Fix Spectre in Hardware!
Why and How

M. Anton Ertl[1]

TU Wien

Abstract. Spectre can be fixed in hardware by treating speculative microarchitectural state in the same way as speculative architectural state: On mis-speculation throw away all the speculatively-performed changes. The resource-contention side channel needs to be closed, too. This position paper also explains how Spectre works, why software mitigations are not sufficient.

1 Introduction

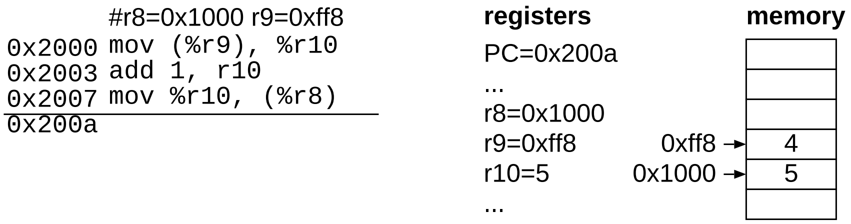
Spectre [SSLG18] is a hardware vulnerability that has been reported to hardware manufacturers such as AMD and Intel in June 2017, and to the general public on January 3, 2018. Unlike Meltdown, which has been published at the same time and has been fixed in hardware relatively quickly[1] (or, in the case of AMD, not built into the hardware from the start), even the latest CPUs with speculative execution from all manufacturers are vulnerable to Spectre, and hardware manufacturers leave it to software to "mitigate" these vulnerabilities.

New Spectre variations that bypass existing mitigations are discovered regularly, e.g., the recent discoveries of Intel's DownFall [Mog23] and AMD's Inception [TWR23] vulnerabilities. Intel lists[2] 6 "transient execution attacks" published in 2018–2021, and, as of this writing (August 2023), 5 published in 2022-2023 that require software mitigations (sometimes with hardware support) even on Intel's most recent Sapphire Rapids server CPU. This includes the original two Spectre variants (v1 and v2) reported to Intel in June 2017.

In this position paper I present a general approach to fix Spectre in hardware (Section 9) that would fix not only Spectre v1 and v2, but also, e.g., the recently-discovered Downfall and Inception vulnerabilities. In order to make this work understandable to a wide audience, Section 2 explains architecture and microarchitecture, Section 3 side channels, Section 4 speculative execution, Section 5 Spectre and Section 6 its relevance. Section 7 argues why we should not seek the solution to the problem in software mitigations. One possible hardware fix for Spectre is to eliminate speculative execution, but the performance impact is unacceptably big (Section 8). Instead, a better fix is to eliminate the side

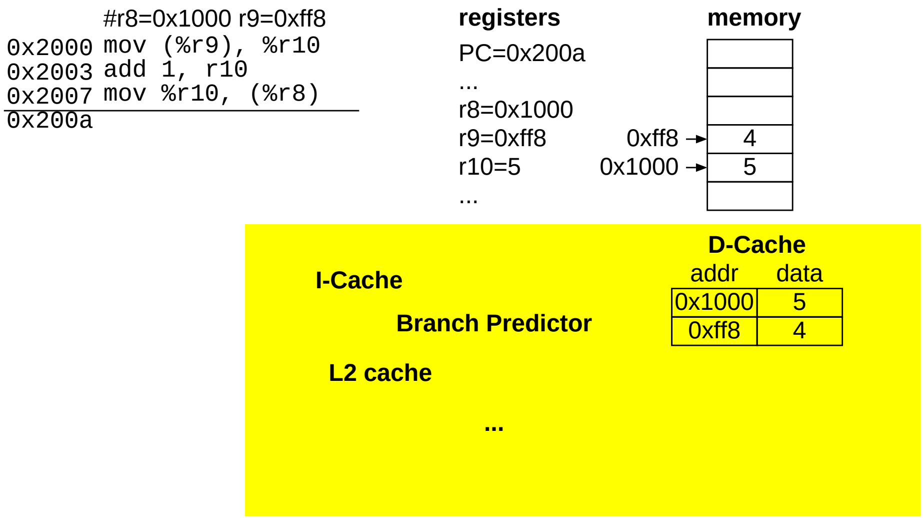
[1] https://www.anandtech.com/show/13450/intels-new-core-and-xeon-w-processors-fixes-for-spectre-meltdown

[2] https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html



```
               #r8=0x1000 r9=0xff8
0x2000 mov (%r9), %r10
0x2003 add 1, r10
0x2007 mov %r10, (%r8)
0x200a
```

Fig. 1. Architectural state: register and memory contents; this example shows the architectural state right after the instrucion at `0x2007`



```
               #r8=0x1000 r9=0xff8
0x2000 mov (%r9), %r10
0x2003 add 1, r10
0x2007 mov %r10, (%r8)
0x200a
```

addr	data
0x1000	5
0xff8	4

Fig. 2. Microarchitecture (yellow background) is normally invisible to software, apart from its performance effects

channels back from the speculative world into the committed world (Section 9). Section 10 discusses the costs of this fix. Finally, Section 11 is a call to action for computer buyers, researchers and CPU manufacturers.

