

FENCE.T “timing fence” or SEC.FLUSH “security flush”

Gernot Heiser, Richard Newell, Andy Glew

following up on 2021-01-04, slide set
discussed by Gernot Heiser, Richard Newell and Andy Glew

Enhanced with hopefully clearer proposal and request for help

Security Risks

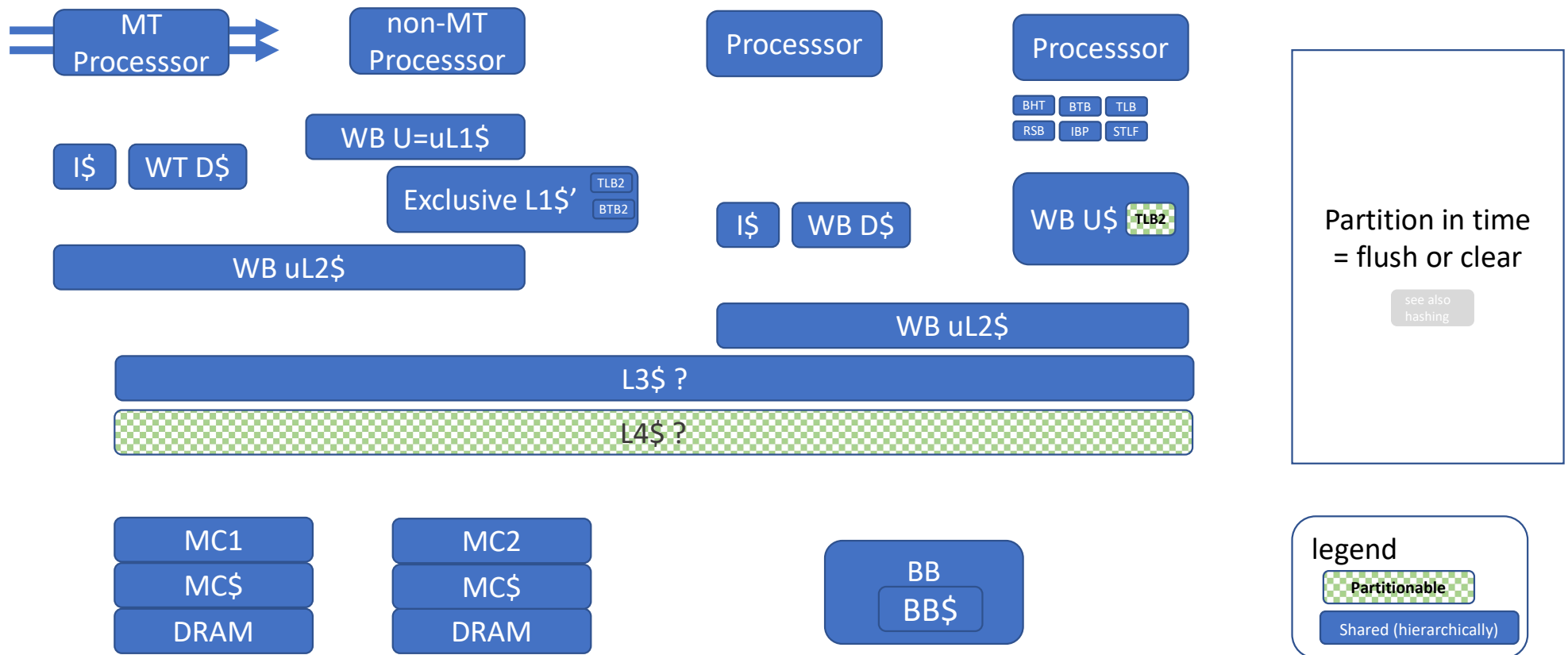
infiltration

- malware already in system
 - insider, supply channel
- buggy SW
 - binary code injection
 - code hijacking “ROP”
 - mitigation: CHERI capabilities, J extension Pointer Masking, CHERI
- speculation
 - unprivileged speculative access to privileged state
 - part of Ri5 ISA
 - nonprivileged training influencing speculation of privileged code

exfiltration


- any of your favorite microarchitecture channels
 - cache
 - branch predictor
 - TLB
 - hardware prefetchers
 - MC open bank policies
- beyond scope of this talk
 - DOS
 - ? soon in Ri5 - cache way/entry locking ?
 - Power channels
 - EMF
 - audio

Example system - partition in space or in time



seL4 Context Switch - Stealing Gernot's slide

Temporal Partitioning: Flush on Switch



Must remove any history dependence!

1. $T_0 = \text{current_time}()$
2. Switch user context
3. Flush on-core state
4. Touch all shared data needed for return
5. $\text{while } (T_0 + \text{WCET} < \text{current_time}()) ;$
6. Reprogram timer
7. return

Latency depends on prior execution!

Ensure deterministic execution

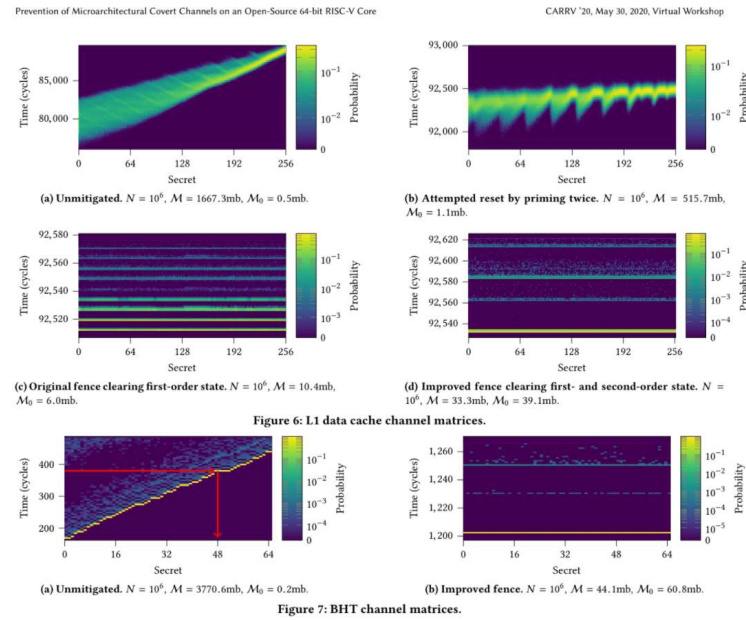
Time padding to remove dependency

Western Digital Security Workshop, Dec 20

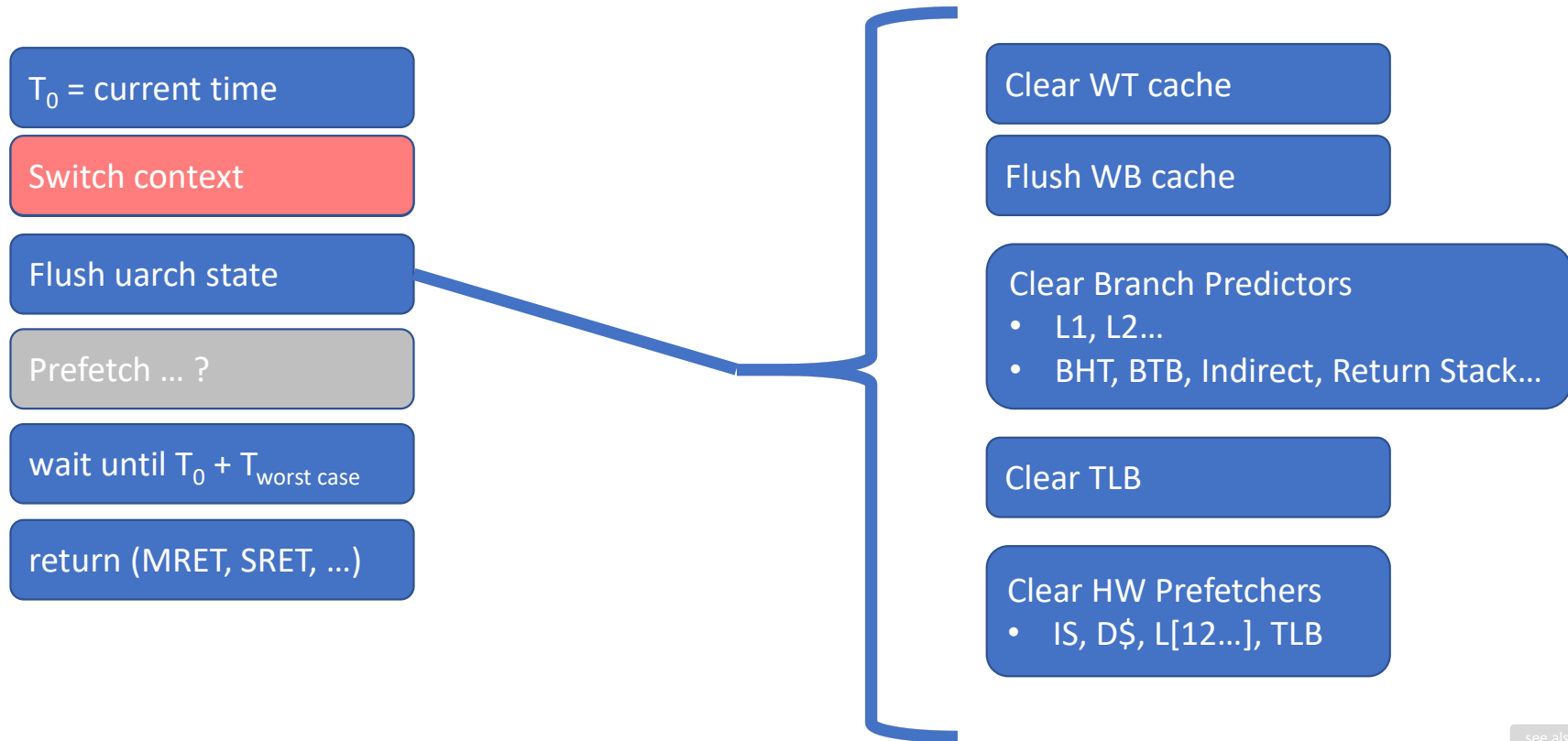
Gernot Heiser: The seL4 Microkernel

21 |

Stealing Some More – Cool Channel Graphs



Expanding – many flushes, what order?



