the control flow, while adding a defense against speculative execution. To make the mitigation fully effective, one would do this in a program where all branches are protected with SLH/Triosecuris, and also use the misspeculation predicate as part of the assertion. Out approach removes branches that otherwise would need to be protected by SLH/Trisecuris, updating the speculation predicate and manufacturing a safe value, and instead protects the load in the sequential and transient domain using the data-flow assertion.

In the case of bound checking, an alternate solution could also use the `dfence` instruction [6], to guarantee the bound value is correct, instead of using a SLH/Triosecuris misspeculation predicate.

Listing 1e implements a standalone memory address defense (without protections against incorrect speculation elsewhere in the program). In the sequential domain, applying the protection to the load result would result in undefined behavior (sequential out-of-bound access), as per the C standard, hence we must instead protect the load address.

To improve performance, back to the level of the load value masking variant of SLH, which hides more of the load latency, Listing 1f adds a prefetch to the unchecked address, placed after the data assertion in the sequential program order, to avoid undefined behavior, but which can be executed earlier in the out-of-order back end. It also uses the misspeculation flag, integrating into a SLH/Triosecuris protection against misspeculation in the whole program. For a store, this sequence could be altered to use a suitable prefetch for write instruction.

6 RISC-V extension proposals

We propose two variant for this extension. In either case, because RISC-V instructions have at most two source operands and one destination operand, we have to split the assertion into two parts. A first instruction computes the condition, if needed, and a second instruction takes the source register, and the condition as sources, and the destination register as the destination operand.

The two solutions differ in how the condition is managed: in the first proposal the condition is stored in a general purpose register; while in the second case, we extend the instruction set to support a set of eight 1-bit predicate registers. This reduces the register pressure and avoids wasting whole general purpose registers to store a single bit of information.

6.1 Simple RISC-V approach

To generate the conditions, we can use existing `setcc` instructions (e.g., `stl`, set less than), as well as `and` and `or`. We only need an R-type data-flow assertion instruction: `dassert rd, rs1, rs2`, occupying very little encoding space.

6.2 Predicate registers

Predicate registers are 1-bit registers. Introducing them in RISC-V would be a much more significant extension; this strategy would be more suitable for architectures which already include such registers. Variants of the `setcc` instructions would be added, writing to one of 8 predicate registers, using 3 bits to encode the destination. We propose the mnemonic `setpcondition pn, rs1, rs2`

However, operand read could use a mask to select several predicates to be and-ed or or-ed together: `dassertpand rd, rs1, p{i,j,...k}` , `dassertpor rd, rs1, p{i,j,...k}` , `andp pd, p{i,j,...k}`, `orp pd, p{i,j,...k}`. These instructions are encoded as I-type instructions, with 8 of the 12 immediate bits used for the predicate mask. 4 extra bits remain to distinguish other instructions, or variants.

Following from this idea, this extension could probably also include constant-time conditional moves, as well as branches on predicate register(s). An alternate design would do away with the ability to refer to arbitrary subsets of the predicate

registers as inputs, and should piggy-back on the existing design of the predicates for ISAs already including such a feature.

In any case, such a design allow would SLH to use predicates for the misspeculation flag computation, and incidentally give compilers more options to store boolean variables, both of which reduce register pressure.

7 Extension to other forms of speculation

Value Prediction is a form of speculation that originates in the back-end. Unrestricted value prediction undermines all techniques relying on data flows, including all SLH variants. In particular, the value speculation masks, or our assertion conditions should not be predicted, as this would make the value meaningless.

However, requiring that all inputs to the assertion be non-value-speculative (though possibly control-flow speculative), may be overly conservative. Indeed, for an array bound check, we need the bound to be correct, but if an incorrect but in-bounds index is used, this does not cause out-of-bounds access, as long as the checked index value is the one used to perform the load (such *time of check, time of use* (TOCTOU) vulnerabilities can only be solved in hardware).

A second kind of speculation originates in the back-end: Speculative store-to-load forwarding, and more generally speculation on address aliasing in the memory queues. These optimization play a large part in hiding load latency, and more work is needed to address these issues at an acceptable performance cost.

As a general principle, some of the available bits used to set the condition instructions could be used to specify which input must be non-speculative, then using the same sort of hardware-implementation as `dfence` [6] to track the speculative status. We would also need to guarantee the absence of prediction on the path from the condition setting to the data-flow assertions. If the ISA provides a sufficient set of instructions whose outputs are guaranteed never to be value predicted, it is also possible to combine the simple version of data assertion instructions with `dfence` to cover value predictions.

8 Evaluation strategy and discussion

8.1 Implementation work

To evaluate this mitigation, the first step is to implement it on an open-source out-of-order CPU, which ideally also has implementations of other comparable HW software mitigations, such as NaxRISC-V used by Andrianatrehina et al. [2], Proteus, used in [6], or BOOM [13].

Separately, compiler toolchains must be adapted to use the software extension. SLH variants are implemented on LLVM, and could be adapted to use the new instructions. One would also want to add some intrinsics to allow programmers to insert manual asserts, and the rust compiler should be adapted to make use of the assertion instruction for its release bound checking, instead of branches.

