Spectre: What do we know so far

Khan Shaikhul Hadi
dept. of Computer Science
University of Central Florida

***Abstract*—Spectre is a new kind of hardware vulnerability that let attacker leak secret data exploiting speculative execution. This is also know as speculative execution attack. Since it's first disclosure since Januray 2018, there are many variant of Spectre attack that have been exposed. Spectre attack mainly consist of two main component: transient execution or mispredicted speculative execution to create microarchitectural footprint and microarchitectural side channel or cover channel attack that actually leak the information to the attacker. Speculative execution is a fundamental feature in modern architectures that provides significant performance boost while covert channel are unintended channel that could leak information. Spectre attack have motivated researchers from both industry and academia to rethink the design of the processor and hardware defense as spectre is a fundament hardware design flaw. To make it easy to understand spectre vulnerability and what is the current state of our system to mitigate this threat, this paper presents a short review. This paper talk about some fundamental architectural properties, explain spectre attack and how it works. This also gives information about mitigation's that are in place in current system and some proposals to mitigate this threat at different stages.**

***Index Terms*—Speculative Execution, Branch Prediction, Branch Target Buffer, Out-of-Order Execution**

I. Introduction

Computation performed by physical devices often leave observable side effect that ideally may not be readable as they are not part of regular output. Side channel attack focused on this type of side effect to leak information that attacker have no permission to read. While some attack exploit software vulnerabilities, other software attack focus on hardware vulnerabilities to leak sensitive information. To launch this type of attack, attacker mast have extensive knowledge of targeted system.

Several microarchitectural design techniques have been introduced to increase performance in modern CPU. Out-of-order(OOO) execution and speculative execution reduce bubble in the architecture pipeline and thus, provide a significant performance gain. Out-of-order execution execute instructions out of order but maintain order to retire each instruction which creates an illusion of in-order execution to the programmer. It makes programmer's life easy, but as programmer do not know which instruction will be executed after which instruction in the pipeline, it is very difficult to control what kind of microarchitectural effect they will produce before hand. A user with malicious intention could utilize this to produce unintended microarchitectural footprint and later use side channel attack to leak secret information. Spectre based attack is one of such attacks where attacker utilize speculative execution to execute unauthorized data access and combined it with side channel attack, most commonly cache based side channel attack to leak targeted data.

In this paper will will see a short review on spectre vulnerability that focus on:

- spectre vulnerability and it's working principle.
- Example of major spectre variants
- Existing mitigations
- Different type of proposed solutions

II. Background Knowledge

A. Out-of-order execution

Most of the early processors executed instructions in order. One of the major bottleneck of in-order execution is that, if there is a data dependency faced, rest of the instructions have to wait no matter either later instructions are dependent on that data or not. At the same time, many stages of the pipeline will remain idle which incurs bubble and cause wastage of resources. Out-of-order execution idea was introduced to minimize this issue. *Out-of-Order* execution utilize processors component by allowing instructions further down the instruction stream of the program to be executed in parallel with or before

preceding instructions based on data availability. Modern processors internally work with micro-ops, emulate instruction set architecture [1] and issue micro-ops out of order to the pipeline. When micro-ops corresponding to the instruction along with preceding instructions are completed, the instruction can retired and their changes are committed in the architectural states,i.e. registers. Thus processors are able to maintain illusion of in-order execution as architectural states are being updated in order while executing instructions in *Out-of-Order*.