see also

Expanding – Deterministic Timing

$T_0 = T_{\text{now}}$ current time

Switch context

Flush uarch state

Prefetch ... ?

wait until $T_0 + T_{\text{worst case}}$

return (MRET, SRET, ...)

loop until end time
• leaks information

either ☹️
re-flush affected predictors

IMHO 😊
 $T_{\text{wait}} = T_0 + T_{\text{worst case}}$
 $\text{PAUSE}(T_{\text{wait}})$

but $\text{PAUSE}(T_{\text{wait}})$ or $\text{PAUSE}(T_{\text{abs}})$
• **Rejected when Ri5 PAUSE was defined**
• Time need not be exact, \geq

Best Proposal: **PREFETCH**(t_{wait}) or PREFETCH(t_{abs})

IMHO almost a no-brainer

but the security channel may be small
and foolish people may think it unnecessary

Best Proposal: **hide FLUSH/CLEAR ugliness**

Clear WT cache

Flush WB cache

Clear Branch Predictors

- L1, L2...
- BHT, BTB, Indirect, Return Stack...

Clear TLB

Clear HW Prefetchers

- IS, D\$, L[12...], TLB

```
x1 ← 0
L: x1 ← SEC.FLUSH.type x1
   BNEZ x1
```

where Implementation determines
which predictors and caches get
enumerated for any type of security flush
e.g. “flush everything before partitioned L2”

CMOs TG already rejected “address ranges”

These are “microarchitecture ranges”
but SEC.FLUSH nevertheless returns a value
which causes FUD

Hide ugliness: KA & AW's neat idea

Clear WT cache

Flush WB cache

Clear Branch Predictors

- L1, L2...
- BHT, BTB, Indirect, Return Stack...

Clear TLB

Clear HW Prefetchers

- IS, D\$, L[12...], TLB

```
x1 ← 0
L: x1 ← SEC.FLUSH.type x1
   BNEZ x1
```

where Implementation determines which predictors and caches get enumerated for any type of security flush e.g. “flush everything before partitioned L2”

Implementations may

- Implement a line at a time
- Implement as a state machine
- Tap and emulate
 - lower overhead than per line
 - easy to extend to “full system”
- Use “instant” clears for WT caches, etc
- Use different line sizes for different caches
- Make strong/weak security guarantees
- Can be as secure as possible for Linux

Worst Proposal: Hybrid Mongrel

Clear WT cache

Flush WB cache

Clear Branch Predictors

- L1, L2...
- BHT, BTB, Indirect, Return Stack...

Clear TLB

Clear HW Prefetchers

- IS, D\$, L[12...], TLB

CPU Internal Stuff

```
SEC.FLUSH.instant
FOR j FROM 0 to I_size DO
    SEC.INVALID.I j
FOR j FROM 0 to Ln_size DO
    SEC.FLUSH.Ln j
```

CPU External Stuff

```
FOR j FROM 0 to Bus_Bridge_size DO
    SEC.FLUSH.MC j
FOR j FROM 0 to MC_size DO
    SEC.FLUSH.MC j
```

- Requires cache + MC Parameter discovery (TBD)
- Expensive to trap and emulate
- Therefore less likely to be able to handle external caches
- Less secure for Standard Linux

Middling Proposal: SBI

Clear WT cache

Flush WB cache

Clear Branch Predictors

- L1, L2...
- BHT, BTB, Indirect, Return Stack...

Clear TLB

Clear HW Prefetchers

- IS, D\$, L[12...], TLB

Implementation dependent SBI call

SBI_call # SEC_FLUSH_type

Does not add to ISA for such implementation dependent features as caches and predictors.

Always slow

see Slow Syscalls
Considered Dangerous

Impediment to SW portability

somebody has to write the SBI call

Can be as secure as possible for Linux

Conclusion and HELP !!!

- The IMHO best proposals
 - PREFETCH(t)
 - Hide Ugliness $j \leftarrow \text{SEC.FLUSH.level } j$
were already rejected by relevant Ri5 TGs
although not for security purposes
- The other proposals – Mongrel and SBI – have problems
- BKM for as secure as possible involves intermediate cleanup state
 - OK for Microkernels like seL4, not the way Linux currently does things
- Non-causal speculation and fetching
 - is a real problem cache flushes in general
 - probably not a real problem for branch predictors
 - but need a theoretical/abstract model to constrain non-causality
 - information theoretic?

Real-world microarchitecture considerations

You cannot always instantly flush an array

Prevention of Microarchitectural Covert Channels on an Open-Source 64-bit RISC-V Core

Nils Wistoff
ETH Zurich and FORTH Aachen
and HENSOLDT Cyber GmbH
Zurich, Switzerland
nwistoff@ethz.ch

Moritz Schneider
ETH Zurich
Zurich, Switzerland
moritz.schneider@inf.ethz.ch

Frank K. Gürkaynak
ETH Zurich
Zurich, Switzerland
kgf@inf.ethz.ch

Lucia Benini
ETH Zurich
Zurich, Switzerland
benini@inf.ethz.ch

Gernot Heiser
UNSW Sydney and Data61 CSIRO
Sydney, Australia
gernot@unsw.edu.au

CARRV '20, May 30, 2020, Virtual Workshop

other than the L1-D). In contrast, the temporal fence, which we have found to be highly effective against *all* channels, only adds 320 cycles (less than 30%) to the cold-cache latency. With a switch rate of no more than 1 kHz, this adds negligible cost.

The dominating contribution to the direct latency of the fence is the instruction is the **cache flush**. A **write-through** cache is flushed by clearing all valid bits. This is a constant-time operation, which could in theory be performed in a single cycle. However, in Ariane's write-through cache, the **valid bits are stored together with the tags** in sequentially accessible SRAM, allowing for an **invalidation of only one set per cycle**, and thus resulting in a latency of 256 cycles. All other state can be reset in a single cycle.

A write-back L1-D cache would be more expensive to flush, as each dirty line must be written back to the L2. Since the L2 of our platform can process up to 8 B per cycle, the theoretical latency for a write-back variant varies between 0 cycles (clean cache) and 4,096 cycles (all lines dirty). In such a case of a variable latency, the OS must pad to the worst-case latency, to prevent the flush latency becoming a covert channel [10].

