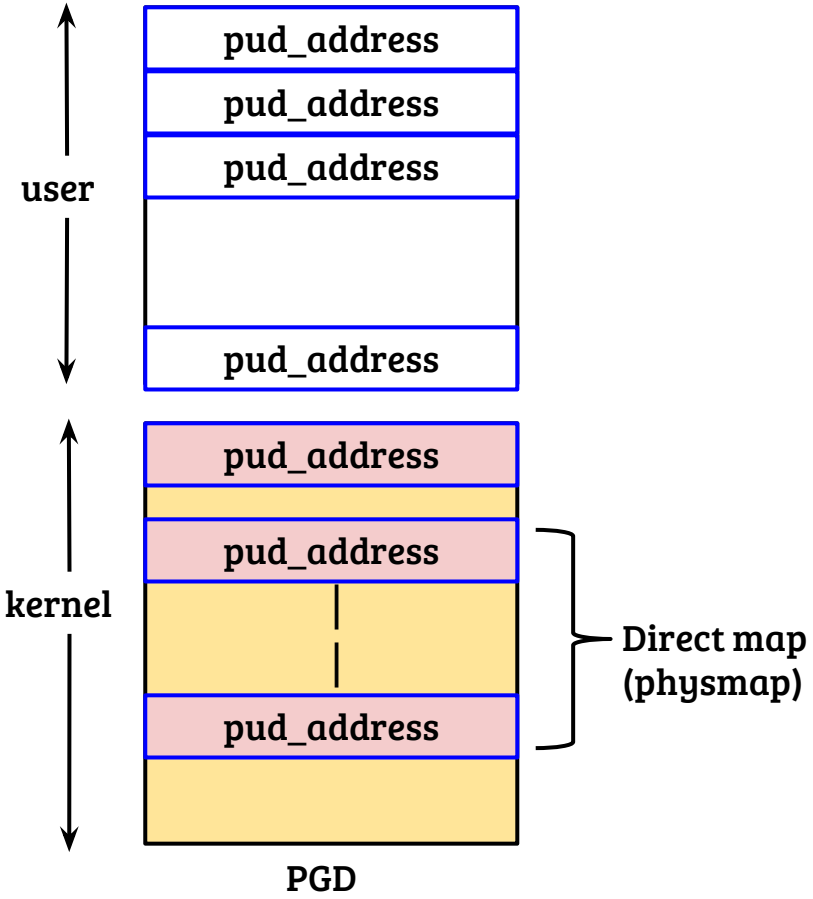
Topics in Operating Systems

Meltdown and mitigation

Debadatta Mishra, CSE, IITK

Recap: process address space in Linux



- Virtual address space is split into two parts, user VA and kernel VA
- Kernel mappings are isolated from user through **S/U** bit of page table entry
- 64 TB of kernel VA maps the complete physical memory
- Advantages: isolation + efficiency

Isolation enforcement

```
char array[256 * 4096];        //__alligned(4k);
char secret = *(char *) 0xffff888000000000;
array[secret  << 12] = 0;
```

- This program will result in an exception → Segmentation fault
- Everything seems to be under control. What is the problem then?

Information leakage through out-of-order execution

1. mov RCX, $0xFFFF888000000000;
2. mov RBX, $array;
3. mov AL, [RCX];
4. Shl RAX, $0xC;
5. mov RBX, qword [RBX + RAX];

Executed out-of-order

Exception handler

1. cmp CR2, $userend;
2. Jg raise_segv;
3. ……
4. ……..
5. raise_segv:
6. ……

- By the time the instruction in line#3 is committed (and a fault is raised), instructions in line#4 and #5 are completed out-of-order

Side-effect: access footprint

1. char array[256 * 4096]; //__alligned(4k);
2. char secret = *(char *) 0xffff888000000000;
3. **array[secret << 12] = 0;**

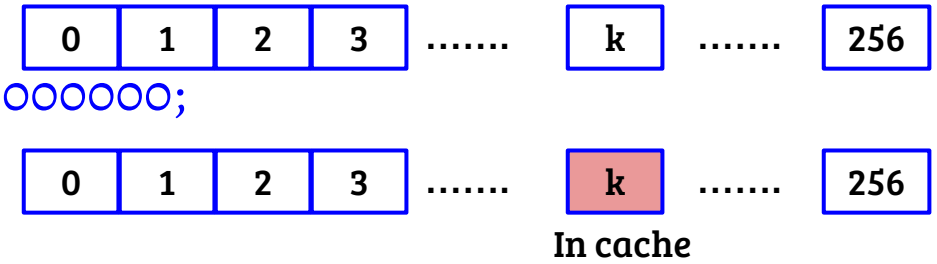
Array (before the program execution) : block 0 == {0 - 4095} etc.

