EXPLAINED:

MELTDOWN & SPECTRE

Jens Neuhalfen

https://github.com/neuhalje/presentation_meltdown_spectre

EXPLAINED:

MELTDOWN & SPECTRE

for normal people

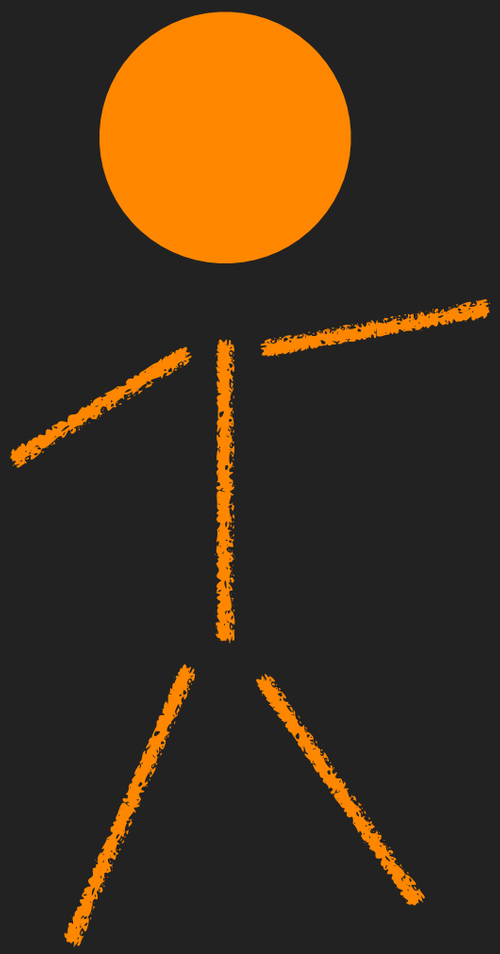
https://github.com/neuhalje/presentation_meltdown_spectre

Jens Neuhalfen

WHO AM I?

- Jens Neuhalfen
- Age: Forty something
- IT since: ever
- Skills: Bridge between IT and business, IT-Security Management, writing software
- https://github.com/neuhalje

SECURITY GOALS



SECURITY GOALS: APP ISOLATION

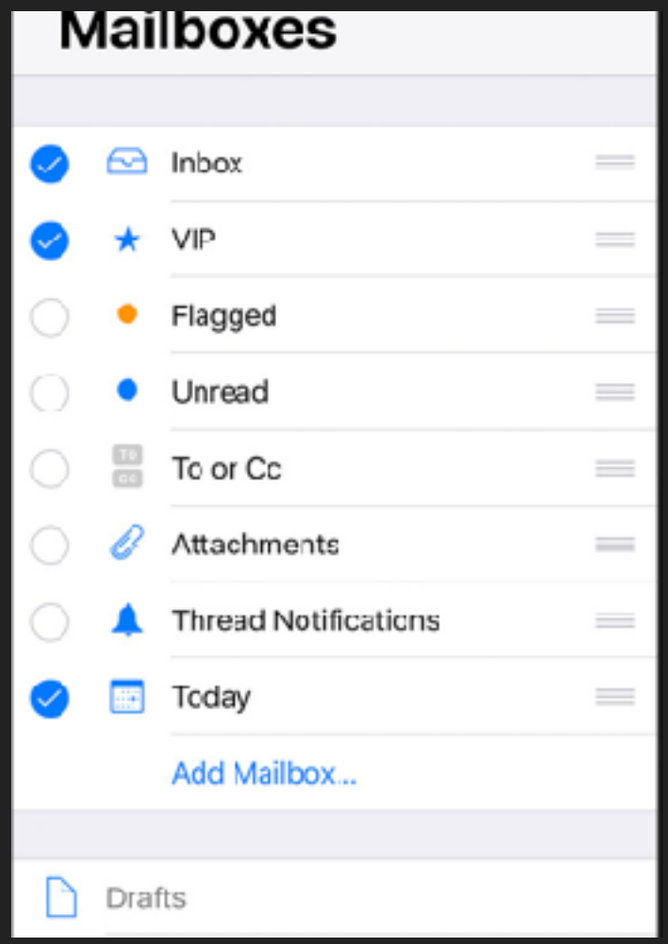
SECURITY GOALS: APP ISOLATION

You don’t want

this

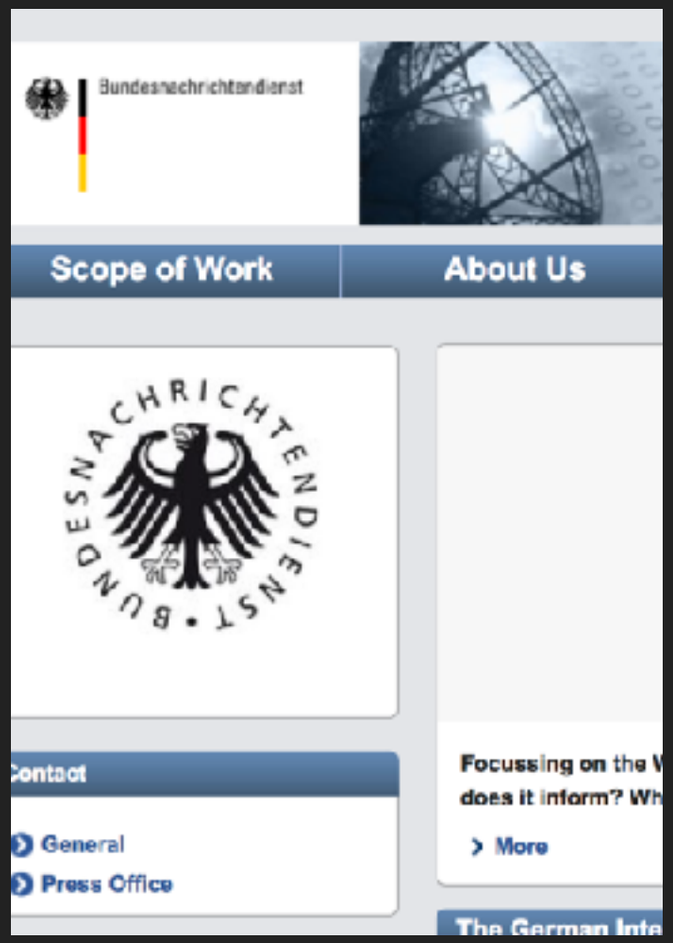
SECURITY GOALS: APP ISOLATION

You don't want
this



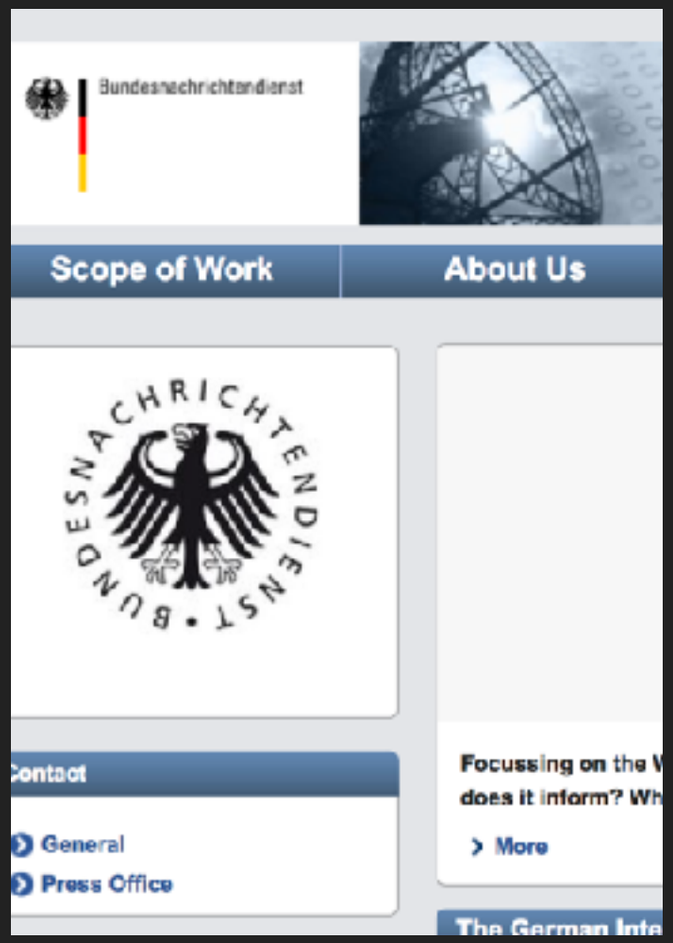
To read
that

SECURITY GOALS: BROWSER TAB ISOLATION

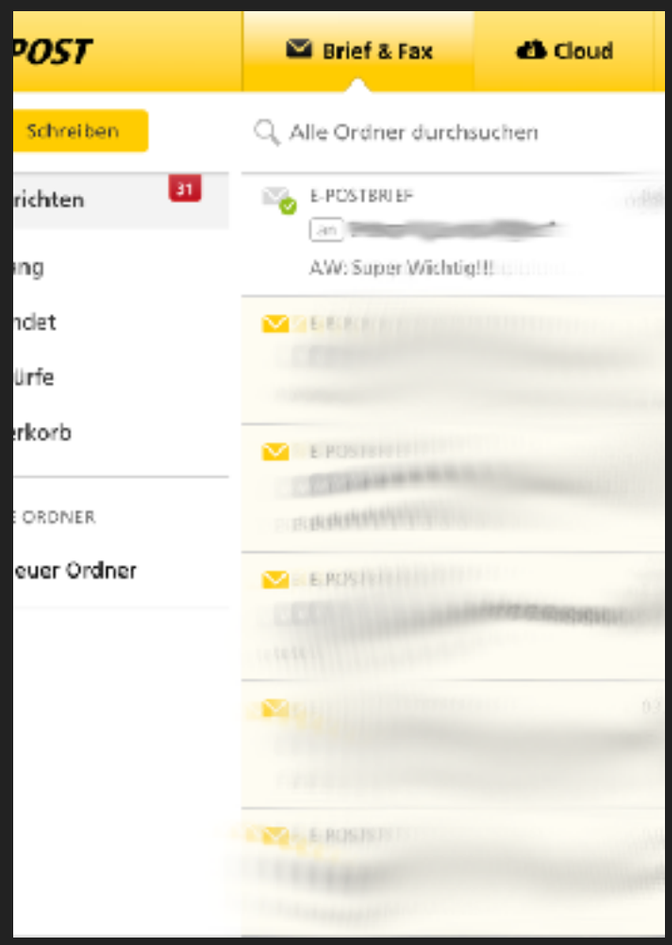


You don’t want *this*

SECURITY GOALS: BROWSER TAB ISOLATION



You don't want *this*



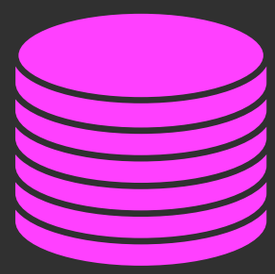
To read *that*

SECURITY GOALS: CLOUD ISOLATION

You don’t want

this

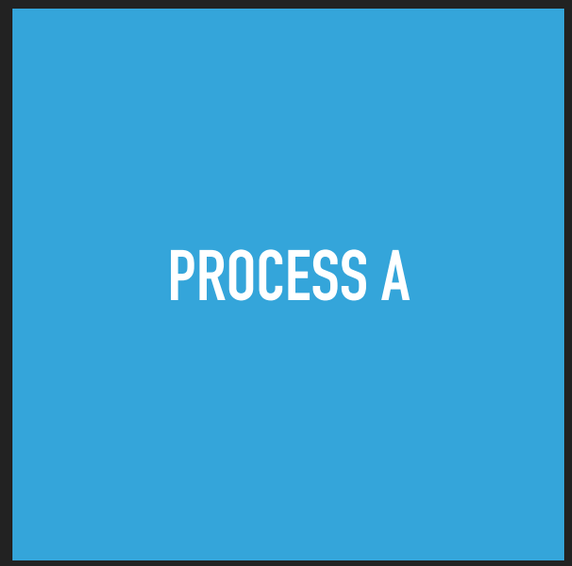
SECURITY GOALS: CLOUD ISOLATION



You don't want
this

To read
that

SECURITY GOALS: MEMORY ISOLATION



You don’t want

this

SECURITY GOALS: MEMORY ISOLATION

PROCESS A

PROCESS B

You don't want *this*

To read *that*

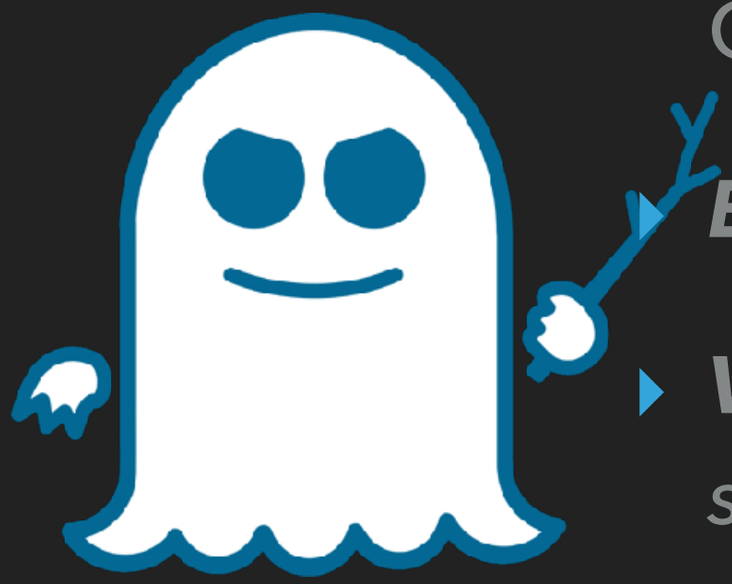
A PROCESS MUST BE ISOLATED (PROTECTED) FROM OTHER PROCESSES!

You

MELTDOWN

- ***Result***:Programs can read memory it should not
- ***Affects***: All modern CPU/OS
- ***Break out of VM***: No
- ***Vector***: Uses *speculative execution* to read forbidden memory and *cache timing* as side channel to exfiltrate data
- ***How bad***: Very bad
- ***Fixes***: Mainly OS patches. Negligible performance impact on modern CPU. High impact on older CPU

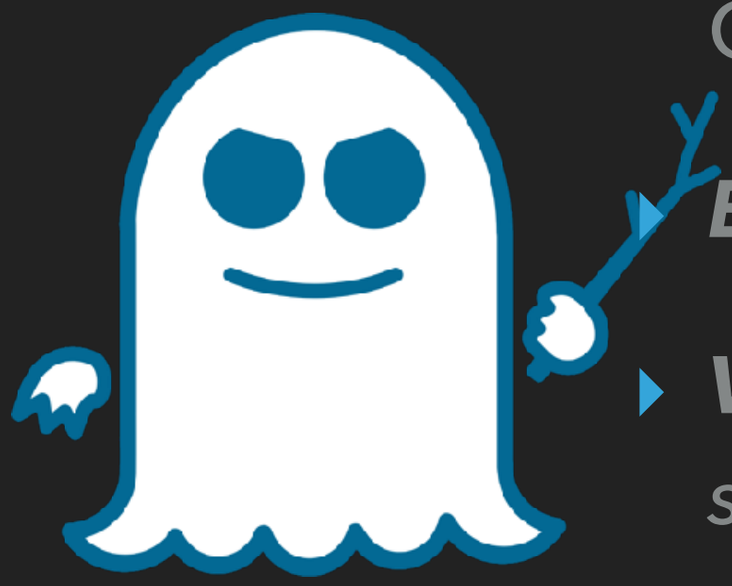
SPECTRE



- ***Result***:Programs can read all memory
- ***Affects***: All modern CPU/OS
- ***Break out of VM***: No
- ***Vector***: Uses *speculative execution* to read forbidden memory and *cache timing* to exfiltrate data
- ***How bad***: Very bad
- ***Fixes***: Mainly OS patches. Negligible performance impact on modern CPU. High impact on older CPU

SPECTRE

TODO



- ***Result***: Programs can read all memory
- ***Affects***: All modern CPU/OS
- ***Break out of VM***: No
- ***Vector***: Uses *speculative execution* to read forbidden memory and *cache timing* to exfiltrate data
- ***How bad***: Very bad
- ***Fixes***: Mainly OS patches. Negligible performance impact on modern CPU. High impact on older CPU

SIDE CHANNEL

- Unintended leak of information
- Light on –> Someone at home
- Power consumption high –> CPU busy
- Memory read fast –> Data is in cache

SIDE CHANNEL

TODO

- Unintended leak of information
- Light on –> Someone at home
- Power consumption high –> CPU busy
- Memory read fast –> Data is in cache

OUT OF ORDER EXECUTION

- ...

OUT OF ORDER EXECUTION

TODO

- ...