Anyone with enough experience

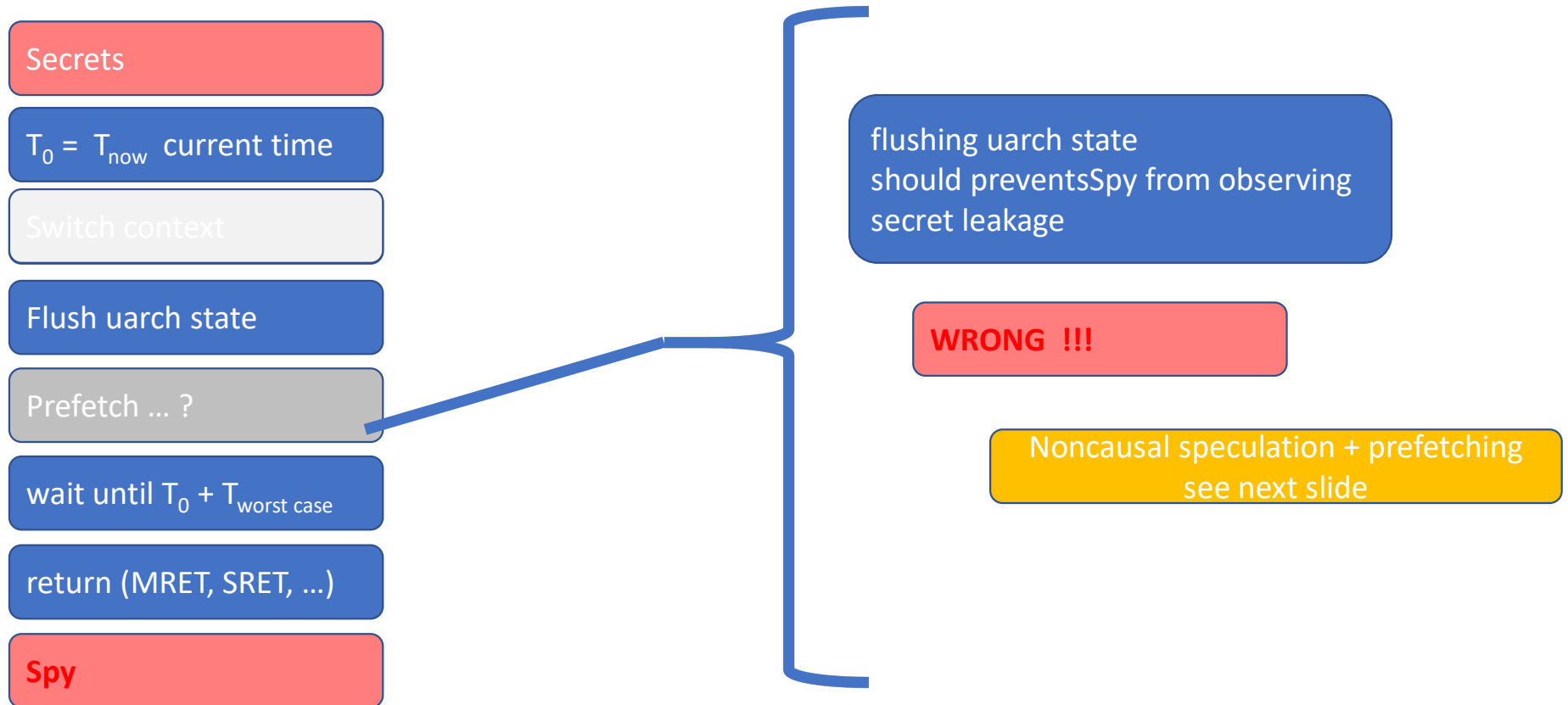
- has encountered non-WB caches (WT D\$, clean only I
- that could be instantly cleaned (in some circuit technologies)
- but which are not in some implementations
- Just recording this evidence for those who have not encountered this.

Also applies to large BTBs and TLBs and any large clean array

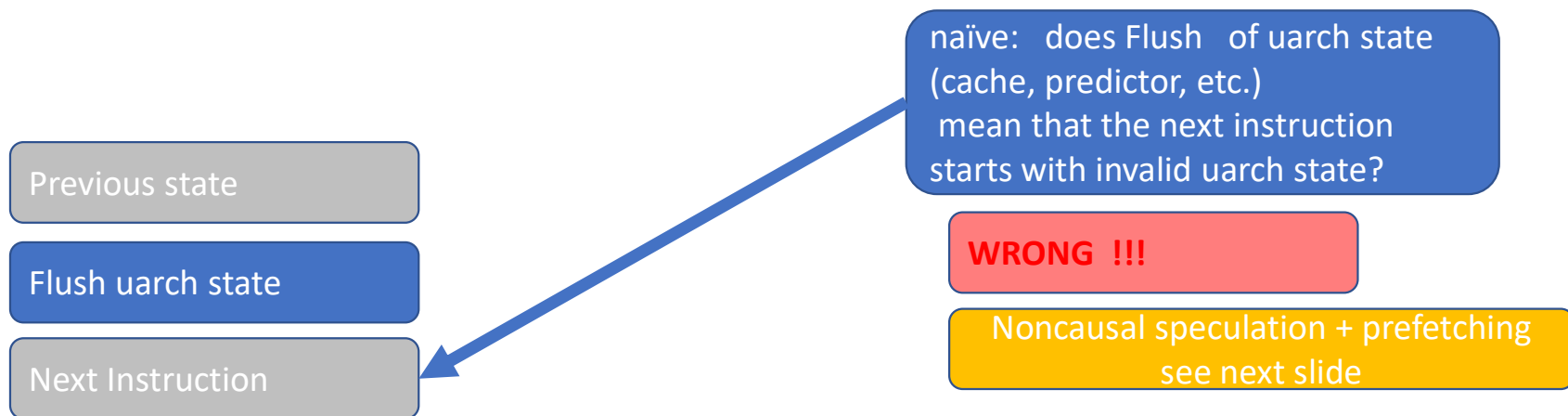
[links to here](#)