2 What is architecture and microarchitecture?

The architecture (aka instruction-set architecture, ISA) is the interface between the hardware and the software. Software sees main memory and registers, and instructions that work on them (see Fig. 1).

On the hardware side of this interface the highest design level is called microarchitecture. Microarchitecture is generally not visible in the functionality presented to the software, only through the performance. I.e., instructions generally deal only with architectural features such as memory and registers, not with microarchitectural features such as caches.[3]

E.g., the cache is a microarchitectural feature, and the CPU works functionally in the same way with the cache as without it (or with caches with different sizes); the only difference is that CPUs with caches run faster. While an access to main memory takes several hundred cycles on a modern general-purpose CPU, accessing the level-1 (L1) data (D) cache costs 3–5 cycles. However, the (L1) D-cache is much smaller (32-128KB on recent CPU cores), and contains only recently-accessed data.

Speculative execution is another microarchitectural feature and is discussed in Section 4.

3 What are side channels?

A side channel (aka covert channel) reveals data not directly by letting attacker read the secret data, but through ancillary properties of data processing.

E.g., if the run-time a program takes depends on the secret, an attacker can often use this fact to extract the secret (this kind of attack is known as a timing attack). E.g., a program could contain an `if`-statement where the condition depends on the secret, and the run-time of the two branches differs. For program code that deals with the dearest secrets (encryption keys and passwords), avoiding secret-dependent branches has long been best practice.

More generally, the best practice has been to write code that runs in constant time with respect to the secret.

The timing of memory accesses depends on the input address, thanks to caches. Caches provide such a big performance boost that we prefer to keep them and deal with the security implications in some other way rather than use CPUs without caches.

One case where memory access timing has played a role is AES encryption. It has been designed in a way that is hard to implement without loads from an address that depends on the secret key. While that dependence is quite convoluted, Bernstein has found a way to use the timing variation due to loads in such AES implementations to extract the key [Ber05].

3.1 Defending against side-channel attacks

The defense against side-channel attacks first requires realizing that there is a side-channel, and then taking measures that eliminate the leakage of secret information through that side channel.

[3] There are a few cases where microarchitectural features are managed by software, and there are instructions for that, e.g., instructions for fetching data into caches (prefetch), instruction-cache management, or for draining the pipeline to ensure strictly in-order execution, but these instructions are not relevant for the rest of this paper.

As mentioned above, for timing side channels this has usually been done by writing the pieces of code that deal with the dearest secrets as constant-time code. These pieces of code tend to be miniscule (hundreds of lines) compared to the huge amounts of code (millions of lines) for complete programs like a web browser or an operating system.

While this makes the defense sound like being the responsibility of the software people alone (and this perception may have contributed to the lack of efforts on hardware fixes for Spectre), the software people cannot do it without support from hardware manufacturers.

In order to write constant-time code, the programmer needs guarantees that the timing of the used instructions does not depend on the input. Such guarantees have been historically hard to come by (and were only specified for specific implementations rather than the architecture), but recently Intel has guaranteed the input-independence of a subset of instructions for all of its implementations.[4]

In the AES case, the hardware manufacturers helped, not by making load timing address-independent (which would be impractical, as mentioned above), but by providing instructions that perform the problematic steps of AES in constant time without needing loads.

The discipline of writing constant-time code is used only for cryptographic and password-handling code, because it requires additional effort and specialized competencies, because it often results in slower run-time, but also because it is too limiting and impractical for implementing the requirements of most code. E.g., while the contents of spreadsheets of big companies and intelligence agencies may be considered by their users to be at least as secret as the encryption keys of ordinary citizens, to my knowledge nobody has tried to write a spreadsheet program with content-independent timing.

