Mitigating transient execution attacks with focus on performance-security tradeoff

Bi-annual seminar-I Report(MS by research)
by

Sumon Nath
(21Q050007)

Supervisor:
Prof. Biswabandan Panda

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076 (India)

22 December 2022

Table of Contents

List of Figures

Introduction

Modern processor employ a plethora of techniques to get high performance from the available silicon. It is not just the increasing clock speed that has led to the high performance but the ability to pump out multiple instructions in a single clock cycle while maintaining the program correctness, has led to improvement of performance by multiple folds. Techniques like Out-of-order and speculative execution, multi-threading, caching and hardware prefetching are some of the key contributors for the same. But, this narrow approach of improving performance over the past few decades without critically analyzing their side effects has led to an insecure foundation when it comes to the aspect of security. Recently, there has been multiple proposals which identifies and exploits such vulnerabilities created by some of the techniques which improves performance. For example, Spectre[1] is one of the more lethal attacks which takes advantage of the side channel created by out-of-order and speculative execution to leak secret data. In this section we will discussing some of the techniques employed in modern high-speed processor.

Out-of-order: This is one of the techniques used in modern high performance processors which can execute multiple instructions simultaneously and in an out-of-order fashion. This takes advantage of the fact that some instructions may take a long time to execute while other can complete in a short duration, and executes as many independent instructions as possible while the long latency instruction finishes. Although, to maintain program correctness, all the instructions must come out of the pipeline in the order in which it entered. This is where a structure called the re-order buffer (ROB), comes into play. The ROB buffers the information related to all currently executing instruction and in-order. Instructions from the head of the ROB are taken out of the pipeline whenever it completes and it halts the following instructions from getting out of the pipeline maintaining program order. Even while pipeline is halted due to a long latency instruction, the following instructions can complete their execution and update the ROB entries as and when required as show in figure 1.1. This allows bursts of completed instructions to commit from the pipeline when they reach the head of the ROB, resulting in a high performance improvement.

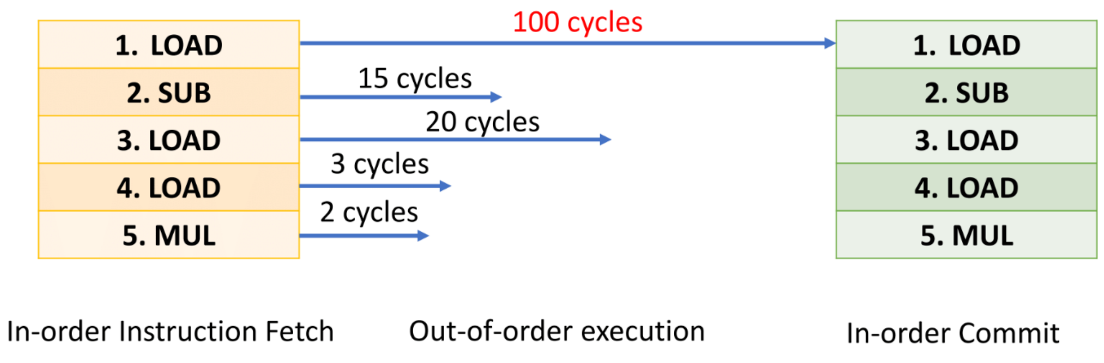


Figure 1.1: Out-of-order execution

speculative execution: It may happen that the processor is not aware of the future instruction stream. This occurs when a conditional branch instruction is encountered in the pipeline. In such cases modern processors guesses the most probable path the branch might take using a structure called the branch predictor, preserves its current state and starts fetching and executing instructions from the predicted path. When the branch eventually resolves at a later point of time the processor can take the appropriate action. In the best case, if the predicted path was correct, the results of the speculatively executed instructions are committed and normal execution continues. In case the prediction is wrong, the processor restores it state to the one just before it executed the conditional branch. This way, in the best case the processor can execute 1000s of instructions way ahead of its time resulting in high performance improvement and in the worst case it flushes out the entire pipeline of wrongly predicted instructions, without affecting the correctness of program.

branch prediction: As discussed while the processor speculatively executes instructions, it uses a structure called the branch predictor to predict the direction of a branch instruction. Modern branch predictors are very sophisticated in nature and uses the history of previous branch outcomes to learn and predict the outcome of a newly encountered branch. These predictors can predict both indirect and direct branch outcomes. For indirect branches where the target address is unknown before the completion of the branch instruction, the branch predictor uses another structure called the branch target buffer(BTB) which buffers all the target addresses which the branch instruction referred to previously. For direct branches there is no need to predict the target address as the the address is available in the instruction itself as an immediate. Some branch predictors in modern processors are based on machine learning techniques which leads to very high prediction accuracy. Most branch predictors in modern systems have a prediction accuracy of more than 95%.