Aside: Noncausal Speculation + Prefetching

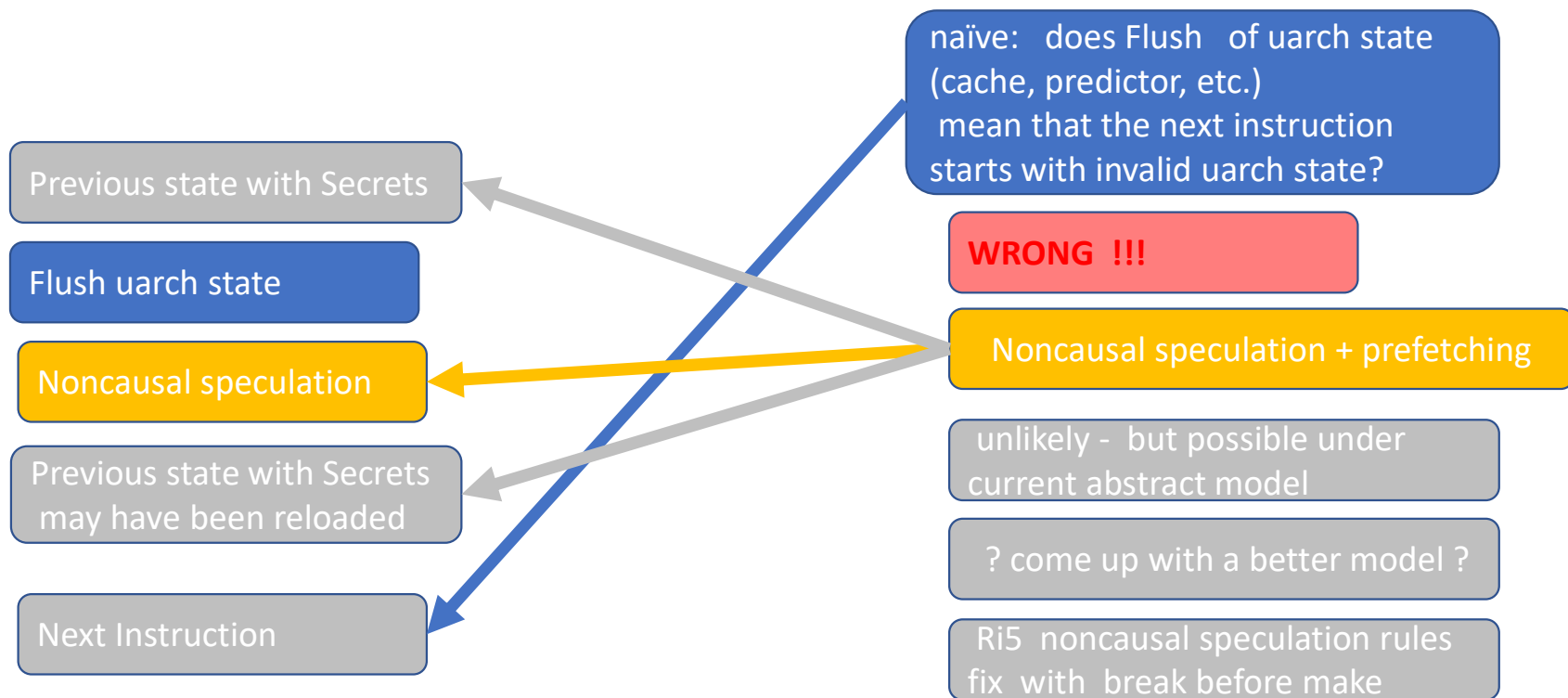
Expanding – Accessibility



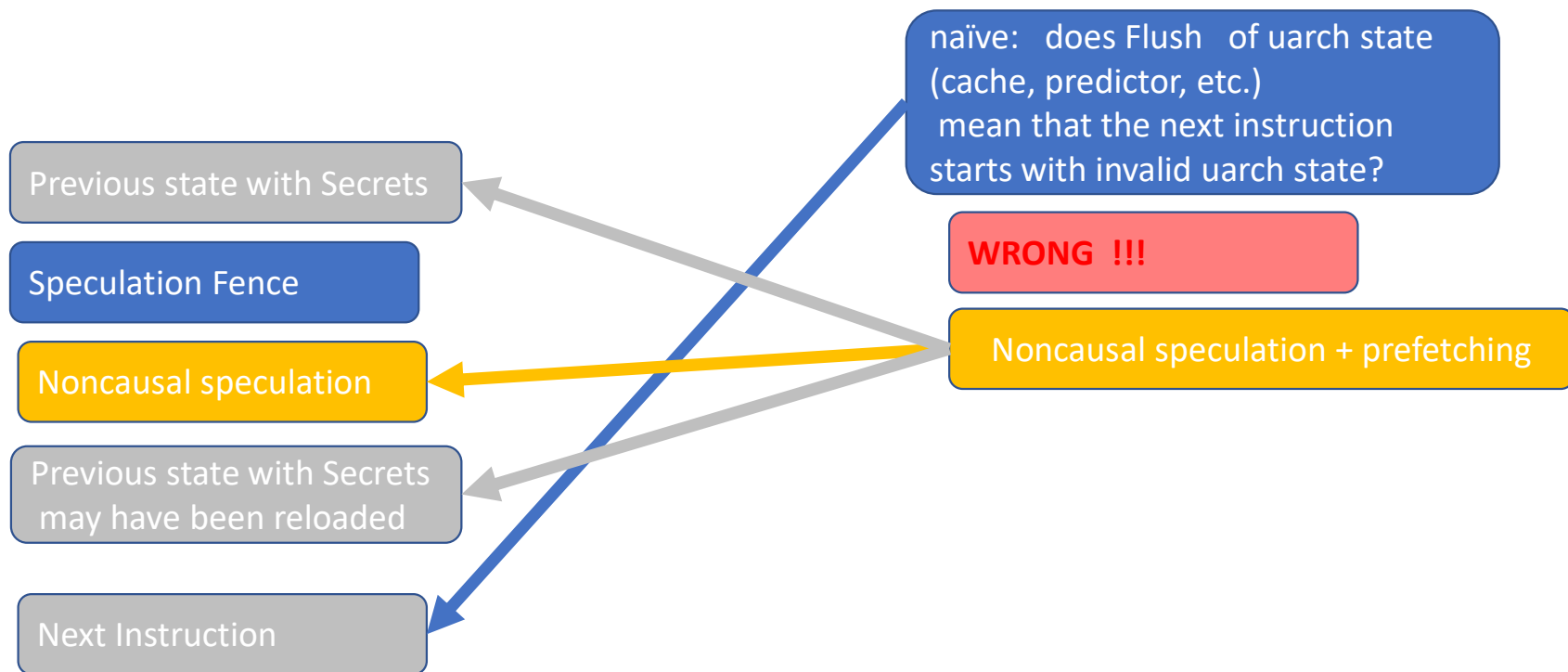
Non-causality



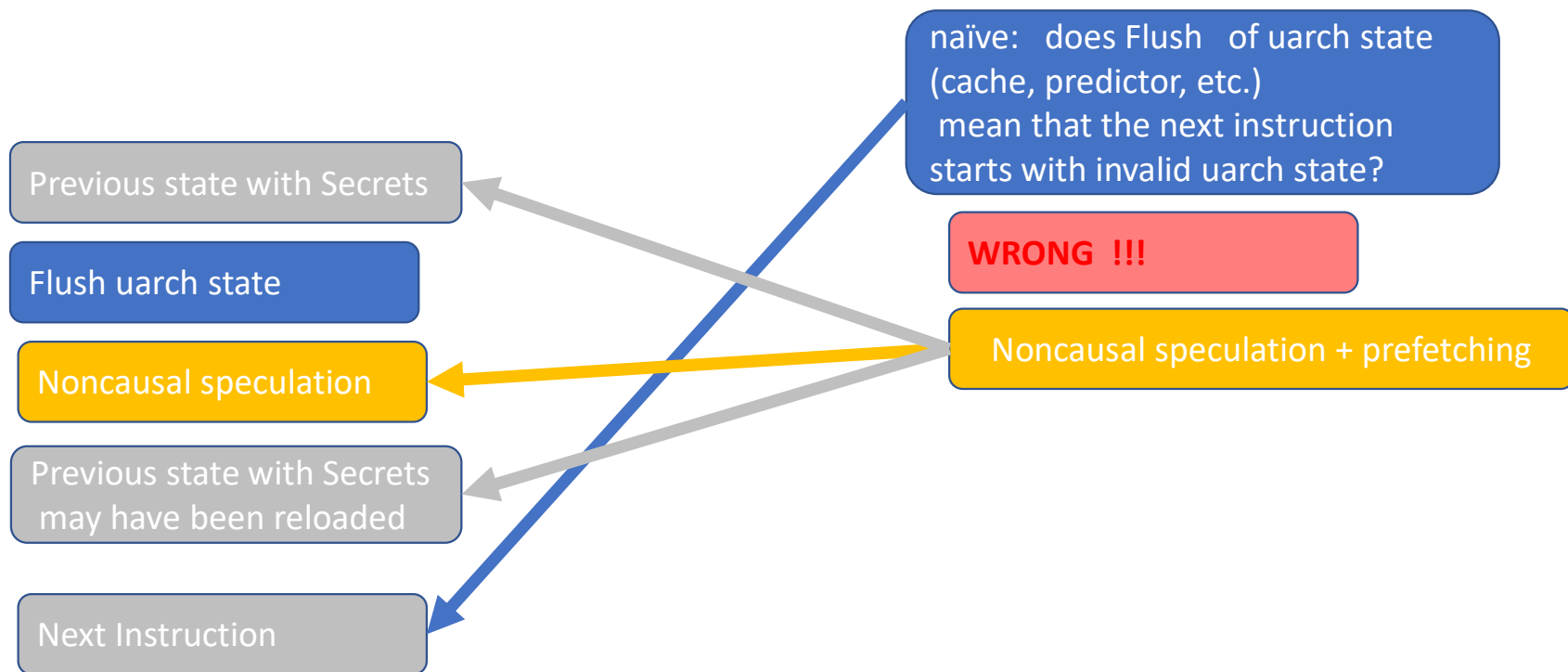
Non-causality => Information leaks



=> Speculation fences don't work (leakage)

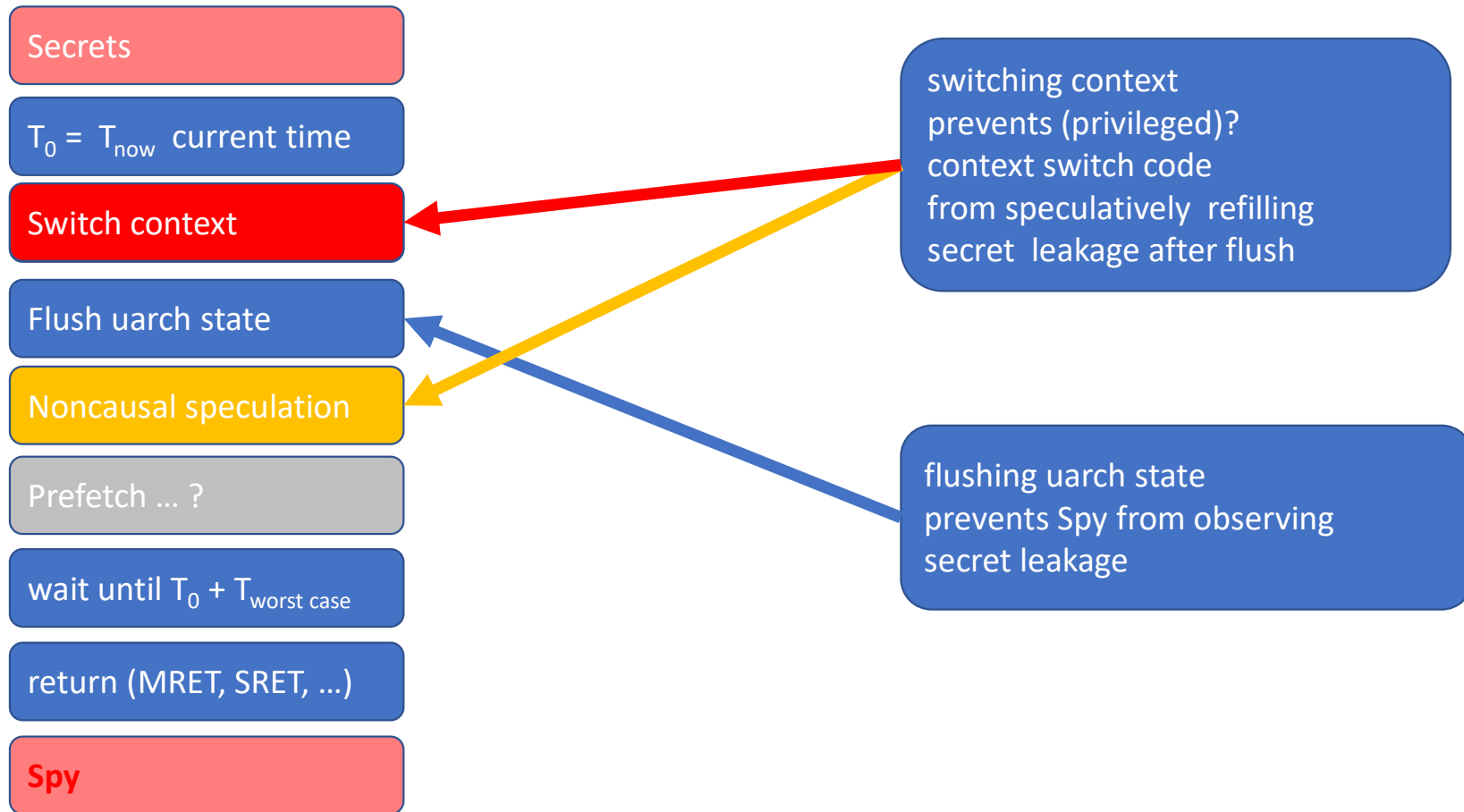


=> Speculation fences don't work (leakage)

