A New Memory Type against Speculative Side Channel Attacks

Ke Sun, Kekai Hu, Rodrigo Branco

Intel - STrategic Offensive Research & Mitigations (STORM)

v1.51 (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. developing proof-of-concept exploits, new attack/threat identification, mitigation design/evaluation, etc.), we've discussed many ideas as potential mitigations (some end up in real implementations, some are disclosed as patents or published in other forms). This whitepaper shares one of the ideas that our team considered, including the theory and possible implementation options, with the objective to give a ground base for other researchers to improve upon and also for the industry to consider.

After the initial release of this paper, we have received information about recent works on similar topic (Ref[13, 14]). Although our work was carried out independently, we have added the citation and highlighted some of the differences between our work and those cited in the new Section 2.6.

1. INTRODUCTION

Modern CPUs rely on speculative execution to avoid pipeline stalls and achieve better performance. However, speculative execution may be leveraged by a malicious attacker, leading to information disclosure through cache-based side-channel attacks such as Spectre (Ref[1][2]), Meltdown (Ref[3]), L1 Terminal Fault (L1TF) (Ref[4]), Speculative Store Bypass (SSB) (Ref[5]), etc. Existing mitigations include hardware and software-based solutions. As hardware-based mitigations, Intel proposed the indirect branch restricted speculation (IBRS), single thread indirect branch predictors (STIBP) and indirect branch predictor barrier (IBPB) as mitigations against the Spectre Variant 2 (Branch Target Injection), the Speculative Store Bypass Disable

(SSBD) against SSB (Ref[6][7]). The software-based solutions include memory fences against Spectre Variant 1 (Bounds Check Bypass), retpoline (Ref[8]) and randpoline (Ref[9]) against Spectre Variant 2 and kernel page-table isolation (KPTI) – heavily based in the Kaiser reference implementation (Ref[10]) - against Meltdown.

This paper proposes a consolidated hardware approach as an alternative mitigation to the existing solutions of speculative execution side-channel attacks, by introducing a new memory type that can be used to protect memory content against speculative access.

This new memory type is expected to mitigate multiple known speculative execution vulnerabilities (including Spectre Variant 1, Variant 2, Meltdown, L1TF, SSB, etc.) with one consolidated solution by removing one of the common primitives of different speculative execution attacks. It could also be effective against unknown vulnerabilities in the same category. Our proposal provides more flexibility to software and is also expected to improve performance by containing the mitigation overhead to specific memory accesses.

2. OUR IDEA

2.1 Overview

We propose a new memory type, speculative-access protected memory (SAPM), as a consolidated solution for multiple speculative execution side-channel attacks. SAPM can be applied to specific memory ranges, with the attribute that any memory access to such memory type will be instruction-level serialized, meaning that any speculative execution beyond the SAPM-accessing instruction will be stopped pending the successful retirement of this SAPM-accessing instruction.

Such memory type gives software the flexibility and choice to put only the secret and sensitive data in dedicated memory regions to protect them from speculative side-channel attacks.

2.2 The Idea

Based on most known cases, a speculative execution side-channel attack can be in general viewed as containing two parts, which, in this paper, are mentioned as “front-end” and “back-end” for simplicity. The “front-end” refers to the part of an attack from the beginning to the point of speculatively reading the secret, which corresponds to the “speculation primitive”, “windowing gadget” (existent in certain attacks only) and part of the “disclosure gadget” according to MSRC’s definition (Ref[11]). The “back-end” refers to the part that transforms the speculatively loaded secret into an architecturally measurable trace (e.g., cache-line loading) and eventually gets resolved by the attacker, which corresponds to the “disclosure gadget” and the “disclosure primitive” steps in MSRC’s definition (Ref[11]).

The front-end of an attack varies case-by-case and generally includes preparation of a certain condition that leads to a speculative execution that bypasses architectural security checks. For example, a speculative load of the secret as a result of conditional branch prediction (Spectre Variant 1), indirect branch prediction (Spectre Variant 2), general protection fault (Meltdown), page fault (L1TF), or latency in older store (SSB). Such a secret-leaking load cannot happen architecturally but can take place speculatively and is not limited by the security constraints.

While the front-end varies across different attacks, the back-end is similar for most cases in that it involves a cache-based covert-channel: it transforms the speculatively loaded secret into a secret-dependent cache-line loading that is measurable using timing side channel analysis. The back-end needs to be executed speculatively AFTER the speculative load of the secret.

The purpose of the proposed new memory type, SAPM, is to prevent the common back-end (the part of the “disclosure gadget” after the speculative load of the secret, to be specific) of different attacks from being speculatively executed: any access to SAPM region will cause instruction-level serialization and speculative execution beyond the SAPM-accessing instruction will be stopped until the successful retirement of that instruction.