Array (after out-of-order execution of #3) {assume secret = k}

Accessed

OOO vulnerability + Flush-Reload

```
1. unsigned time[256];
2. char array[256 * 4096];
3. flush_array(array);
4. char secret = *(char *) 0xffff888000000000;
5. array[secret  << 12] = 0;
6. for(i=0; i<256; ++i)
7.         access_and_time(array, time, i);
8. secret = find_index_with_min_time( time);
```



- Result: indirectly read the value of secret
- Meltdown is easy…. Some subtle points still remain

Fault handling

1. unsigned time[256];
2. char array[256 * 4096];
3. flush_array(array);
4. char secret = *(char *) 0xffff888000000000; //SEGFAULT and Terminate
5. array[secret << 12] = 0;

   ……..

- Solutions?

Fault handling

```
1. unsigned time[256];
2. char array[256 * 4096];
3. flush_array(array);
4. char secret = *(char *) 0xffff888000000000;  //SEGFAULT and Terminate
5. array[secret  << 12] = 0;

   ……..
```

- Custom signal handler
- Fork( ) based solution: Child faults and gets killed, parent extracts the secret
- Exploit H/W support for transactions: Intel TSX

Handling non-determinism

1. mov RCX, $0xFFFF888000000000;
2. mov RBX, $array;
3. mov AL, [RCX];
4. Shl RAX, $0xC;
5. mov RBX, qword [RBX + RAX];

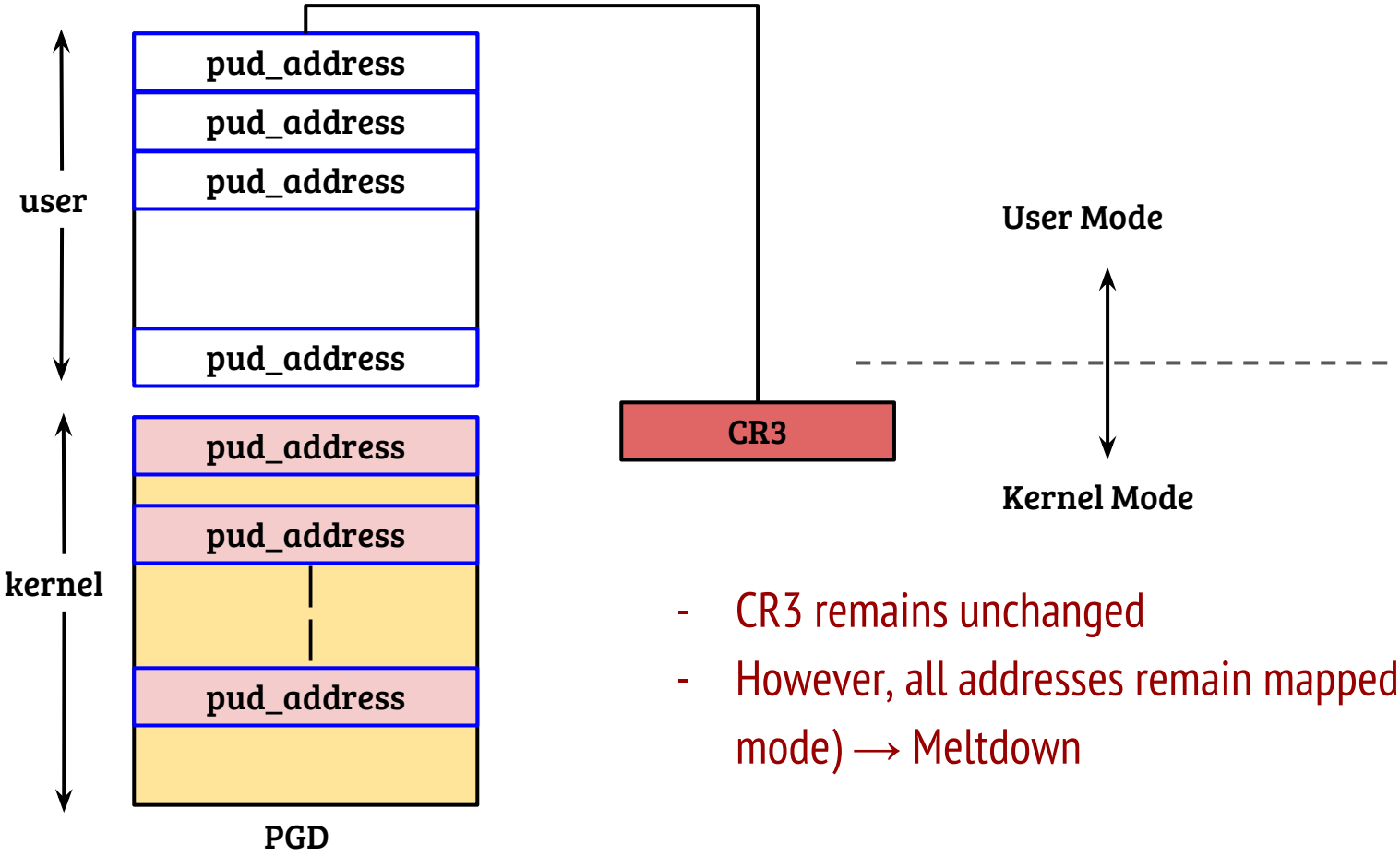
Executed out-of-order → Not always guaranteed