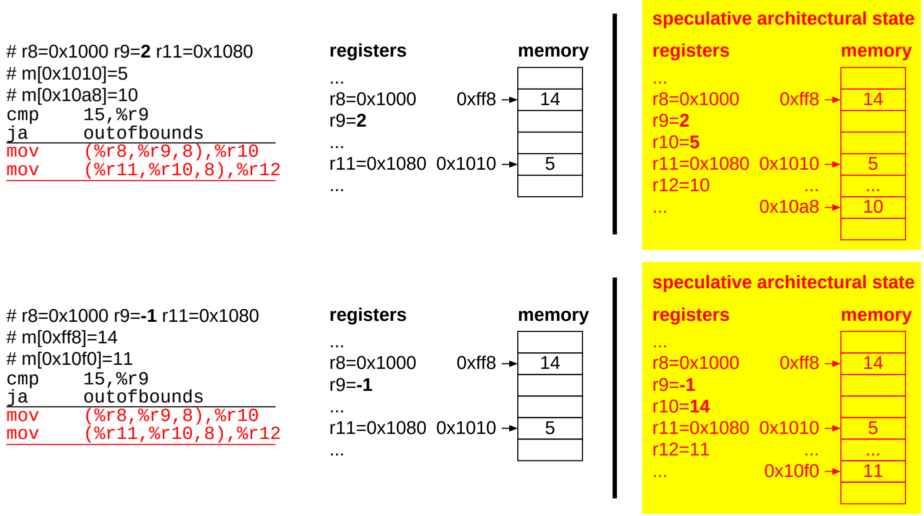
4 What is speculative execution?

Most modern general-purpose CPU core use speculative execution, a microarchitectural technique that works as follows:

The core's branch predictor predicts a likely future execution path and then executes (but does not commit) instructions on that path. The catch is that the prediction may turn out to be incorrect. In that case the architectural state (registers and memory) must not be changed in the way indicated by the misprediction prediction. If the speculation turns out to be correct, the changes can be committed (see Fig. 3).

Modern CPUs with speculative execution do this by conceptually dividing the core into a speculative part, which contains architectural results-to-be of unconfirmed speculative execution, and a committed part which contains the actual architectural state at the current architectural program counter (PC). So when the core architecturally processes an instruction (by committing (aka retiring)

[4] https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html



```
# r8=0x1000 r9=2 r11=0x1080
# m[0x1010]=5
# m[0x10a8]=10
cmp    15,%r9
ja     outofbounds
mov    (%r8,%r9,8),%r10
mov    (%r11,%r10,8),%r12
```

```
# r8=0x1000 r9=-1 r11=0x1080
# m[0xff8]=14
# m[0x10f0]=11
cmp    15,%r9
ja     outofbounds
mov    (%r8,%r9,8),%r10
mov    (%r11,%r10,8),%r12
```

Fig. 3. Two examples of speculative execution, in both cases the `ja` instruction is predicted as being not taken. Above: The prediction is correct, and the speculative architectural state is eventually committed. Below: The prediction is incorrect, and the speculative architectural state is squashed.

it in the reorder buffer), that instruction has often been speculatively executed some time earlier, and its result is lying around, waiting to be committed; and the commit takes this result and turns it into committed architectural state.

However, when a branch turns out to be mispredicted, and this branch is committed, all the speculative results after the branch (i.e., on the wrong path) are thrown away, and the processor starts executing on the correct path.

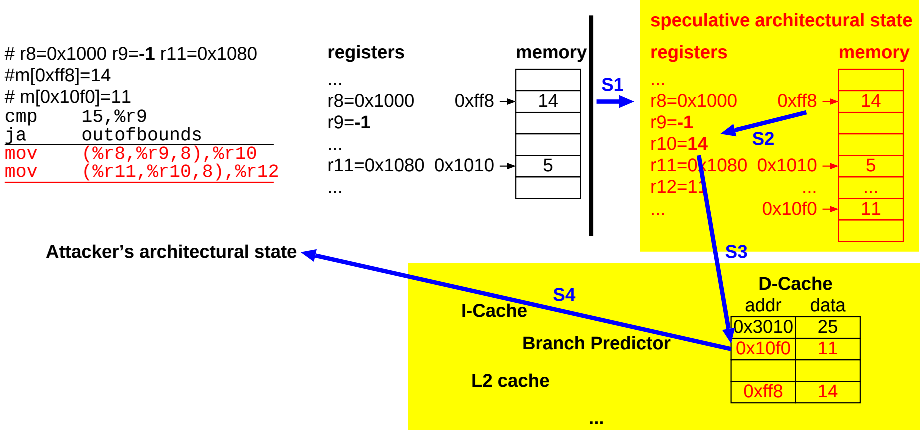
Note that this speculative execution not only contains register updates, but also stores to memory, possibly including several stores to the same memory location, and (speculative) loads from that location in between.