2.3 Value of the Idea

Compared to existing solutions, the main value of this idea can be divided in two parts:

1. While most of the existing solutions are case-by-case, this proposal is a general solution for many speculative execution side-channel attacks. It reduces the mitigation complexity and saves patching resource of applying multiple mitigations for different scenarios. It may also avoid the potential corner cases for an attack-specific mitigation and could also be effective for unknown future attacks of the same category.

2. Potential performance benefit. Unlike the existing solutions that try to prevent the unsafe speculative execution (by flushing branch prediction arrays at multiple conditions like context switches or mode changes, forcing each indirect branch to mis-predict, or disabling certain speculative behavior in general), the proposed solution only addresses the speculative execution that actually accesses the secret the software cares about. The instruction-level serialization only happens when the user-defined secret is accessed in the SAPM region. Although the performance cost for each memory access to SAPM is relatively big, considering such operations shall only be a very small portion of the total software execution, the overall performance overhead is expected to be low and potentially less than the performance impact of current mitigations.

2.4 Possible Implementation of the Idea

There can be different ways to implement SAPM and any real implementations are architecture and micro-architecture specific. Here we give an example of possible implementation that is compatible to the pipelined modern processor with out-of-order (OOO) execution.

In modern processors with pipeline and OOO execution, instructions are fetched in order, executed out-of-order with speculative execution, and eventually retired in order to take effect architecturally.

To implement SAPM, a clear-and-re-fetch approach can be used:

1) Add a logic after instruction decoding and before memory access to check whether the target address of a memory accessing instruction (load, store, etc.) points to an SAPM region (At this point, although the instructions after this memory-accessing instruction could already be in the pipeline, no secret can be accessed/leaked before the resolving and checking of the secret's address).
2) If the data address of the memory-accessing instruction is inside an SAPM region, then immediately clear/flush the pipeline.
3) Re-fetch instructions only up to the SAPM-accessing instruction, with further fetch depending on the successful retirement of the SAPM-accessing instruction. Thus the data-dependent speculative execution beyond SAPM-accessing instruction cannot happen until the successful retirement of this instruction. In this way, speculative secret-leaking can be effectively prevented regardless of how exactly the speculative execution happens.

While the example above is just a simplified model, there can be large room of optimization for the implementation of such serialization, depending on the pipeline stages and the microarchitecture.

Given the general idea of SAPM, there can also be many variations in the actual design. For example, SAPM can be implemented either by virtual address or by physical address. If SAPM is associated with virtual address, it can be implemented through the paging data structures, using one of the reserved bits in page table entries (and also cached in TLB as one of the TLB flags). If SAPM is associated with physical address, it can be implemented by a group of SAPM address range registers. In either case, the address check should be carried out before the decoded instructions dispatched for memory execution.

One thing to notice is that if implementing SAPM through paging data structures, it is assumed the operating system (OS) is trusted to manage SAPM, thus it is not applicable to the scenarios that OS itself is the attacker. The mitigation scope of SAPM covers the speculative execution side-channel attacks that speculatively read data from cache/memory, such as Spectre Variant 1, Spectre Variant 2, Meltdown, SSB, L1TF (variants covered depending on SAPM implementation), etc. For such attacks, as long as the secret data is placed in an SAPM region, any speculative access to the secret will trigger instruction-level serialization and the speculative secret-leaking that bypasses security constraints will be mitigated, regardless of the specific attack scenario. It is not in the scope of SAPM if the data is leaked through other processor internal buffers (e.g. MDS, RIDL) (Ref[12]).

Depending on the need and the purpose, SAPM can also be implemented per thread, per processor, or to be associated with a specific processor mode. For example, in the case of virtualization, it could be configured to differentiate between VMX root and VMX non-root mode, or it could be configured to be associated with specific CPU thread(s) that are assigned to a guest virtual machine (VM); in that way SAPM can protect the rest of the host memory (including memory of other VMs) from speculative access attacks. Per-thread, per-core or mode-specific implementation options could have performance implications in practice for such cases. The memory access types of SAPM can also be further divided into a finer level and made configurable if desired (e.g., load only or both load and store, etc.).

2.5 Comparing with Memory Fences

In some aspects, SAPM might be considered similar to memory fences, such as *lfence* or *mfence*, which serializes memory-accessing instructions with respect to the fence instruction. However, there are some fundamental differences between the two:

1. Memory fences are instruction-based while SAPM is data-based (address-based). When memory fences are used as mitigation, they need to be inserted in all possible locations of vulnerable code pattern and will serialize every time they are executed regardless of which part of memory is accessed, while SAPM only serializes when SAPM region is accessed.

2. Memory fences are used to mitigate the front-end (e.g., using *lfence* to serialize before vulnerable conditional branches for Spectre Variant 1), while SAPM is aiming to mitigate the back-end (it is not practical to use memory fences to serialize back-end which requires adding fence instructions to all data-dependent cache-loading code patterns).