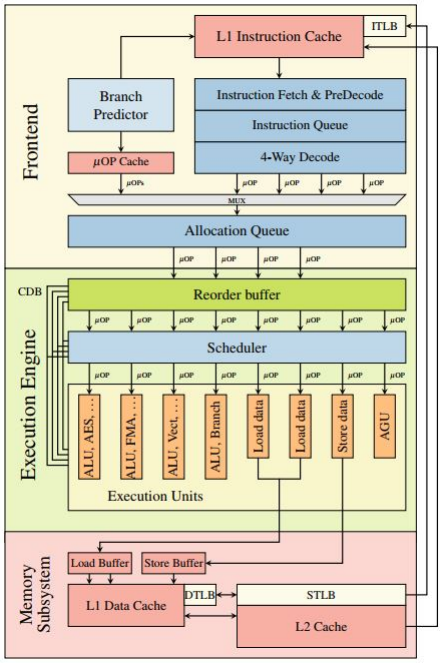

Fig. 1. Simplified illustration of a single core of the Intel's Skylake microarchitecture.Instructions are decoded into micro-ops and executed out-of-order in the execution engine by individual execution units [2]

B. Speculative Execution

For many instances, future instructions stream depends on instructions that are being issued but yet to produce output. For example, sometimes out-of-order execution reaches a conditional branch instructions whose direction depends on preceding instruction whose execution yet to be completed. If processor wait for the outcome, it will incur bubble in the system resulting performance loss. In such cases, processor preserve it's current register state, speculate about the path of next instruction stream and start executing along the path *speculatively*. If the prediction turns out to be correct, the result will be committed, resulting performance gain. Otherwise, if processor's speculation turns out to be incorrect, it abandons the work it performed speculatively by reverting its register state and resuming along the correct path. As a result, it does not have any architectural effect. But it may leave some microarchitectural footprint.

Instructions that are executed due to wrong prediction but may leave microarchitectural traces are referred as *transient instructions*. Speculative execution on modern CPUs could run several hundred instructions ahead as it's limit is typically governed by the size of the reorder buffer in the CPU. For instance, on the Haswell microarchitecture, the reorder buffer has sufficient space for 192 micro-ops [1].To speculate next instruction steam, modern processor use many techniques like branch predictor [1], branch target buffer(BTB), data speculation etc.

C. Branch Prediction

During speculative execution, instead of randomly predicting next instruction stream, processor use branch predictors to increase probability to predict right path. Better prediction improve performance as it increases number of speculatively executed instructions that will be committed. Modern processors use multiple prediction mechanism for direct and indirect branches and could predict with extremely high accuracy. For direct branches, prediction varies between either taken or not taken while for indirect branch, predictor have to predict next instruction address. To compensate this, direct jumps and calls are optimized using at least two different prediction mechanism [3].Intel [3] describes that the processor predicts

- "Direct Calls and Jumps" in a static or monotonic manner,
- "Indirect Calls and Jumps" either monotonic manner or varying manner depending on recent program behavior,
- "Conditional Branches" use branch target and predict whether the branch will be taken or not.

D. Address Spaces

Modern processors support virtual spaces to maintain isolation among processors. A virtual ad-

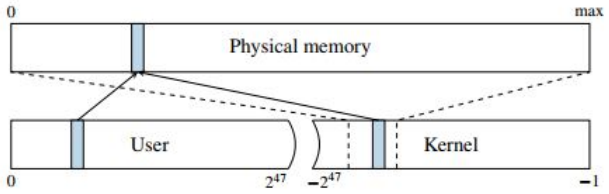

Fig. 2. The physical memory is directly mapped in the kernel at a certain offset. A physical address(blue) which is mapped accessible to the user space is also mapped in the kernel space through the direct mapping. [2]

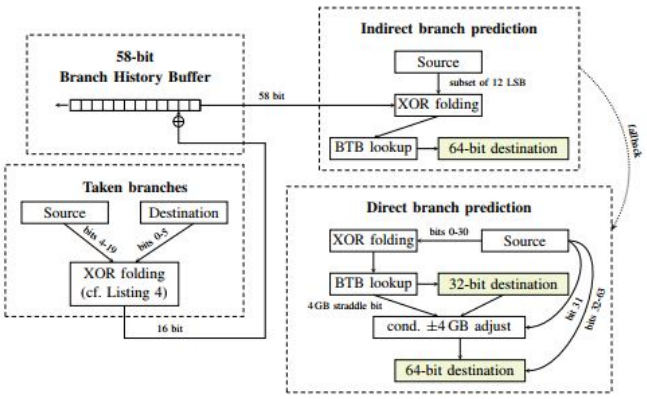

Fig. 3. Multiple mechanisms influence the prediction of direct,indirect and conditional branches [4]

