Simulating Modern CPU Vulnerabilities on a 5-Stage MIPS Pipeline using Node-RED

Samuel Miles, Corey McDonough, Emmanuel Obichukwu Michael, Valli Sanghami Shankar Kumar, and John J. Lee

Abstract This paper proposes a simulation of the 5-stage pipelined MIPS processor using Node-RED and illustrates the basic effects of modern CPU vulnerabilities. Demonstrated in this study are Spectre vulnerability attack and Load Value Injection (LVI) transient-execution attack. The storing of secret data within the cache is shown for Spectre, and through the use of an attacker's injected page number after a page fault has occurred, we demonstrate LVI's ability to access the host secrets via simulated memory hierarchy. The persistence of the secret data in the cache can also be observed in the case of both attacks. The characteristics of such security vulnerabilities are successfully simulated with the proposed Node-RED-based processor simulator.

Samuel Miles
Department of Electrical and Computer Engineering at IUPUI, Indianapolis, IN 46202, e-mail: sammiles@iupui.edu

Corey McDonough
Department of Electrical and Computer Engineering at IUPUI, Indianapolis, IN 46202, e-mail: cojomcdo@iupui.edu

Emmanuel Obichukwu Michael
Department of Electrical and Computer Engineering at IUPUI, Indianapolis, IN 46202, e-mail: emobmich@iupui.edu

Valli Sanghami Shankar Kumar
Department of Electrical and Computer Engineering at IUPUI, Indianapolis, IN 46202, e-mail: vshanka@iupui.edu

John J. Lee
Department of Electrical and Computer Engineering at IUPUI, Indianapolis, IN 46202, e-mail: johnlee@iupui.edu

This is the author's manuscript of the article published in final edited form as:

Miles, S., McDonough, C., Michael, E. O., Shankar Kumar, V. S., & Lee, J. J. (2022). Simulating Modern CPU Vulnerabilities on a 5-stage MIPS Pipeline Using Node-RED. In P. Verma, C. Charan, X. Fernando, & S. Ganesan (Eds.), Advances in Data Computing, Communication and Security (pp. 707–716). Springer Nature. https://doi.org/10.1007/978-981-16-8403-6_65

1 Introduction

Transient execution CPU vulnerabilities are defined as attacks in which a speculative execution optimization within microprocessors can be used by an attacker to access secret information. Due to the parallel nature of modern computers, if an operation is not able to be performed due to a previous operation that is taking more time and has not yet completed, the microprocessor may try to predict the result of the aforementioned previous operation. This is called speculative execution and is what enables many modern CPU vulnerabilities, such as Spectre [1] and Meltdown [2], which are capable of accessing secret information. Many transient-execution attacks abuse transient instructions to encode unauthorised data in a victim program. In this study, we demonstrate the effects of two modern CPU vulnerabilities by showing how speculative and injection-based attacks can load secret information into cache using a processor simulator built with Node-RED [3].

2 Prior Work

Recent research works on transient execution attacks have predominantly focused on miss-prediction (diverting control flow) and data extraction type attacks (exploiting illegal data flows). Several studies talk about using page-faults or microcode assists to obtain data from various micro-architectural elements such as caches [4] and store buffers [5]. Others discuss the way that transient-execution attacks have evolved [6] and what defenses exist [7]. However, these studies primarily focus on the breadth of capabilities for the attacks by fully dictating the attack vector, rather than clearly illustrating and demonstrating some of these effects in a tangible and intuitive format. Specifically, the idea of simulating Spectre-type vulnerabilities is discussed by researches in [8] and is not new, but there is no current works using Node-RED to simulate CPU vulnerabilities. The goal of this study is to provide a visual, evidence-based framework for illustrating the effects of these exploits in a novel and intuitive way. Specifically, our contributions include:

- Simulation of a Node-RED-based 5-stage pipelined MIPS processor.
- Demonstration of the effects of modern CPU's transient execution vulnerabilities.
- Simulation of the two-phase operation of a real CPU by using both rising and falling edges of a clock.