There have been many speculative-execution implementations of various architectures since the 1990s, and almost[5] all of them implemented the handling of architectural state correctly, both for correctly predicted branches and for mispredictions, for various kinds of registers, and for memory.

5 What is Spectre?

For microarchitectural state, e.g., the contents of the cache, existing processor cores do not discern between speculative and committed changes. After all, the

[5] The recently-published Zenbleed bug in AMD's Zen2 core (https://lock.cmpxchg8b.com/zenbleed.html) is the exception that proves the rule.



```
# r8=0x1000 r9=-1 r11=0x1080
#m[0xff8]=14
# m[0x10f0]=11
cmp    15,%r9
ja     outofbounds
mov    (%r8,%r9,8),%r10
mov    (%r11,%r10,8),%r12
```

addr	data
0x3010	25
0x10f0	11
0xff8	14

Fig. 4. A Spectre v1 attack starts with a misprediction (S1), loads the secret (in `0xff8`) into speculative architectural state (S2), changes the cache state in a secret-dependent way (S3), and finally uses cache timing to extract the secret into the architectural state of the attacker (S4).

idea is that microarchitectural state is invisible anyway. If a speculative load puts a line in the D-cache (and evicts another line), this has no architectural significance, so the hardware designers have had no qualms at performing this change speculatively, without a mechanism that cancels it in case of a misprediction.

Unfortunately, this approach opens a side channel that allows to leak data from the otherwise ephemeral world of misspeculation.

Figure 4 shows an example: The `cmp` and `ja` instructions architecturally prevent an out-of-bounds access to the array in `r8`, but if the branch is mispredicted to be not-taken, the following code is speculatively executed, and it reads the address `0xff8` speculatively; by using any other index, any other 64-bit value in the address space of the process could be accessed, including, e.g., secret keys or passwords that are there for cryptographic or authentication purposes. Let us assume that the secret is in memory location `0xff8`. In itself the load of the secret value into the speculative r10 does not appear a problem, because this is still the ephemeral world of misspeculation, and it cannot get out, right?

Unfortunately, on current hardware it can get out through a cache-based side channel: The second `mov` instruction loads a value into the D-cache, and the address of this load depends on the secret. The loaded cache line replaces a line that used to be in the cache, and which cache line is replaced depends on the address. An attacker can repeatedly access a number of memory locations in order to prime the cache, and can see from the timing of the cache accesses whether a cache line has been replaced, and in this way learn something about the secret.

There are a number of steps involved in Spectre attacks:

S1 The speculation itself: In this example (which is a Spectre v1 attack) a conditional branch causes speculative execution, but there are others. E.g., Spectre V2 uses indirect branches. There are also other speculative mechanisms in modern cores, such as speculating whether a memory load is to a different or the same address as an earlier store, and this has also been used in a number of attacks.

S2 The mechanism for getting the secret data into speculative architectural state. In the example above it is the load from `a[i]`. In the recently-published Downfall vulnerability [Mog23], it's gather instructions as implemented on some Intel microarchitectures.

S3 The sending side of the side channel for getting the data out of the misspeculation realm. In the example above it's the access to `b[j]` that channels information about `j` through the cache side channel. But other microarchitectural state can also be used, such as the power state of the AVX unit [SSLG18].

S4 The receiving side of that side channel. For a cache side-channel this consists of the attacker priming the cache and monitoring through timing which lines are replaced by the victim.

There is a lot of variation on all of these steps, leading to the stream of vulnerabilities that have been found up to now and continue to be found. For more details (and more aspects) there is a survey of the Spectre and Meltdown attacks until December 2020 [XS21]. A term that has been used to cover all these vulnerabilities and attacks against them is "transient execution vulnerabilities/attacks", but in this paper I just use "Spectre" in the same meaning as referring to all of these speculation-based side-channel attacks.

6 How relevant is Spectre?

Has Spectre been used in the wild? It's hard to know. Consider the case where attackers use Spectre to determine your password or encryption key. If they use that to decrypt your files, you may never know; maybe it was bad luck that your competitor undercut you by a narrow margin. But even if somebody does something very blatant like publish your documents on the Internet or encrypt your files and demand ransom, you usually don't know how the attacker got at your password or your encryption key.