dress space is divided into a set of pages that can be individually mapped to physical memory through a multi-level page translation table. Translation table define actual virtual address to physical mapping and also protection properties that are used to enforce privilege checks like readable, writable, executable and user accessible. In the context of context switching, operating system updates register with the next process's translation table address in order to implement per-process virtual address spaces thus prevent one process to referencing data that does not belong to it's virtual addresses. Each virtual address space itself is split into a user and a kernel part. Kernel address space can only be accessed if the CPU is running in privileged mode while normal user address space can be accessed by the running application. Consequently, entire physical memory is typically mapped in the kernel [Figure 2]. So, if a normal application can get higher privilege to read kernel space, eventually it could read all physical memory space that may not belong to that particular user and cause security breach for other users.

III. Spectre Attack

Spectre attacks involve manipulating speculation to unintentionally make victim perform operations that would not occur during correct program execution to create microarchitectural footprint based on secret data and then leak that confidential data to the adversary through side-channel. Most common way to mislead user program to execute unintentional program execution is to manipulate branch predictors. Spectre attack consist of three major step:

In first step, attacker mistrain branch predictor. Branch prediction mechanism could be influenced many way targeting different predictors operating principle [see Figure 3]. Attacker determines what is the mechanism of a particular predictor, create a training program that create attacker's desired predictor pattern in the predictor.

```
if(x<array1_size)
    y = array2[array1[x]*4096]
```

Listing 1. Bound Check Conditional Branch

In second step, attacker run victim's program as victim have the access privilege to read secret data. To protect secret data, there are many checking parameter in place in the system. But It takes time to complete all the checking. To gain performance, processor continue executing instructions speculatively. As attacker already manipulated predictor to predict according to their desire, victim function unintentionally read secret data and left microarchitectural footprint.

Finally attacker leak secret data based on side channel attack or covert channel attack. Even though this is the final part of the attack, some portion of the attack code may need to be executed to put microarchitectural states in base state before running victim code. After first publication of spectre attack [4](variant 1), it opens up a new way of research in the field of secure computer architecture. This leads to subsequent findings of many variations of spectre attack. If we combine that with existing side channels attacks, it will create myriad instances of attack. Theoretically attacker could utilize more that hundred variant of cache based side channel attack [5] just to leaked the confidential data that left footprint in the cache.As it will not possible to

cover all attack models, we will cover four major variants of spectre attack.

A. Variant 1: Exploiting Conditional Branch

In variant one, attacker mistrain branch prediction to predict true, then access out of bound memory while bypassing bound checks [4], [6].Lets consider a array access bound check condition shown in listing 1. This code fragment start with checking if variable `x` is less than `array1_size` so that memory outsize `array1` could not be read. This is a bound check to ensure unauthorized memory access protection. In ideal case, no value of `x` that would read outside the boundary of `array` could execute next line. But due to speculative execution, situation may diverted from ideal cases.This situation is illustrated in Figure 4 where we could see, if prediction was false but in actuality it would be true, that does not let attacker read secret value. But if bound check is predicted true while in reality it would be false, attacker could read secret value.