SPECULATIVE EXECUTION

- ...

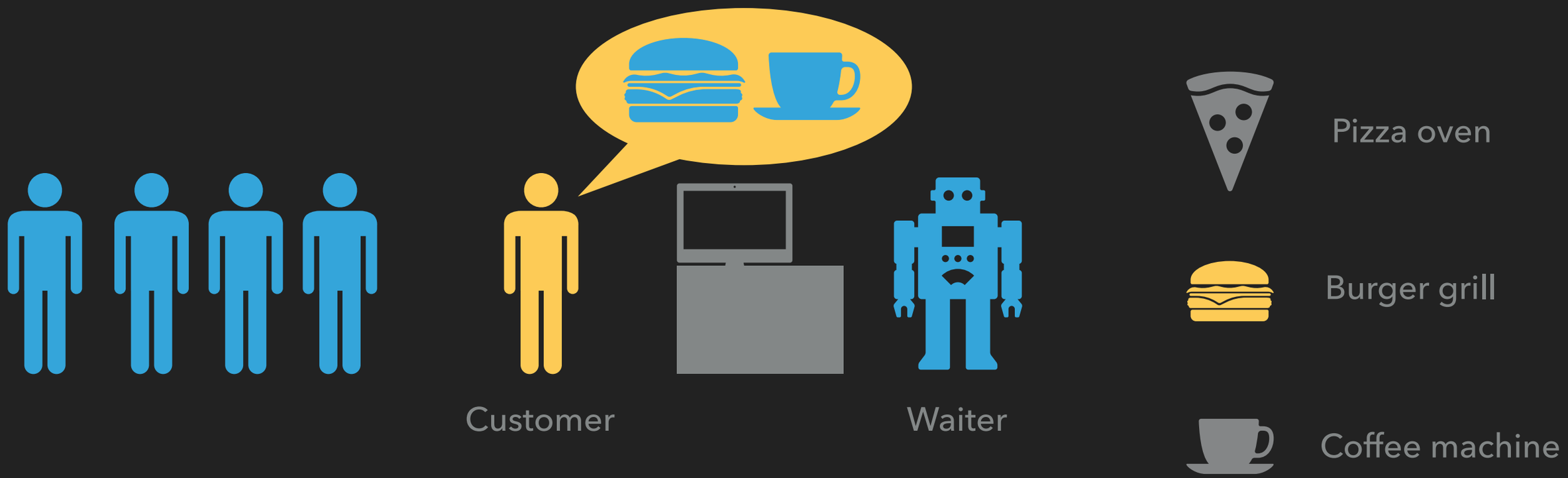
SPECULATIVE EXECUTION

TODO

- ...

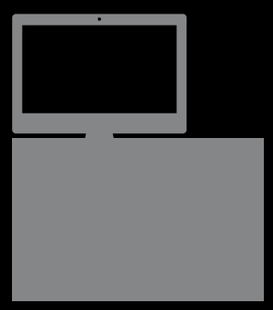
CONFIDENTIAL BURGERS INC.

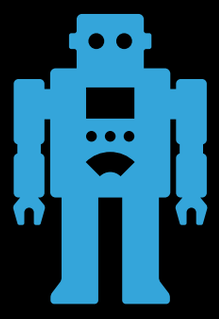
Confidential Burgers inc. sells burgers, pizza, and coffee.



CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION

Customer





Waiter

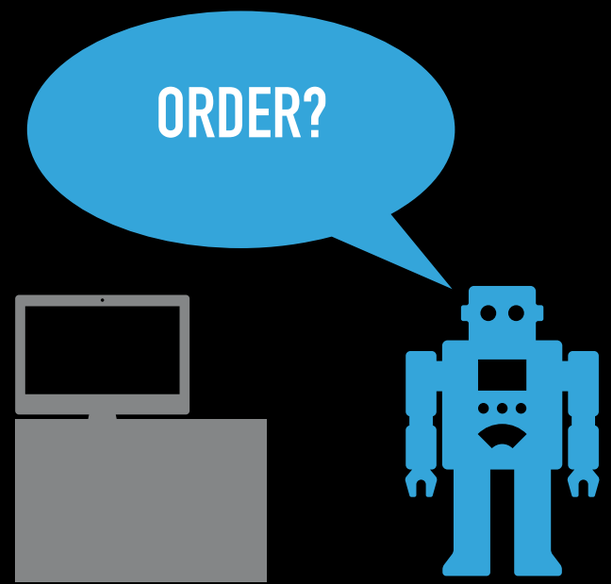
Pizza oven

Burger grill

Coffee machine

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION

Customer



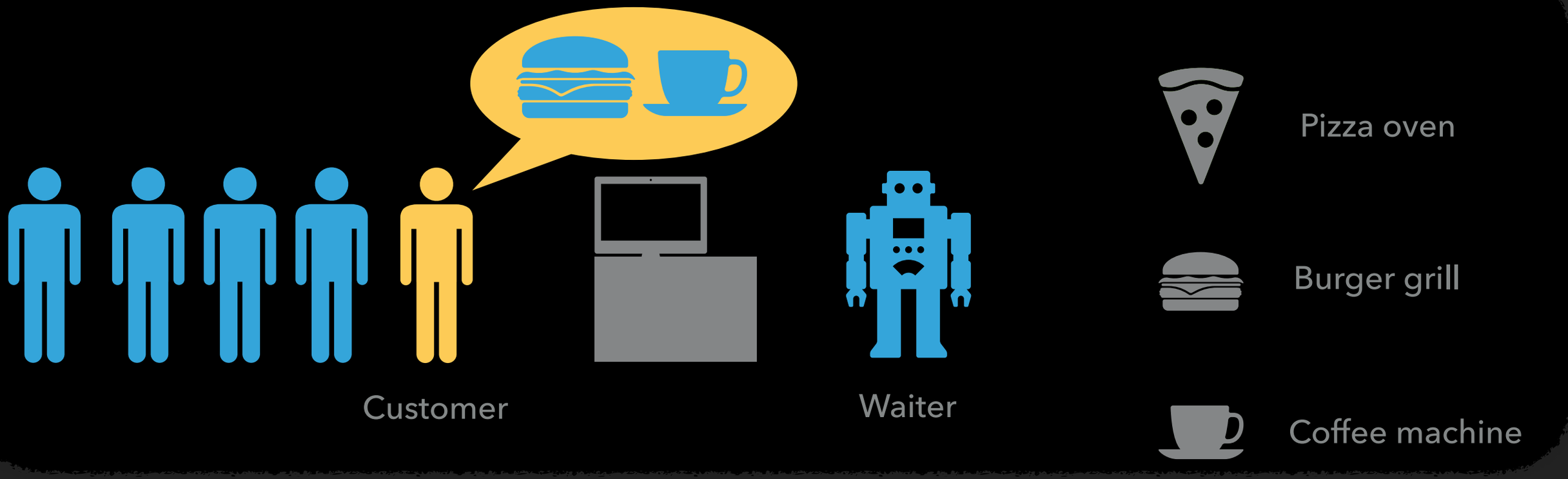
Waiter

Pizza oven

Burger grill

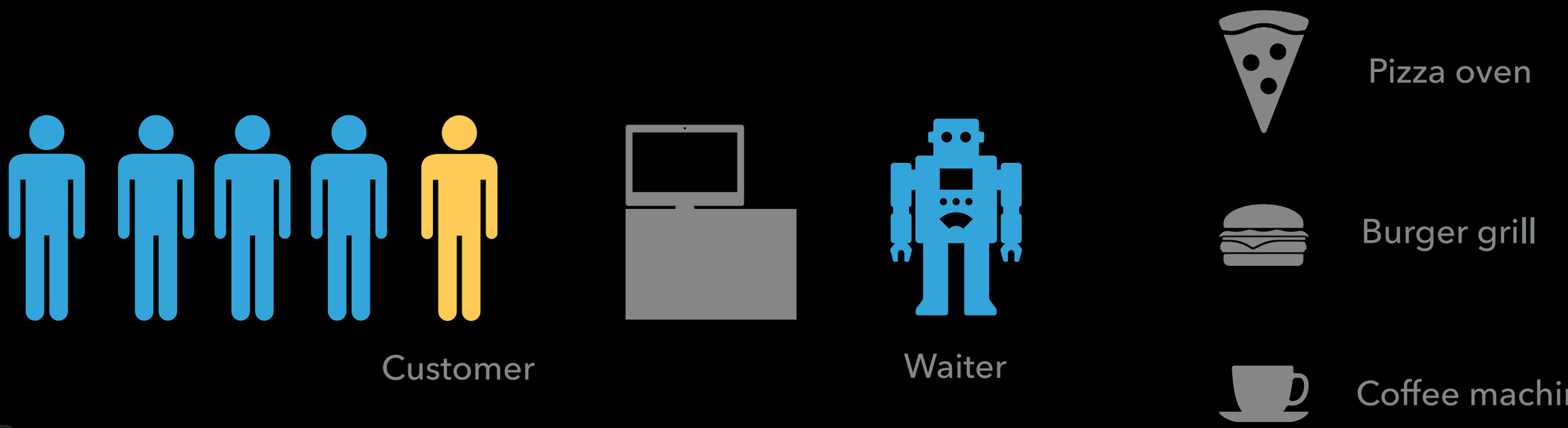
Coffee machine

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



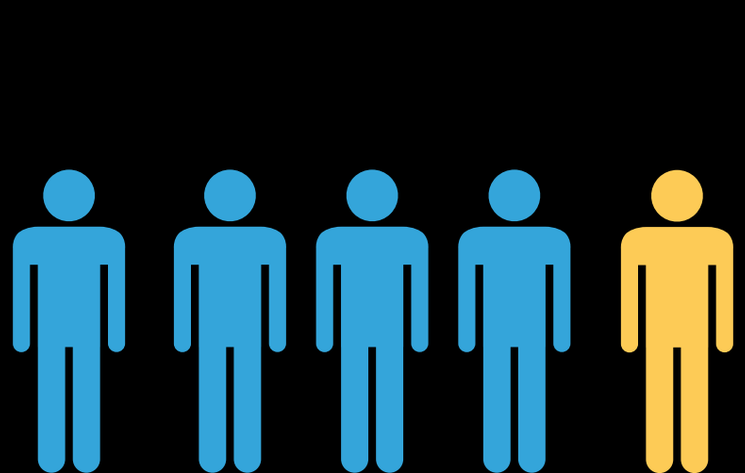
- Decode instruction into μOPs

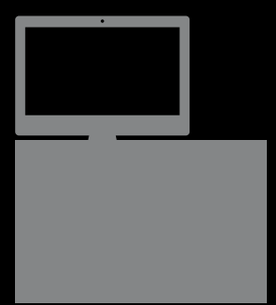
CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION

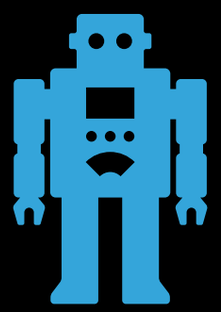


- Decode instruction into µOPs
- Schedule µOPs

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION







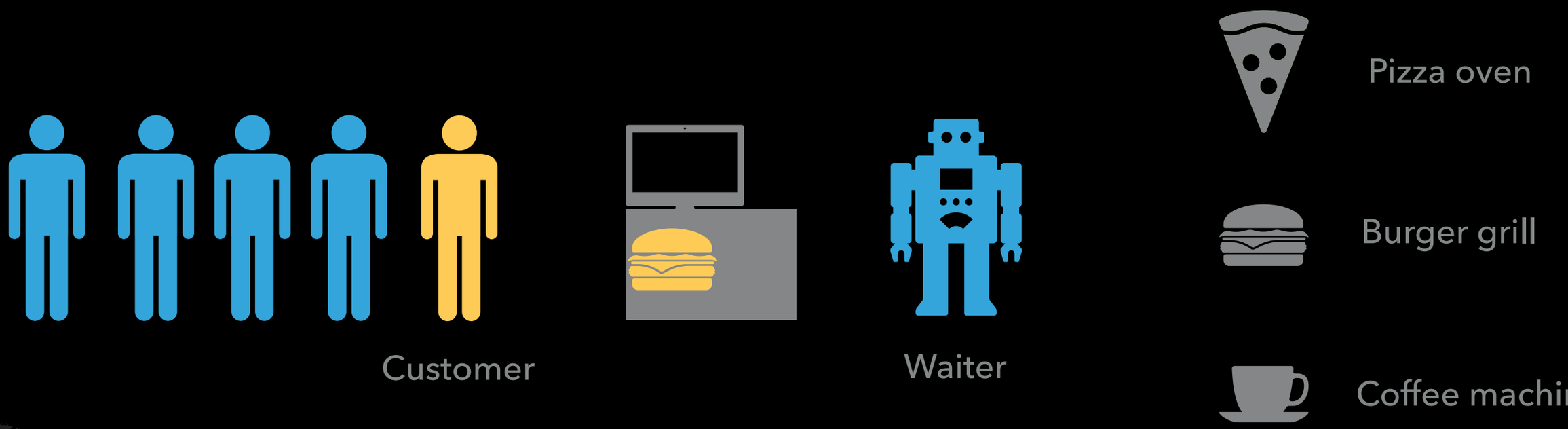
Pizza oven

Burger grill

Coffee machine

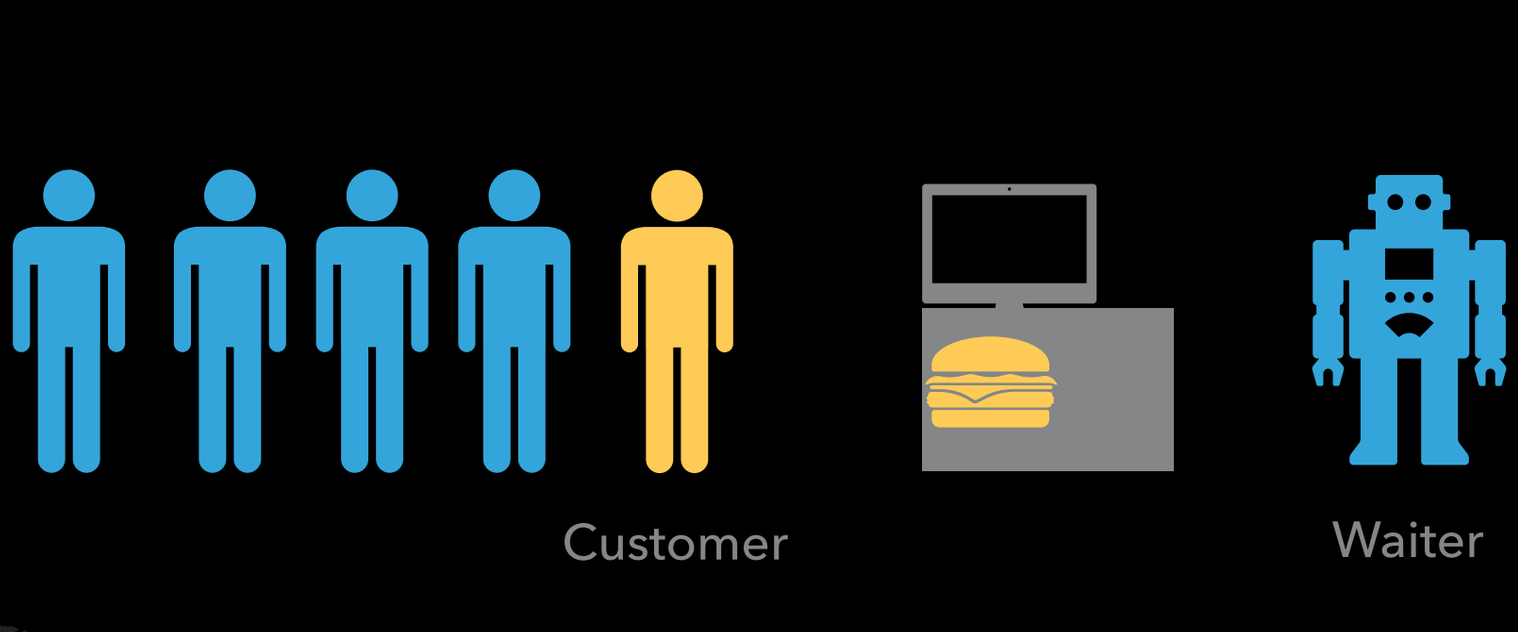
- Decode instruction into µOPs
- Schedule µOPs
- run 1st µOP

