# A New Type of Branch Instruction with Configurable Speculation Behavior

**Kekai Hu, Ke Sun, Rodrigo Branco**

***Intel - STrategic Offensive Research & Mitigations (STORM)***

***v1.42 (For the latest version, please check: https://github.com/intelstormteam/Papers)***

## 0. OBJECTIVES

During our work in battling various speculative side-channel vulnerabilities (e.g. proof-of-concept exploits writing, new attack/threat identification, mitigation design/evaluation, etc.), many ideas for mitigations have been discussed (some end up in real implementations, some are disclosed as patents or published in other forms). This paper shares one of the ideas that our team considered, including the theory and possible implementation options. The idea itself has not been fully implemented or tested, it is shared with the objective to give a ground base for other researchers to improve upon and also for the industry to consider.

## 1. INTRODUCTION

Modern CPU cores rely on speculative execution to avoid pipeline stalls and achieve better performance. However, non-committed speculative execution may leave measurable traces through cache-based covert channels and eventually lead to attacks such as the work in (Ref[1-6]). This proposed idea intends to improve the existing solution for speculative side-channel attacks, by introducing a new type of branch instruction that has configurable speculation behavior. The proposed design involves minor changes to the existing instruction set architecture (ISA) and can be implemented with compiler support. It provides more flexibility to the software and is also expected to have less performance overhead compare to the existing mitigation solutions.

## 2. EXISTING SOLUTIONS

Intel provides different solutions for their legacy micro-architectures against different variants of the speculative side-channel attacks (Ref[7, 8]).

For Variant 1 (Bounds Check Bypass) – Spectre V1, Intel recommends software modifications that inserts a barrier (fence) to stop speculation in appropriate locations.

For Variant 2 (Branch Target Injection) – Spectre V2, Intel provides three hardware-based mitigations: Indirect Branch Restricted Speculation (IBRS), Single Thread Indirect Branch Predictors (STIBP) and Indirect Branch Predictor Barrier (IBPB). Software mitigations for Variant 2 include "retpoline" (recommended by Intel), which converts vulnerable indirect branches into "*push + ret*" instructions to prevent branch target injection (Ref[9]). Another software option (introduced by our team, but not necessarily recommended) is called "randpoline" and it is a non-deterministic defense (Ref[10]).

For Variant 3 (Rogue Data Cache Load) - Meltdown, Intel proposes a software mitigation, KPTI (Kernel Page-Table Isolation) (Ref[11]), to ensure that privileged pages are not mapped when executing user-mode code in order to protect against user mode speculative side-channel attacks.

## 3. OUR IDEA

### 3.1 Overview

In this paper, we propose to enhance the current branch instructions by adding extra opcode bits to represent different speculation behavior. Based on a configurable compiler option, a user can have the flexibility to control whether certain branch instructions can proceed with speculative execution or not. In this way, a better balance between performance and security could be achieved.

### 3.2 Details of our idea

Current branch instructions do not have speculation control information, thus, protecting these instructions from speculative side-channel attacks can induce extra performance overhead. In some cases, the existing mitigations against such attacks involve forcing branch miss-prediction, inhibiting the speculative execution of all indirect branches, or flushing the whole branch prediction buffers. While insignificant in many cases, the performance impacts of existing mitigations can vary by case and by workload, making it desirable for them to be further optimized.

As proposed by this paper, one improvement (as supplement or alternative) to the existing solutions is to add speculation control information in the branch instructions so that the processor can identify the branch type in terms of speculative execution and act accordingly in the branch prediction process. There are multiple ways to implement the idea, such as redefining a completely new set of branch instructions or just adding speculation control flags to existing branch instructions. In this paper, we elaborate on the latter as an example implementation since it introduces less changes to the existing ISA and requires less design complexity.

For better illustration of this idea, a simple code snippet of the speculative side-channel attack Spectre V1 is used as an example:

```
0: mov r8, [some_data]
1: lea rdi, [base_addr]
2: mov rax, [p_offset]
3: add rax, rdi
4: mov rcx, [p_boundary]
5: mov [some_addr], r8
6: cmp rax, rcx
7: jge abort
8: mov rbx, [rax]
```

9: mov rdx, [rsi + rbx*0x100]

In a serialized execution flow without speculative and out of order execution:

