Spoiler Alert!

Julio Soto  
Department of Computer Science & EngineeringUniversity of Colorado DenverDenver, USA  
julio.m.soto@ucdenver.edu

Kushal Kusram  
Department of Computer Science & Engineering  
University of Colorado DenverDenver, USA  
kushal.kusram@ucdenver.edu

Mien Nguyen  
Department of Computer Science & Engineering  
University of Colorado DenverDenver, USA  
mien.nguyen@ucdenver.edu

Jeremy Henry   
Department of Computer Science & Engineering  
University of Colorado DenverDenver, USA  
jeremy.henry@ucdenver.edu

*Abstract* — Intel CPUs make up 80 percent of the global CPU market, thus, any type of security vulnerability existing in these chips could potentially expose tens of millions of customers’ private and sensitive information. Spoiler: a previously unknown microarchitectural leakage in the Intel processor that has the potential to expose critical information about physical page mappings, using a limited set of instructions, was recently discovered and is operating system independent and is present in all Intel Core CPU family.

Keywords—Speculative Execution, Speculative Loads, Store Forwarding, Memory Attacks, Cache Attacks, Rowhammer, Intel Dependency Logic, Intel Memory Order Buffer

# Introduction

Almost all modern microarchitectures have optimization techniques such as speculative loads and store forwarding with their main purpose being to improvise memory bottleneck to increase performance. This research project focuses on recreating the simulation, the researchers from our primary research paper conducted, to compute and visually represent the CPU clock cycles. Increase in clock cycles exposes the 4k Aliasing conflict resolution time. Showing this increase in time directly confirms the proportionality to the dependency resolution logic managing and handling addresses which creates the opportune moment for the Spoiler attack to occur.

Spoiler is a previously unknown vulnerability in all Intel Core processing units dating back to the very first Intel Solo Chip and into today’s Core i9 running Coffee Lake architecture released October 2018. The flaw was in their proprietary dependency resolution logic which exploits the Memory Order Buffer (MOB) during speculative execution. Researchers have dubbed this vulnerability “Spoiler,” which allows a hacker to, potentially, obtain virtual and physical memory addresses and access sensitive data present at memory locations even through sandboxed environments such as JavaScript with a higher probability of successfully implementing memory attack techniques such as Rowhammer to steal information.

The motivation behind this research project is the scope of vulnerability, as well as, how recent the vulnerability has been discovered. Spoiler not only has the potential to be executed from user-space but also from sandboxed environments such as JavaScript from the browser.

## Background

Understanding speculative execution and how it affects CPU performance is essential to this research project. Speculative loads and store forwarding are functions of speculative execution that most processor manufacturers have implemented in their chips to enhance performance, but how?

Memory disambiguation, store forwarding, and speculative loads.

## Memory Order Buffer

The Memory Order Buffer (MOB) is proprietary to Intel and is responsible for managing memory operations [1]. The MOB is tightly coupled with the cache and ensures memory operations are executed efficiently and according to Intel’s own memory ordering rule in which stores are executed in-order and loads can be executed out-of-order [1]. Components of the MOB include a store buffer, load buffer and circular buffers [1]. *Figure 1* displays the architecture of the MOB.

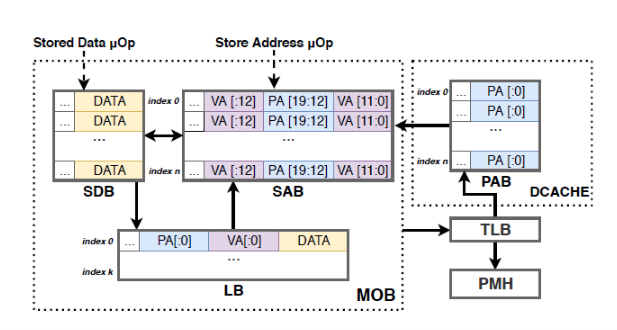


Fig.1 Intel Memory Order Buffer

## Specutlative Load

A load operation can be executed speculatively before preceding store operations. During the speculative execution of the load, false dependencies may occur due to the unavailability of physical address information. These false dependencies need to be resolved to avoid computation on invalid data [1].

It is these false dependencies and their increase in clock cycles that the processor needs to resolve the conflicts that we have focused on for our project.

Since Intel uses their own proprietary memory disambiguation and dependency resolution logic in their processors to predict and resolve false dependencies that are related to speculative loads [1], we will not be comparing other processor brands. Researchers found Intel’s dependency logic suffers from and unknown false dependency independent of the 4k aliasing and found it to be occurring during the 1 MB aliasing of speculative memory accesses which is then exploitable to secure information about physical page mappings [1].