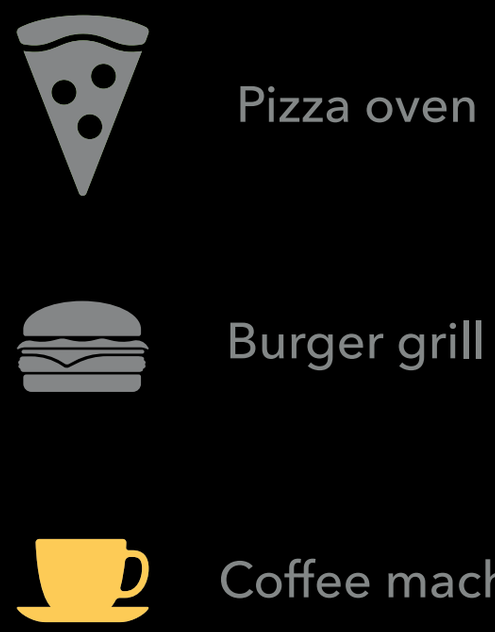
CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



- Decode instruction into μOPs
- Schedule μOPs
- run 1st μOP

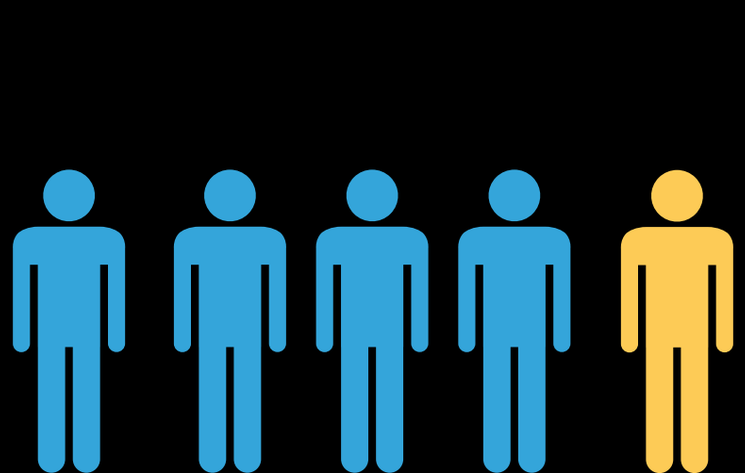
CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION

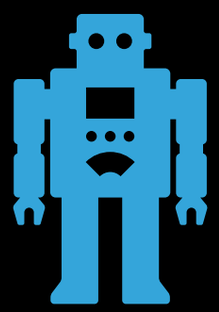




- Decode instruction into µOPs
- Schedule µOPs
- run 1st µOP
- run 2nd µOP (serial execution)

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION





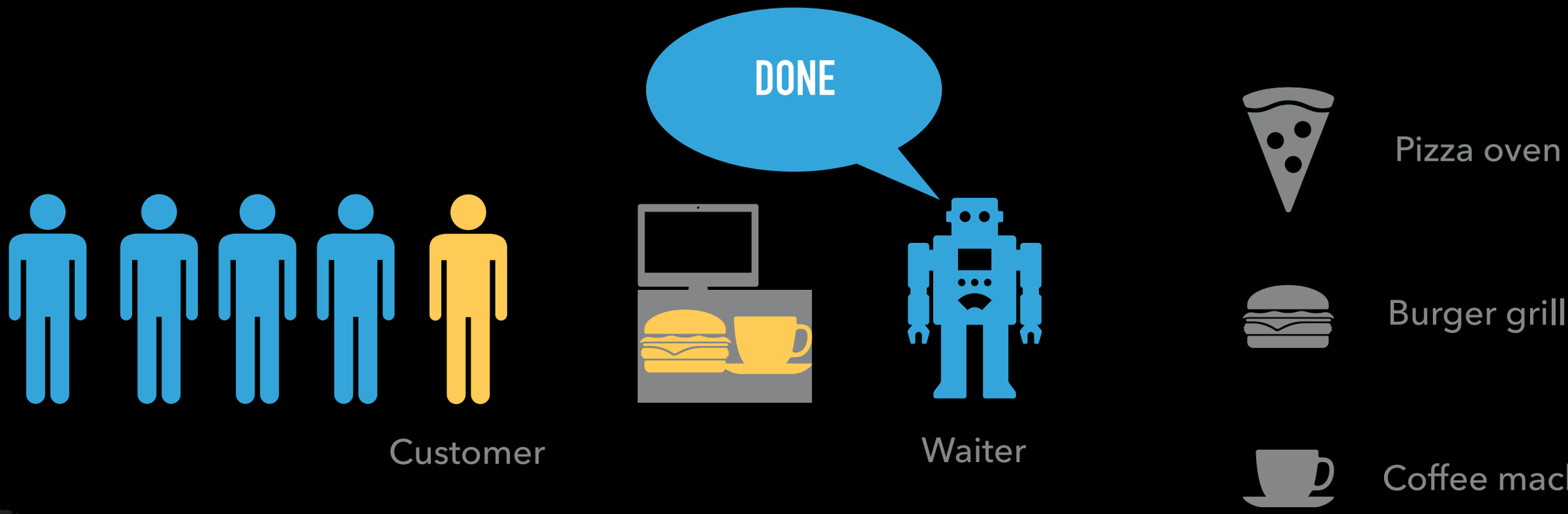
Pizza oven

Burger grill

Coffee machine

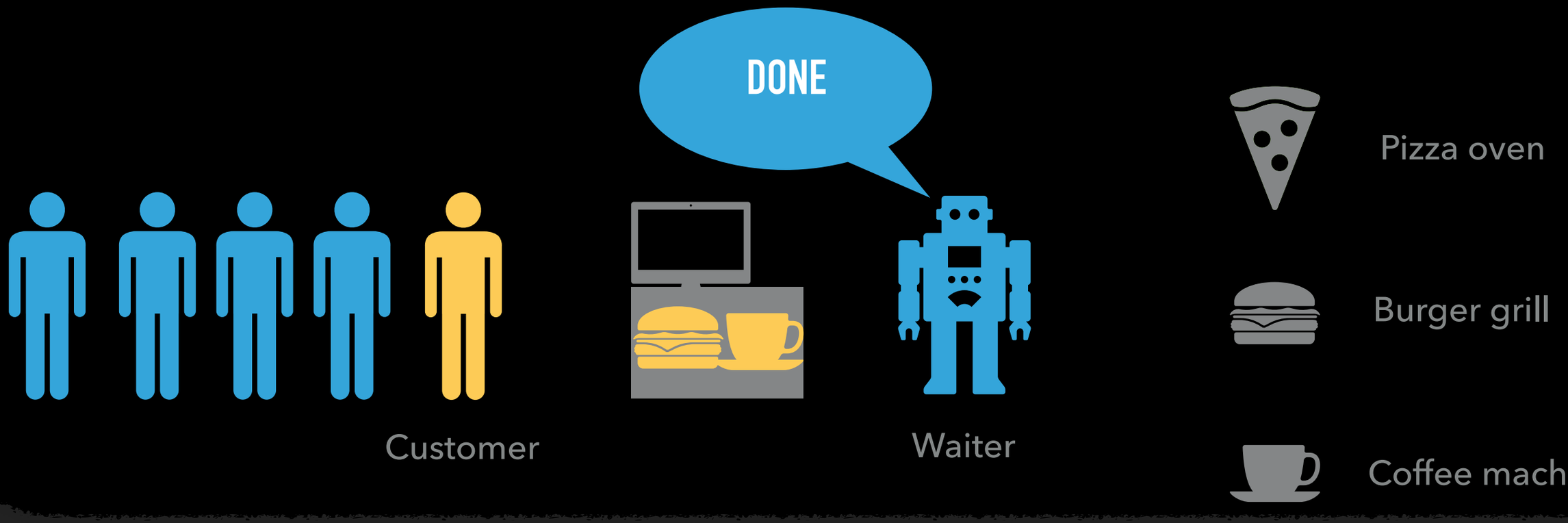
- Decode instruction into µOPs
- Schedule µOPs
- run 1st µOP
- run 2nd µOP (serial execution)

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



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

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



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

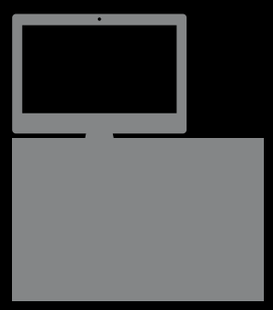
I.e. first the burger, then the coffee

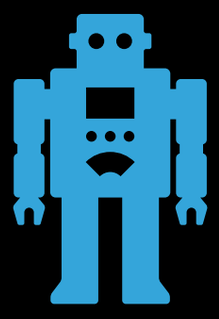
- PRO: Easy to implement and understand
- CON: Slow because resources[3] not utilised fully

[1] customer == CPU instruction [2] part == µOP - micro operation [3] oven, grill, coffee machine

CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION

Customer





Waiter

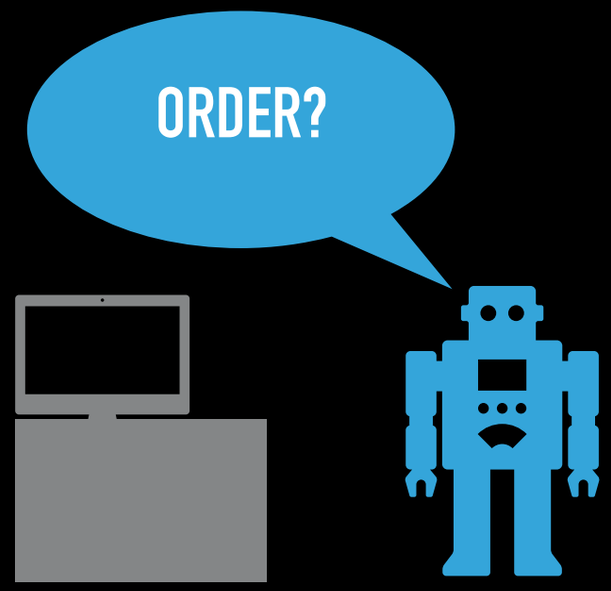
Pizza oven

Burger grill

Coffee machine

CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION

Customer



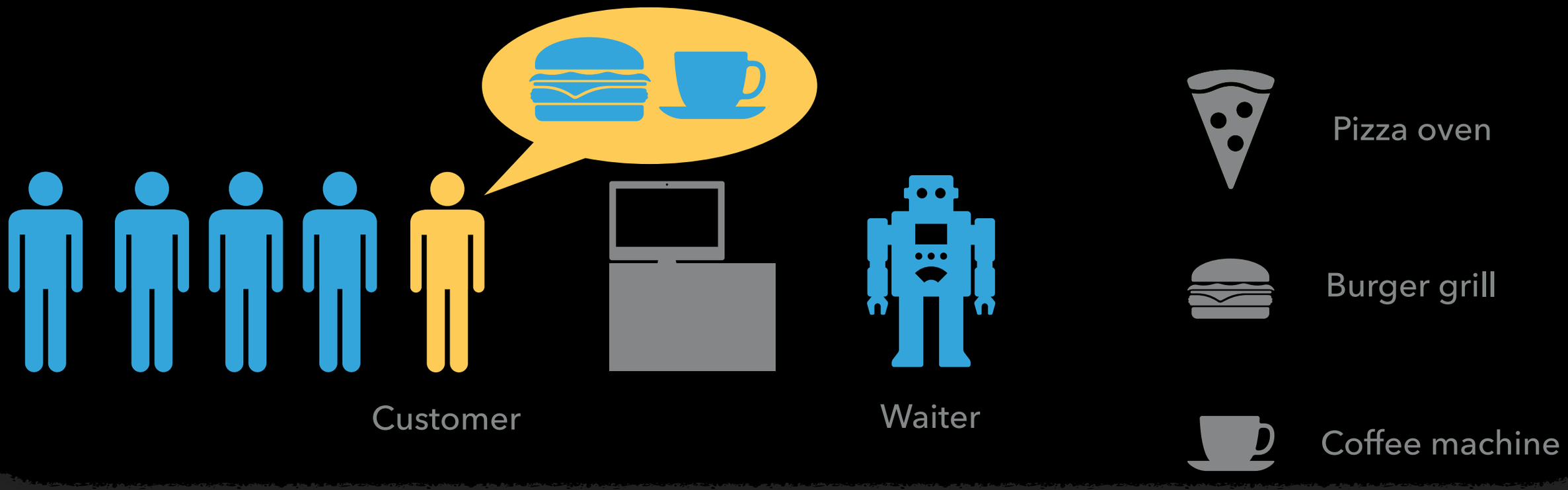
Waiter

Pizza oven

Burger grill

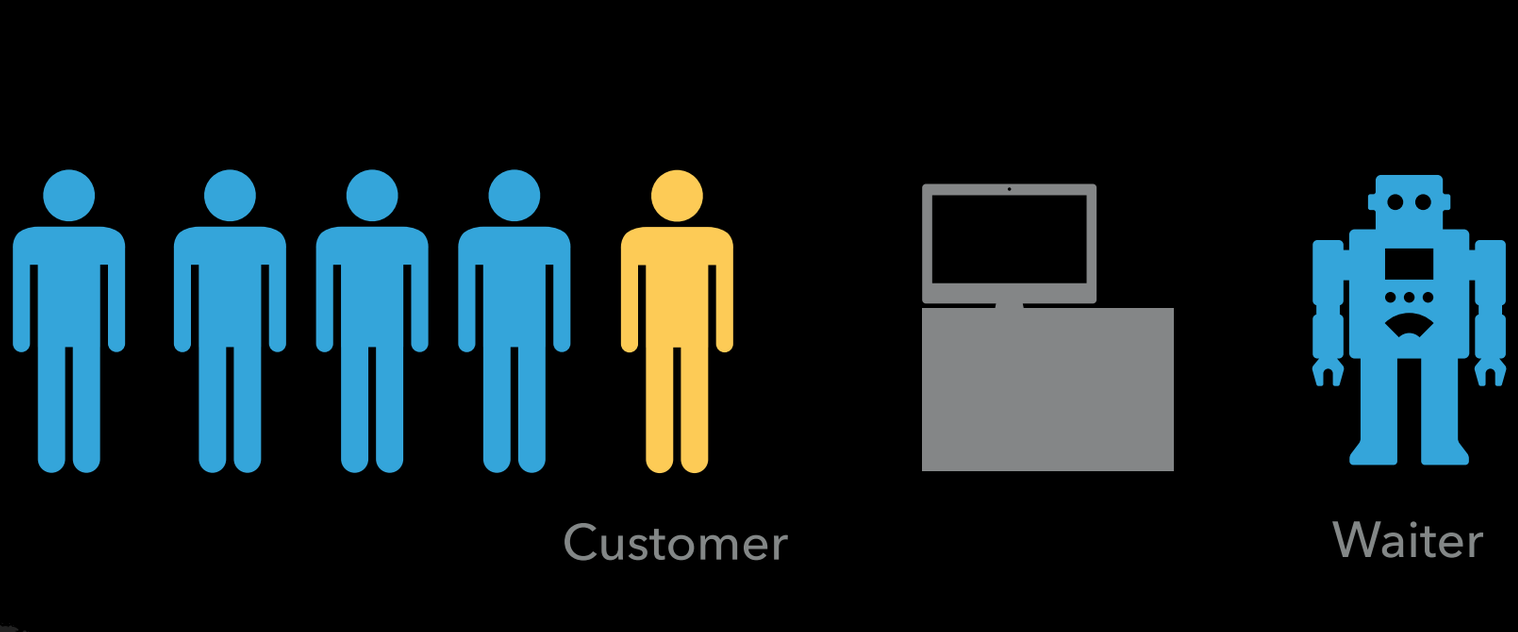
Coffee machine

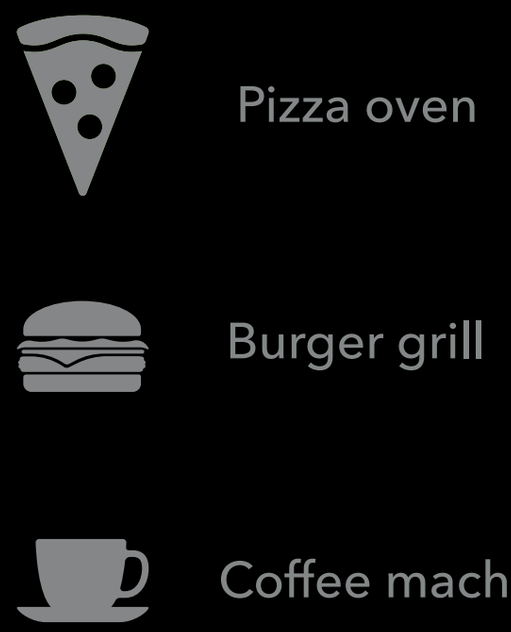
CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION