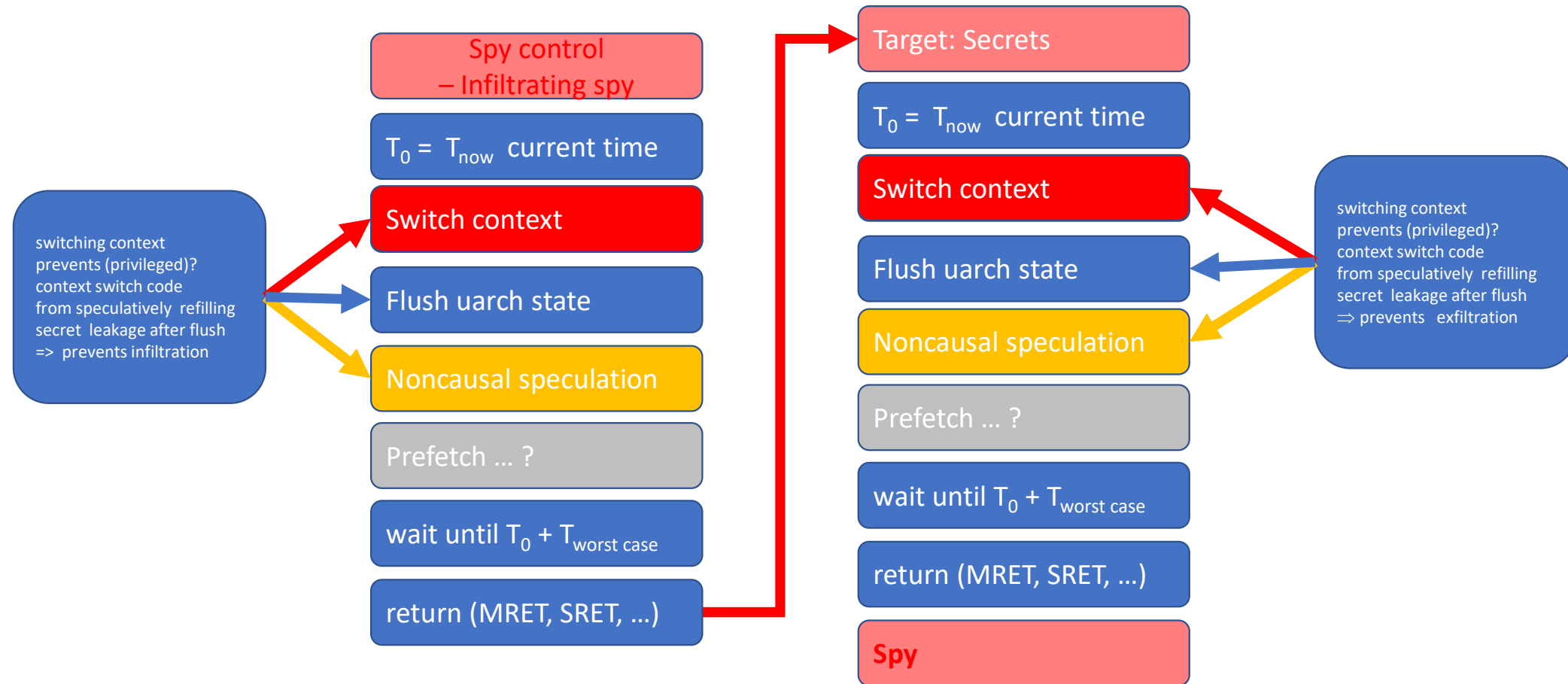


End of Aside: Noncausal
Speculation + Prefetching

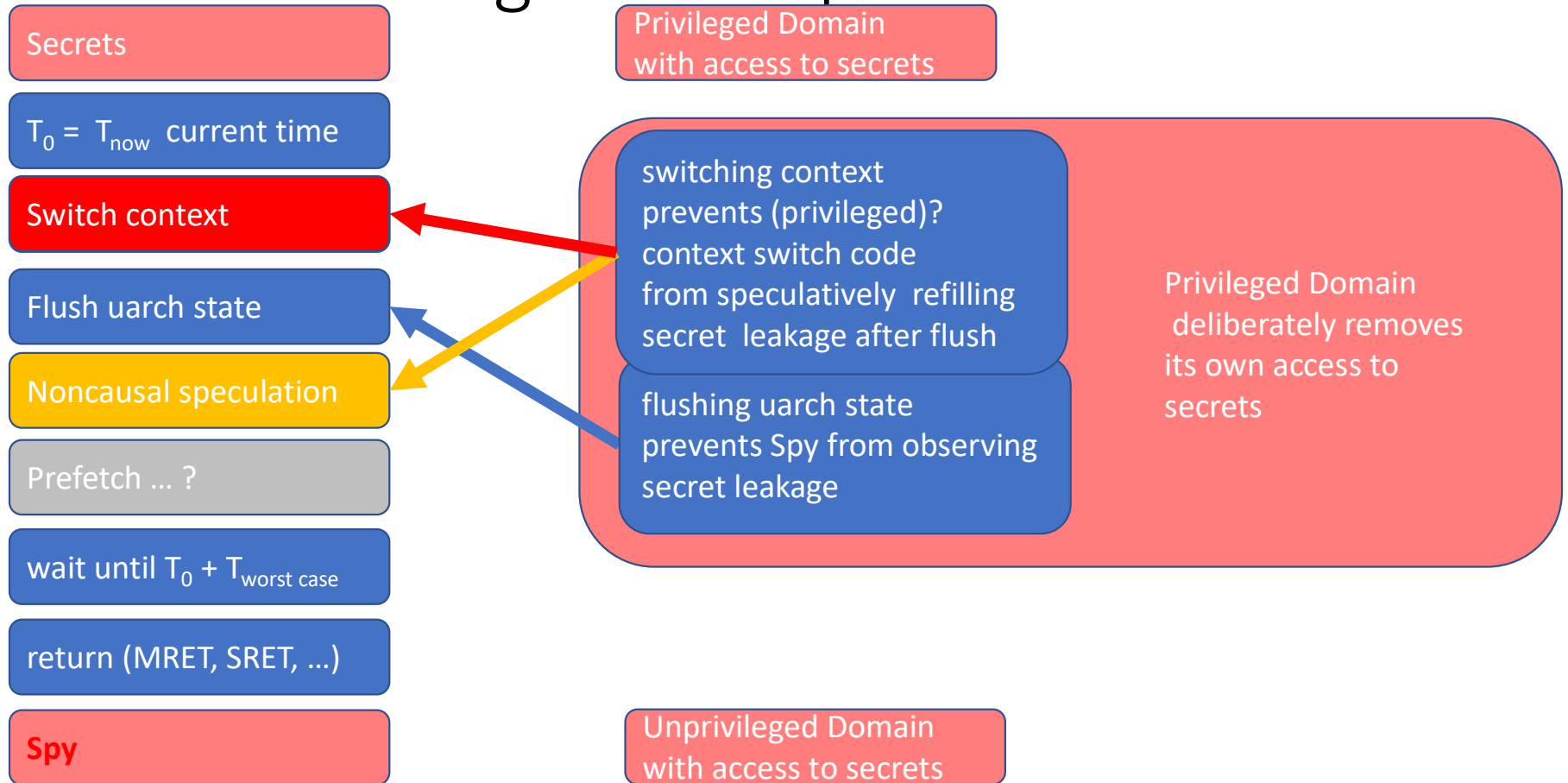
Expanding – Exfiltration



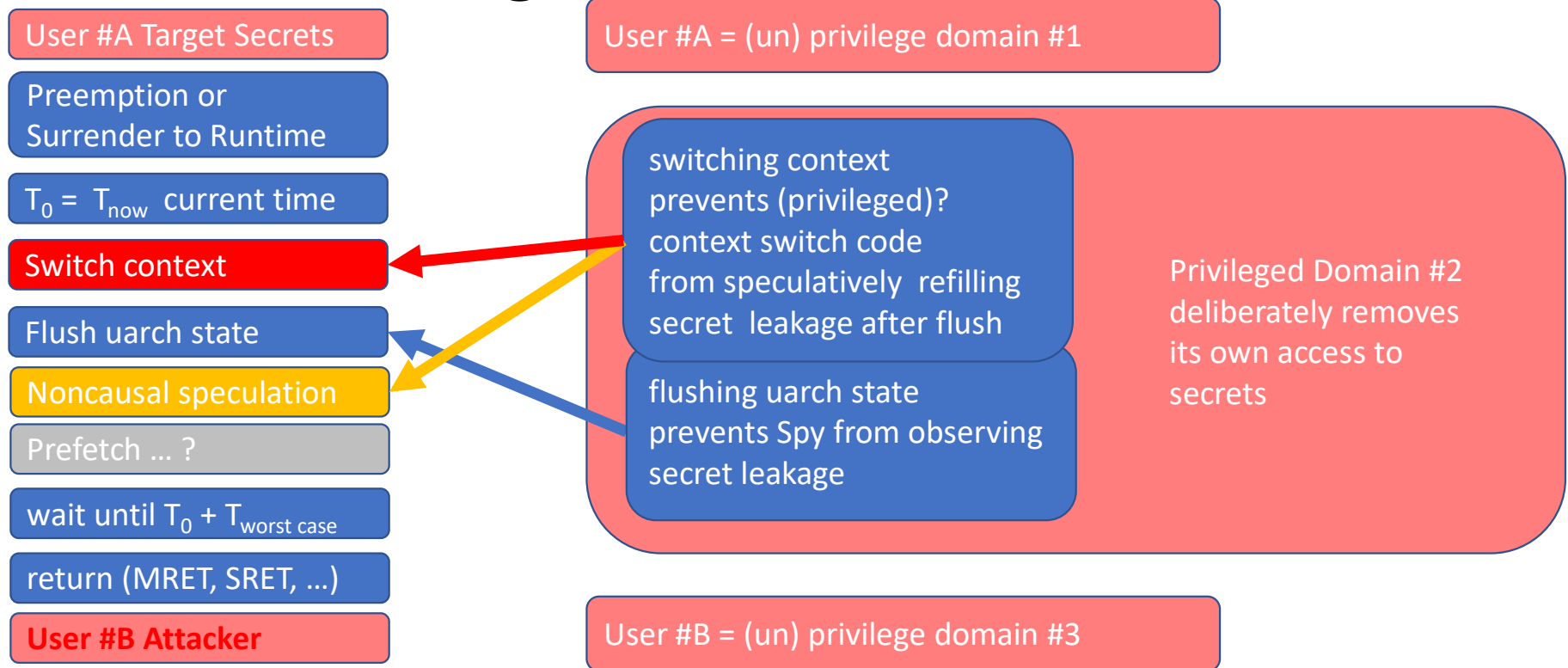
Expanding – Infiltration and Exfiltration



Privileged → Unprivileged transition through cleanup state



Unprivileged → Unprivileged transition through cleanup state



What is this Intermediate Cleanup State?

Possibilities include

- S-mode kernel, but changed page tables
- M-mode
 - ideally some PMPs that M-mode can change
 - so that these PMPs restrict M-mode (speculation++)

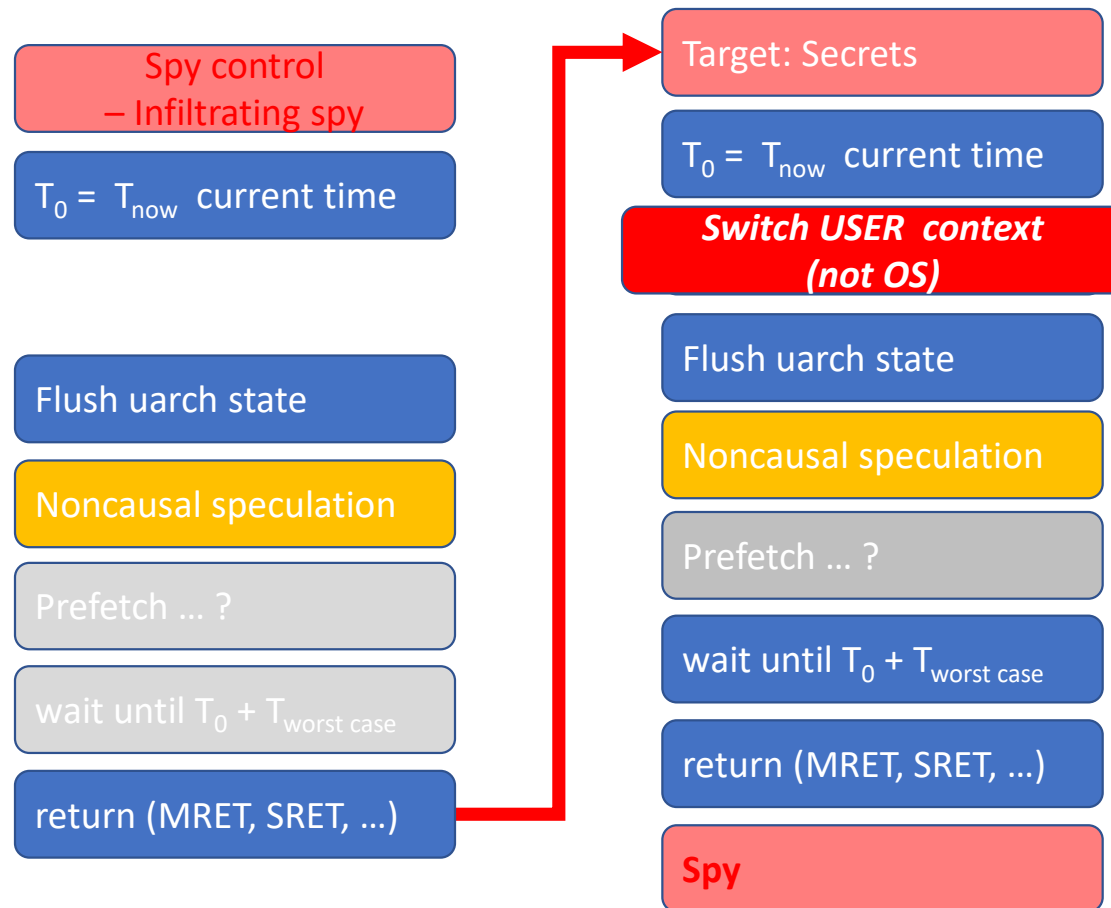
PROBLEM

micro kernels may do this

Linux does not

user usually mapped
into kernel top half

Expanding – Infiltration and Exfiltration



Mitigations?

Forbid speculation across user/kernel boundary
⇒ slower system calls

Or.. structure kernel so that secrets are not mapped into top half

i.e. secrets in microkernel kernel processes (not threads),
monolithic kernel for syscalls)

???

Slow system calls considered dangerous

when system calls are slow
developers abandon Principle of Least Privilege
(e.g. NT GDI, kernel device drivers)

[SBI slide links here](#)

PROBLEM

want user → user transitions without syscalls
e.g. User level sandboxes in web browsers
(assuming syscalls are slow)

like Linux, except user cannot manipulate
page tables or PMPs

Atomicity and Non-Causal cacheability

The real problem

(Non) Atomicity

WLOG assume: multiple FENCE.T instructions needed at any transition

- Wish: PrivDom1 --- FENCE.T ---> PrivDom2
 - If your hardware can do this, we want the ISA to support it
- Probably:
 - PrivDom1 --- FENCE.bpred.T, FENCE.D\$.T ---> PrivDom2
 - Maybe : PrivDom1 --- FENCE.bpred.T, LOOP { FENCE.T.DS(set.way) } ---> PrivDom2
 - Abstract: LOOP {FENCE[j].T } for all modules j
 - merging all the microarchitecture units into the same index space