3 Simulator Implementation

The implementation is broken into sub-sections that discuss the details for each aspect of the pipeline as well as the additional pieces of the simulator that were

developed in order to properly replicate the necessary functionalities for vulnerability demonstration. The diagram for the proposed simulator is shown in Figure 1.

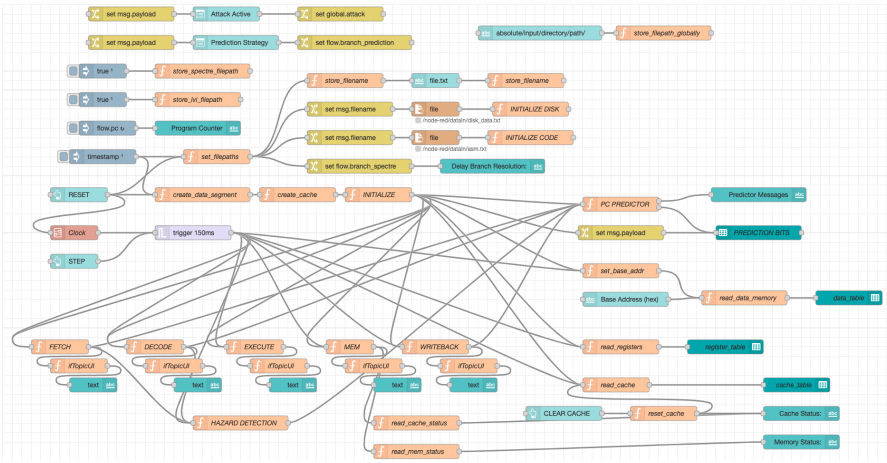

Fig. 1 Proposed 5-stage pipelined MIPS processor.

3.1 Implementation Details

The simulator illustrates the basic function of a 5-stage pipelined MIPS processor [9], while demonstrating the effects of modern CPU vulnerabilities. It relies on either an internal clock or by manually stepping through cycles in order to process each instruction. Assembly instructions are read from a text file where they have been encoded in hexadecimal.

3.1.1 Clock Cycles

Clock cycles in the pipeline are coordinated by a trigger node that sends a rising and falling edge signal. Changing the Clock Period alters a Timer Delay node that repeats the signal. Once all appropriate actions are performed during rising edge, the nodes save all data necessary for the next stage to local Node Memory. On the falling edge of the clock cycle, the Node Memory is set to Flow Memory buffers to be read by the next stage of the pipeline during the next cycle. Some operations such as fetching data from registers during the decode stage occur during the falling edge. This ensures the write-back operation can be completed before attempting to access the registers during the same clock cycle. Branch and PC calculations also occur during the falling edge to ensure all data can be forwarded before proceeding.

3.1.2 Fetch

During the rising edge, the Fetch stage of the pipeline reads the current PC register value from Flow Memory. The block verifies that code is loaded and the program has not finished execution. It then fetches the next instruction from the code relative to the PC value and sets the data into the pipeline register during the falling edge.

3.1.3 Decode

During the rising edge, the node reads the instruction register written by the previous stage and parses the information. This includes OP code, instruction type (i, j, r), rs, rt, rd, immediate, and address fields. During the falling edge, the parsed instruction is moved to the pipeline buffers. If the instruction is a branch, the node determines if the branch is predicted to be taken or not. The PC predictor evaluates miss-predictions to update the branch history table (BHT). If the Spectre is enabled, resulting prediction information is sent during the write-back stage instead.

3.1.4 Execute

During the rising edge, the data written by the decode stage is read and executed. Forwarded operands from the execute/memory and memory/write-back stages replace values if a data hazard is detected. On the falling edge, the results are written to pipeline buffers for the next stage.