- Decode instruction into μOPs

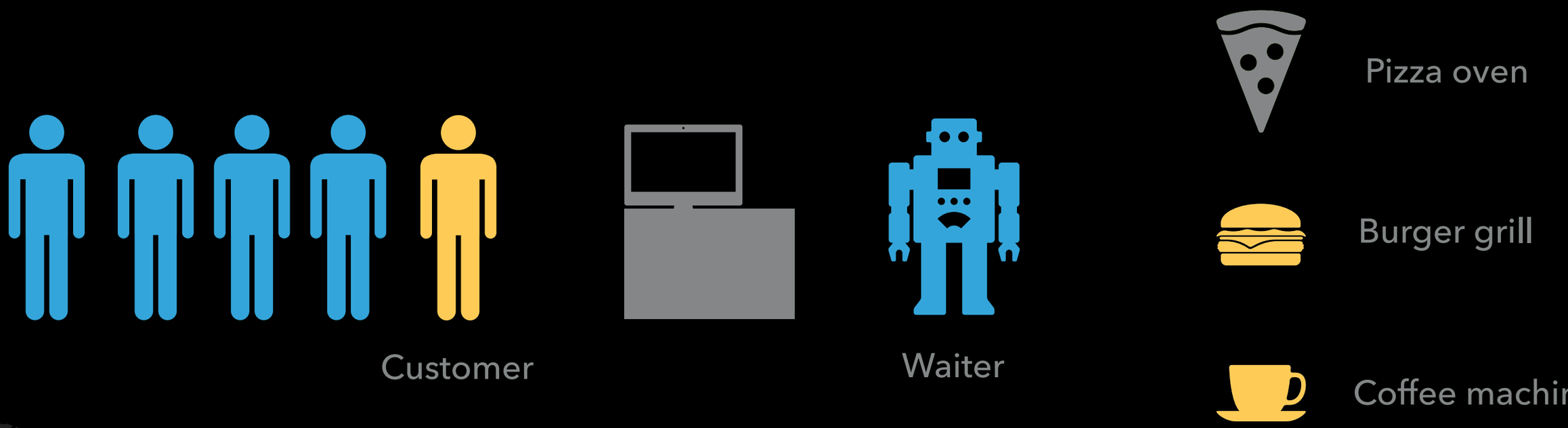
CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION





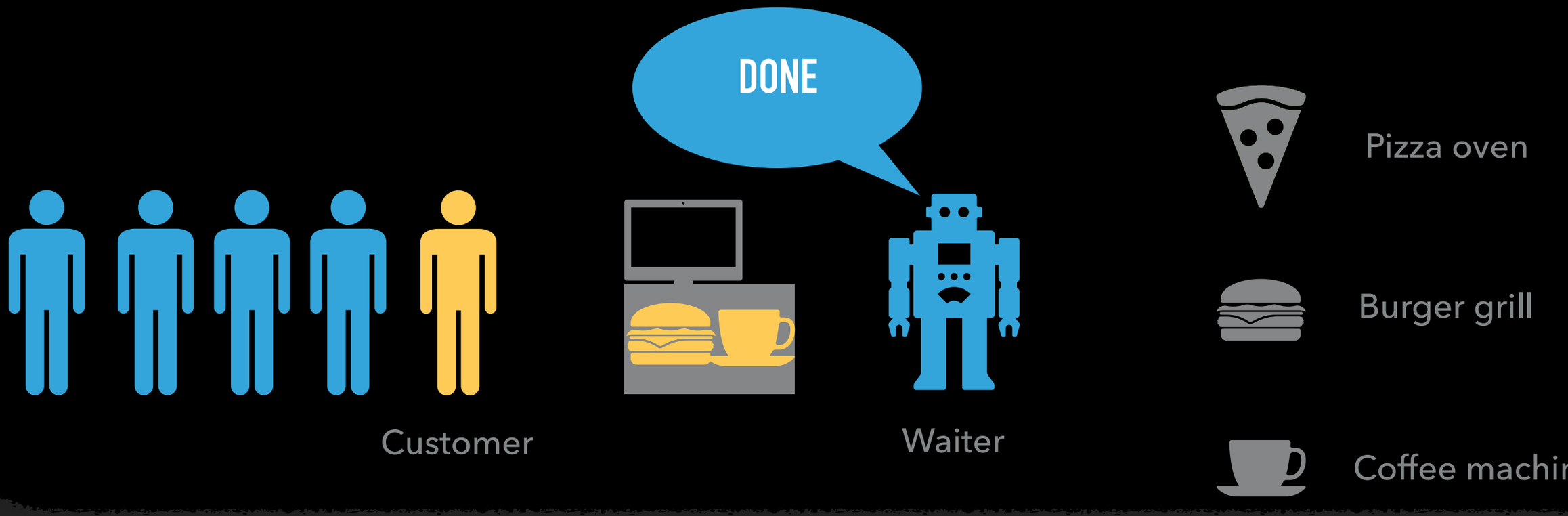
- Decode instruction into µOPs
- Schedule µOPs

CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION



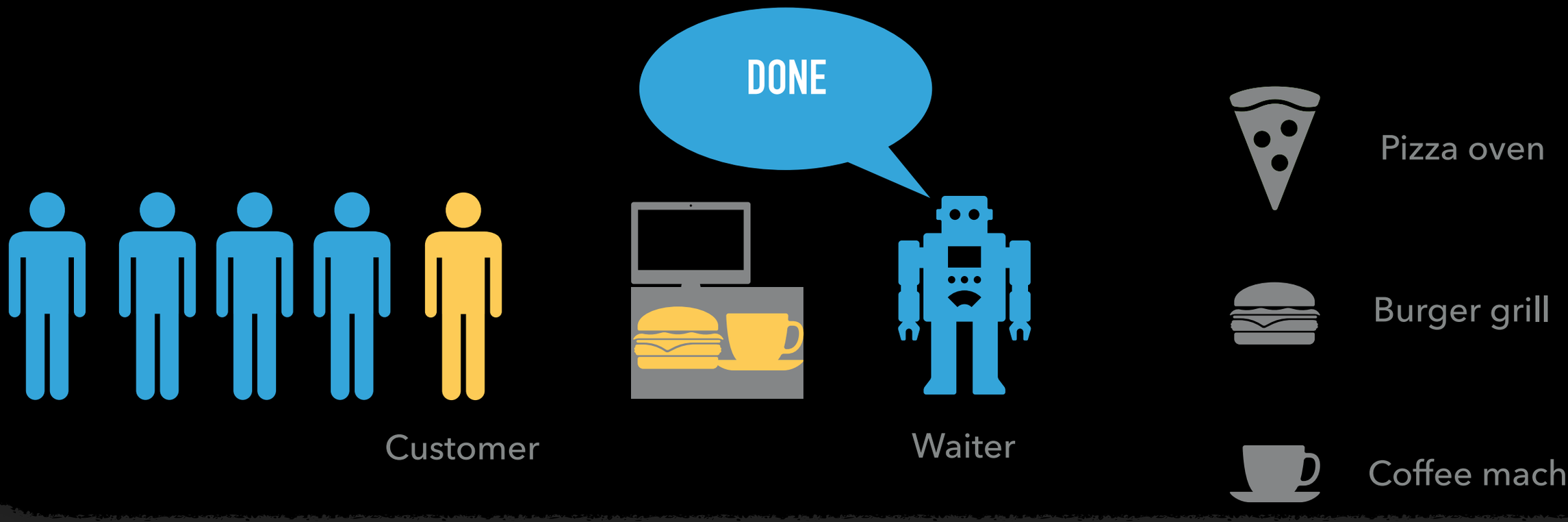
- Decode instruction into μOPs
- Schedule μOPs
- run 1st μOP and 2nd μOP (parallel execution)

CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION



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

CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION

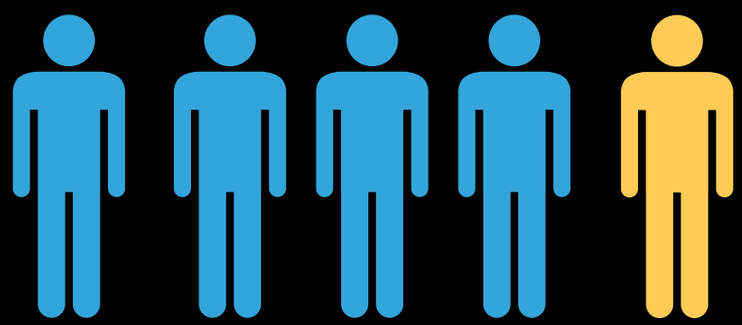


- One customer[1] after another (**in order**)
- Each part of the order [2] executed **in parallel**

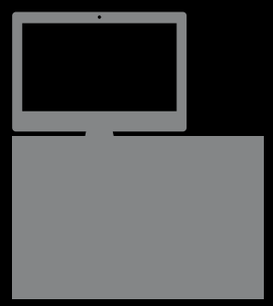
I.e. burger and coffee prepared at the same time

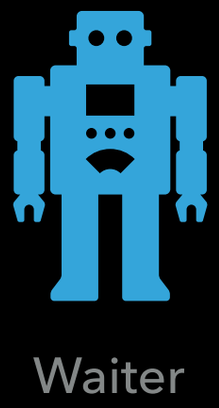
- PRO: **Faster bc. of better resource utilisation.**
- CON: Still not perfect, more complex

CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



Customer





Waiter

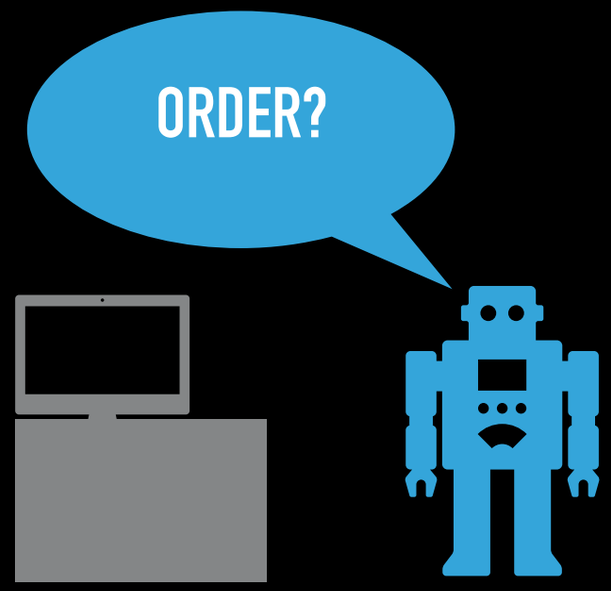
Pizza oven

Burger grill

Coffee machine

CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION

Customer



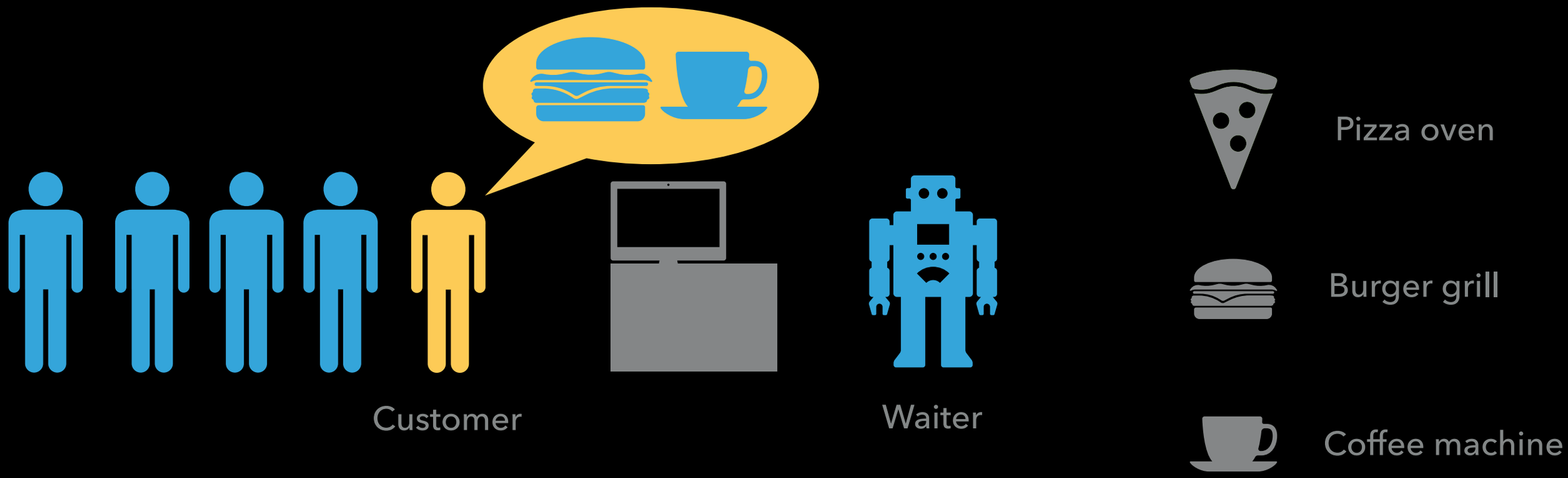
Waiter

Pizza oven

Burger grill

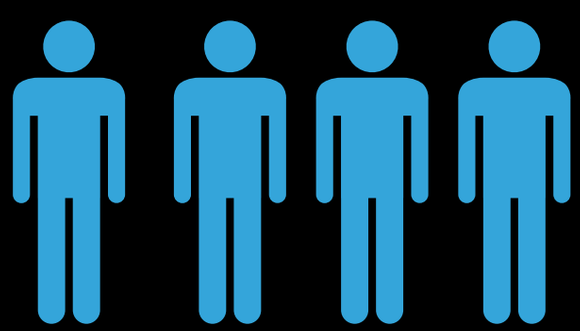
Coffee machine

CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION

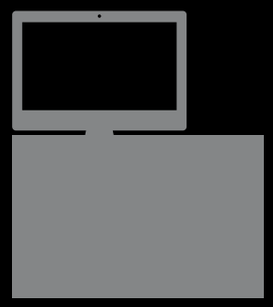


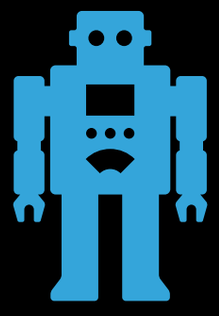
CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION

CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



Customer





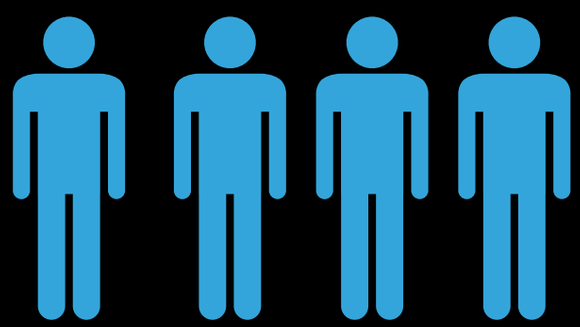
Waiter

Pizza oven

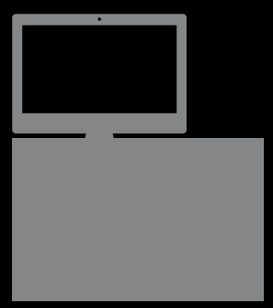
Burger grill

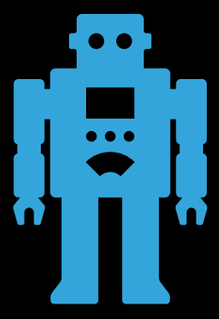
Coffee machine

CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



Customer





Waiter

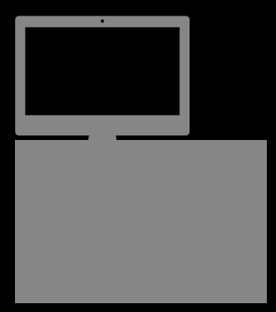
Pizza oven

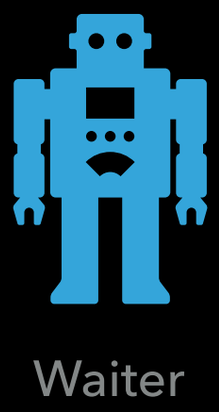
Burger grill

Coffee machine

CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION

Customer





Waiter

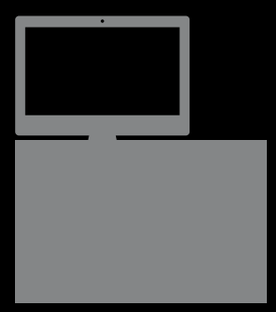
Pizza oven

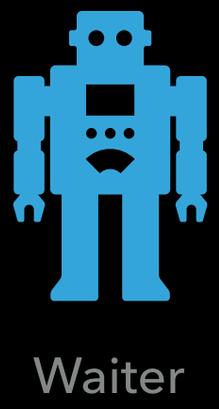
Burger grill

Coffee machine

CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION

Customer





Waiter

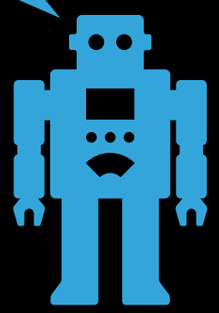
Pizza oven

Burger grill

Coffee machine

CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION

Customer

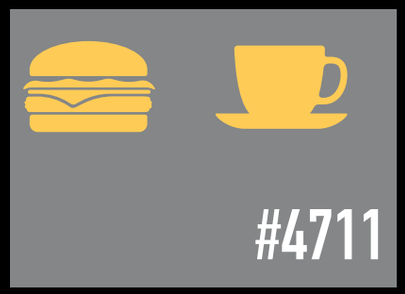


Waiter

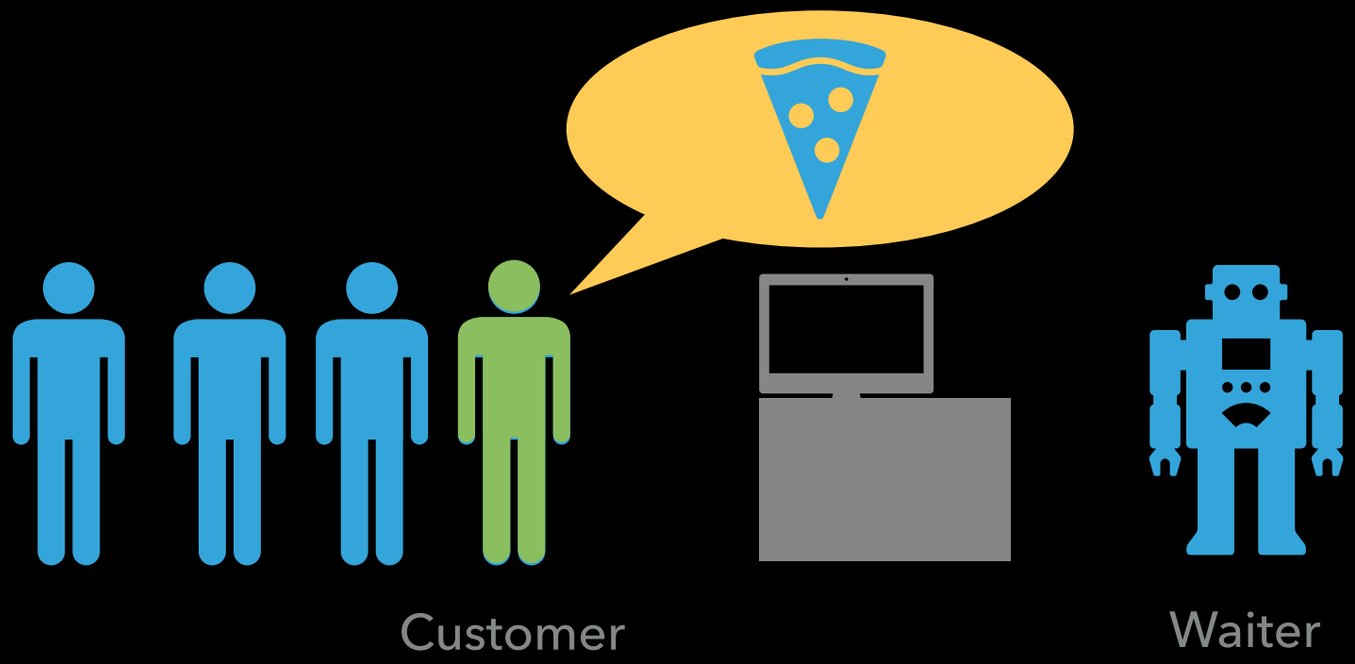
Pizza oven

Burger grill

Coffee machine



CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION




Pizza oven


Burger grill

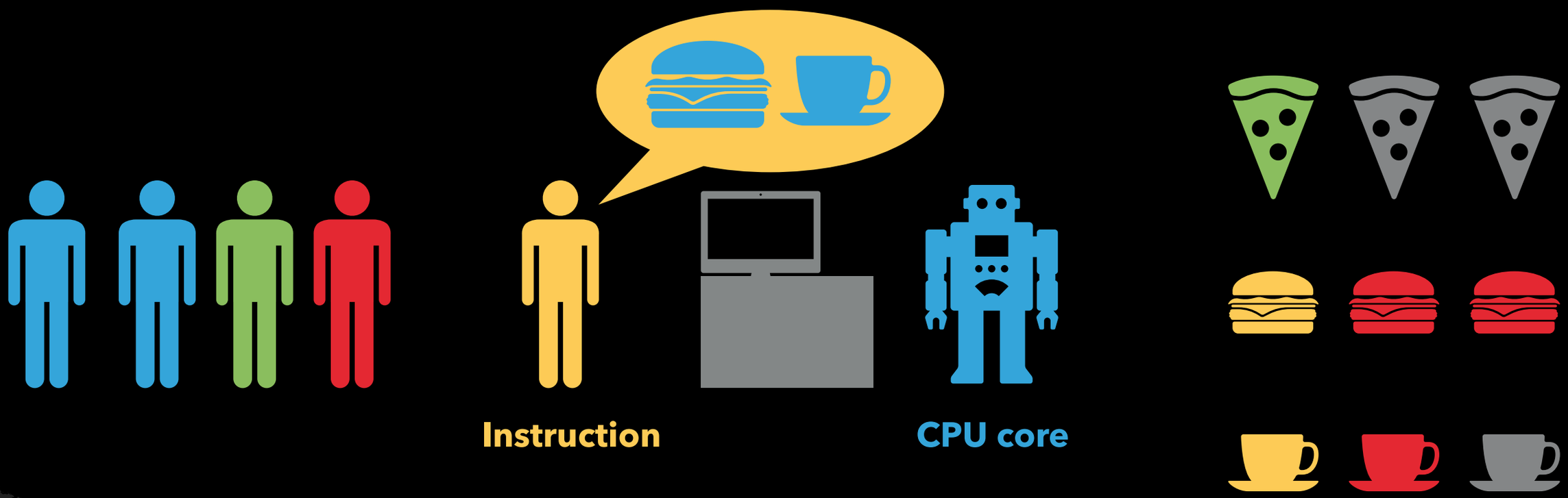

Coffee machine

- Multiple customers' orders **executed in parallel** and **delivered** (retired) **in order**

I.e. multiple orders prepared at the same time

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

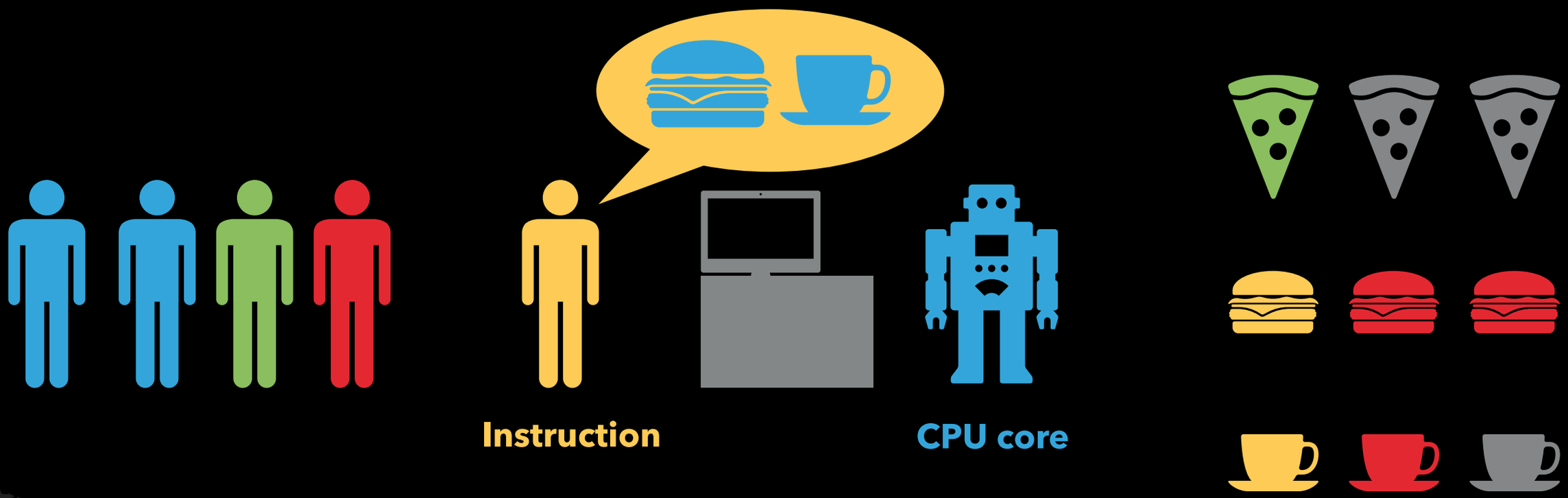
CONFIDENTIAL BURGERS INC.



Adding more resources increase parallelism & throughput

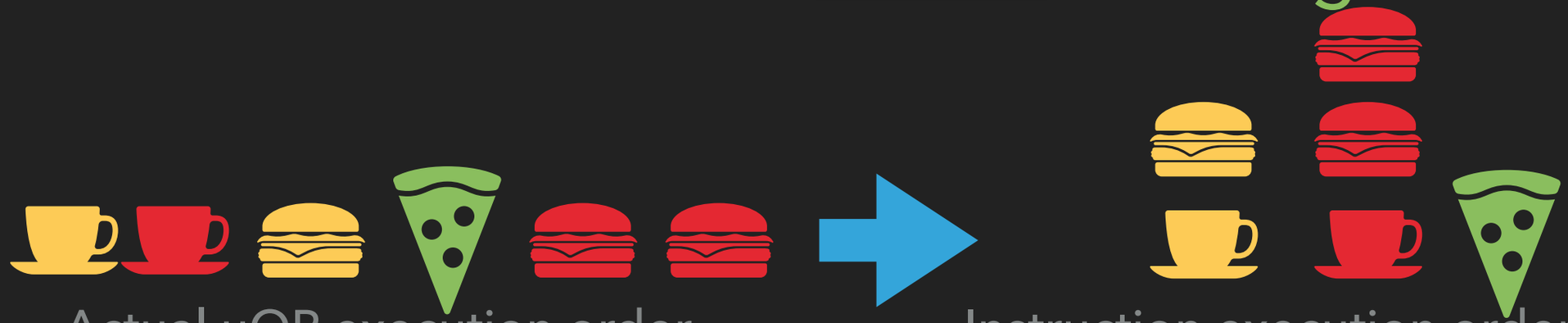
This is all on one CPU core

CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



The green instruction will finish before the red instruction

The CPU ensures that red is seen before green



MELTDOWN

Meltdown basically works like this:

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

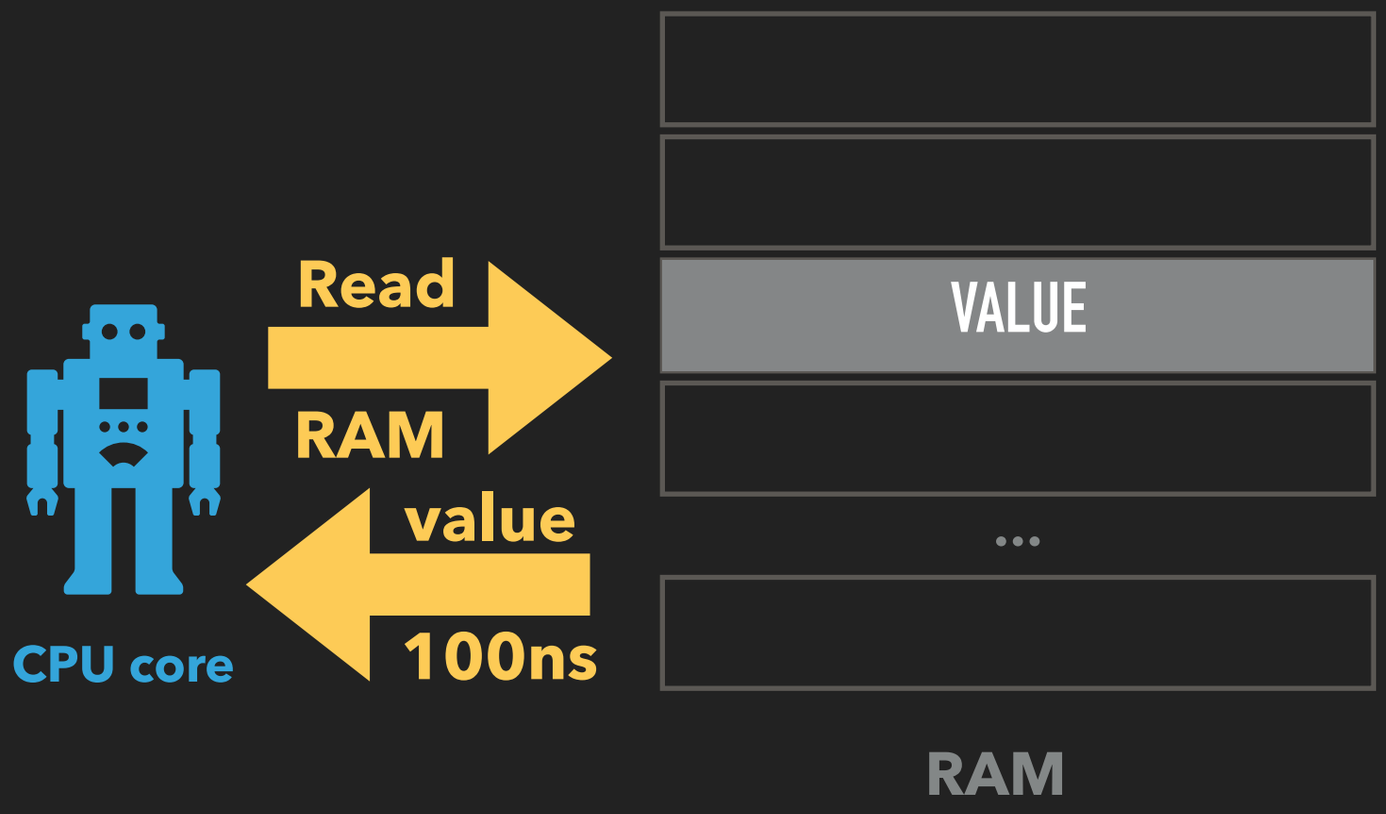
MELTDOWN

"Stashing" and "retrieving" the secret works via *side channels*.

Side channels are *observable side effects* of actions.