## Store Forwarding

Memory stores are executed in-order and memory loads can be executed out-of-order [1]. The order in which the processor executes either of these instructions in controlled by the MOB, which includes circular buffers, a store buffer and a load buffer [1].

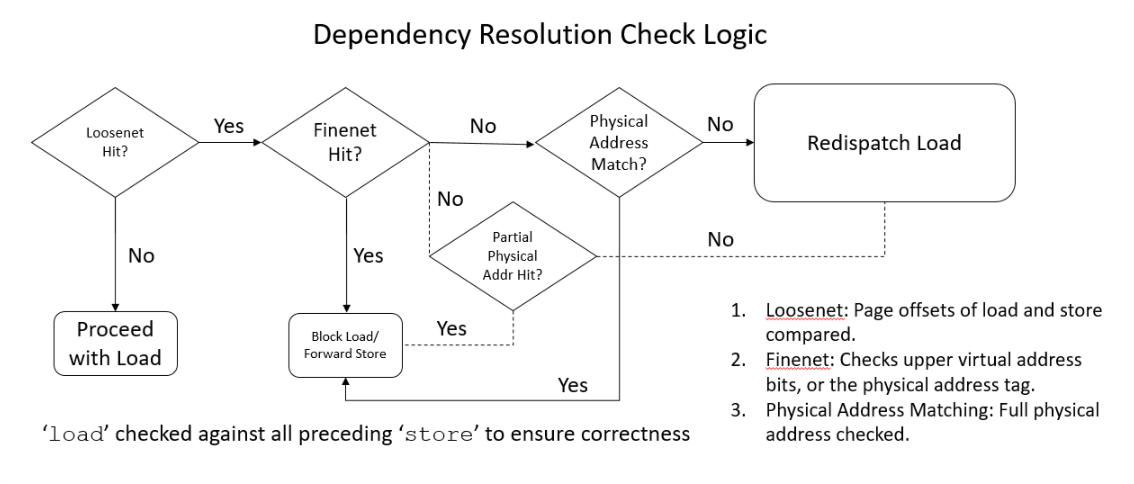
A store will be decoded into two micro ops to store the address and data respectively [1]. This in turn, enables the processor to continue execution of other operations without having to wait for the store to commit to memory.

Store forwarding is an optimization mechanism that sends the store data to a load if the load address matches any of the store buffer entries [1]. However, while trying to find any information about the store buffer in Intel’s architectures, the researches were unable to find any documentation.

If there were to be flaw in Intel’s resolution software, the resulting effect could be that the processor may falsely forward incorrect data although the physical and the virtual addresses do not match. The complete resolution will be delayed until the load commitment, since the MOB needs to ask the Translation Look-aside Buffer (TLB) for the complete physical address information, which is time consuming. Additionally, the data cache may hold the translated store addresses in a Physical Address Buffer (PAB) with equal number entries as the store buffer [1].

## Dependency Resolution Logic

Fig. 2 Intel’s proprietary Dependency Resolution Logic



Intel uses their own dependency check logic to resolve the conflicts between store and load instructions. The designed 3 stages to handle a conflict are:

* Loosenet: Checks the page offset bits and if they match it forwards to the next stage to check, else proceeds with a load instruction.
* Finenet: Checks the upper address bits (i.e. virtual address bits or also called the physical address tag). Upon a match load instruction is blocked or store instructions are forwarded. Otherwise, if there is no match then proceeds to the next stage.
* Physical Address Match: In this stage the entire physical address field is checked and if there is a match then the load instruction is re-dispatched.

# Spoiler

The researchers designed an experiment to observe timing measurements of speculative loads. They propose *Algorithm 1*which jams the buffer with multiple stores and then execute a speculative load [1]. For each window w, the algorithm iterates over multiple pages and for each page, performs a store to that page and for each previous page within a window [1]. After each store, the speculative load operation is timed from a different memory page. Using 1000 different virtual addresses when executing Algorithm 1, various step-wise peaks were observed with a high latency [1]. To evaluate whether the step-wise peaks have any correlation with actual physical page numbers, the authors’ utilized the Linux *pagemap* file. *Pagemap* is a set of kernel interfaces which allows user space processes to examine page tables. According to Figure 4 in the research page, the average time of execution is 200 cycles for load operations, and the step-wise peaks observed occur in every 256 pages, noting that the least 20 bits of physical address for the load address matches the addresses for performing stores to virtual pages [1]. Even conducting the experiment multiple times, one would always observe a matching of the least 20 bits of physical address, which shows the existence of 1MB aliasing, and it is this aliasing that exposes a physical address mapping to the user space [1].