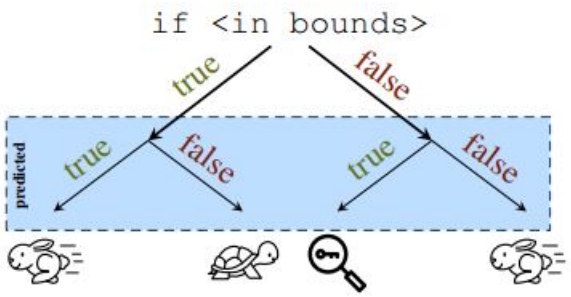

Fig. 4. If branch is predicted branch is taken, while in reality it will not be,it will let attacker transiently read secret data. [4]

In the example of Listing 1, value of `x` is chosen by attacker to be out-of-bound so that `array1[x]` will indicate to a secret byte `k` somewhere in the victim's memory. `array1_size` and `array2` are uncached but secret byte `k` is cached.Also previous operations received value of `x` in the bound that leads to train the branch predictor to predict `if` will likely to be true. For current operation, this leads to speculatively read `array1[x]` value to `k` where `x` is out of bound. As bound check is yet to complete, processor speculatively continue to execute instructions which leads to bringing value of `array2[k*4096]` into the cache. As eventually bound check will be completed and processor would realize that `x` is out of bound, it will not retire the speculatively executed instructions and made no architectural change. But as memory related to value of `k` is brought into cache, using cache based side channel attack, attacker could leak that byte value.

B. Variant 2: Poisoning Indirect Branches

Another branch prediction mechanism is Branch Target Buffer (BTB) that keeps track of indirect jumps and try to predict next time indirect jump happens based on jump instruction's address pattern. Attacker need to figure out before hand how BTB of a particular system correlate jump instruction to jump address. Then adversary mistrains the branch predictor with malicious destination, in a way that speculative execution continues at a location chosen by the adversary. In the figure 5 , we see that attacker mistrains the branch predictor in the attacker-controlled context A. In context B, which most likely to be victim's context, branch target buffer makes it's prediction on the context A and leads to speculative execution at the attacker chosen address which corresponds to the location of the spectre gadget in the victim address space. As context B run on user's privilege which may have higher privilege than that attacker, attacker effectively promotes his privilege to read higher privilege memory address.

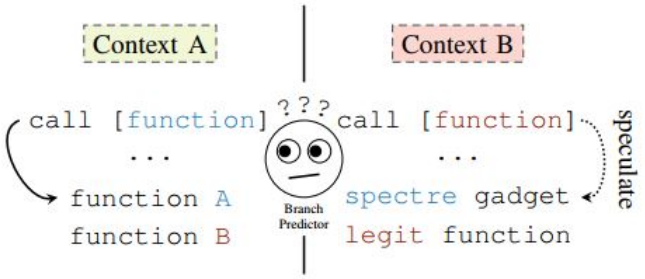

Fig. 5. The branch predictor is (mis-)trained in the attacker controlled context A. In context B, the branch predictor makes its prediction on the basis of training data from context A, leading to speculative execution at an attacker-chosen address which corresponds to the location of the Spectre gadget in the victim's address space [4]

C. Variant 3: Meltdown

In meltdown attack, attacker could read kernel memory space data by speculatively executing read operation utilizing out-of-order before exception

handling could handle unauthorized memory access[see Figure 6]. More sophisticated attack could utilize exception suppression to prevent termination of attack thread.

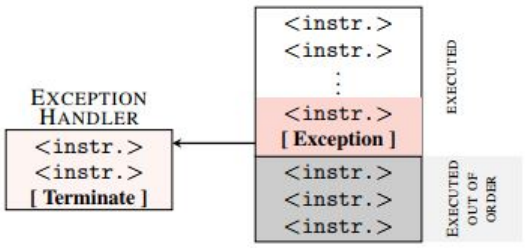

Fig. 6. If an executed instruction causes an exception, diverting the control flow to an exception handler, the subsequent instruction must not be executed. Due to OOO execution, the subsequent instructions may already have been partially executed, but not retired. However, architectural effects of the execution are discarded. [2]