8.2 Experiments needed

Unprotected Software performance impact This is a hardware-software co-design solution, that aims to protect programs containing sensitive data, but non-sensitive programs would run unmodified. Since hardware modifications are involved, it is important to evaluate the performance impact on general programs. When data-flow assertions are used to replace defense in depth checks like Rust's panicking bound-check, evaluating the difference between a branch and an assertion-based implementation for unprotected programs should also be covered.

Protected software performance impact The performance of security sensitive programs should then be evaluated, to measure the full protection overhead.

Protection effectiveness checking Each defense effectiveness should be evaluated on a data set of transient execution vulnerabilities such as Andrianatrehina et al.'s [2], to evaluate its coverage, and compare it with defenses with similar overhead.

8.3 Expectations

We expect an implementation of the simple RISC-V solution to provide a small improvement to SLH techniques, due to the lessened pressure on execution units, and the reduction in register pressure. We expect the transformation of defensive branches into assertions, to provide significant improvements over an address masking SLH. The performance difference between a value masking SLH and the optimized defense in Listing 1f seems harder to predict.

Depending on the cost of adding predicate registers, we would expect significant improvements to SLH due to the reduced register pressure. The implementation of the faulting behavior of the assertion should be similar to how other faults now avoid Meltdown-like attacks.

9 Conclusion

We have presented *data-flow assertion instructions* as a new hardware primitive supporting Spectre mitigations, designed to transfer security invariants encoded in the sequential control flow into the transient data flow. We demonstrate how they may improve the performance of mitigations such as SLH [3, 5, 14, 15] or Triosecuris [4], and use their sequential behavior of assertion to further improve their performance on defense in depth checks. We have also specified two possible RISC-V extensions to include such instructions, and present a strategy for evaluation. Lastly, we have discussed how this idea could be extended to account for backend speculation, e.g., value prediction or store-to-load forwarding.

Acknowledgments. This work has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101020415). The authors would like to thank J. Baumann and J. Reineke for their constructive feedback.

References

1. AMD, Security Analysis of AMD Predictive Store Forwarding. Whitepaper, AMD (2023). https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/white-papers/security-analysis-of-amd-predictive-store-forwarding.pdf (visited on 06/26/2026)
2. Andrianatrehina, H., Lashermes, R., Paturel, J., Rokicki, S., Rubiano, T.: Exploring Speculation Barriers for RISC-V Selective Speculation. In: ARES (2025). https://doi.org/10.1007/978-3-032-00627-1_9
3. Baumann, J., Blanco, R., Ducruet, L., Harwig, S., Hritcu, C.: FSLH: Flexible Mechanized Speculative Load Hardening. In: IEEE CSF (2025). https://doi.org/10.1109/CSF64896.2025.00023
4. Baumann, J., Kim, Y., Farba, Y., Hriţcu, C., Leatherman-Brooks, J.: Triosecuris: Formally Verified Protection Against Speculative Control-Flow Hijacking. In: IEEE CSF (2026). https://doi.org/10.1109/CSF68417.2026.00036
5. Carruth, C.: Speculative Load Hardening, (2018). https://llvm.org/docs/SpeculativeLoadHardening.html
6. Davoli, D., Bognar, M., Daniel, L.-A., Grégoire, B., Piessens, F., Rezk, T.: dfence: Fine-Grained Speculation Barriers for Efficient and Effective Hardware-Software Protection in the Spectre Era. In: CCS (2026). https://arxiv.org/abs/2608.06124
7. Fiolhais, L., Sousa, L.: Transient-Execution Attacks: A Computer Architect Perspective. ACM Comput. Surv. (2023)
8. Horn, J.: Speculative Execution, Variant 4: Speculative Store Bypass, (2018). https://project-zero.issues.chromium.org/issues/42450580 (visited on 06/26/2026)
9. Intel, Intel [n. d.]. Fast Store Forwarding Predictor, (2022). https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/fast-store-forwarding-predictor.html (visited on 06/26/2026)
10. Intel, Intel [n. d.]. Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115, (2018). https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html (visited on 06/26/2026)
11. 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: 40th IEEE Symposium on Security and Privacy (S&P'19) (2019)
12. 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)
13. Liu, Y., Hsieh, C., Kuo, S.: Boomerang: Physical-Aware Design Space Exploration Framework on RISC-V SonicBOOM Microarchitecture. In: IEEE ASAP (2023). https://doi.org/10.1109/ASAP57973.2023.00026
14. Patrignani, M., Guarnieri, M.: Exorcising Spectres with Secure Compilers. In: Kim, Y., Kim, J., Vigna, G., Shi, E. (eds.) CCS (2021). https://doi.org/10.1145/3460120.3484534
15. Zhang, Z., Barthe, G., Chuengsatiansup, C., Schwabe, P., Yarom, Y.: Ultimate SLH: Taking Speculative Load Hardening to the Next Level. In: USENIX Security (2023)