Cache hierarchy: The infamous 3 level cache hierarchy is another technique which improves processor performance by reducing the average memory access time by storing a subset of the memory in a smaller and faster memory, known as cache, with the expectation that the subset will contain the most frequently used data/instructions. A hit in the cache can give 10x to 50x less memory access time depending on the level at which the data is found, also called a cache hit. So a high cache hit rate leads to high overall performance improvement. The cache is usually divided into fixed size blocks called lines and uses different mapping policies to map a particular memory block to a corresponding cache line. Caches also uses sophisticated replacement policies which decides which line to evict from the cache in case it is full. Replacement policies play an important, as a bad replacement policy can kick out a block from the cache which will be referenced very soon in the future instead of a block that will be used farther in the future leading to a high latency load.

Speculative attack

As discussed in the last section, modern processors can speculatively execute instructions by predicting future program behaviour. In case of wrong prediction the cpu reverts back to state as the wrongly predicted branch and following instructions were never executed. But it overlooks the fact that the speculatively executed instructions may affect other microarhitectural structures. For example, the cache may contain speculative data even after the transient instructions are flushed out of the pipline. The reason being, the cache is not flushed when a wrongly predicted path is detected. Usually the cache is microarchitecutural structure which can not be referred directly from the programmer side and thus it is assumed that the programmer will not be able to retrieve anything meaning from the cache. But in recent literature it is shown that we can use timing channels to leak information from the cache. This leads to a highly vulnerable environment in a multiprogramming setup where multiple programs can run at the same time via time-sharing or in parallel in multiple cores. The attacker can leak secret information from data present in the cache which was brought in unintended into the cache by the victim process as a result of speculative execution. Next we will be discussing an attack called Spectre which demonstrates such an attack can be possible in a real system.

2.1 Spectre attack

Figure 2.1 shows the code which can potentially leak victims data. The scenario is such that the attacker can invoke the victim with an argument which sets the value of x. And the array2 is a shared data structure between the victim and the attacker. The goal of the attacker is to retrieve the secret key array1[x] where x is an out-of-bounds value. The preliminary step for the attack to be successful is to mistrain the branch predictor by invoking the victim with a value of x which always satisfies the condition in line 1 of the victim code. Then the attacker invokes the victim with an out-of-bounds value of x,say 19, in the hope of leaking the secret key which is array1[19] = 49. Now even though the condition would be false, the processor will continue to execute the wrong

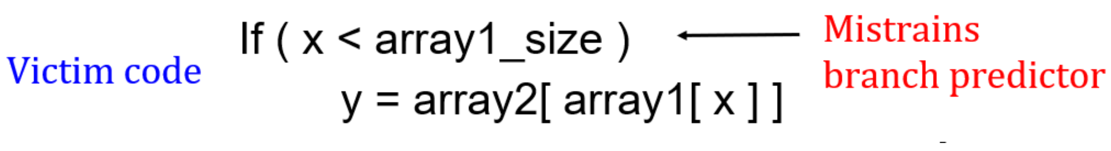


Figure 2.1: Victim code to demonstrate the spectre attack

path instruction till the branch gets resolved. This was possible as the branch predictor was mistrained to always predict the condition as true. Because of this line 2 in victim code will be executed and the value of array2[49] = 129 will be accessed from memory and stored into cache. Even after the branch is resolved and the misprediction is detected and all the wrong instructions are flushed from the pipeline, the transient data still resides in the cache. The attacker can use any timing channel attacks like FLUSH+RELOAD [2] or PRIME+PROBE [3] to get the location of array2 which was accessed. Using the location it can infer the secret key as show in figure 2.2 Figure 2.3 shows the attack in

