Security

Chickens come home to roost
(In no predictable order)

Out of Order Execution

- Enables scheduling to hide latencies
- Independent instructions launch ahead of others
- If an earlier instruction causes an exception, they are squashed
- Since the architectural registers aren’t changed, it’s like the instructions never executed
- What could possibly go wrong?

Cache Gone Wrong

- Micro-architectures have a lot of non-ISA state
  - Caches, write buffers, MSHRs, victim buffers, branch predictors, TLBs…
- Squashing merely abandons changes to that state
- So if it’s not exposed to the ISA, why is that a problem?

Squashing is Sloppy

- Once instructions are issued, they are on the fast path
- Checks (cache tags, TLB hits, privilege levels, etc.) come later
  - They depend on lookups from other sources that take time to arrive
- Violations will eventually get caught
- If squashing happens, it makes checks “unnecessary”
  - Micro-state changes are left laying around like dirty laundry on the floor

Meltdown: Reading Kernel Memory from User Space

Moritz Lipp, et. al.

Simple Example

- Instruction that forces and exception but has a dependency
- Independent instruction that accesses memory
  - The latter will issue first
  - The former will trigger squashing of the latter
  - But the memory access may have already started cache miss handling

Getting Tricky

- Suppose the memory access uses a value outside of user space as an indirect address
- Privilege violation gets checked after TLB lookup
- Squashing will happen first, so the violation is ignored
- But the cache will load a line at an address corresponding to the value
  - That’s an invisible leakage of information from the other address space

Making the Invisible Visible

- If a cache flush was forced before the exception, it's empty…
- Except for the line loaded as a result of the squashed instruction
- Scanning cache while checking the response time finds that one fast line…
- Whose address is the value of the data in the location used as the pointer

Why This is Bad

- It’s not a software bug
- It’s a hardware design feature
- It bypasses all security
- There is no easy hardware fix
  - OoO is a deeply integrated performance enhancement
  - Fixes either reduce performance or require major changes

Practicalities

- Need to multiply data by page size to avoid prefetcher interference
- Violation may still cause termination
  - So fork the access first, or set up a signal handler if allowed to catch it
  - Suppress the exception by hiding it in a transaction, or speculative code
- Indirection by a full byte requires checking 256 lines
- Indirection dependent on a single bit only requires checking one, which is less faster and thus less susceptible to noise, although it takes more accesses

Kernel Address Space Layout Randomization (KASLR)

- Different machines map the kernel to different locations in virtual memory
- Virtual memory is big (2^{40} bytes)
- But we only need to see one responsive location to find it
- Scanning at intervals of the RAM size (e.g., 2^{33}) makes that feasible

Mitigation

- Disable OoO — not practical
- Check permissions earlier — would slow all memory access
- Memoize and revoke changes — needs more storage
- Hard partition of kernel/user memory space with an address bit
  - Issues with some kinds of virtualization (recursive, different guest OSes)
  - Limits physical memory to 512GB - shortsighted

KAISER

- Traditionally, kernel space is mapped into user space but protected, to enable faster access to OS services
- KAISER maps the kernel outside of user space
- Intel requires some kernel addresses to be in user space for e.g., interrupts need pointers to service routines
- Replace interrupt service calls with trampoline functions using a different randomization offset to indirectly call services
- Linux optimized version called Kernel Page Table Isolation (KPTI)
- May still leave vulnerabilities

Discussion

Speculation

- Given a branch prediction…
- Start executing the predicted path
- If it turns out to be a mispredict, squash the results in the pipeline
- No architectural state is changed, so no problem
- Sound familiar?

Training a Predictor

- Repeat a branch with a consistent outcome
- Trains the predictor and the branch target buffer
- Recall that predictors suffer from branch aliasing

Exploiting the Predictor

- Find a branch in the OS as part of a system call
- Find a “gadget” in the OS that does an indirect load using a register that isn’t overwritten before the branch in the system call (e.g., LDR R2, [R1], LDR R3, [R2])
- Position a branch in user space at a location that aliases the OS branch
- Train the BTB entry for that branch to jump to a location with the same user virtual address as the gadget’s address in OS space
- Load the register (e.g., R1) and make the system call
- The system call speculatively branches to the gadget, which does the loading
- Scan cache for the fast line, whose address is the value at the target location

Why this is Worse than Meltdown

- Nothing exceptional happened — it's all on the branch predictor
- The OS did the dirty work inside its own address space (wherever that is)
- There is no good way to protect against this

Why this is Not Worse

- It requires detailed knowledge of the OS
- It requires reverse engineering the branch predictor (the example was for a simple, local history predictor)
- It takes time to set up each attack — leakage is slow
- The training pattern can be detected when run in an interpreter
- But it could still be useful for high value data

Why this is Scary

- The paradigm isn’t limited to cache
- There are many variants that could leak, e.g., registers, patterns of execution, etc., which could facilitate other attacks

Mitigation

- Disable speculation — severe performance penalty
- Browsers execute every page in a separate process
- Unroll speculatively generated state
- Keep privileged level indirect branches from using prior predictions
- Flush the BTB on entry to the OS
- Use special branches that avoid prediction

Discussion