- If exception is raised before line #5 is executed OOO
  - Value of RAX depends on architecture, mostly 0
  - Retry N times if value of RAX == 0

Conclusion

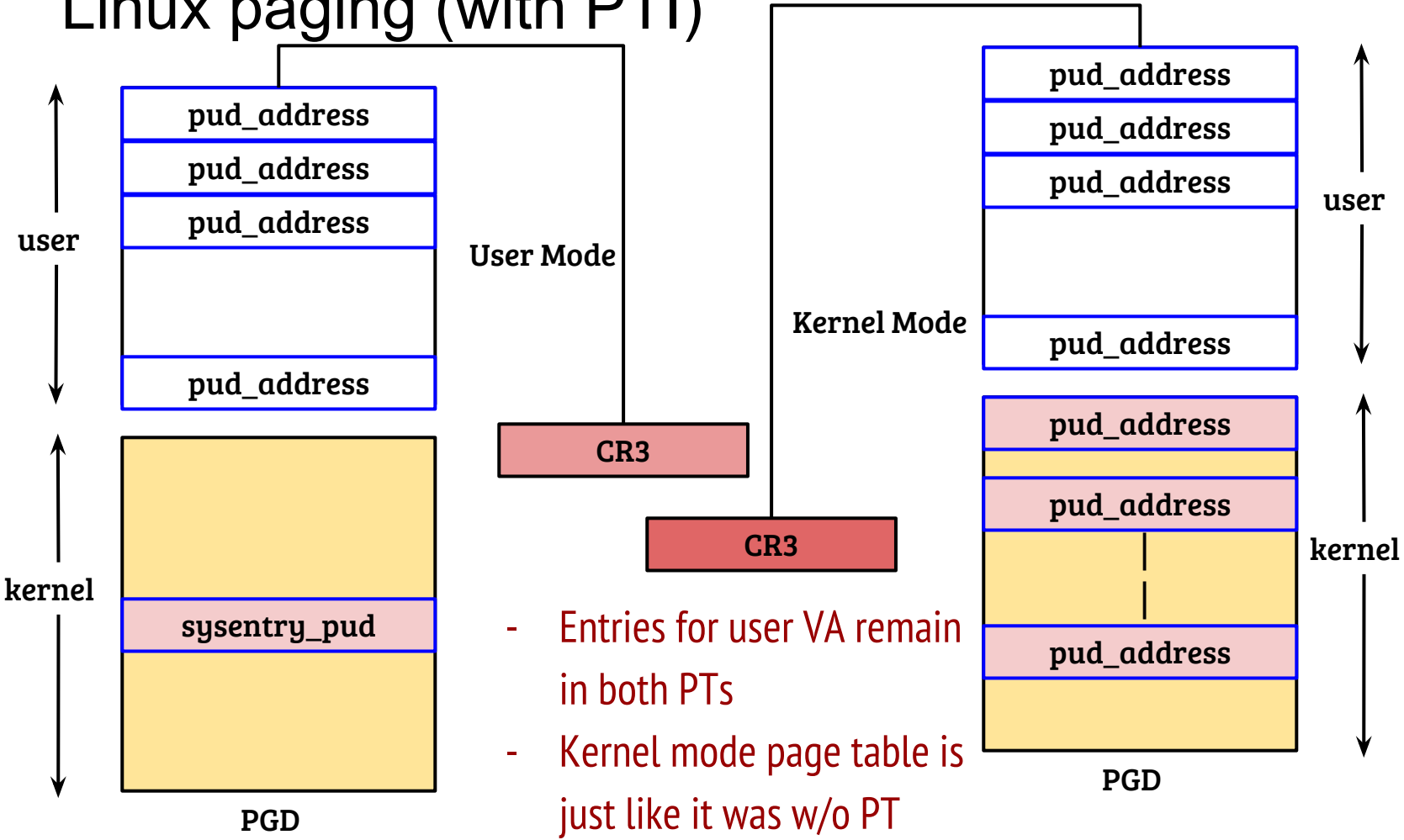
- Meltdown proven to be a powerful attack
  - Accurate and fast
  - Works in presence of traditional defence mechanisms
- Hardware fix should be easy!
- OS community (including Linux) provided software fixes quickly
- Next: Linux page table isolation (PTI, KAISER)