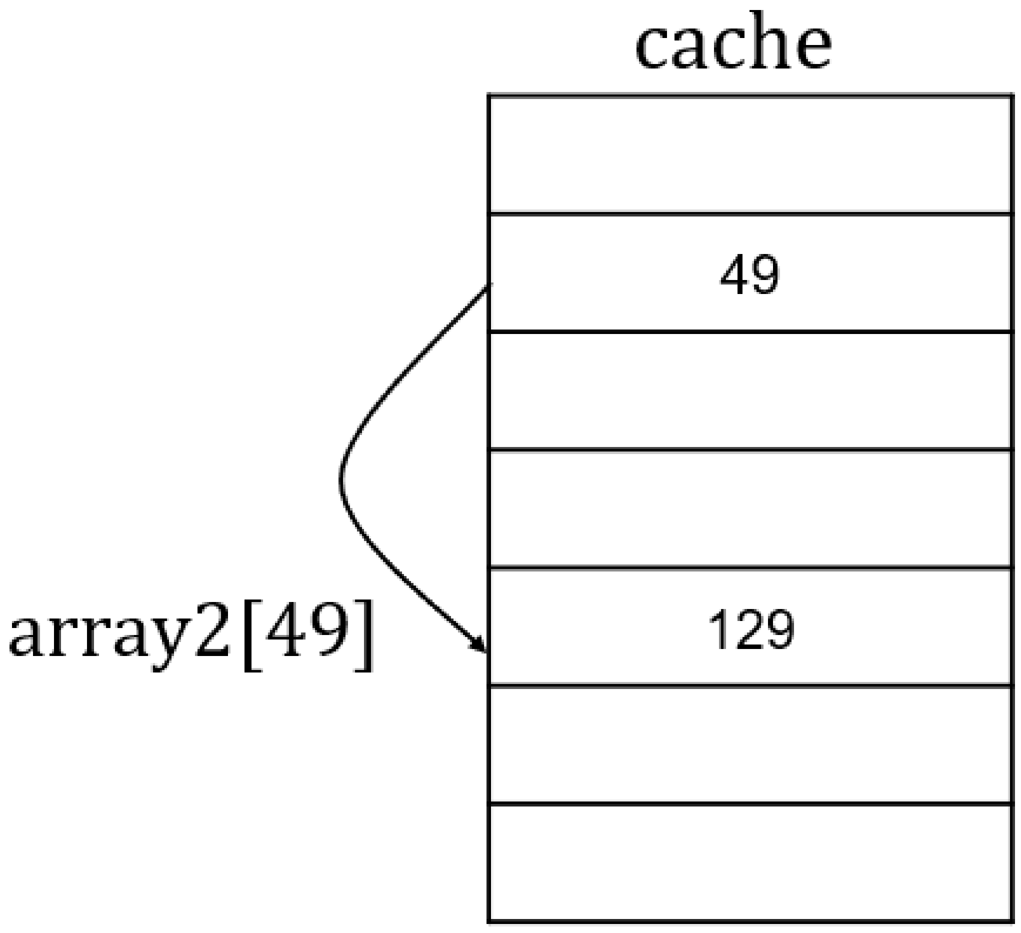


Figure 2.2: Cache state in the spectre attack

a nutshell, where the attacker initially primes the shared array. Then invokes the victim with an out-of-bounds value of x. The victim accessed the secret key and using the secret key as an index accesses the shared data structure, leading to eviction of one the lines in the shared array. Then attacker can infer the secret key by finding out the location of the evicted data by using one of the timing-based attacks.