Listing 2 shows a toy example of core part of a meltdown attack. This portion of the code use out-of-order execution to leak kernel information which attacker could retrieve using cache side channel attacks. In line 5, byte value located at the target kernel address, stored in the `RCX` register, is loaded into the least significant byte of the `RAX` register represented by `AL`. Move instruction is fetched by the core, decoded into micro-ops, allocated and sent to the reorder buffer. Architectural register `RAX` and `RCX` are mapped to underlying physical register enabling Out-of-order execution. To utilize the pipeline, rest of the instructions from line 6 to 8 are already decoded and allocated. By the time `MOV` instruction is retired, rest of the instructions were already executed due to out of order execution and waiting in the reservation station. As this `MOV` instruction is trying to read kernel data, when it retires, the exception is registered and the pipeline is flushed to eliminate all results. But if line 8 already completed it's execution before `MOV` is retired, it already created microarchitectural footprint using `rax` value which contains kernel address space data. Now attacker could mount a predefined side channel attack like Flush+Reload [7].

As triggered exception will terminate the execution, this portion of the code must run in separate thread if exception suppression is not used.

```
1   ; rcx=kernel addresss
2   ; rbx=probe array
3   xor rax, rax
4   retry:
5   mov al, byte [rcx]
6   shl rax, 0xc
7   jz retry
8   mov rbx, qword [rbx + rax]
```

Listing 2. Core code of Meltdown [2]

D. Variant 4: Predictive Store Forwarding

This is a variant 4 vulnerability that is recently found in AMD processor [8] that utilize *predictive store forwarding(PSF)* techniques (also known as Store-to-Load Forwarding) to manipulate victim process to forward secret data in attacker targeted load that leads to microarchitectural footprint of that secret data.

```
1   void fn(int idx){
2       unsigned char v;
3       idx_array[0]=4096;
4       v=array[idx_array[idx]*(idx)];
5   }
```

Listing 3. Example code for PSF where `idx_array[0]` will be forwarded speculatively to read out of bound value

If we follow the code block of listing 3 with `idx=0`, we could see that `idx_array[idx]*(idx)` becomes zero. So `array[0]` will be accessed which is a perfectly valid execution considering `array` is defined propoerly. Let's assume `idx_array` with default value of zero. If we use `idx=1`, that leads to executing `v=array[1]*1` which itself is also a valid execution as `array[1]=0`. But, if we call function `fn` multiple times with `idx=0`, PSF will predict that value 4096 from line 3 most likely to be used in line 4. So when attacker call function `fn` with `idx=1`, until `idx_array[1]` value is resolved, processor will continue execution with value 4096 which will leads to reading out of bound memory if `array` length is less than 4096. By varying value of `idx_array[0]` attacker could read unauthorized memory data speculatively. This speculative read will leave maliciously intended microarchitectural footprint that attacker could use to leak data.

Outside this four major variant, there are some version of the attack that does not fall into any

major variant which due to space constraint, are not explained here [9]–[12].

IV. Deployed Mitigations

As Out-of-Order execution and speculative execution are providing use performance gain for decades, majority of the existing CPUs from Personal computer to server to SoC in our mobile, all became vulnerable to this threat. So, most of the major tech companies were quick to deploy patches to mitigate this threat even at the cost of performance gain. One of the major solution is `lfence` released by Intel [13] that will block Out-of-order execution until values of preceding instructions are resolved. AMD also recommends to use `lfence` [14]. The safes but slowest approach to protect conditional branches would be to add such an instruction on the two outcomes of every conditional branches. Microsoft also released update for C compiler [15] that has switch *Qspectre* that will automatically place `lfence` in the vulnerable portion of the code. Intel and AMD also deployed microarchitectural update [13], [14] where they let programmer enable Indirect Branch Restricted Speculation (IBRS), Single Thread Indirect Branch Prediction (STIBP) & Indirect Branch Predictor Barrier (IBPB) to prevent branch poisoning. Google suggest an alternative approach called *retpoline* to mitigate branch poisoning where indirect branch will be replaced with return instructions [16].There are also some application targeted mitigation that protect secret data when classification of secret data is not ambiguous. Google Chrome browser have provided patch to execute each web site in a separate process so that one website could not read other websites data [17]. Unfortunately, even though it may prevent spectre variant one attack, but other variants that are able to break isolation could exploit speculative vulnerabilities. In order to mitigate Speculative Store Bypass, intel let programmer to set *Speculative Store Bypass Disable(SSBD)* bit [13]. Programmer can disable speculative store bypass on a processor by setting `IA32_SPEC_CTRL.SSBD` to 1. To prevent predictive store forwarding vulnerability, AMD suggested to utilize *Predictive Store Forwarding Disable(PSFD)* or *Speculative Store Bypass Disable(SSBD)* that disable predictive store forwarding [8]. Vulnerability related to Meltdown associated with how virtual address is mapped between user space and kernel space. It was tested that KAISER patch by Gruss [18] implements strong isolation between kernel and user space resulting mitigation of meltdown as a byproduct. It is recommended to active KAISER in linux patch by default.