However, a working exploit for reading normally unaccessible files on Linux has been discovered by Julien Voisin.[6] There is no proof that this exploit has been used for an actual attack, but given that it is widely available, it would be surprising if it has not.

Some people argue that Spectre is not relevant because there are many software vulnerabilities that may be used for subverting your system; so why, they argue, should an attacker bother with Spectre, which is supposedly harder to use. On the other hand, software vulnerabilities may be discovered and fixed at

[6] https://dustri.org/b/spectre-exploits-in-the-wild.html

any moment, while Spectre exists unfixed on all desktop and server hardware, and is not even mitigated against in most software. So Spectre may be more attractive to attackers than some people give it credit for.

7 Why is software mitigation not a good way to deal with Spectre?

The mitigation of non-speculative timing attacks is to write the few hundred lines of code that deals with keys and passwords in a constant-time way. Can we not just deal with Spectre in the same way?

Unfortunately, for Spectre *all* the software that has the secret in its address space can potentially be used for an attack, and consequently would have to be hardened. For a web browser or an OS kernel that is typically millions of lines of code.

As an example of a mitigation, for the Sprectre V1 example in Fig. 4, speculative load hardening has been proposed. A simple variant would change the code as follows:

```
    cmp    15,%r9
    ja     end
    mov    $0x0,%rax
    cmova  %rax, %r9
    mov    (%r8,%r9,8),%r10
    mov    (%r11,%r10,8),%r12
end:
```

Here the `cmova` hardens the following load, by setting `r9` to 0 if `r9>15`. While this condition cannot architecturally be true at that place, it can be true during misspeculation. The `cmova` instruction uses the same flags as the `ja` branch, but the mitigation assumes that `cmova` does not speculate.

In reality speculative load hardening is substantially more complicated, because it also has to also deal with possible speculation on earlier branches [ZBC$^+$23].

Software mitigations have apparently led to the impression that Spectre is under control and no hardware fix is necessary, but they have a number of problems:

7.1 Still vulnerable

It has often turned out that many mitigations do not even completely close the vulnerability for which they are designed.

One reason for that is that the mitigation defends against a too-narrow attack scenario. E.g., speculative load hardening (SLH) has been implemented in the LLVM compiler and is intended to close the Spectre V1 vulnerability presented above, but it still has some leakage; this was then improved in Strong SLH [GP19] and recently Ultimate SLH [ZBC$^+$23].

Another reason is that the mitigation relies on assumptions about microarchitectural mechanisms that turn out to be wrong; e.g., earlier Spectre V2 mitigations assumed that returns would only be predicted from the return stack, but there are some microarchitectures that use the general indirect-branch predictor to predict returns when the return stack is empty (so returns can also be used in Spectre V2 attacks).

Also, these mitigations tend to work only against a specific variant, but new variants are discovered all the time.

7.2 Performance

These mitigations cost performance, for the whole program (because with Spectre the whole program can be used to reveal the secret). E.g., Zhang et al. report a factor of around 2.5 slowdown (compared to no mitigation) for SPEC CPU 2017 (int and fp, rate and speed) [ZBC$^+$23]. I saw a slowdown (compared to using no mitigation) by a factor 2–9.5 from compiling Gforth with the fastest retpoline mitigation against Spectre V2[7].

7.3 Effort

Because the slowdowns that you get from applying compiler-based mitigations across the board are often considered to be unacceptable, there is the idea that programmers should be more selective and analyse whether each specific place in a program can actually be used by an attacker, and only harden those places, lowering the performance cost.

However, this requires a huge amount of effort, and it takes only one place in the potential attack surface that the programmer mistakenly has not hardened, and the program is still vulnerable.

And when the next vulnerability and mitigation shows up, you have to do it all again. And when the program is changed (due to, e.g., new requirements), you have to analyse more than just the changed lines.[8]

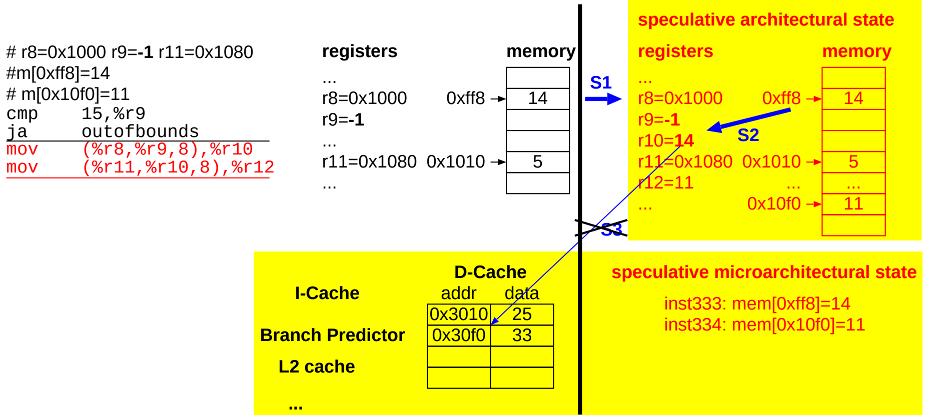
8 Why is the first idea for a fix not so great?