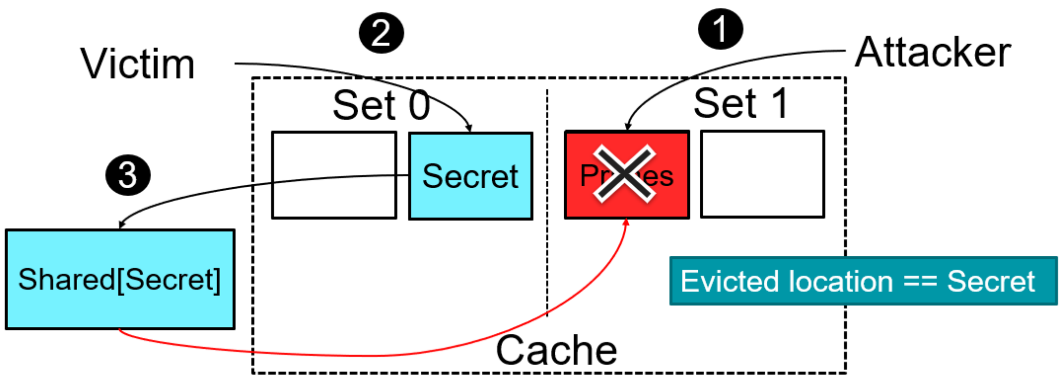


Figure 2.3: Overview of the spectre attack

Mitigation strategies

Numerous mitigation strategies have been proposed since the attack came out. We will now discuss some of the counter measures to prevent the spectre attack. Preventing speculative execution is one the ways we can prevent spectre attacks as it is one the necessary requirements for a successful spectre attack. The way to do is to stop speculative execution whenever we reach a code which accesses some secret data. But as of now there are functionality in modern processors to switch off speculative execution completely. An alternative solution is to use instructions which can block speculative instruction. One of the instruction that can accomplish this the lfence instruction. But the disadvantage of this method is that there needs to be a change in the software level and it causes high performance overhead. Another way to mitigate spectre attacks is to prevent access to secret data. This can be done by replacing bounds checking code by some other code which does not use conditional branch instructions. Index masking is one such technique which can replace array bounds checking. Again the problem with this method is, it is a software approach and all binaries need to be modified to mitigate the attack. Recently, a few hardware approaches have been proposed namely MuonTrap[4] and GhostMinion[5] which mitigates the Spectre attack and its variants with minimum performance overhead. Next we will be discussing the MuonTrap mitigation strategy.

3.1 MuonTrap

The key problem identified by the authors which allows the spectre attack is the sharing of transient data across domain boundaries. Where a domain boundary can be a context switch between two processes. One solution is to flush the entire cache on a context switch, but it is not a practical solution as it will lead to a high performance degradation. The idea is to introduce another cache in the cache hierarchy where the new cache is called the filter cache as show in figure 3.1. The filter cache is a very small cache which can be flushed in a single cycle. The filter cache stores all the speculative data and is flushed when there is a switch in domain boundary which in most cases is a context switch. The rest of the cache hierarchy must not be affected by speculative data requests, which means, speculative data is not filled into the rest of the cache hierarchy. Only when an instruction commits, that is, it is not speculative anymore, the data associated with it is written back to the level 1 cache and then eventually to the rest of the cache hierarchy. Recently, more variants of Spectre has come where rollback of speculative instructions is