3. Memory fences are only effective for certain cases (e.g., Spectre Variant 1), while SAPM is expected to be generally effective for most speculative execution side-channel attacks that leak secret from cache/memory.

2.6 Comparing with Other Ideas

After the initial release of this paper, some researchers reached out to inform us of potentially similar works, which are discussed in this section.

In the work of M. Schwarz *et al* (Ref[13]), the researcher discussed a similar idea of introducing a new memory type against speculative side-channel attack. M. Schwarz *et al* also extended the idea to registers and used taint analysis for secret tracking and offered a detailed discussion and evaluation of their implementation, while our work focused on the possible hardware implementation of SAPM itself in a modern processor.

While being similar in terms of the goal and the approach that uses a dedicated memory type to protect data from speculative leakage, there are important differences between the aforementioned work and the idea proposed in this paper. In the work of M. Schwarz *et al* (Ref[13]), the authors define the instructions that have been executed (in speculative or OOO execution) but not architecturally committed as “transient instructions” and the corresponding execution as “transient execution”. The core of their idea is that the secret data in the new memory type is “not available when executing in the transient-execution domain”: “instead of the actual value, the CPU uses a dummy value (e.g., ‘0’) when accessing a non-transient memory location during transient execution”, indicating the CPU acts differently upon transient and non-transient execution when the CPU accesses the protected memory.

Given that, in a general view, any instruction can be viewed as speculatively executed until it successfully retires (since it can be revoked due to branch mis-prediction, fault triggered by itself or other instructions, memory access reordering, etc.), there is no easy way of telling, from the CPU's view, if an instruction is speculatively executed upon the point of execution without the potential for performance impact. The SAPM as we proposed does not prevent speculative access to the secret per se, but only forces post-serialization after the access to SAPM in order to allow architectural constrains to take effect, which induces no change to the OOO and execution engines and avoids the challenge of needing to determine if an instruction is speculatively executed upon execution time. The SAPM access also does not have to be as slow as uncacheable (UC) memory accesses: it only holds further execution upon its successful retirement, which by itself can be as fast as any regular memory access.

In the recent work by J. Fustos *et al.* (Ref[14]), a similar type of memory is proposed and evaluated using Gem5 simulation. While the core idea is similar, their work focuses on mitigating Spectre-type attacks, specifically Spectre Variant 1.

Another potentially similar feature exists in the PowerPC platform. According to the processor reference guide, in PowerPC architecture (Ref[15]), "speculative accesses can be prevented by assigning the guarded storage attribute (G) to memory locations. An access to a guarded memory location is not performed until that access is required by the sequential-execution model and is no longer speculative", which also seems related to this paper and the ideas mentioned above. Yet there are two major differences that likely make this feature not as effective of a mitigation as the one proposed in this paper. According to the processor reference guide, "marking a memory location as guarded does not completely prevent speculative accesses from that memory location", "If the memory location is already cached, the location can be speculatively accessed". This means it will not be an effective mitigation to most of the known speculative side-channel attacks. According to the processor reference guide, speculative access is defined as "any access not required by the sequential-execution", with a typical example as a mis-predicted branch. It is not clear and likely not including the transient execution in the latency of handling an exception (e.g., Meltdown, L1TF) in sequential execution.

3. CONCLUSION

In this paper, a new memory type, speculative-access protected memory (SAPM), is proposed to remove a common primitive from speculative execution side-channel attacks that leak secret from cache/memory. It is expected to mitigate most of the known attacks so far in this category as a consolidated solution that offers more flexibility and potential performance improvement over the existing mitigations.

Acknowledgement

Researchers Brad Spengler (grsecurity) and pipacs (PaX project) reminded us of the PowerPC guarded storage (Ref[15]). Researcher Daniel Gruss informed us of the work in (Ref[13]) and also helped reviewing the new version of this paper. Hypervisor Architect, Liran Aron, recommended us to elaborate more on the importance of per-thread/per-core implementation options for performance consideration in scenarios involving virtualization.

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.

References

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

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

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

4. Intel, “Q3 2018 Speculative Execution Side Channel Update”, Intel Security Advisory 00161, Aug 2018

5. Matt Miller, “Analysis and mitigation of speculative store bypass”, MSRC, 2018

6. Intel, “Intel Analysis of Speculative Execution Side Channels”, Jan 2018

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

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

9. Intel Storm team, “A Software Mitigation for Branch Target Injection Attacks”, Aug 2019

10. Daniel Gruss, Moritz Lipp *et al.*, “KASLR is Dead: Long Live KASLR”, Jun 2017

11. Matt Miller, “Mitigating speculative execution side channel hardware vulnerabilities”, MSRC, Mar 2018

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

13. Michael Schwarz *et al.,* “ConTExT: Leakage-Free Transient Execution”, May 2019

14. Jacob Michael Fustos, *et al.*, “SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks”, Design Automation Conference (DAC), June 2019

15. IBM, “PowerPC Processor Reference Guide”, Jan 2010