The first idea many people have for fixing Spectre is to eliminate speculative execution. While this certainly fixes Spectre by preventing step 1 of the exploits, the performance impact of this measure is pretty bad: E.g., the core without speculation that shows the best performance on our LaTeXbenchmark[9] is the 1800MHz Cortex-A55 on the Rock 5B single-board computer. The Cortex-A76 core (with speculative execution) running at 2275MHz on the same computer is

[7] news:<2023Jan15.105348@mips.complang.tuwien.ac.at>

[8] The idea that you do not have to reanalyse code when the requirements change resulted in the Ariane 501 explosion.

[9] https://www.complang.tuwien.ac.at/franz/latex-bench



```
# r8=0x1000 r9=-1 r11=0x1080
#m[0xff8]=14
# m[0x10f0]=11
cmp     15,%r9
ja      outofbounds
mov     (%r8,%r9,8),%r10
mov     (%r11,%r10,8),%r12
```

Fig. 5. Separating speculative microarchitectural state from committed microarchitectural state eliminates the S3 part (and therefore also S4) of a Spectre attack, as far as microarchitectural state is concerned; resource-limitation side channels also need to be addressed, see text.

3.3× as fast for this benchmark, and the 3000MHz Apple Firestorm is 7.8× as fast.

Given the performance impact, it's no surprise that we have not seen a resurgence of microarchitectures without speculation. The number of customers that would exchange this much performance for security against Spectre is small. The customers' reasoning is as follows: There are lots of vulnerabilities in the software we use, so fixing Spectre is not going to make our computers that much safer. Therefore we are not willing to sacrifice that much performance for this benefit.

9 How to fix Spectre in hardware?

The less costly and therefore better way to fix Spectre is to prevent step S3.

9.1 Side channels based on microarchitectural state

In particular, for the side channels through microarchitectural state, we can use the same approach for microarchitectural state as for architectural state: keep the speculative state separate from the committed state, and squash it when it turns out that the speculation is wrong. This goes for all microarchitectural state: D-cache, I-cache, branch predictor, TLBs, etc.

In the case of D-cache, several papers [YCS$^+$18,KKS$^+$19,AJ20] have already proposed ways of doing that, probably because the cache side channel has been the most popular one for Spectre-type attacks. But of course, the other state-based side channels need to be closed, too.

Some attempts at fixes for state-based side channels have tried to undo the changes when a speculation turns out to be false, e.g., CleanupSpec [SQ19]. However, I think that it is better to keep the speculative changes separate until commit time, for the following reasons:

- The microarchitectural state is changed, albeit only for a short time, and this is visible to potential attackers, given enough effort.
- It is harder to reason about the correctness of an undoing approach than about an approach that never speculatively changes the non-speculative state.
- Undoing approaches have been tried for architectural state [DA92], but committing approaches have won. It is likely that the same reasons will make undoing of microarchitectural state unattractive.

9.2 Side channels based on resource contention

Apart from the popular state-based side channels, another kind of side channel is contention for resources such as execution ports, functional units, or cache ports. SMoTherSpectre [BSN$^+$19] attacks another SMT thread on the same core by using execution port contention by the speculatively executing victim as a side channel. Even worse, speculative interference attacks [BSP$^+$21] use resource contention to affect the timing of an older (eventually committing) instruction in the same thread.

For the SMT side channel, a solution is to have a fixed partitioning of resources between the threads, so that no thread can use resource contention as a side channel. This means that resources that exist only once have to be time-shared. E.g., if there is an non-pipelined divider that takes 20 cycles for the division, there are fixed 20-cycle time slots for each thread, and when a thread does not have a division ready at the start of its time slot, that time slot goes unused. This fixed partitioning will cost some performance; it could be made optional, allowing the full benefit of SMT to be used in settings where the sibling threads are believed to not spy on each other.