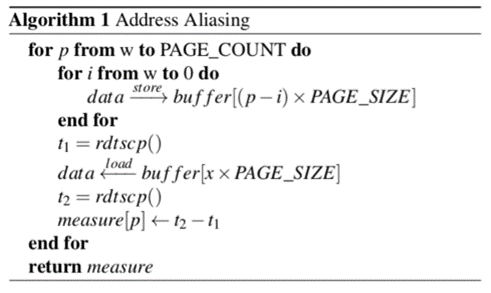
Though JavaScript is difficult to perform attacks and exploits due to its sandboxed nature, there is still a workaround using WebAssembly [1]. WebAssembly is a virtual machine used to support multi-language applications on modern web browsers to facilitate faster execution of page scripts and is designed to run alongside JavaScript for increased performance and speed. Through the use of WebAssembly, the researchers were able to penetrate the abstraction layers, and utilizing *Algorithm 1* and Figure 4 in the research paper, were able to expose the aliased addresses and these addresses were used for creation of eviction sets, which are then used for cache attacks such as Prime+Probe [1]. Essentially, eviction sets are collection of virtual addresses that are each mapped to the same cache set. The danger of constructing and obtaining these eviction sets is that it allows an attacker to perform several clandestine operations. The contents of a cache set can be completely replaced by the attacker, which then allows the attacker to bring the cache set into a controlled state. From here, an attacker would be able to probe whether the cache has been modified by the user. The authors utilized an existing algorithm with a couple of tweaks for eviction set creation. Testing against the Intel Core i7-4770 with four physical cores and a shared 8MB 16-way L3 cache, the experiment yielded an 80% accuracy rate in finding all 8192 eviction sets in a pool of 4096 addresses, with the eviction set creation process taking an average of 46 seconds [1]. Using the *Spoiler* attack, eviction set creation gained a dramatic speedup. Because only aliased addresses from *Spoiler* were used, on average, one in 256 addresses will be aliased instead of 1048576, providing a speedup of 4096 for eviction set creation, which increases the efficiency of other cache attacks [1].

## Hardware Mitigation

There are currently no software mitigations in place to completely resolve this vulnerability. Spoiler exploits the fact that when a load operation occurs after multiple store instructions, the clashing of physical addresses creates an erratic, high timing behavior. While the timing behavior can be curbed by inserting store fences between load and store operations, the user still has the ability to leak the physical addresses. Most attacks on JavaScript can prevent *Spoiler* by removing accurate timers, such as the SharedArrayBuffer (disabled by most modern browsers due to Spectre and Meltdown attacks) but there are multiple timers with varying degrees of precision and removing all would be unattainable [1].

## Software Mitigation

The proprietary memory disambiguator currently used by Intel to allow speculative behavior could be modified to prevent leakage of physical addresses but doing this would drastically affect performance [1]. Also, patches hardware on legacy machines are difficult to apply and could possibly take years to deploy.



rdtscp;

mov %eax, %esi;

mov (%rbx), %eax;

rdtscp;

mfence;

sub %esi, %eax;

**Fig. 3 Timing Measurement of Speculative Load [1]**

# implementation

The basis for our testing platforms is as follows:

## Suitable Test Platform(s)

Researchers from Worcester Polytechnic Institute, Massachusetts, and the University of Lübeck in north Germany have discovered this flaw in Intel processors. As such, we shall require Intel processors from the Core family and a suitable test bed to show this flaw. In this experiment we used an Intel i7-7700 clocked at 3.60 GHz with 4 physical and 8 logical cores coupled with 16GB of RAM. Hyper-V was enabled on this system. This was the base configuration, upon which a layer of virtualization was placed thereby providing us with a sandbox environment to run our experiment. The configuration of this virtual environment was an Intel i7-7700 clocked at 3.60GHz but with 2 physical and 4 logical cores coupled with 8GB of RAM. The virtual system had Ubuntu 18.04LTS 64-bit running. VMware was the tool used to virtualize *virtual hardware performance counter* support had to be enabled in order to access rdtscp().

## Measurment/Detection Tool

The experiment utilizes rdtscp() to record the cycle count to execute a speculative load after performing multiple speculative stores in a window of size 64 pages. To gather accurate results, the project makes use of inline assembly functions in C. The function rdtscp() has been implemented directly in C. The program results are redirected into a CSV file to perform analysis.