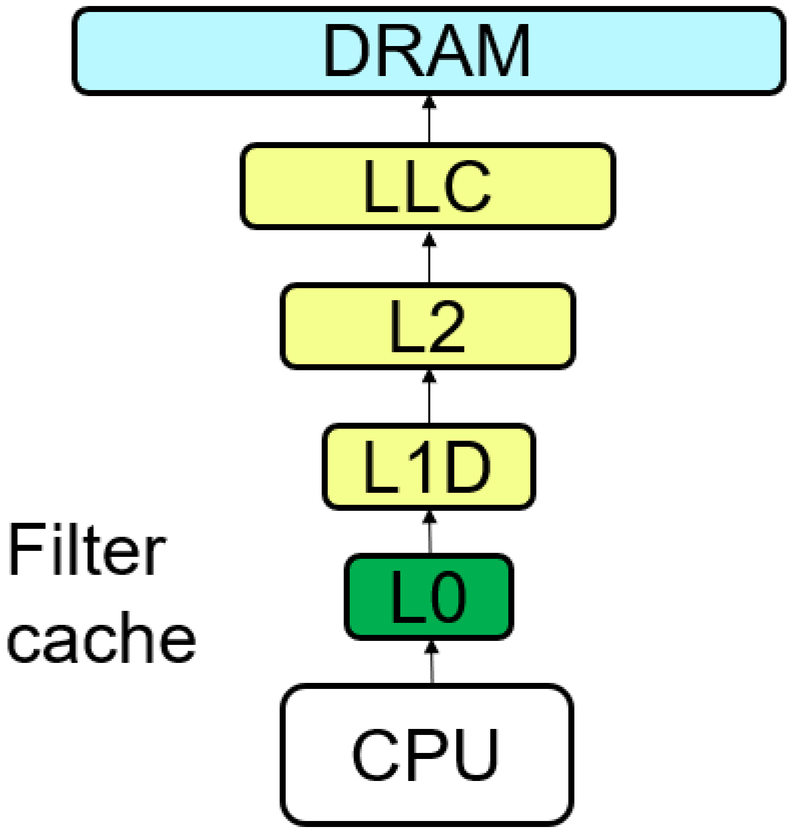
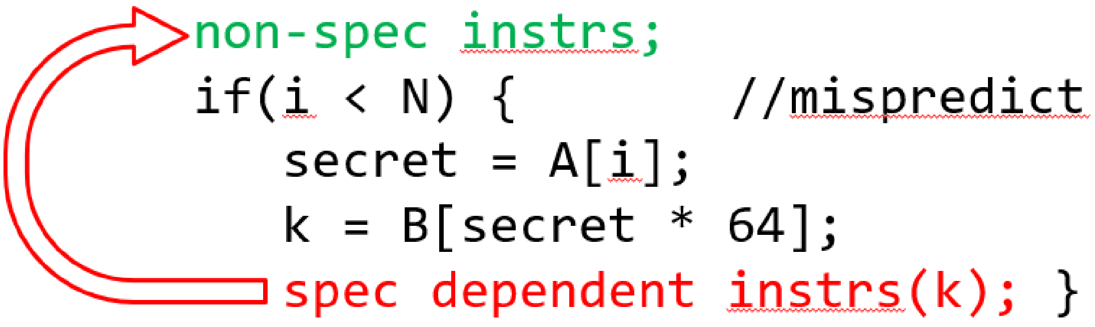


Figure 3.1: Modified cache hierarchy for MuonTrap

not enough. Attacks like SpectreRewind[6] and Speculative Interference[7] uses the fact that speculative instructions can cause the timing effects on logically earlier instructions which can change the order in which the non-speculative instructions execute as we are dealing with an out-of-order processor. As show in figure 3.2 a speculative instruction affects the order in which the non-speculative instructions execute, which can in turn change the microarchitectural state and creates side-channels that can be exploited by attackers. To mitigate these new attacks and Ghostminion was proposed which proposes a more generalized solution which can and will be able to mitigate a larger of pool of spectre attacks. In the next section we will be discussing GhostMinion in details.



```
non-spec instrs;
if(i < N) {         //mispredict
   secret = A[i];
   k = B[secret * 64];
   spec dependent instrs(k); }
```

Figure 3.2: Backward-in time channel

3.2 GhostMinion

Similar to MuonTrap it uses a filter cache to filter all of the speculative requests. The distinguishing factor of GhostMinion is that, it employs multiple restrictions to the filter cache which makes it resilient toward the new attack variants. Here are the distinguishing factors of GhostMinion:

Parallel access: The filter cache also called the GhostMinion is accessed in parallel to the L1 cache, rather than having a sequential access to L1 cache as show in figure **??**. This is done to avoid the overhead of accessing the filter cache even when the data is not present in the filter cache. **TimeGuarding:** To ensure a temporal ordering of instruc-

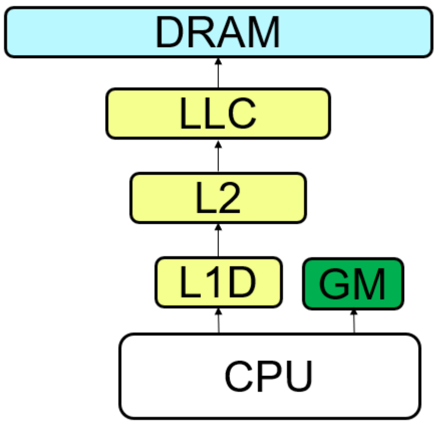


Figure 3.3: Modified cache hierarchy for GhostMinion

tions, GhostMinion uses the timeguarding technique. Every cache line is associated with a timestamp of the associated instruction which last accessed the cache line. It restricts a read from the filter cache if a newer instruction tries to read the data of an older instruction. While filling a line in the filter cache, it restricts a newer instruction to evict a data associated to an older instruction as shown in figure 3.4.