V. Proposed Solutions

As current solutions are either not robust or incurs significant performance penalty, researchers are trying to find more efficient and robust solutions. We could classify this kind of solution in two basis. We could classify based on at what point protections are placed. One kind of solutions target speculative execution itself [13], [19] and prevent out of order execution until speculation is resolved. Some solution propose to create different isolated microarchitectural space to process speculative data and are inaccessible to the programmer [20]. There is also suggestion to prevent data load in conditional block [21], [22] but their effectiveness only proven in variant 1. Another solutions suggest to execute speculative read but prevent later dependent instructions [23]. Another suggestion was to prevent reading secret data speculatively [19], [24] but shortcoming is what data to be considered secret is purely subjective.There are also some work to detect which block to protect [25], [26] to reduce performance penalties.

We will talk more detail about another classification based on how to implement the protection as it is more relevant to the complexity of deploying the protection itself.

A. Hardware Based Solutions

`lfence` is too restrictive to prevent speculative vulnerability that leads to major performance loss. Context-Sensitive Fencing [19] propose a hardware based solution that will monitor the activity or the system, mostly cache hit-miss and relevant properties to predict probable spectre type attack pattern in the system and actively place or remove fence for optimized performance. Non-speculative Data Access (NDA) [23] suggest to execute speculative instruction but do not broadcast it's execution completion. This will prevent issuing dependent instruction that attacker have to use to create microarchitectural footprint based on secret

data. More costly implementation proposal is to implement InvisiSpec [20] that will create isolated microarchitectural area to cache speculatively read data. As it will be isolated from rest of the system and programmer will have zero access, attacker could not read microarchitecture footprint to leak data. There is also machine learning based proposal where ML hardware implementation is proposed to detect real time potential spectre attack and active protection mechanism [26].

B. Software Based Solutions

Software based solutions mainly target ways to detect potential vulnerable area to increase performance or potential secret data to deploy protection itself. SpecFuzz [25] is one of such proposal that suggest specialized fuzzing technique to detect potential vulnerable code block that needs protection. Speculative Load Hardening [22] suggests to use data dependency to ensure that until speculation is not resolved, no speculatively read data is used in later instructions. YSNB [21] propose different variant of dependency mechanism to deploy same protection. Both cases, it will prevent attacker to create desired microarchitectural footprint to leak data while out-of-order execution will not be blocked. It provides less performance penalties than `lfench` but only tested for variant one vulnerabilities.

C. Hardware-Software Co-design Solution

Hardware-Software co-designed proposals are more complex to implement. SpectreGuard [24] propose to implement non-speculative bit in memory hardware and utilize OS to mark secure data by updating non-speculative bits. Architecture will not read non-speculative marked data speculatively. Context [19] took much more complex approach. In this case, programmer only define variables that they want to protect. Compiler will lump them together in same memory space, OS will assign them in a non-speculative page table and hardware will be designed to not to read from non-speculative page table. This will make programmer life more easy compared to previous solution but still both case, main issue remains the same,"What data to be considered secret?"