More than one instruction ➔ atomicity issues

- e.g. interrupt in between
 - block interrupts: unprivileged users cannot... => leads to POLP violations
 - even guest OSes are users from the point of view of a hypervisor
- but it's not just that...

Speculation → atomicity issues

Even “ideal” has problems: PrivDom1 --- FENCE.T ---> PrivDom2

E.g. returning from syscall

OS-97: ... *earlier code, may manipulate sensitive data ...*

OS-98: FENCE.T *ideal? ... no*

OS-99: ERET – *return to user mode*

USER-100: ... *user code, which may be attempting to investigate OS state*

What can happen between OS-98 and USER-100?

- interrupt? - assume OK (interrupt handler flushes/rekeys)
- *the real problem is SPECULATION and NON-CAUSAL CACHEABILITY*
 - *ANYTHING can happen between OS-98:FENCE.T and OS-99:ERET*
 - *as long as it does not affect architectural state, only micro architecture timing state*
 - *in the current noncausal cacheability model*

Speculation and Non-Causal Cacheability

OS-97: ... *earlier code, may manipulate sensitive data* ...

OS-98: FENCE.T *ideal?* ... *no*

OS-99: ERET – *return to user mode*

USER-100: ... *user code, which may be attempting to investigate OS state*

Q: is cache (and other uarch state) empty/rekeyed between FENCE.T and ERET? A: no...

All that is guaranteed is that the state **was** flushed/rekeyed during and immediately after FENCE.T ... but anything may have been loaded back into the calf and predictors **before** the ERET. Even if there was no interrupt.

it is correct according to the formal architectural model to instantaneously reload every piece of microarchitectures state that the FENCE.T flushed or rekeyed

- Is this likely to happen? No.
- But it is allowed to happen in the formal model of speculative behavior
 - which I defined for P6
 - which RISC-V has inherited in a slightly modified form
- I would like to define a formal model that makes it do what we want.