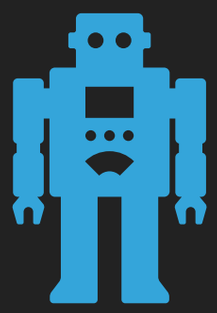
- READ secret from forbidden address
- Stash away secret by caching a memory location that depends on the secret
- Retrieve secret by finding which memory location is cached

MELTDOWN: STASHING AWAY – SIDECHANNEL

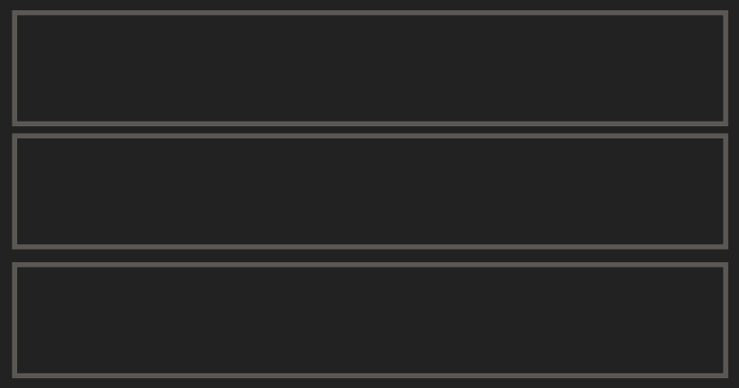


- Data is stored in RAM
- RAM is very slow
- Reading one byte stalls the CPU for hundreds of µOPs

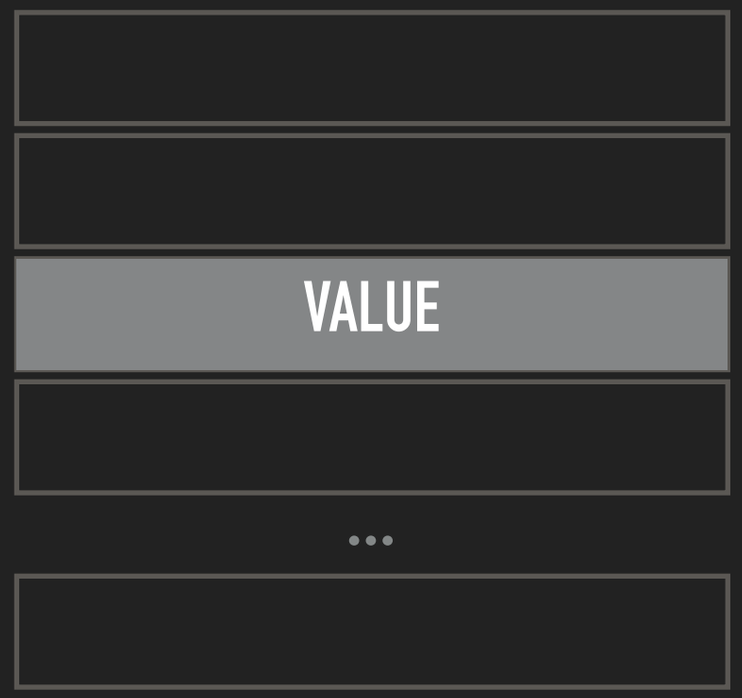
MELTDOWN: STASHING AWAY – SIDECHANNEL



CPU core



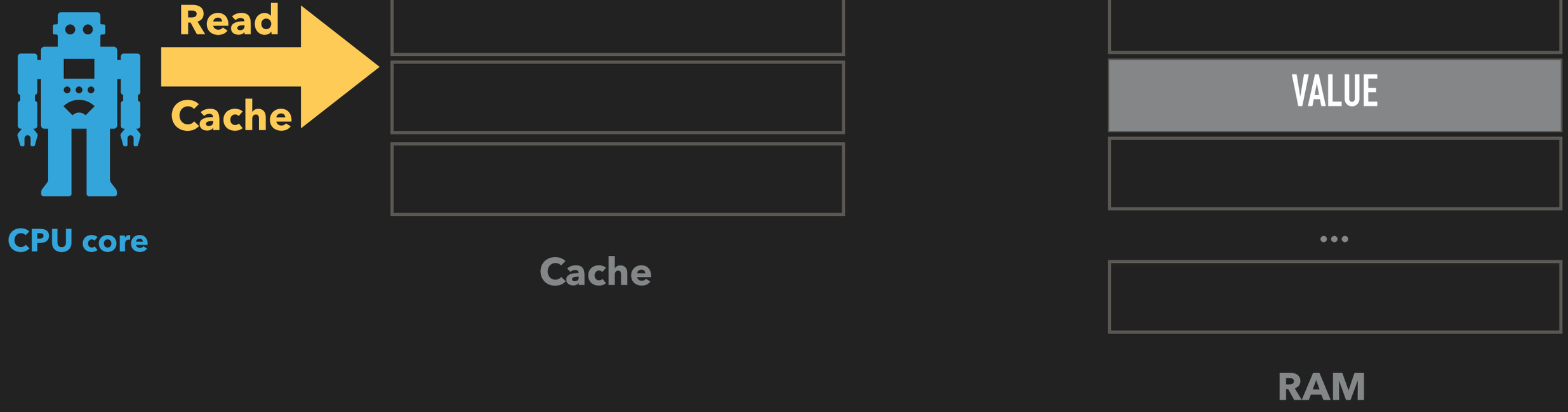
Cache



RAM

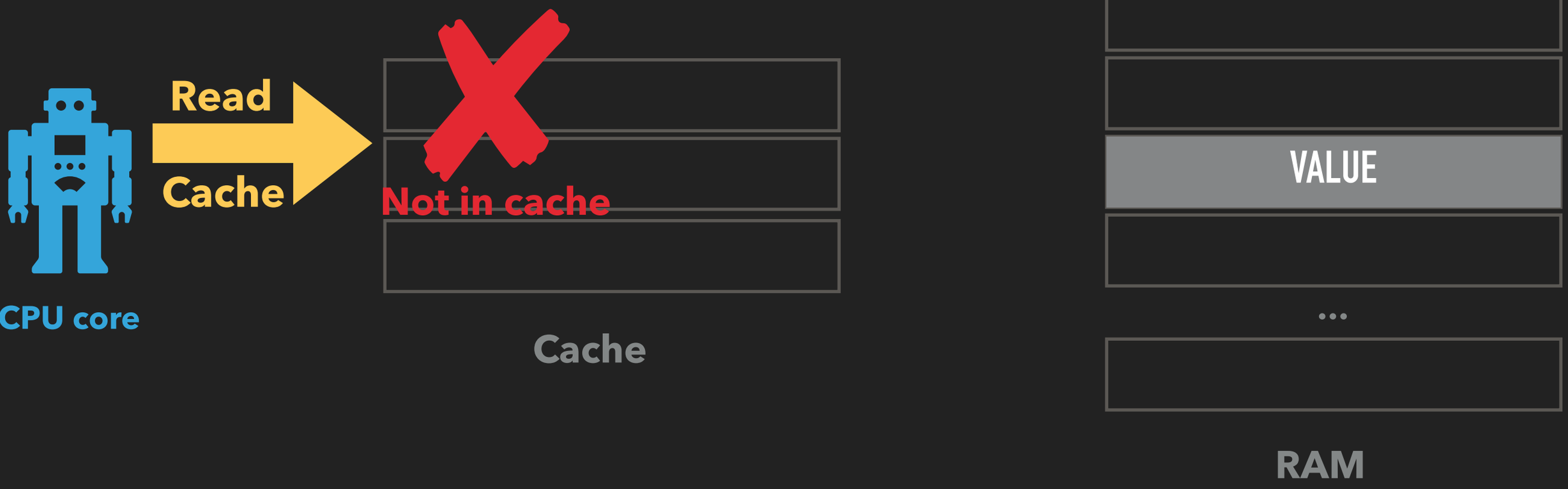
- Reading one byte stalls the CPU for hundreds of µOPs
- CPU caches considerably speed this up
- E.g. reading cached takes 3ns, reading uncached 103ns

MELTDOWN: STASHING AWAY – SIDECHANNEL



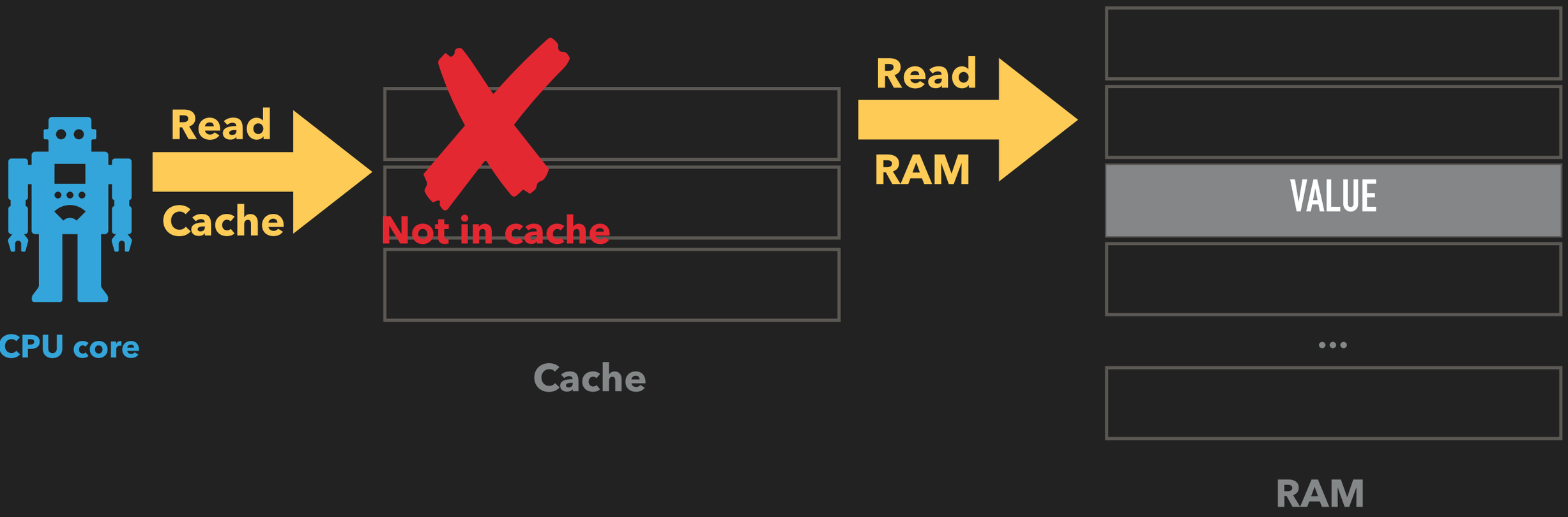
- Reading one byte stalls the CPU for hundreds of µOPs
- CPU caches considerably speed this up
- E.g. reading cached takes 3ns, reading uncached 103ns

MELTDOWN: STASHING AWAY – SIDECHANNEL



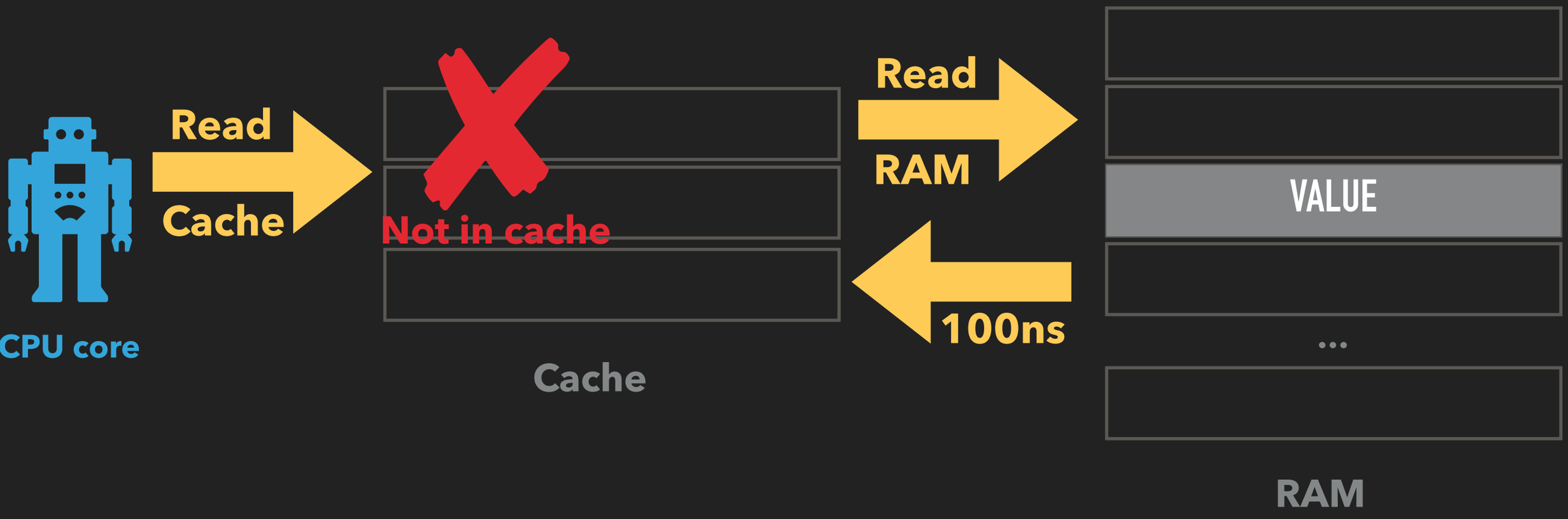
- Reading one byte stalls the CPU for hundreds of µOPs
- CPU caches considerably speed this up
- E.g. reading cached takes 3ns, reading uncached 103ns

MELTDOWN: STASHING AWAY – SIDECHANNEL



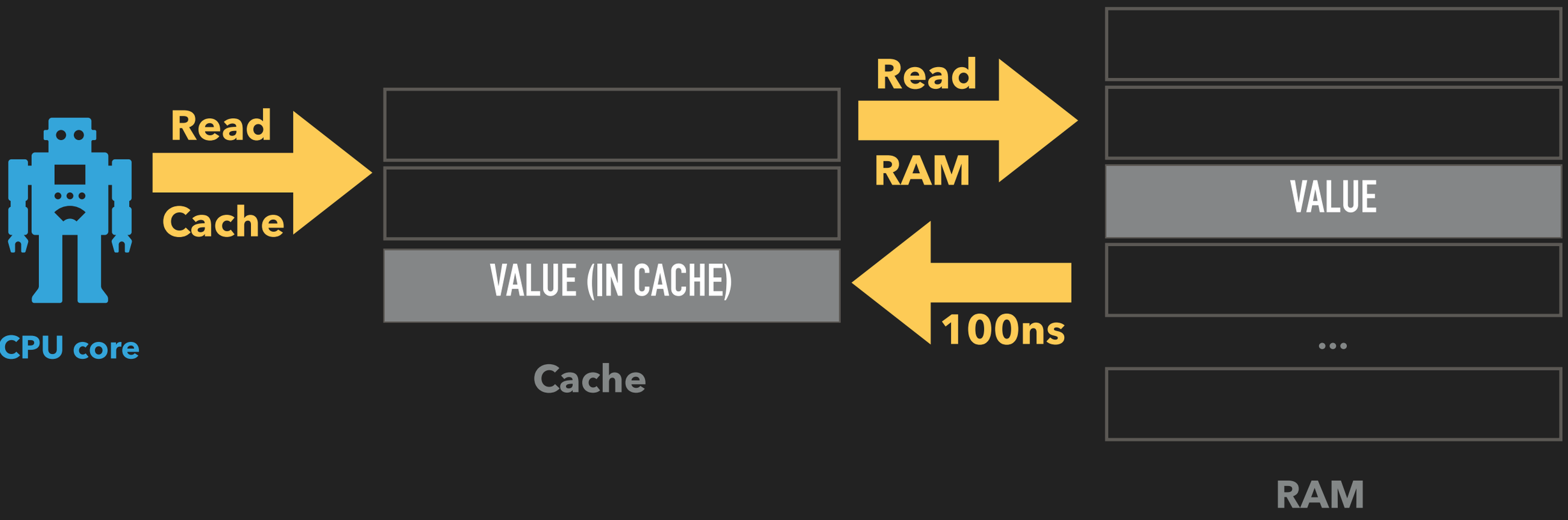
- Reading one byte stalls the CPU for hundreds of µOPs
- CPU caches considerably speed this up
- E.g. reading cached takes 3ns, reading uncached 103ns

MELTDOWN: STASHING AWAY – SIDECHANNEL



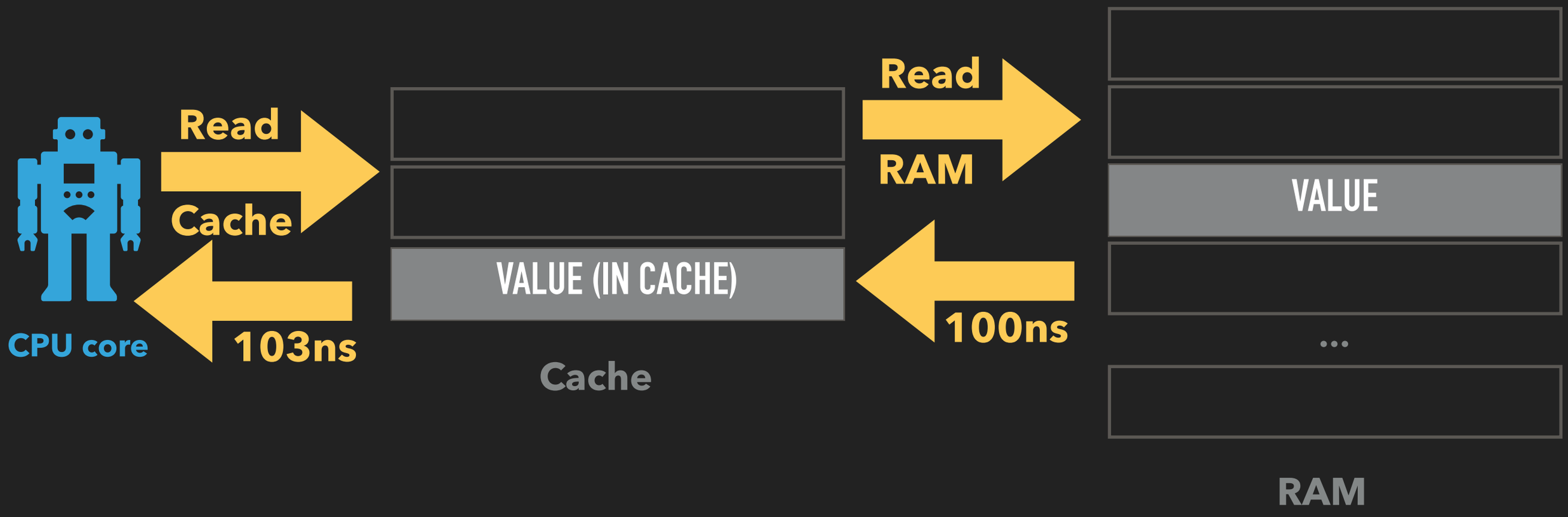
- Reading one byte stalls the CPU for hundreds of µOPs
- CPU caches considerably speed this up
- E.g. reading cached takes 3ns, reading uncached 103ns