VI. Discussion

Spectre arises due to hardware design flaw. This flaw was hidden in our existing system for decades. As this is inherited in our hardware, it is always a possibility that there will be new ways to exploit this vulnerability. Speculative execution became integral part of our system that we could not consider totally giving up on this due to significant performance penalty. For the time being, it seems that mitigating the risk is only way forward. But, this spectre vulnerability changed our perspective of how we see research in the context of performance and security. Fixing this problem permanently will take significant time and monetary penalty. It seems like performance and security will always be at odd to each other and we have to find a proper balance between them. This leaves the question in front of us,"How much security we are willing to give up for the sake of performance?".

VII. Conclusion

This work provides a small survey on spectre vulnerability to give the reader a basic idea about the threat we are facing. This paper first gives some background knowledge on some performance feature of computer architecture that are essential to understand spectre type vulnerability. Later, it explains spectre attack and how four type of spectre vulnerability works. Later this talk about what are the mitigation mechanism that are already deployed and different proposal of mitigation that researchers have brought forward.

References

[1] A. Fog, "The microarchitecture of intel, amd and via cpus." http://www.agner.org/optimize/microarchitecture.pdf, 2017. [Online].

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, and D. Genkin, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, pp. 973–990, 2018.

[3] Intel, ""intel 64 and ia-32 architectures optimization reference manual"," 2016.

[4] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, and T. Prescher, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1–19, IEEE, 2019.

[5] S. Deng, W. Xiong, and J. Szefer, "A benchmark suite for evaluating caches' vulnerability to timing attacks," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 683–697, 2020.

[6] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv preprint arXiv:1807.03757*, 2018.

[7] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pp. 719–732, 2014.

[8] AMD, "security-analysis-predictive-store-forwarding.pdf," 2021.

[9] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *European Symposium on Research in Computer Security*, pp. 279–299, Springer, 2019.

[10] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 991–1008, 2018.

[11] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution," 2018.

[12] J. Stecklina and T. Prescher, "Lazyfp: Leaking fpu register state using microarchitectural side-channels," *arXiv preprint arXiv:1806.07480*, 2018.

[13] Intel, "Intel analysis of speculative execution side channels." https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, 2018. [Online].

[14] AMD, "Software techniques for managing speculation on amd processors." https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf, 2020. [Online].

[15] A. Pardoe, "Spectre mitigations in msvc." https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/, 2018. [Online].

[16] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," *URL https://support. google. com/faqs/answer/7625886*, 2018.

[17] T. C. Project, "Site isolation." https://www.chromium.org/Home/chromium-security/site-isolation. [Online].

[18] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "Kaslr is dead: long live kaslr," in *International Symposium on Engineering Secure Software and Systems*, pp. 161–176, Springer, 2017.

[19] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "Context: A generic approach for mitigating spectre," in *Proc. Network and Distributed System Security Symposium. https://doi. org/10.14722/ndss*, vol. 10, 2020.

[20] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," 2018.

[21] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer, "You shall not bypass: Employing data dependencies to prevent bounds check bypass," *arXiv preprint arXiv:1805.08506*, 2018.

[22] C. Carruth, "Speculative load hardening," 2018.

[23] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "Nda: Preventing speculative execution attacks at their source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 572–586, 2019.

[24] J. Fustos, F. Farshchi, and H. Yun, "Spectreguard: An efficient data-centric defense mechanism against spectre attacks," 2019.

[25] 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.

[26] S. Mirbagher-Ajorpaz, G. Pokam, E. Mohammadian-Koruyeh, E. Garza, N. Abu-Ghazaleh, and D. A. Jiménez, "Perspectron: Detecting invariant footprints of microarchitectural attacks with perceptron," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1124–1137, IEEE, 2020.