Linux paging (before PTI)



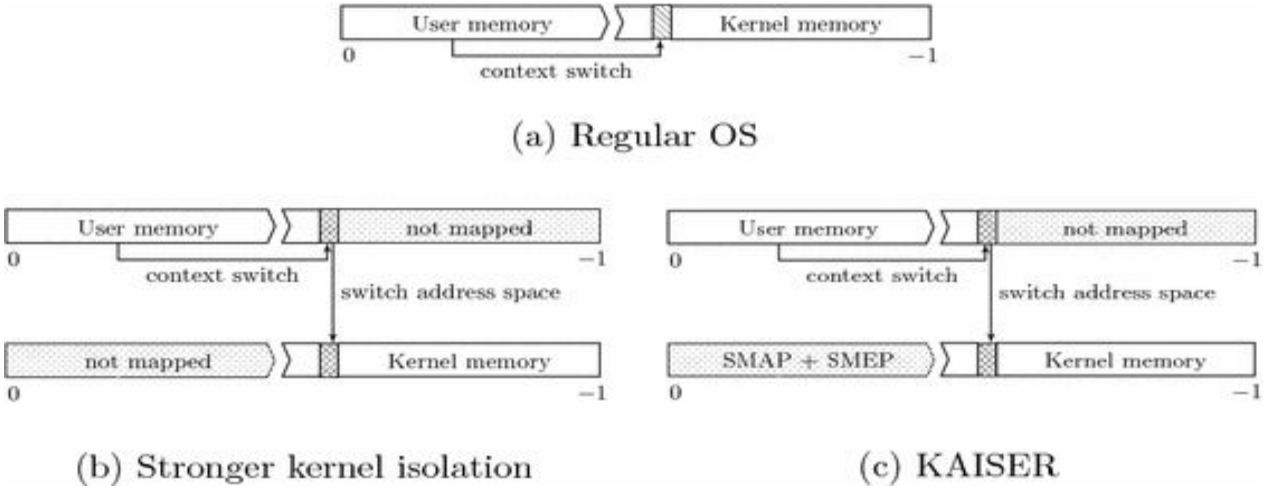
- CR3 remains unchanged
- However, all addresses remain mapped (even in user mode) ⟶ Meltdown

Linux paging (with PTI)



- Entries for user VA remain in both PTs
- Kernel mode page table is just like it was w/o PT

KAISER/PTI: Kernel entry and exit [1]



- CR3 switch overhead (~100's of cycles)
- Without ASID support ⟶ Larger overheads
- Additional kernel stack switch on entry required

1. **Image is used from the paper by Daniel Gruss et al. KASLR is Dead: Long Live KASLR**

Page tables management overheads

- Any change in user PGD should be synced with the kernel PGD (only @pgd level)
- On fork( ), both user PGD and kernel PGD should be copied
- TLB flush overheads

Context switch overhead

- Flush the user and kernel entries out of the TLB
- Increased context switch overheads due to additional TLB misses