3.1.5 Memory and Cache

In the memory stage, for store and load word (SW/LW) operations, during the rising edge, the address to memory access passed from the execute stage is read. Then, the cache is checked first. A fully associative write-through cache with Least Recently Used (LRU) replacement scheme has been implemented. If the address is found in the cache, the cache is accessed. For SW, the cache and memory are updated. For LW, if miss occurs, the cache checks if there is an empty block. If cache is full, blocks are replaced based on the LRU scheme. In case of cache miss, the system will check the simulated RAM storage for the specified virtual memory address. We implement a small amount of simulated memory that can be directly mapped into four equal size pages. In case of RAM miss, a page fault will occur and the desired page will be loaded from the simulated DISK. The memory hierarchy is simulated using RAM and DISK storage representations. The RAM is set to be 1KB, with a 4KB DISK. The DISK is initially loaded with data from an external text file where most entries are set to 0 except for some *TRUSTED* and *SECRET* values, which are represented as strings containing the representative text “TRUSTED” and “SECRET” for the purposes of simulating load value injection attacks, located at

specified physical memory addresses used in the simulation. Cache hit/miss statistics as well as declared memory are shown in the UI and updated every clock cycle.

3.1.6 Write-Back

During the rising edge, the data from the memory stage is read. If the instruction writes back to memory, the result is written back. If the Spectre attack is enabled, the branch calculation determined during the decode stage is sent to the PC predictor. No results are written to pipeline buffers during the falling edge because the pipeline is complete.

3.1.7 Hazard Detection and Stalls

The hazard detection block receives information from the fetch and decode stages. The node parses necessary information from the fetched instruction and determines if there is a true dependency caused by a LW instruction. If a dependency is detected, a stall signal is sent to the PC predictor. The PC predictor will not increase the PC for a stall cycle and will override the value in the current fetch-stage register with a stall instruction. This will be decoded next cycle and future stages become idle when they read the instruction. If the Spectre attack is enabled, the PC predictor will also flush the decode, execution, and memory pipeline buffers.

3.1.8 PC Predictor

The PC predictor receives messages from the Fetch stage informing the node to continue updating/predicting the PC. If the instruction address is in the BHT, the predictor will update the PC based on the prediction strategy selected. The PC predictor also receives branch messages from the decode stage stating the results of branch calculations (calculated taken or not-taken) and the address of the instruction. The node updates the BHT and flushes the fetch-stage register if the branch was miss-predicted. Various branch prediction strategies can be chosen between. On hit/misses, the BHT is updated with the most current predictions for each strategy. On first encountering a branch instruction, the program will always predict not taken and create a new entry in the BHT using the branch address.

3.1.9 Simulation Settings

The simulator gets input instructions as a file of hexadecimal instructions separated by newlines. A program file can be created using a MIPS-32 assembler provided by MARS [10] was used for this purpose. The simulator can start or stop with a button and can be resumed from a stop. A stepping function is present to step through one

instruction at a time to better visualize the pipeline. A reset button clears the memory, registers, cache, pipeline, BHT, and reloading the input files. The branch prediction strategy defaults to 1-bit prediction and the attack settings default to off. Simulating either attack can be done through the drop-down menu in the UI. Both the active and inactive states for each attack can be demonstrated. Enabling Spectre activates the delay branch prediction setting in the system. This forces branch resolution to occur during the Write-Back clock cycle. LW and SW operations can alter the cache if executed directly after a miss-predicted branch instruction. Any write to memory will be invalidated if the branch was miss-predicted. In LVI the malicious page number is injected to load data onto RAM on a page fault. Data from outside the allowed access page for the process is loaded into RAM and the cache, which allows attackers to access the host *SECRET* information.

4 Simulation of Vulnerabilities

Spectre attack and Load Value Injection (LVI) attack on MIPS processor are vulnerabilities that affect modern processors through techniques like branch prediction and out-of-order execution that were originally designed for performance purposes.