- Instruction 0 loads some data from memory to R8
- Instruction 1 loads the base address to register RDI
- Instruction 2 moves the offset value to register RAX
- Instruction 3 adds the base address from RDI with the offset in RAX and stores the result to RAX
- Instruction 4 loads the boundary value from memory to RCX
- Instruction 5 stores some data from R8 to memory
- Instruction 6 compares RAX with RCX, if RAX is greater or equal to RCX
- Instruction 7 jumps to an abort routine, thus instruction 5 will not be executed, otherwise
- Instruction 8 loads the value stored in the location of RAX to RBX
- Instruction 9 loads from memory with the cache-line accessed depending on the value in RBX

This contains a simplified boundary checking code that checks whether the offset (originally loaded in RAX) plus base address (loaded in RDI) is within the expected upper boundary (loaded in RCX). Only if the check passes, the value stored at the target address (in RAX after instruction 3) will be read to RBX, otherwise, the flow will jump to abort routine and the value will not be loaded.

This code works fine if all the instructions are executed in a serialized order. However, speculative branch predictions and out-of-order execution in modern CPUs make Spectre V1 attack possible which could bypass the boundary check in a speculative path.

In such scenario, the definitive result of the comparison at line 6 can be slow if the boundary value is not cached and needs to be fetched from memory. Rather than waiting for the result, the processor may predict that the check will pass (based on previous branch history) and

speculatively read the value at target location (in RAX after instruction 3) to RBX. The processor can then continue with the cache-line loading that depends on the RBX value, even if at this point RAX has an address outside the boundary. By the time the check resolves (and fails), the speculative execution will be dropped; however, the cache-line loaded during the speculative path will not be reverted and thus it is measurable by an attacker through a timing side-channel.

The current solution for this type of speculative side-channel attack is serialization, i.e., adding fence instructions (e.g., *lfence*) between instruction 5 and instruction 6 so that no instruction from 6 onward begins execution until all prior instructions up to 5 have completed. Per Intel Software Developer's Manual (Ref[12]), "LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes".

Although the serialization mitigation prevents the speculative side-channel attack like Spectre V1, it stalls the pipeline pending the completion of ALL prior instructions, including the ones that are not related to the branch. If the code can tell the CPU to just avoid speculative execution at branch 6, the side-channel vulnerability will be prevented as the pipeline will stall and wait ONLY for the instructions the branch has dependencies on (which can be resolved by the out-of-order execution engine as is currently implemented), instead of ALL the instructions prior to the branch (e.g., it does not need to wait for instruction 0 and instruction 5 if they take long time to complete). This is expected to improve the performance in certain scenarios in comparison to the existing mitigations.

While the discussion above uses the example of Spectre V1 regarding conditional branches, the proposed idea is also applicable to Spectre V2 with indirect branches. An indirect branch configured as non-speculative will wait for the resolving of its target instead of proceeding with speculative branch target from the branch prediction unit (BPU). Considering that, in the general case, a vulnerable indirect branch (in terms of Spectre V2) needs to be in a reachable path with certain attacker controlled contexts, this idea gives the software more flexibility to distinguish and mark only the vulnerable indirect branches as non-speculative.

As few as one bit of speculative information can be added to existing branch instructions to represent the expected speculative behavior. If it is set to 0, the branch instruction works as normal, if it is set to 1, the CPU will not carry out branch prediction on this branch. This bit can be added in a compatible manner to the existing branch instructions, e.g., as an extended opcode bit or as a bit in the prefix. One thing to notice is that marking a branch as "non-speculative" will only force the dependencies of this branch to be resolved, thus protecting the branch itself from speculative side-channel attacks. However, marking as "non-speculative" does not guarantee the control flow that contains the marked branch is not on a speculative path. For example, in a nested-branch case like the code pattern below:

```
0:    If (slow_conditional_flag){
1:          (*func_ptr)();
2:    }
```

An indirect branch (generated by calling the function pointer at line 1) is put inside a conditional branch (generated by the *if* conditional at line 0). If only the indirect branch from line 1 is marked as non-speculative, although the indirect branch itself is protected from branch target injection attack (Spectre V2), it may still be executed on a speculative path miss-predicted by the conditional branch in line 0 (essentially a Spectre V1 case). The defense of such case relies on marking the conditional branch from line 0 non-speculative if it is deemed vulnerable.

