ECE 571 – Advanced Microprocessor-Based Design Lecture 21

Vince Weaver

https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

25 October 2024

Announcements

- Homework #6 Due
- Homework #7 will be posted
- Midterm Exam Next Wednesday
  - Can bring one page of notes
  - Benchmarking, skid, power/energy, energy delay, branch predictors (static), caches, virtual memory

Real World Examples

Haswell Virtual Memory

- ITLB
  - 4kB: 128 entry, 4-way, dynamic between Hyperthreads
  - 2MB/4MB: 8, fully assoc, duplicated ht
- DTLB
  - 4kB: 64-entry, 4-way, fixed partition
  - 2MB/4MB: 32 entry, 4-way
  - 1GB: 4-entry, 4-way (!?)
- STLB (second level)
  - 4kB/2MB: 1024 entry, 8-way

Computer Architecture Security

- We've made fast chips, but at what cost?
- All the tricks to hide memory latency can introduce information leaks
- This was actually known for a while, but the big names in computer architecture didn't take it seriously until 2018 when repeatable exploits were released

Side Channel Attacks

- Leak info in unexpected ways
- With hardware access can tell what chip is doing based on RF noise, temperature, power draw, etc
- Timing Attacks
  - If can run code at same time someone else's, info can leak
  - Measuring any shared resource can tell if someone else is using it (caches, branch predictors, stalls on hyperthreads, context switch times)

◦ Any way of accurate time measurement (time syscalls, hardware performance counters) lets you do this

Side Channel Mitigations

- Hardware: shielding the hardware
- Writing timing-invariant code (for encrypting, the time to encode a 0 and a 1 need to be exactly the same so someone monitoring can't tell the difference)
- Disable HW perf counters (sad), remove high-res timers
- Add random noise to some measurements

Meltdown (January 2018)

- `https://meltdownattack.com/`
- Problem: speculative execution can load data into caches even if you don't have VM permissions to do so
- Many OSes map kernel into VM address space of each process (this speeds up operating system calls a lot) Kernel often includes a full mapping of physical RAM as well

  so in theory you can read out *all* of memory
- This primarily an Intel bug, all chips with speculative

execution, dating back to Pentium Pro?

- Some very high-end ARM (Cortex A75) too as well as IBM Power? *Not* AMD though.

Cache Side-Channel Attacks

- Just loading speculative values into cache shouldn't be a problem as you can't actually read the values
- UNLESS there is some sort of side-channel
- If you can use the speculative data as an index to a memory load, you can bring yet another cache line in, the line depending on the value you shouldn't know.
- If you can determine if these lines are in cache you can know the content of the data.

Determining if a Cache line is in Cache

- Evict and Time – run and time. Then evict just one line, then run again. If ran slower, it depended on data in that line
- Prime and Probe – load cache full of your data. Wait until code of interest runs. Then see how many of cache lines got kicked out.
- Flush and Reload
  - Single cache line granularity
  - Use `cflush` or similar to kick out a cache line

- ○ Reload and time it, can tell if someone else had reloaded it in the meantime by how fast it loads

Meltdown – Toy Code

```
raise_exception()
access(probe_array[data*4096])
```

- The access should never happen, as the exception (segfault, etc) will trigger
- If exception slow enough, the access will likely be speculatively executed
- In theory the results of the access are thrown out so you cannot know it happened
- The speculative load might end up in the cache though, and you can probe to see if it did

- The *bug* is that on Intel chips you can speculatively access kernel data from userspace and it will be cached despite the permission mismatch.
- Why multiply by 4096? Spread across multiple pages so the prefetcher doesn't get in the way.

Probing the Cache

- Secret data in cache, but how can you read it out from a different process?
- Timing attack
- If you can speculatively execute something like

  ```
  if (secret_data&1) load_one_address;
  else load_few_cachelines_away_address;
  ```

  You can then probe which one of those got loaded into the cache with timing analysis, and now you know one bit of the secret
- Repeat for all the bits of the secret data

Performance

- Up to 500K/s read out

Meltdown – Issues

- Exception Handling – either a signal handler, or else forking a child to cause the exception
- Exception suppression transactional memory
- Limitation: Can get false zeros. Repeat until sure.

Workarounds

- Hardware
  - Turn off out-of-order (not possible, expensive)
  - Not allow user speculation to access kernel addresses
  - Not put data into cache until permission check completes
- Software
  - KASLR (Kernel Address Space Layout Randomization). If want arbitrarily read out kernel info need to find kernel

Turns out that with 40 bit address space and large physical memory (8GB) it doesn't take too many tries to find kernel

- ◦ KAISER (KPTI) – map kernel in separate address space.