For the same-thread problem, Behnia et al. [BSP$^+$21] describe the high-level principle: “a speculative instruction must not influence the execution of a non-speculative instruction”. And they describe two rules that ensure that:

- “No instruction ever influences the execution time of an older instruction.” They propose to achieve this by giving priority to older instructions in case of resource contention. They discuss several options how to deal with non-pipelined execution units. The slot idea above is another way to deal with that: If a thread can start using the execution unit only at the start of a slot, the priority approach works for non-pipelined units (although one might wish for better performance).
- “Any resources allocated to an instruction at the front end and the execution engine are not deallocated until the instruction becomes non-speculative”. This rule ensures that misspeculated code cannot produce timing variations by congesting the front end.

9.3 Other side channels

Another known side channel is energy consumption. In particular, Meltdown-Power [KJG$^+$23] uses speculation for S1 and S2, and then a power-based side channel for S3 and S4. However, it requires that the speculative load updates the cache, which does not happen with the fix for speculative microarchitectural state outlined above, so fixed hardware would be immune against this particular attack.

Still, one can imagine that the energy consumption of e.g., functional units working on mis-speculatively loaded data could reveal something about the data. At the moment I have no good hardware answer for that. On the other hand, the question is if such an attacks can be made practical (i.e., leak relevant amounts of data in realistic time frames).

10 How much does the fix cost?

The fixes certainly cost design complexity. Hardware architects have been remarkably good at handling the increasing complexity of modern high-performance CPUs, and I expect them to rise to the challenge of designing fixed hardware, if they are given the task.

The resulting CPU cores will require more area, for the speculative state. E.g., if we want to be able to buffer, say, 30 cache lines loaded from outer cache levels in speculative microarchitectural state, the memory for these 30 cache lines is needed, as well as the infrastructure to look up data in them and deal with snoop messages. Compared to the 224 physical ZMM registers (each with 64 bytes) in Intel's Sunny Cove core, this does not seem to add that much area; and I expect that the area for other microarchitectural features will be even smaller.

Concerning performance, the additional buffers can even help, and for Muon-Trap [AJ20] the Parsec benchmarks indeed see a speedup by a factor 1.05. However, SPEC 2006 sees a slowdown by a factor 1.04 compared to vulnerable hardware. And then there is the question of how much speed the additional area could have produced if it was invested just in performance. On the other hand, compared to applying software mitigations to all software (e.g., a factor 2.5 for defending only against Spectre v1), even the SPEC slowdown and the opportunity performance cost of the additional area are small.

One may want to compare with the more selective hardening approach that is used in, e.g., the Linux kernel. This kind of hardening has not been applied to the SPEC benchmarks, and the hardware fixes have not been measured on the benchmarks that are typically used for measuring the Linux kernel performance, so a direct comparison is not possible. Looking at Michael Larabel's results for how the kernel mitigation of just Inception[10] and the firmware mitigation of just Downfall [11] slows down applications, the slowdowns are often larger than

[10] https://www.phoronix.com/review/amd-inception-benchmarks

[11] https://www.phoronix.com/review/intel-downfall-benchmarks

what has been reported as slowdown from hardware fixes for the cache side channel. While these are numbers for different programs and mitigations/fixes for different vulnerabilities, and both comprehensive software mitigations and comprehensive hardware fixes will have higher cost, I expect that the majority of the performance cost of a hardware fix is in dealing with the cache (because of stuff like cache coherence), so I don't expect the cost of a comprehensive hardware fix to be that much higher than the cache-only approaches we have seen yet, while on the software mitigation side, every vulnerability seems to require its own mitigation, with a program-dependent performance impact, sometimes very expensive, as discussed above.

11 What should I do?

As computer **customers**, we should keep asking the CPU manufacturers when they will finally fix Spectre in hardware; we should tell them that software mitigations are not good enough.

And when one of the manufacturers comes out with a CPU with a Spectre fix, we should prefer these CPUs in our buying decisions even if they are a little slower at running unmitigated software (or software with mitigations that are unnecessary for the fixed CPUs). After all, such a CPU will be safer than an unfixed CPU when both run unmitigated software (the usual case). And such a CPU will be faster and at least as safe (probably safer) when the fixed CPU runs software without mitigations and the unfixed hardware runs software with mitigations.

When CPU manufacturers claim that they have fixed Spectre, only believe them when they explain how they did it (and only if that explanation does not have holes); don't accept hand-waving along the lines of "Differences in AMD architecture mean there is a near zero risk of exploitation"[12].