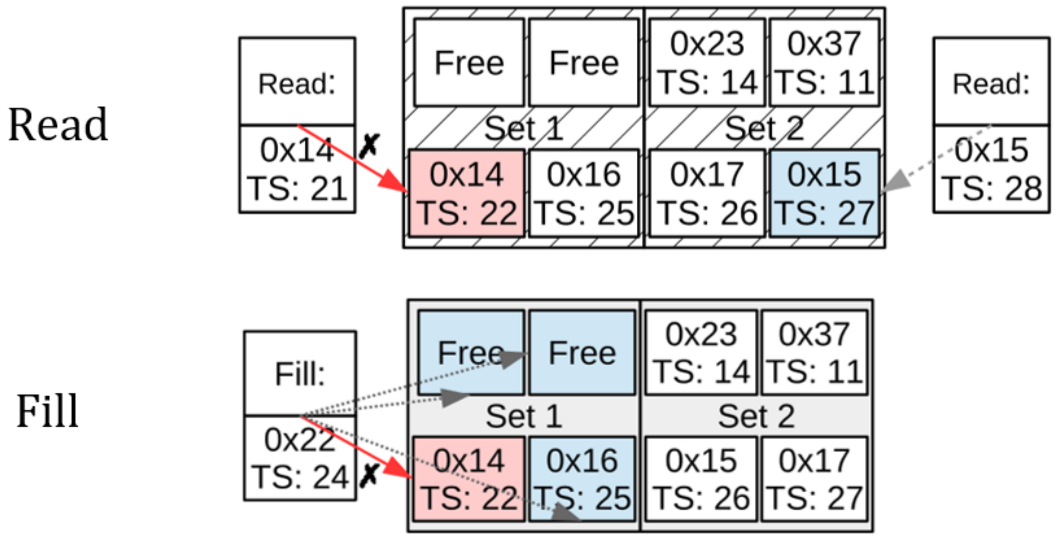


Figure 3.4: Timeguarding technique used in GhostMinion

Free-slotting: To ensure enough space is there in the filter cache, whenever an instruction is committed, the associated cache line is evicted from the filter cache and written back to the L1 cache.

LeapFrogging: To avoid backwards-in-time attacks through the MSHR, GhostMinion allows an older instruction to evict MSHR entry of a newer instruction whenever the MSHR is full. The new instruction retires whenever the pipeline is free a later point of time.

Problem statement

The major problem with the above two ideas is that the evaluation setup does not model modern processors with heterogeneous workloads. The evaluation setup uses a small ROB and Load-store queues as a result the experiments are not applicable for a modern processor with larger ROB size and Load-store queues. GhostMinion only prefetches for instructions which are committed, which results in bad prefetcher timelines. So the major task is to analyze the performance impact of GhostMinion on an updated evaluation setup. Next we will discussing some of the implementation details.

4.1 Implementation details

We are currently implementing the GhostMinion cache along with all its features and the plan is to evaluate its performance on the updated evalutaion setup. We are implementing it on a trace based simulator called Champsim. The features implemented so far are as follows:

- Implemented the basic structure of GhostMinion.
- Integrated it with the rest of the cache hierarchy.
- Implemented the features timeguarding and free-slotting.

Some of the features like leap-frogging and extending the GhostMinion idea to TLBs is yet pending. The major complexity of the implementation was to make the GhostMinion cache a VIPT cache and make it work with rest of the cache hierarchy as shown in figure 4.1.