4.1 Spectre

Spectre attack is a security vulnerability that was discovered as part of Google's project *Zero* which was launched in 2018 to identify security issues with modern day processors [1]. Spectre exploits speculative execution on modern processors to steal secret data [1]. Speculative execution utilizes branch prediction to execute a set of instructions and avoid stalling while trying to decide if a branch is taken or not. This is used whenever there is a branch or when there is a control dependency between the branch instruction and an earlier instruction that has not written back to a register. The speculated instructions are flushed on a branch miss-prediction. However, footprints left in the processor allows for the Spectre exploits. A simple spectre attack involves training a branch to access data from an array with a check on the array size. Once the attacker is confident that the branch is sufficiently trained to always access this array, it then uses this behavior to try to access secret data in an unreachable memory space using the array source address. Since the branch is always taken, the data is accessed and loaded in the cache before the system realizes that the branch was miss-predicted and flushes the pipeline leaving the cache untouched. A series of timed cache accesses are then used to grab this secret data from the cache. Spectre attacks can occur in virtually any computing environment.

As demonstrated in Figure 2, secret data (0x6970) is initially stored at address 0x400. An array of size 10 is initialized (array is set to all zeros for the demo but can be set to any value) starting from the base memory address 0x108. A loop is used

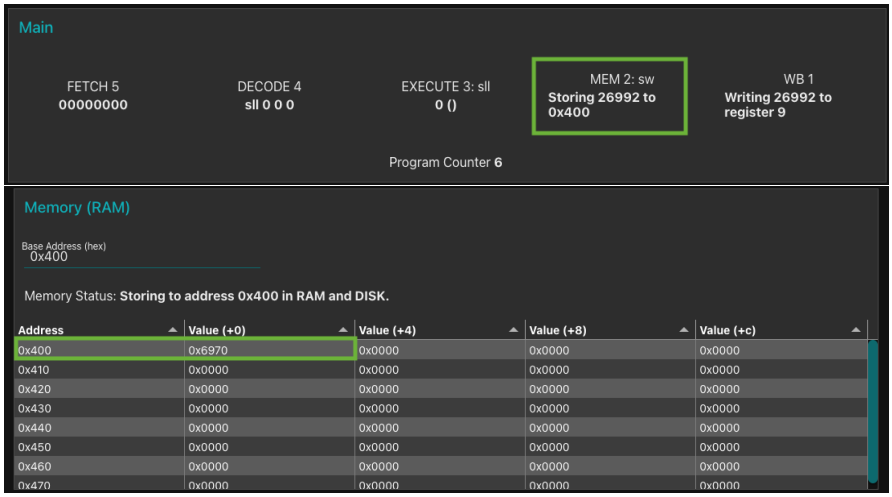

Fig. 2 Secret data is stored by the program in memory address 0x400.

to train the array access branch that checks the array access index to be less than ten. The looping check uses branch prediction to train the BHT entry to "Always Taken". As the array size is always flushed from the cache, memory access for store operations will require additional clock cycles, creating a window for the Spectre attack on a miss-prediction. This loop is run four times, allowing array access and giving the process the false impression that array access is always going to be taken. After a miss-prediction, a full pipeline flush occurs but the secret is already cached. Figure 3 shows the outcomes of the spectre attack simulation.

Fig. 3 Secret data is cached by the Spectre attack.