As **computer architecture researcher**, you can work at designing and evaluating mechanisms for fixing Spectre. Even if there is already some work in that direction, there is probably still some microarchitectural state or other side channels that have not been covered yet. And even for the microarchitectural state that has been covered, there are probably ways to improve on it, i.e., a solution that costs less area and/or less performance.

If your research leans more towards **theory**, you could work out a formal description of speculative side channels, and a way how computer architects could prove that they have closed these side channels. I do not know if they worked out such an approach to make sure that speculation works correctly for architectural state; it may be (usually) good enough to validate the architectural design by running test programs, but for microarchitectural state and other side channels, such an approach is needed, because the side channel does not show up in the usual architectural validation.

[12] https://web.archive.org/web/20180104014617/https://www.amd.com/en/corporate/speculative-execution

If you work at a **CPU manufacturer** (or CPU design house), you have the best opportunity to fix this problem. If the decision is up to you, go ahead and decide that you will make a Spectre-immune high-performance CPU core. If the decision is up to someone else, make a case that convinces them that the fix is worth the development and manufacturing costs by making your CPU safer than the competition, and to put a stop to the constant stream of new Spectre- and Meltdown-type vulnerabilities (and the slowdowns from firmware and software mitigations). Also, imagine what happens if your competition is first at presenting a Spectre-immune CPU.

12 Conclusion

Attacks like Spectre that extract speculative state through a side channel are different from earlier side-channel attacks in being impractical to mitigate in software: not just the small piece of code that deals with the secret, but all software in the same address space as the secret (including libraries) needs to mitigate these attacks; E.g., an automatic compiler approach against Spectre v1 alone costs a factor 2.5 in performance, and that does not defend against all Spectre attacks (e.g., not against Spectre v2). One way to reduce this cost taken in, e.g., the Linux kernel, is to try to identify places that can be attacked and only harden those; this costs a lot of programmer effort, has the potential danger of leaving a hole open, and when another attack is discovered, this effort often has to be repeated.

Therefore the right way to deal with Spectre is to fix it in hardware. For speculative microarchitectural state, it should be treated just like speculative architectural state: During speculation, keep it separate from the committed state; and when the speculation turns out to be wrong, just squash the speculative state (including speculative microarchitectural state). When the speculation is correct, turn the speculative state into commited state (e.g., during instruction commit).

In addition to state-based side channels, resource contention can also provide a side channel. This can be addressed with a fixed partitioning of resources in an SMT setting, by always prioritizing older instructions in resource conflicts, and by managing front-end resources in a specific way.

A hardware fix for Spectre costs some chip area and often also performance compared to a vulnerable core, but much less than applying a software mitigation against just Spectre v1 across the board.

References

[AJ20] Sam Ainsworth and Timothy M. Jones. MuonTrap: Preventing cross-domain Spectre-like attacks by capturing speculative state. In *International Symposium on Computer Architecture (ISCA)*, pages 132–144, 2020.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES. 2005.

[BSN+19] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *Conference on Computer and Communications Security*, 2019.

[BSP+21] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christo pher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. Speculative interference attacks: Breaking invisible speculation schemes. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, pages 1046–1060, 2021.

[DA92] Keith Diefendorff and Michael Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, pages 40–63, April 1992.

[GP19] Marco Guarnieri and Marco Patrignani. Exorcising Spectres with secure compilers. *CoRR*, abs/1910.08607, 2019.

[KJG+23] Andreas Kogler, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Collide+Power: Leaking inaccessible data with software-based power side channels. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7285–7302, Anaheim, CA, August 2023. USENIX Association.

[KKS+19] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with leakage-free speculation. In *Design Automation Conference*, 2019.

[Mog23] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *32th USENIX Security Symposium (USENIX Security 2023)*, 2023.

[SQ19] Gururaj Saileshwar and Moinuddin K. Qureshi. CleanupSpec: An undo approach to safe speculation. In *International Symposium on Microarchitecture*, page 73–86, 2019.

[SSLG18] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. *CoRR*, abs/1807.10535, 2018.

[TWR23] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. Inception: Exposing new attack surfaces with training in transient execution. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7303–7320, Anaheim, CA, August 2023. USENIX Association.

[XS21] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Computing Surveys*, 54(3), May 2021.

[YCS+18] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *International Symposium on Microarchitecture*, page 428–441, 2018.

[ZBC+23] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate SLH: Taking speculative load hardening to the next level. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7125–7142, Anaheim, CA, August 2023. USENIX Association.