MELTDOWN: STASHING AWAY – SIDECHANNEL



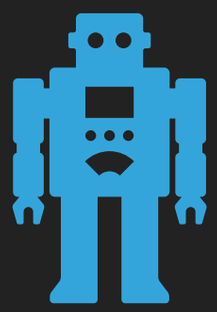
- Reading one byte stalls the CPU for hundreds of µOPs
- CPU caches considerably speed this up
- E.g. reading cached takes 3ns, reading uncached 103ns

MELTDOWN: STASHING AWAY – SIDECHANNEL



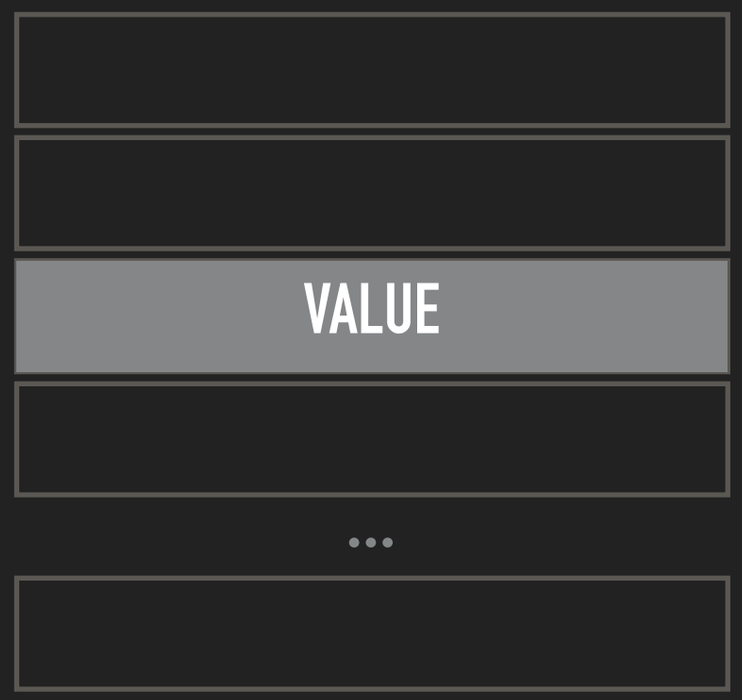
- Reading one byte stalls the CPU for hundreds of µOPs
- CPU caches considerably speed this up
- E.g. reading cached takes 3ns, reading uncached 103ns

MELTDOWN: STASHING AWAY – SIDECHANNEL



CPU core

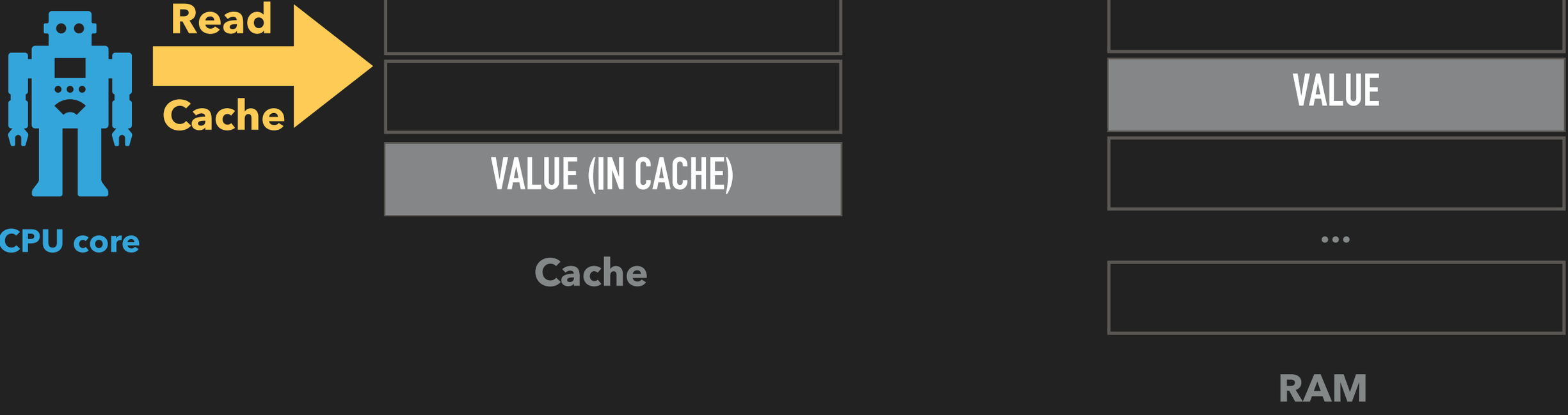
Cache



RAM

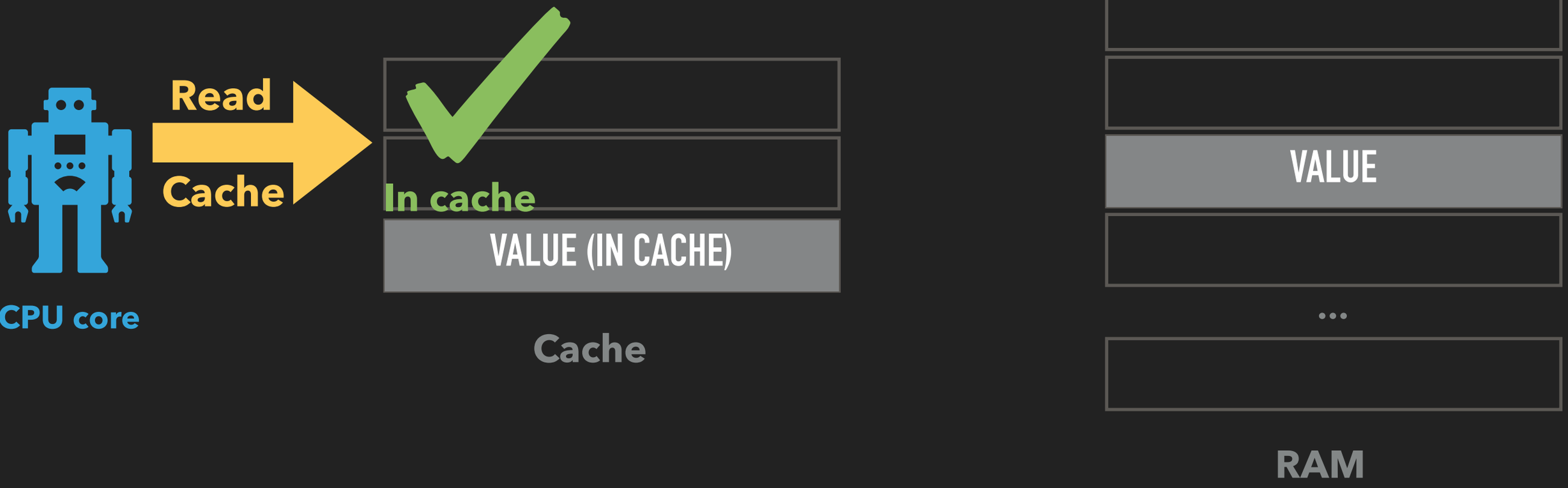
- Reading one byte stalls the CPU for hundreds of µOPs
- CPU caches considerably speed this up
- E.g. reading cached takes 3ns, reading uncached 103ns

MELTDOWN: STASHING AWAY – SIDECHANNEL



- 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

MELTDOWN: STASHING AWAY – SIDECHANNEL



- 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

MELTDOWN: STASHING AWAY – SIDECHANNEL

- 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

“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

“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.

In our burger example:

1. Customer orders a burger
2. Customer has not enough money
3. Customer does not get his burger

[1] Register: The CPUs scratchpad

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)

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 1 is only seen *iff* 1 succeeds.

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.

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 beca
that is only seen *iff* succe

Unless is able to hide the secre
attacker can find it later.

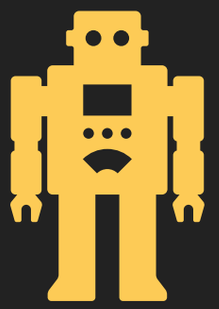
In our burger example:

1. **Customer orders a burger**
2. **Customer gets his burger**
3. **Customer has not enough money**
4. **Customer runs away with burger**

MELTDOWN

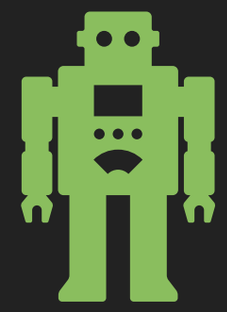
For Meltdown two actors are needed

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



Spy

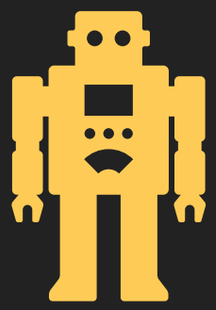
The **spy** will “steal” the secret and stash it away. The CPU will kill him for accessing the secret information.



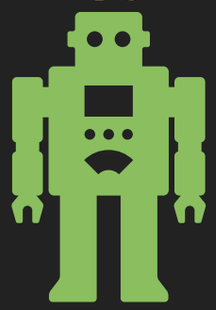
Collector

The **collector** will find the stashed away secret.

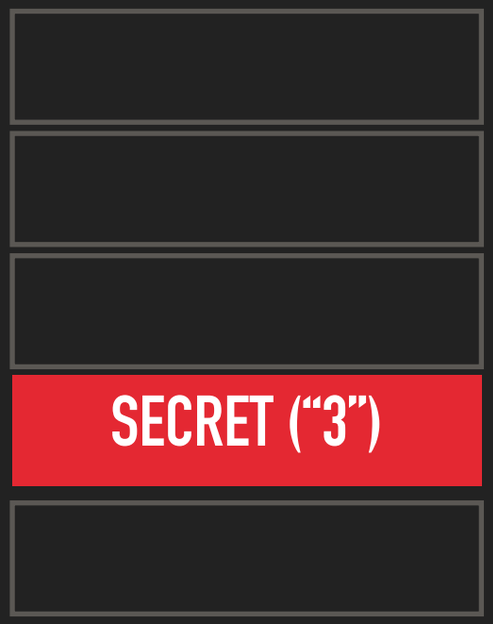
MELTDOWN: THE ATTACK



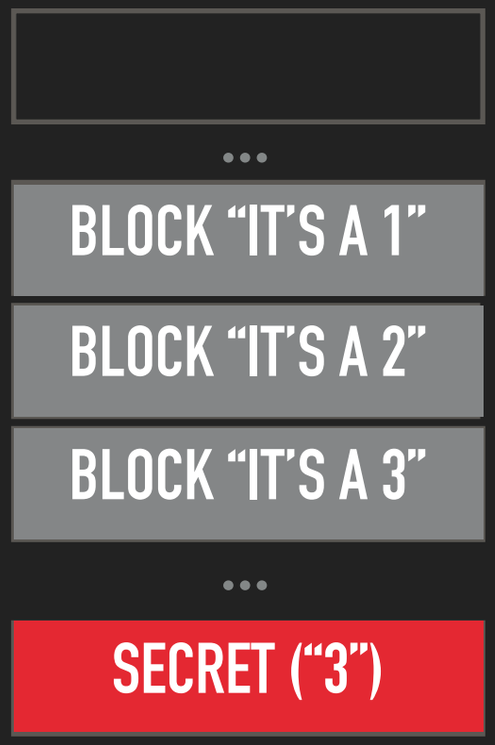
Spy



Collector

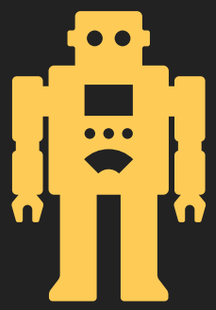


Cache



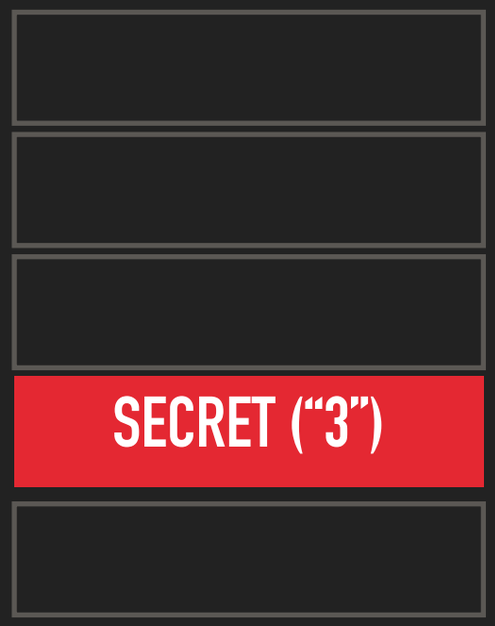
RAM

MELTDOWN: THE ATTACK

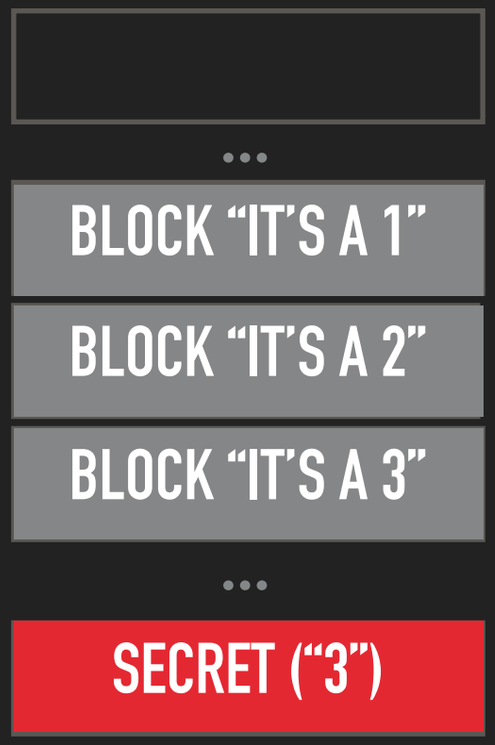


Spy

Collector



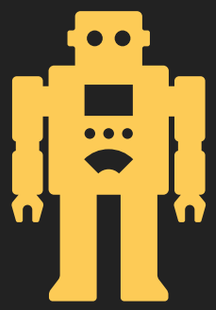
Cache



RAM

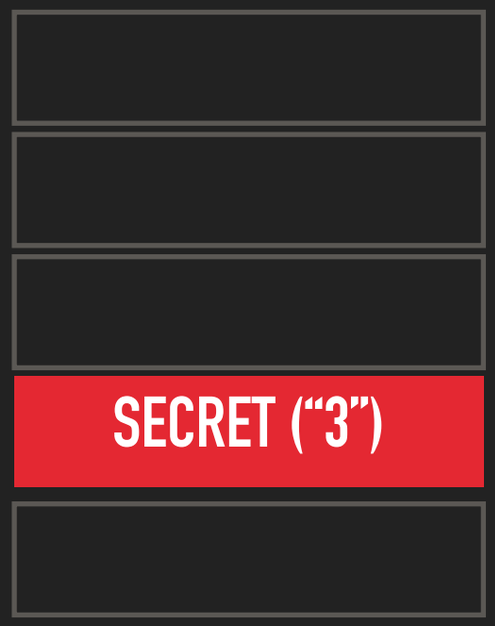
- Meltdown needs some precondition

MELTDOWN: THE ATTACK



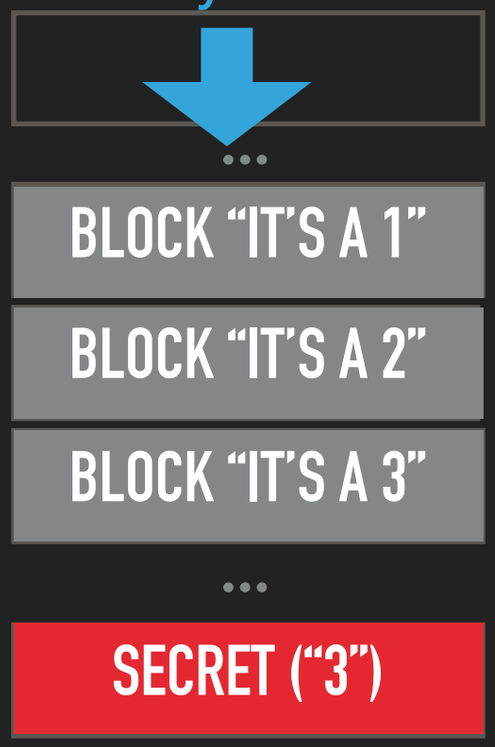
Spy

Collector



Cache

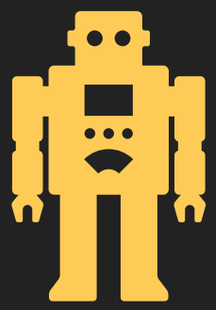
grey box:
memory block
tested by **Collector**