4.2 Load Value Injection (LVI)

LVI attack reversely exploits Meltdown-type micro-architectural data leakage that was discovered by researchers in 2019. Meltdown is a vulnerability allowing a process to read all memory in a given system [2]. This attack was initially thought to target only Intel SGX enclave but authors in [11] have shown that LVI-type of attacks are applicable in other domains. LVI attack is a lethal attack since it mimics certain existing attacks but none of those attack's defences can actually mitigate it. LVI combines Spectre-style code gadgets with Meltdown-type illegal data flows to bypass existing defenses but none of the existing Spectre/Meltdown defenses can mitigate LVI. LVI is one of the most advanced exploitation techniques presented to date since it combines page table manipulations and invalid transient executions. An attacker can use poisoned data loaded into various micro-architectural buffers in the legitimate victim program. It is shown that these buffers can be injected with a value in advance [11]. Recent research also suggests the existence of different LVI variants [11]. Some suggest an LVI-type attack to hijack the control flow through an injection of a poisoned address upon returning from a branch or jump instruction. This allows the attacker to execute malicious code on the host system, further illustrating the dangers of LVI attacks. By targeting load-type instructions, LVI drastically widens the spectrum of incorrect transient paths. In order to successfully exploit LVI, it needs to be possible to induce a page fault or microcode assists during execution on the host. LVI attacks are said to occur in three phases:

- Micro-architectural poisoning,
- Faulting loads,
- Secret data transmission.

Figure 4 shows the first step in LVI, where it can be observed that the cache is loaded with the *SECRET*, as defined in Section 3.1.5 previously. Figure 5 shows that even after another page fault occurs and resets the RAM, it remains in the cache.

The system is designed such that typically when a page fault occurs, the page that is being used to map the storage changes based on what physical address is desired. In the case of simulating LVI, the page number that normally is calculated based on the desired physical address is instead replaced with an attacker's page number that then allows for memory access outside of the currently accessible process to be breached. Through a page fault, the attacker can access *SECRET* and even upon an additional page fault where RAM is cleared, *SECRET* is still accessible to the attacker from the data cache. Currently with LVI-type attacks, mitigation requires the serialization of the processor pipeline with "lfence" (load fence)[12] instructions after every memory load. Research shows these defenses lead to performance penalties anywhere from 2x to 19x loss [13]. Intel has developed LLVM-based compiler pass which inserts lfence instructions in an optimal way to reduce overhead, but even this technique has proven to introduce more than a negligible penalty [11].

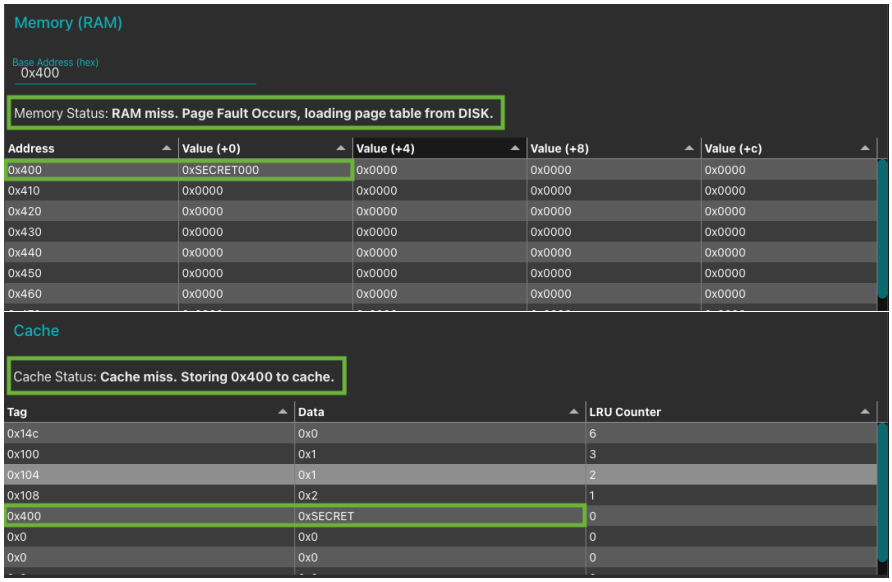

Fig. 4 LVI loads secret data into the cache.