Why did I do this? (Non-Causal Cacheability)

OS-97: ... *earlier code, may manipulate sensitive data ...*

OS-98: FENCE.T *ideal? ... no*

OS-99: ERET – *return to user mode*

USER-100: ... *user code, which may be attempting to investigate OS state*

There could be a branch prediction made at OS-99 that goes and speculatively re-executes everything that you wanted to flush/rekey.

- But... OS-99:ERET is not that sort of branch?
 - No matter: branch predictions use partial tags
- But flushing at least zeroes – so maybe OS-99:ERET will not be predicted as a bad branch alias.
 - As long as there is no interrupt. (OK, okay...)
- Rekeying makes it worse: with the changed key aliasing is increased
 - Rekeying with a version number
 - but that's a timing channel...

Fixed by causality??

OS-97: ... *earlier code, may manipulate sensitive data* ...

OS-98: FENCE.T *ideal?* ... *no*

OS-99: ERET – *return to user mode*

USER-100: ... *user code, which may be attempting to investigate OS state*

Q: is cache (and other uarch state) empty/rekeyed between FENCE.T and ERET? A: no... how change this answer to yes
If we had a formal model for what things you are allowed to do between the OS-98:FENCE.T and OS-99:ERET... i.e. a model of causality?

1991: I gave up on this for Intel P6, in the general case.

Non-causal cacheability: “Anything that is marked cacheable can be placed in the cache at any time”. only the MTRRs constrain.
No reasoning about uarch state is guaranteed. It may probably work, but it may not always work.

- e.g. can’t trace branches because of partial tagging. Partial instruction caches and partial instruction fetch....

2019: RISC-V = noncausal cacheability

+ “anything that is marked cacheable and is accessible in the present privilege domain can be placed in the cache at any time”.
e.g.. Meltdown not allowed, or most Spectre – but same user level sandbox Spectre still fails. Plus, Ri5 has no MTRRs or dynamic PMAs...

Special cases are possible:

- E.g. we could guarantee that the instruction pair OS-98:FENCE.T and OS-99:ERET is fused, nothing happens between them.
 - CISCy: This is what Intel x86 does for interrupt return CLI;IRET or POPF;IRET
- e.g. how to bootstrap un-paged to paged virtual memory (must have identity mapped page...)
- Plus, simple pipelines obviously can be reasoned about

fused instruction pair atomicity

E.g. we could **guarantee** that the instruction pair OS-98:FENCE.T and OS-99:ERET is fused and that nothing happens between them.

- This is equivalent to defining a single instruction FENCE.T+ERET, although it might take 64 bits
- CISCy: This is what Intel x86 does for interrupt return CLI;IRET or POPF;IRET
- RISCy: delayed branches have very similar problem. hence not in Ri5
- Ri5: considering doing this in a few places, e.g. NTLH / NTSH - but hints only

Doesn't help multiple FENCE.T

multi-instruction sequence atomicity

Fusing FENCE.T and ERET doesn't help
when multiple FENCE.T are needed

- PrivDom1 --- FENCE.bpred.T, FENCE.D\$.T ---> PrivDom2
- Maybe : PrivDom1 --- FENCE.bpred.T, LOOP { FENCE.T.DS(set.way) } ---> PrivDom2
- Abstract: LOOP { FENCE[j].T } for all modules j
 - merging all the microarchitecture units into the same index space

Maybe:

OS-97: ... *earlier code, may manipulate sensitive data* ...

OS.-98.0: FENCE.T.start -- *starts FENCE.T atomic state*

OS-98.1: FENCE.bpred.T

OS-98.2: FENCE.D\$.T

OAg19: ERET – *return to user mode -- implicitly clears the FENCE.T atomic state*

USER-100: ... *user code, which may be attempting to investigate OS state*

Allows composition. Adds CSR bit.

Precedent: MIPSr6 LL2/SC2, LLn...SC

multi-instruction sequence constraints

constraints similar to code allowed between LL SC

- maximum length?
- No loops? ... Oops

Lose iteration => no \$ flushes

- OK: PD1 --- FENCE.T.start, FENCE.bpred.T, FENCE.D\$.T,ERET--> PD2
- FAIL PD1 --- FENCE.T.start, LOOP { FENCE.T(set/way) }--> PD2

Lose abstraction

- OK: PD1 --- FENCE.T.start, FENCE.#1.T, FENCE.#2.T,ERET--> PD2
- FAIL: PD1 --- FENCE.T.start, LOOP { FENCE.T.unit[i] }--> PD2

Mode transition to solve atomicity problems

OS-97: ... *earlier code, may manipulate sensitive data ...*

OS-98: FENCE.T *ideal? ... no*

OS-99: MCALL – *return to M-mode*

M-666.1: ... set up no access to any OS state

M-666.2: ... any instruction sequence you want

- implementation specific: FENCE.bpred.T, FENCE.D\$.T
- abstract: LOOP {FENCE[j].T } for all modules j

M-666.3: MRET

USER-100: ... *user code, which may be attempting to investigate OS state*

This could work. It adds overhead.

Original M mode could not be so restricted, but TEE may allow this.

Or create a new mode. e.g. Jose Renau timing domains....

Wrapping Up

I mainly wanted to show you the problem.

I + Krste + AW have a preferred solution,
but the CMO TG already rejected it

my original vision

FENCE.T.target == CMO.SEC.target

flushes or rekeys all uarch timing state between CPU and target

[P,I\$] → flush/rekey all bpreds, etc. but not I\$

[P,I\$] → flush/rekey all bpreds, etc. and I\$

[P,D\$] → flush/rekey all ALU latency and STLF predictors

[P,D\$] → flush/rekey all ALU latency and STLF predictors and D\$

[P,L2] → flush all upto L2, including LRU bits...

[P,target) is just a number in the instruction. Not a register operand

implementation possibilities

```
L:    x1 ← 0  
      x1 ← FENCE.T.target x1  
      BNEZ x1, L
```

FENCE.T could be a no-op --- does nothing

FENCE.T could instantaneously clear all uarch timing state

FENCE.T could rekey all uarch timing state

FENCE.T could iterate flushing cache lines...

I thought this was a good idea.

Allow a range of implementations

iterate

```
L:   x1 ← 0  
     x1 ← FENCE.T.target x1  
     BNEZ x1, L
```

unfortunately, X1 is a register, as it must be for (set,way)
all units are placed into same index space

e.g 0 = bpred, 1 = BTB, ...

FENCE.T takes an index, and writes a new index

CMO TG rejected

actually, they vociferously rejected the “address range” version of this

- I wish I had not proposed that first, it wasted so much time

but FENCE.T does not need a virtual address range

- it always flushes or re-keys an entire uarch structure, cache or predictor
- or possibly a partition, e.g. way locking

CMO TG phase 1 is doing only single address,

- could not reach agreement on whole cache
 - not “whole cache” - CMO.UR.ALL, sync/async
 - not CMO.UR.IX (set,way)
- phase 2 may look like CMO.SEC.UR == FENCE.T.target
 - but I doubt it, I am so shellshocked
 - worth thinking about possible objections
 - risk: CMO TG pissed off by the FENCE.T.target proposed here

Summary

- rekeying is not a silver bullet
 - but it's good when you can do it
- flushing necessary for things like writeback caches
 - if Spectre/Meltdown important, more instantaneous writethrough flushes

IMHO single operation bad

- certainly fails external, but even inside CPU has problems
- single instruction bad - RISC
- multiple instructions or iteration - good, if accepted

real problems are atomicity and noncausal cacheability

fuse FENCE.T;ERET

multi-instruction sequence FENCE.T.start, FENCE.T.#1, FENCE.T.#2, ERET

transition through mode

- E.g. M-mode SBI (slow, minor changes to an mode needed like TEE)
- E.g. Jose Renau lightweight timing domains

KA/AW/Ag: FENCE.T.target Rsrcsdt

- Solves all the problems I know about
- but CMO TG ... rejected address range, and hasn't had time to really look at security-related flushes and (set,way)

BACKGROUND

Security (Inference Channel) CMO/Flushes

CMO.UR.SEC*. \$id rs1/rd

rs2=x0. s1/rd is srcdst, containing “abstract cache index”, like other CMO.UR

?? rs2 != x0: CMO.UR.SEC*. \$id rs1/rd, rs2:WordGuard_or_other_ID
TBD: need a CSR to avoid virtualization hole (Contentious).

- CMO.UR.SEC.I
 - clear I\$, all branch predictors, etc
- CMO.UR.SEC.D
 - flush L1 D\$, STLF, cache miss predictor, way predictor, etc.
- CMO.UR.SEC.ID
 - I + D.
 - Typically a full CPU/processor
- CMO.UR.SEC.POC_P
 - flush to point of coherence of all processors – included LRU, prefetchers, etc.
 - Typically L2, MLC or LLC depending
- CMO.UR.SEC.POC_P_IO
 - flush to point of coherence of all processors and IO
 - typically DRAM (but not always)
 - DRAM open/close page predictors?
- CMO.UR.SEC.MAX
 - maximal (standard) flush.
 - maybe same as POC_P or POC_P_IO

**Full disclosure: proposed
but contentious.**

if CMO.UR rejected,
will have more distinct instructions
(and IMHO less security,
no user mode, etc.)