If desired, more bits can be added to give more fine-grained control over speculative behaviors rather than just simply on and off. For example, a conditional branch can be configured as always taken/not-taken for security purposes, the performance penalty of which could be better or worse than a non-speculative branch, depending on the actual case. The speculative control can also be configured as mode-specific, which can be either a native processor execution mode (e.g., user-mode, kernel-mode, system management mode, etc.) or a software defined mode. For example, branches in user code can be marked as non-speculative only when executed by kernel-

mode (if not forbidden by SMEP). Software can also define a "trusted mode" versus "untrusted mode" in terms of speculative execution (e.g., based on the mitigations in place, code path, scheduling on the sibling hyper-thread, etc.) and allow branch speculation only in "trusted mode".

In theory it is also possible to limit the speculation of a branch to certain scope (like within the same module, within certain bounds) by memory tagging or target bounds that associate with the branch (similar to the feature in Intel Memory Protection Extensions in a speculative sense). However the nature of speculative attack makes the speculative control flow protection inherently challenging. Different from a control flow hijacking attack like return-oriented programming (ROP), the ultimate goal of speculative side-channel attack is information leak which does not really need to execute code that is restricted from its normal control flow (e.g. ROP gadgets): instead it only needs a cache-loading gadget in speculative execution that typically does a load from memory and a 2$^{nd}$ load with access location depending on the 1$^{st}$ load. Refine granularity control in speculative execution may not be very efficient in preventing such gadgets' execution.

Compiler level options can be added for the programmers to decide the expected speculative behavior of branches, which can be either explicit to a specific branch or implicit for branches with different granularity or certain code pattern.

### 3.3 Comparing with Memory Fences

In some aspects, this proposed idea might be considered similar to memory fences, such as *lfence* or *mfence* in terms of avoiding speculative execution, however, there are two major differences:

1. As mentioned above, memory fence instructions carry out general serialization for all instructions in scope before the fence itself, regardless of the actual dependency with branch instruction, while our proposed idea only serialize pending the resolving of the dependencies of the branch to be protected. Therefore, the overall performance overhead is expected to be less.

2. While memory fences are unconditional instructions that will be executed every time, the proposed branch instructions with configurable speculative information can be encoded to behave differently in terms of speculative execution with respect to the execution context, such as processor mode, etc., which do not necessarily enforce serialization every time thus is also expected to induce less performance cost.

## 4. CONCLUSION

In this paper, we propose a new type of branch instruction that has configurable behavior in speculative execution. Adding speculative information to the branch instructions gives software finer-grained control on the expected behavior of branches based on different use cases and is expected to reduce the performance penalty compared to the current related mitigations.

## Acknowledgement

We would like to thank Linus Torvalds and Daniel Gruss for providing comments on the initial draft of this paper.

## Legal Notices & Disclaimers

This work is disclosed as a security research work for the community to consider, leverage and review. It has not been fully implemented or comprehensively tested for all potential corner cases and therefore is not presented nor guaranteed as a mature solution.

Intel provides these materials as-is, with no express or implied warranties. No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

All products, dates, and figures specified are preliminary, based on current expectations, and are subject to change without notice.

Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration**. No product or component can be absolutely secure**. Check with your system manufacturer or retailer or learn more at http://intel.com.

Intel and the Intel logo are trademarks of Intel Corporation in the United States and other countries.

*Other names and brands may be claimed as the property of others.

## Reference


1. Yuval Yarom and Katrina Falkner, "FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack", 23rd USENIX Security Symposium, 2014

2. Jann Horn, “Reading privileged memory with a side-channel”, Jan 2018

3. Paul Kocher, Jann Horn *et al.*, "Spectre attacks: Exploiting speculative execution", 2018

4. Moritz Lipp, Michael Schwarz *et al.*, “Meltdown: Reading Kernel Memory from User Space”, 2018

5. Jo Van Bulck Jo, Marina Minkin et al. "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution", 27th USENIX Security Symposium, 2018

6. Stephan van Schaik, Alyssa Milburn *et al*., “RIDL: Rogue In-Flight Data Load”, IEEE S&P, May 2019

7. Intel Corporation, “Intel Analysis of Speculative Execution Side-Channels”, Jan 2018

8. Intel Corporation, “Deep Dive: CPUID Enumeration and Architectural MSRs”, May 2018

9. Paul Turner, “Retpoline: a software construct for preventing branch-target-injection”, Google, 2018

10. Ke Sun, Kekai Hu *et al.*, “Randpoline: A Software Mitigation Approach for Branch Target Injection Attack”, Aug 2019

11. Daniel Gruss, Moritz Lipp *et al*., "KASLR is dead: long live KASLR", Jun 2017

12. Intel Corporation, “Intel 64 and IA-32 architectures software developer’s manual combined volumes”, May 2019