# Configuration & Expectations

## Input

* Window of size 64. This is to ensure that the load has aliasing with the maximum number of entries in the store buffer and therefore maximum potential conflicts.
* PAGE\_SIZE of 4096. We obtained this measurement by using the command ‘getconf PAGESIZE’ in the terminal
* PAGE\_COUNT of 1000.
* Buffer array to store addresses. Initialized with size of PAGE\_COUNT \* PAGE\_SIZE
* Variable DATA which used to fill the store buffer during execution of the algorithm
* Measurement array to store the cycle counts of the load operations
* Constant integer variable ‘x’ which determines the different memory pages
* Timing measurement implemented using Inline Assembly

## Output

Returns the timing measurements (cycle counts) of the load operations from different memory pages.

## Results and Analysis

A CSV file is generated with timing data of PAGE\_COUNT number of pages, this valuable information further needs to be opened using any spreadsheet software to plot a graph and analyze. Ideally, one must look for peaks at equal intervals in the graph which shall be interpreted as speculative store/load conflicts. For better accuracy it is advised to perform multiple runs, ideally around 5 to 7 and plot a graph on the average of all these runs.

# Results

The following graph plots show both intermediate results and final results.

Fig. 4: Initial results obtained on running the program while machine is in a random state.

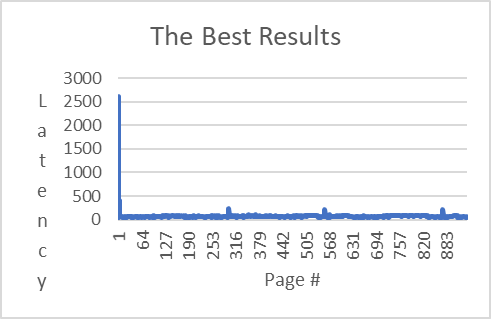
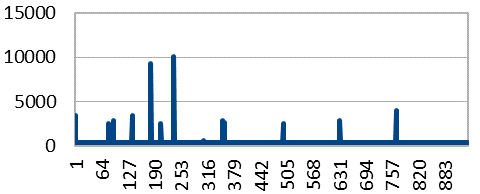


Fig. 5: Results obtained upon a fresh start of the machine.

If we observe Fig. 4 and Fig. 5, the peaks in Fig. 4 indicate that if the program is run when the machine is in a random state the results, we obtained contain peaks that tend to fluctuate and lack consistency. However, in Fig. 5, the results have been collected immediately after a cold start. Therefore, peaks have a pattern and consistency. We believe it is because there are no processes running in the background and the cache is clean on a cold start.

# Milestones

The following table was used in order to keep track of our deadlines. Milestones with asterisks are estimates and were changed depending on guidance.

|  |  |  |
| --- | --- | --- |
| **Milestones** | **Estimated Date** | **Status** |
| Project Proposal and Approval | 14MAR19 | G |
| In-depth Research | 02MAR19 | G |
| Simulation Design | 21MAR19 | G |
| Development | 28MAR19 | G |
| Implementation/Testing | 31MAR19 | G |
| Research Presentation | 09APR19 | G |
| Refinement | 11APR19 | G |
| Project Presentation | 30APR19 | G |
| Final Project Report | 02MAY19 | G |

# Contributions

The following is an outline of what each team member contributed for the duration of research and project. Similarly, each team member will be actively conducting in-depth research, and we have conducted several long late-night synchronization meetings to ensure everyone is on the same page.

## Julio Soto

Conducted research, analysis of the core algorithm, helped in collection of data and ensure the data is formatted to ensure execution for our testing phase. Additionally, aid in the testing and data collection.

## Kushal Kusram

Conducted research and implementation of the algorithm, testing, measuring platforms for our test bed platform. Implemented the test platform to develop and run the algorithm.

## Jeremy Henry

Conducted the research and selection of the proper processor unit(s) utilized for the implementation for the project.

## Mien Nguyen

Conducted the research on how to analyze the data collected through our study in order to relate the project to the course material.

##### References

1. Saad, Moghimi, Ahmad, Bruhns, Ida, Krebbel, Moritz, Berk, Eisenbarth, Thomas, and Berk, “SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks,” [astro-ph/0005112] A Determination of the Hubble Constant from Cepheid Distances and a Model of the Local Peculiar Velocity Field, 01-Mar-2019. [Online]. Available: https://arxiv.org/abs/1903.00446. [Accessed: 08-Mar-2019].
2. M. O. AppleInsider, “New 'Spoiler' vulnerability in all Intel Core processors exposed by researchers,” *AppleInsider*, 05-Mar-2019. [Online]. Available: https://appleinsider.com/articles/19/03/05/new-spoiler-vulnerability-in-all-intel-core-processors-exposed-by-researchers. [Accessed: 10-Mar-2019].