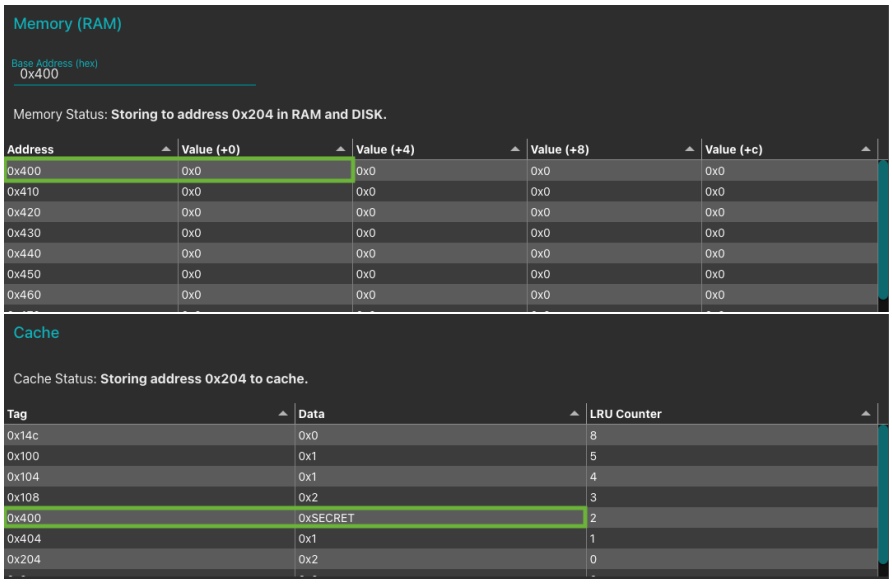

Fig. 5 Secret data remains in the cache, even after another page fault.

5 Conclusions

This study achieves the goal of providing a tangible framework to show the effects of modern CPU exploits. We do this by showing how secret data can be accessed through side-channels and loaded into the data cache to be accessed by the attacker. We contribute an educational simulator of some CPU exploits that shows the effects of these attacks such that understanding them is easier, as well as to encourage further research and development in combating these exploits. Future work would be to add support for J-type instructions, floating point values, and the ability to change any of the cache mapping or replacement policies.

References

1. P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," 2019 IEEE Symposium on Security and Privacy (SP), pp. 1-19, 2019.
2. M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," arXiv:1801.01207, 2018.
3. IBM, Node-RED
https://nodered.org/, Accessed May 7, 2021.
4. J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikxi, F. Piessens, M. Silberstein, T. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," 27th USENIX Security Symposium (USENIX Security 18), pp. 991-1008, 2018.
5. C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant CPUs," Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 769-784, 2019.
6. C. Canella, K. Khasawneh, and D. Gruss, "The evolution of transient-execution attacks," Proceedings of the 2020 on Great Lakes Symposium on VLSI, pp. 163-168, 2020.
7. C. Canella, S. M. Pudukotai Dinakarrao, D. Gruss, and K. Khasawneh, "Evolution of defenses against transient-execution attacks," Proceedings of the 2020 on Great Lakes Symposium on VLSI, pp. 169-174, 2020.
8. O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "SpecFuzz: Bringing Spectre-type vulnerabilities to the surface," In 29th USENIX Security Symposium (USENIX Security 20), pp. 1481-1498. 2020.
9. mips.com, MIPS Architecture For Programmers Volume 1-A, Available via DIALOG.
https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00082-2B-MIPS32INT-AFP-06.01.pdf, Accessed May 7, 2021.
10. K. Vollmar and P. Sanderson, "MARS: an education-oriented MIPS assembly language simulator," in SIGCSE, vol. 6, pp. 239–243, 2006.
11. J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking transient execution through microarchitectural load value injection," 2020 IEEE Symposium on Security and Privacy (SP), pp. 54-72, 2020.
12. Intel, "An Optimized Mitigation Approach for Load Value Injection,"
https://intel.ly/2CSsHwp, Accessed May 10, 2021
13. M. Larabel, "The brutal performance impact from mitigating the LVI vulnerability,"
https://www.phoronix.com/scan.php? page=article&item=lvi-attack-perf.