  Large overhead on switch in/out of kernel (syscalls, context switch)

  Up to 30% on some workloads, almost none on others
- ◦ PCID (intel's ASID implementation on Westmere or newer) helps avoid complete TLB flush

Spectre Vulnerability

- Unlike Meltdown, pretty much *any* processor with speculative execution affected
- Doesn't leak info from kernel, but from one part of program to another
- Why a problem? Well if javascript can read anything in rest of browser (passwords, history, etc)
- SPECulative execution, will haunt us for ages

Spectre – Depends on Branch Predictor

- You can reliably train branch predictor to hit/miss
- You can find if something is in the cache via timing
- Find a place in program where if it branches the wrong way it accesses a value of interest
- Manipulate branch predictor so it always predicts this taken
- Then go with an invalid value, but the predictor is trained to try
- Time analysis to get results

Spectre Variant 1 – Bounds Check

```
if (x<array1_size)
    y = array2[array1[x]*256];
```

- Ideally finds this code already existing in user code
- If mispredicts the check, will speculatively access the out-of-bounds value
- Attacker controls X
- Attacker trains the branch predictor that value is true with lots of runs
- Then passes in a value that is wrong but branch is predicted the previous way.

- array1_size is not cached, so it stalls and execution goes beyond
- Probe the cache much like meltdown

Indirect Branches

- Instead of relying on user code, train up the BTB
- Doesn't have to be the same address space, just has to alias in the BTB
- On many machines only 30 or fewer bits of BTB used to index
- You can then aim the BTB to point to anywhere in memory code you want to speculatively execute

Spectre Variant 2 – Branch Target Injection

- X maliciously chosen
- Branch prediction manipulated to predict wrong
- arrays all kicked out of memory
- array1size was kicked out of RAM, so cache miss and slowly get value for RAM
- meanwhile predicts branch is good and so fetches array2[k*256]
- Eventually figures out and squashes wrong branch, but the fetch already underway into cache

Finding a gadget

- Need to find code that runs with adversarial values are in register
- Not hard, often unused values leak across function calls (if a function doesn't use them)
- Need to find way to trigger a branch in a way that acts on these as pointers.
- Then find existing indirect jump
- Train the BTB to want to jump to our gadget
- clear out cache, perform attack

Notes

- some i7 up to 188 instructions can execute speculatively between
- can be triggered from Javascript. No clflush, but can evict all of cache by reading through an array.
- branch predictors on cpus are independent?

Workarounds

- Software
  - Disable hires timers in javascript
  - Memory barriers – can halt speculation with special instructions, but have to insert them all through code where it might be an issue.
  - Kaiser/KPTI not help
  - Retpoline and IBRS, see next slides

Mitigation: New barriers

- Added by Intel with firmware update, new MSR
- IBRS – indirect branch restricted speculation flush branch predictor on entry to kernel, disable brpred on hyperthread
- STIBP – single-thread indirect branch prediction – disable brpred on sibling thread (currently they share brpred)
- IBPB – indirect br pred barrier – flush branch predictor state

Mitigation: retpoline

- Indirect branches can be used for spectre attack
- Can you (at a performance cost) change all indirect branches to disable branch prediction?
- Original indirect call

```
    jmp *%r11
```

- Replace with this:

```
  call  set_up_target ; skip ahead (ret addr on stack)
capture_spec:
  pause; lfence       ; trap to catch speculation
  jmp   capture_spec  ; can't speculate out
```

```
set_up_target:
  mov   %r11, (%rsp)   ; overwrite ret addr with dest
  ret                  ; call using ret (confuse brpred)
```

- Return trampoline
- Convert indirect branch into a `ret` in a common location, makes it hard to train branch predictor.
- Also adds a code-trap so that if code speculates past the branch it gets trapped in an infinite loop
- Downside: all indirect branches now slower retpoline.

Are you vulnerable?

- On Linux look in `/proc/cpuinfo`
- Also can look in `/sys/devices/system/cpu/vulnerabilities`

Is it worth the mitigations?

- Mitigations can really slow down some processors. How much? 10%? You'd have to check (this might make good class project)
- If you're on a trusted machine with no outside users and you don't run outside code (no web-browser/javascript) maybe turn it off?

Other security things in modern architectures?

- SGX
- Making RAPL counters less accurate “energy filtering”
- Signed firmware
- Secure boot
- etc