- Most security related flushes are “whole cache” => CMO.UR
 - TBD: by address flushes CMO.64B.SEC and CMO.AR.SEC maybe – e.g. flush just security LUTs and key dependent data structures
- All writeback dirty data, and leave all invalid
- In addition, reset/invalidate all uarch perf related state up to specified point => CMO.UR.SEC.*
 - Branch predictors, prefetchers, store to load forwarding predictors, LRU, way predictors
 - Note: such flushes are often idiosyncratic – few busses have good support (although, e.g. AERM DVM has flush bpred)
 - Implementations will take advantage when standard bus support exists.
 - Implementation can trap to M-mode if “mash-up” SOC has support, but not standard on bus. CMO.UR => amortize such trap overhead
 - OVERALL: the RISC-V CMO.UR proposal will encourage SOC/NOC IP, to improve support, while supporting legacy
 - NOTE: most predict state is not coherent => may need to perform CMO.SEC.* on all CPUs involved
- “Abstract cache index” = shows benefit of CMO.UR approach. Not just (set,way) iof a single cache, but may include flash resets or iteration over branch predictors, etc.
 - i.e. the abstract cache index is not just for a single HW cache array
- CMO.UR can solve atomicity issues: e.g. flush bpred before I\$, or after – neither is good enough
- Levels:
 - Security would like to flush everything – up to point where cache is physically partitioned.
 - But security/perf tradeoff – e.g. we browser may flush as much as it can flush without making page too slow. Hence levels here.
 - Expect more levels in a system dependent manner. TBD CSRs?
- User code friendly (including thread migration):
 - e.g. Web browsers want to flush uarch state on transitions between JavaScript in sandbox and user level runtime - without system call overhead
 - Delegation => OS can allow or disallow, depending on security model
 - CMO.UR.* form can be process/thread migration compatible
 - embedded/HPC – often no process/thread migration
 - general-purpose Oss - usually do process/thread migration
 - e.g. CMO.UR.SEC.{ID,POC_*,MAX} are migration safe if traversing a strictly inclusive shared cache level
 - trap/delegate if not inclusive
 - CMO.UR.SEC* stylized code => CO/VMM context switch code can detect and complete or delay migration
 - virtual machines => migration even though OS thinks fixed CPU

**Caveat: ISA supports good inference channel control
but uarch may have incomplete support.**

TBD: how document what implementation provides.
NOTE: CMO.UR.SEC.* supports efficient trap to fix problems discovered once deployed.

Security (channel) TLB flushes

- RISC-V original
 - sfence.vma - local TLB flushes,
 - by virtual address, by ASID, all...
 - constant time and other security-related properties not defined, but that would not be hard
 - I had hoped to address this in the CMO TG, but the CMO TG is bogged down doing non-coherent I/O
 - AFAIK no RISC-V group has worked on “TLB shoot down” instructions
 - not the virtual memory group, not the CMOs group
 - resistance (AW) to requiring ASID constant
 - but ARM and Intel and many others have such instructions
- Obvious extension to already proposed CMO.UR.SEC.*

FLUSH – or NOT

Let's get this out of the way:

Strict partitioning in space and time flushing is not the only way some systems have proposed to mitigate covert/leak channels.

E.g. some change hash keys.

This is great, if it is good enough. But it is not always good enough.

[links here](#)

Flush – or NOT – ?

Let's get this out of the way:

- Gernot: partition in time or partition in space ...
- Partition in time:
 - I hope we agree that flushing microarchitecture data structures - clearing them zeroing them - on transitions between privilege domains is sufficient
- What about non-strict partitioning?
 - e.g. instead of flushing if time sharing
 - e.g. instead of strictly partitioning if space sharing
 - encrypt or hash the microarchitecture state
 - changing keys between privilege domains
 - WLOG I want to call this “keying microarchitecture state

Keying microarchitecture state

BRIEF: this is a great idea, but it has limitations

- I want to allow, implementations, e.g. if you want to make your FENCE.T fast
- but I don't want to assume that all implementations will do this
 - i.e. I do not want the ISA to make assumptions that work for keying but not for flushing or strict partitioning
- Implementations that care about security should document the timing channel parameters and properties
 - just like we used to do for Orange Book in 1980s
 - if keying the microarchitecture state is good enough, great
 - if not, ...

Why (not) keying ?

E.g. QCOM - branch predictor - change hash on context switches

Maybe OK for information leaks

- attacker does not know mappings
 - e.g. of branch addresses (and histories) to branch predictor entries
 - neither for victim nor attacker
 - note: also need to hash prediction taken/not-taken...
- makes it hard to target particular branches as in Spectre/Meltdown

It is not a great covert channel,
not as high bandwidth as some,
but nevertheless...

But covert channel still exists

Cooperating bad guys A+B can signal each other by

- estimate size of branch predictor - occupying all entries
- A fills with highly biased branches
- B signals
 - 0: does nothing
 - 1: fills with highly biased branches
- A re-measures
 - no change: B signalled 0
 - change >~50%: B signalled 1
- Turns: time?

Covert versus Information Leak channels

- The covert channel described in previous slide maybe not high-bandwidth.
 - Covert: cooperating bad guys A+B in different privilege domains
- It may be hard to use this as an information leak channel
 - Leak: attacker A and victim V in different privilege domains
 - ... hard / very hard / may be impossible, but I doubt it lacking a formal proof

STRONG PROPERTY: if there is no covert channel, then there can be no information leak

BESIDES: victim behavior in many places is a well-known. amounts to a stupid bad guy B.

FLUSH – or NOT

- Strict partitioning in space and time flushing is not the only way some systems have proposed to mitigate covert/ leak channels.
 - E.g. some change hash keys.
 - This is great, if it is good enough. But it is not always good enough.
 - I want a FENCE.T ISA that can do both time and key partitioning
 - And I want the user not have to care about the actual implementation
 - Clumsy terminology:
 - partitioning in time: “flushing”
 - strict partitioning in space ...
 - partitioning fiat key/hashing
 - non-strict in time: change key on context switch without flushing
 - non-strict in space: different keys for users sharing/overlapping same data structure
- I wish I had a word that covered both explicit flushing and changing keys. Since I don't know if such a word, I will say “flushing” to include both

Flushing uarch state

and/or key switching...

Lots of uarch timing state

Examples:

- Ifetch: next cache line predictor, conditional, branch location (BTB), return stack, fast L1/slower L2, ...; predecode bits in L2\$...
- Decoder: branch location known, => larger
- Data memory: load latency predictor, STLF predictor
 - D\$ (L1,L2,L3...) way predictor, LRU, bank conflict predictor...
 - Prefetchers – in CPU, and external
 - DRAM page open policy prediction
- ALU latency predictor (multiply, divide, memorization)
- Power predictor: powerdown/retain parts of chip

TBD: diagram: essentially at any point where there's an arrow between two blocks, there can be a predictor.

Some uarch state easy to flush (or key)

E.g. “instantaneous block clear” of valid bits...

- but... one of the first things I learned as a young computer architect
 - is that not all technology can do instantaneous block clear
- ➔ 4 sets of valid bits, switch, state machine clears in background... TC still exists

Sometimes flushing/rekeying hurts performance

- Sometimes big performance benefit for sharing (not flushing/keying)
 - e.g. 2 different user processes accessing *similar* webpages or databases
 - e.g. user may train OS predictors and vice versa
 - OS read() syscall may load cache values that user wants to use

It's my fault:

I deliberately decided not to flush branch predictors on OS/user transitions in 1991 for Intel P6

- why not? -- Gould secure UNIX experience. Performance.
- if you lose performance by partitioning into different privilege domains, users will put things that should be in different privilege domains in the same, violating Principle of Least Privilege.
 - e.g. Microsoft NT GDI - originally in kernel, moved to user mode because kernel crossings too slow
 - Which is more important?: POLP or TC ? A: both?

IMHO: let system software decide when it wants to flush and when it doesn't

- conservative: flush/rekey on all privilege domain crossings
- stupid and insecure: never flush (pre-Spectre)
- smarter:
 - different processes same user → may be don't need to flush
 - user/top-half OS → don't flush if no sensitive info
 - flush if sensitive info
 - hard: same process same user different privilege??

Some uarch state hard to flush (or key)

Semantics

Some uarch state cannot be instantaneously flushed

- e.g. dirty cache lines → must be written out one by one
- shared memory locations → cannot change keys for dirty data

Physics

Some uarch data structures are just plain too big... must iterate

- over entries, sets, ways, banks...

Modularity

- Different functional units may have different keying operations and instructions
 - may not be possible to put in a single instruction
 - maybe not FENCE.T
 - but FENCE.bpred.T, FENCE.D\$.T
 - abstract: LOOP {FENCE[i].T } for all modules i
- even if same instruction, may not occur at the same stage of pipeline
 - Can't even stall pipelines anymore (HF) – replay buffers

Physics + Modularity

+ ... Formal Speculation Model ...

→ Atomicity problems

this is my biggest concern