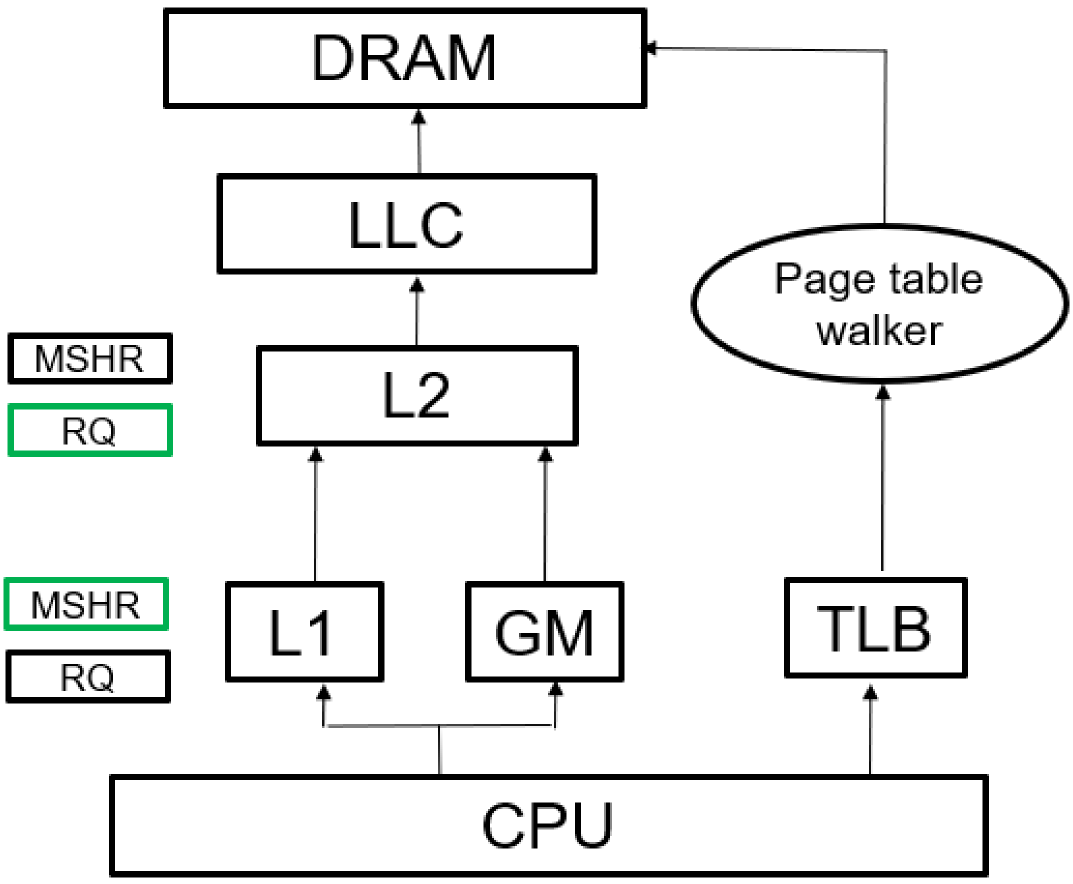


Figure 4.1: Implementation complexity of GhostMinin

Conclusion and Future work

Modern processors utilize techniques like out-of-order and speculative execution to improve performance. However, it results in more vulnerabilities which are exploited by attacks like Spectre. Mitigation techniques like MuonTrap and GhostMinion which are based on the idea of filter cache mitigates such attacks with minimum performance overhead. However, the evaluation setup used in such techniques does not model a modern processor faithfully. In this particular phase we sought to implement and evaluate the aforementioned ideas in a revised evaluation setup.

Acknowledgements

I am grateful to **Prof. Biswabandan Panda** for guiding me and helping me throughout my thesis work.

Sumon Nath
IIT Bombay
22 December 2022

References

[1] Kocher, Paul, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg et al. "Spectre attacks: Exploiting speculative execution." Communications of the ACM 63, no. 7 (2020): 93-101

[2] Y. Yarom and K. Falkner, “Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack,” in USENIX Security Symposium, 2014

[3] Liu, Fangfei, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-level cache side-channel attacks are practical." In 2015 IEEE symposium on security and privacy, pp. 605-622. IEEE, 2015

[4] Ainsworth, Sam, and Timothy M. Jones. "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state." In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pp. 132-144. IEEE, 2020

[5] Ainsworth, Sam. "GhostMinion: A strictness-ordered cache system for Spectre mitigation." In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 592-606. 2021

[6] Kocher, Paul, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg et al. "Spectre attacks: Exploiting speculative execution." Communications of the ACM 63, no. 7 (2020): 93-101

[7] Behnia, Mohammad, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer et al. "Speculative interference attacks: Breaking invisible speculation schemes." In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 1046-1060. 2021