RAM

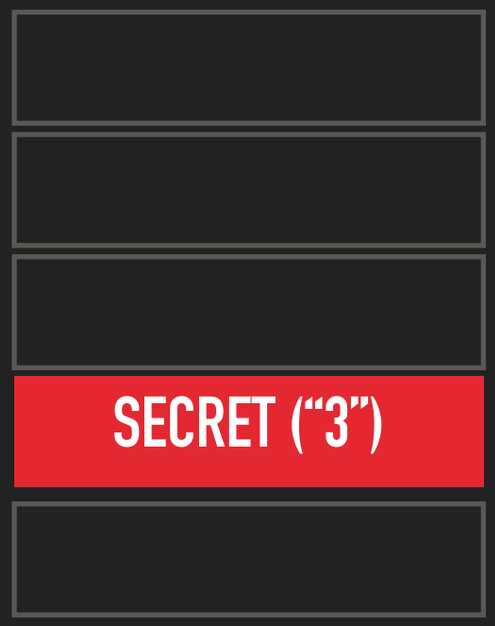
- Meltdown needs some precondition

MELTDOWN: THE ATTACK



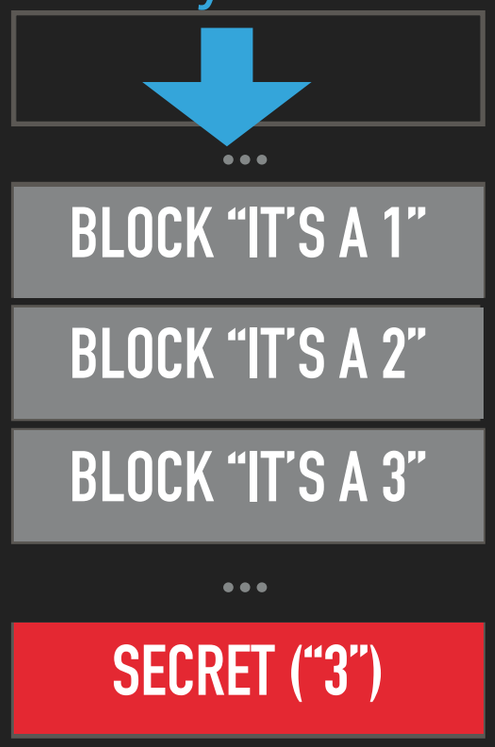
Spy

Collector



Cache

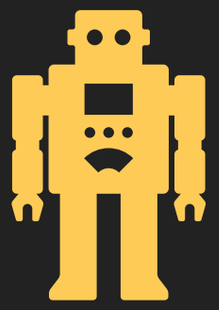
grey box:
memory block
tested by **Collector**



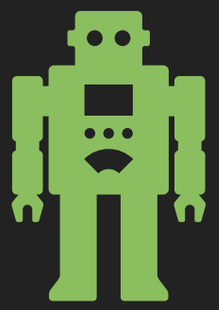
RAM

- Meltdown needs some precondition
- The **secret** is in the cache (value: 3)

MELTDOWN: THE ATTACK



Spy

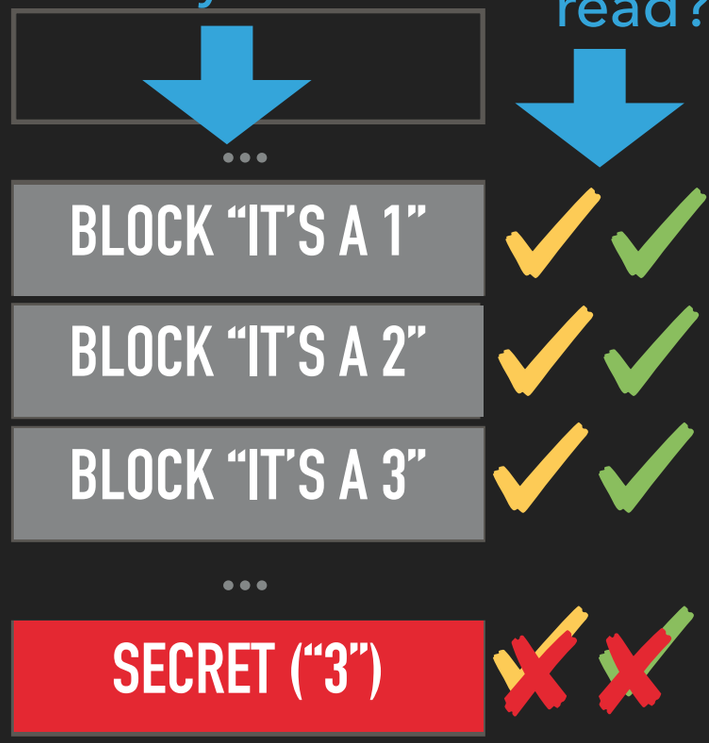


Collector

Cache

grey box:
memory block
tested by **Collector**

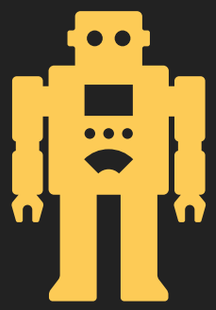
allowed to read?



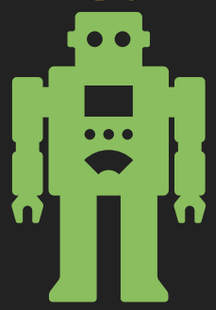
RAM

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

MELTDOWN: THE ATTACK

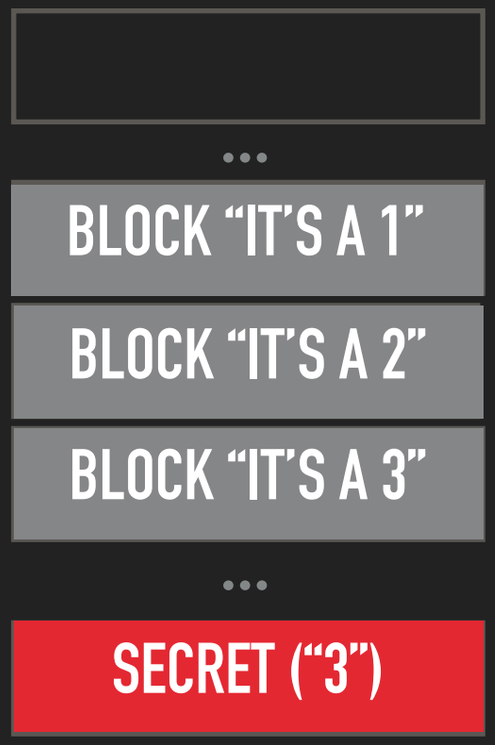


Spy



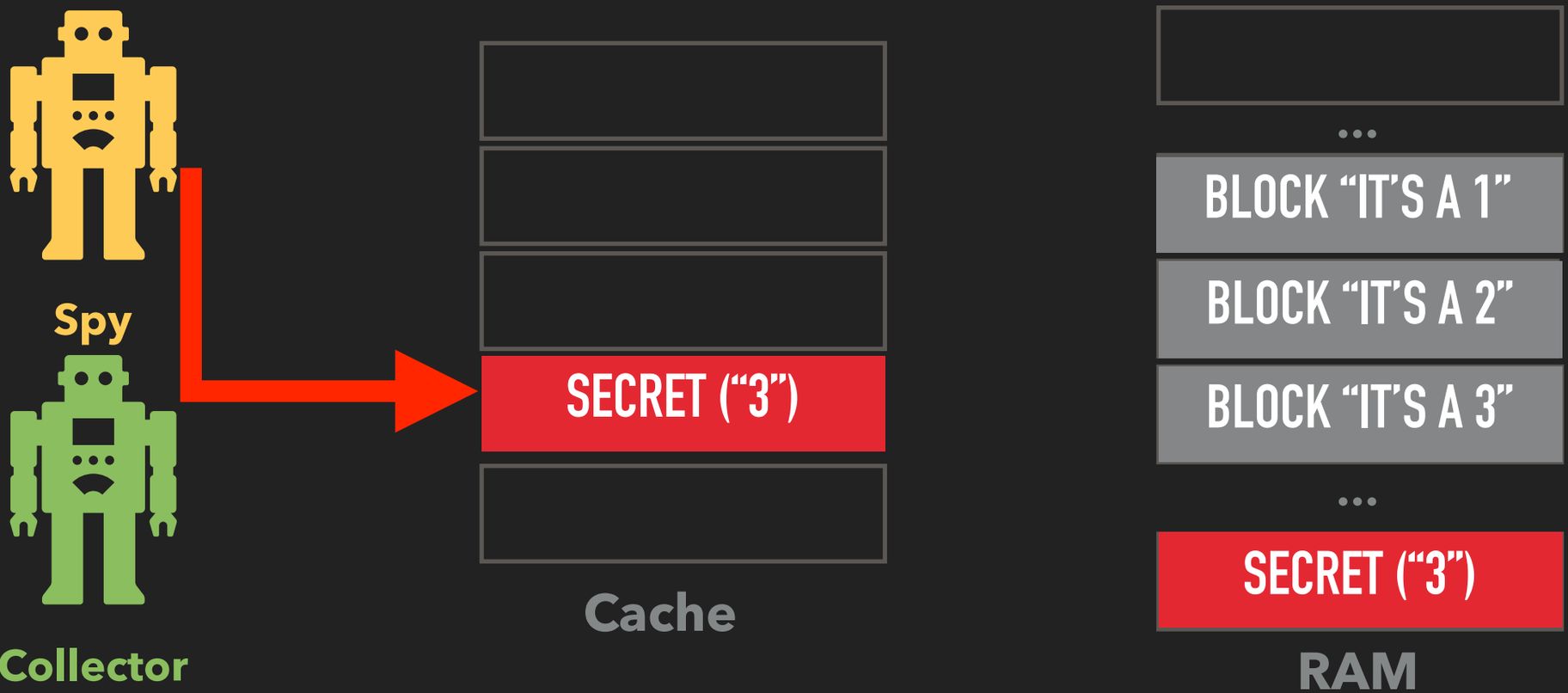
Collector

Cache



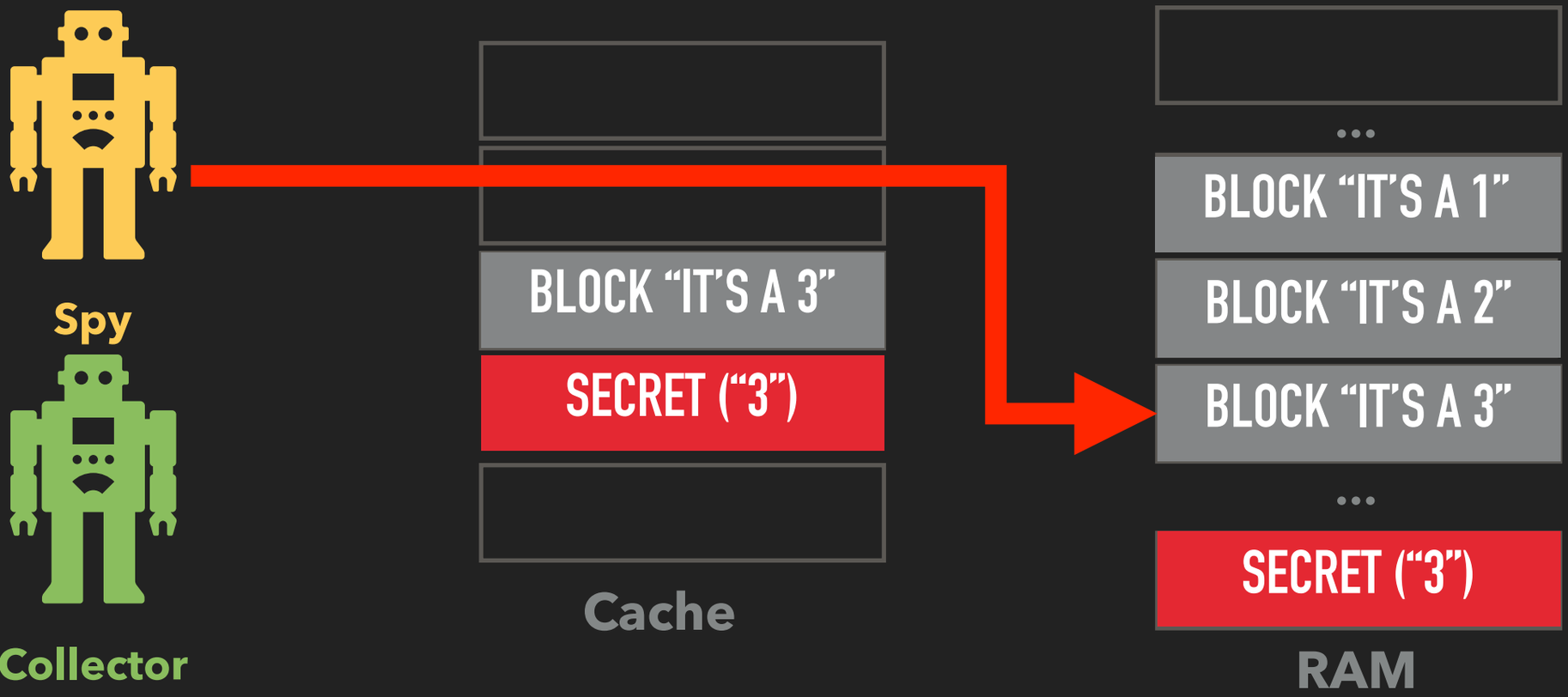
RAM

MELTDOWN: THE ATTACK



1. **Spy** will read the **secret**

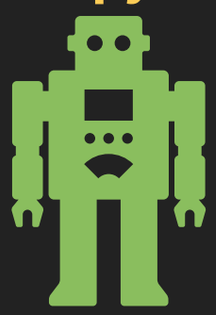
MELTDOWN: THE ATTACK



1. **Spy** will read the **secret**
2. Depending on the **value**, **Spy** will cache a grey block

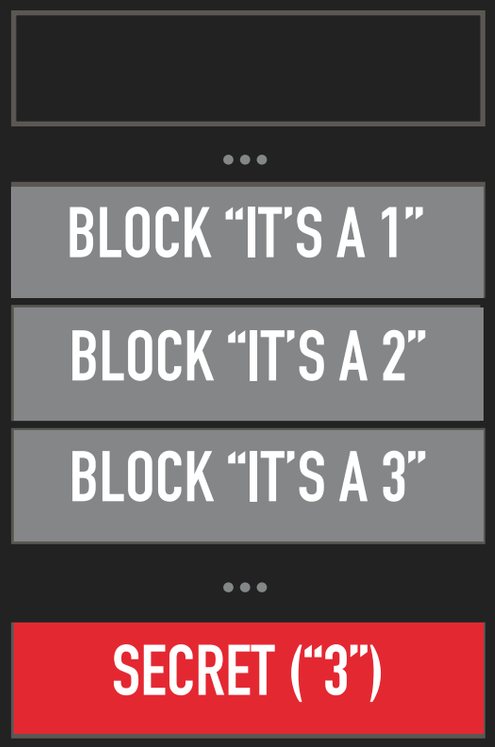
MELTDOWN: THE ATTACK

Spy



Collector

Cache

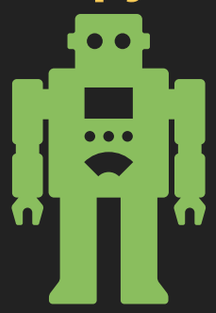


RAM

1. **Spy** will read the **secret**
2. Depending on the **value**, **Spy** will cache a grey block
3. CPU detects **Spys** access validation and terminates **Spy**

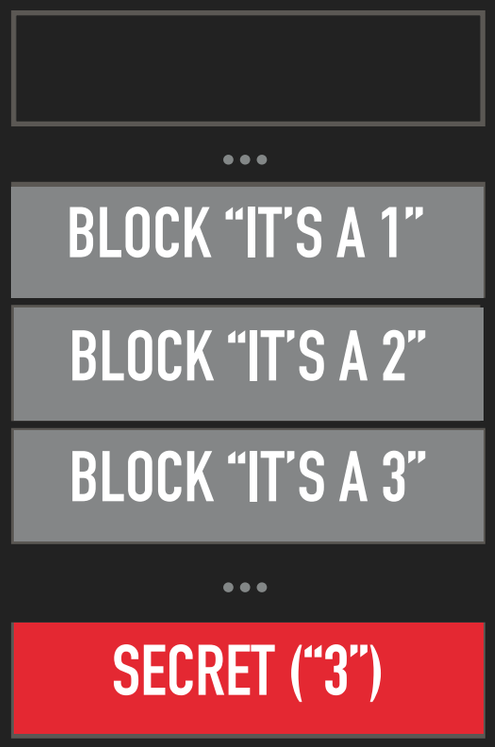
MELTDOWN: THE ATTACK

Spy



Collector

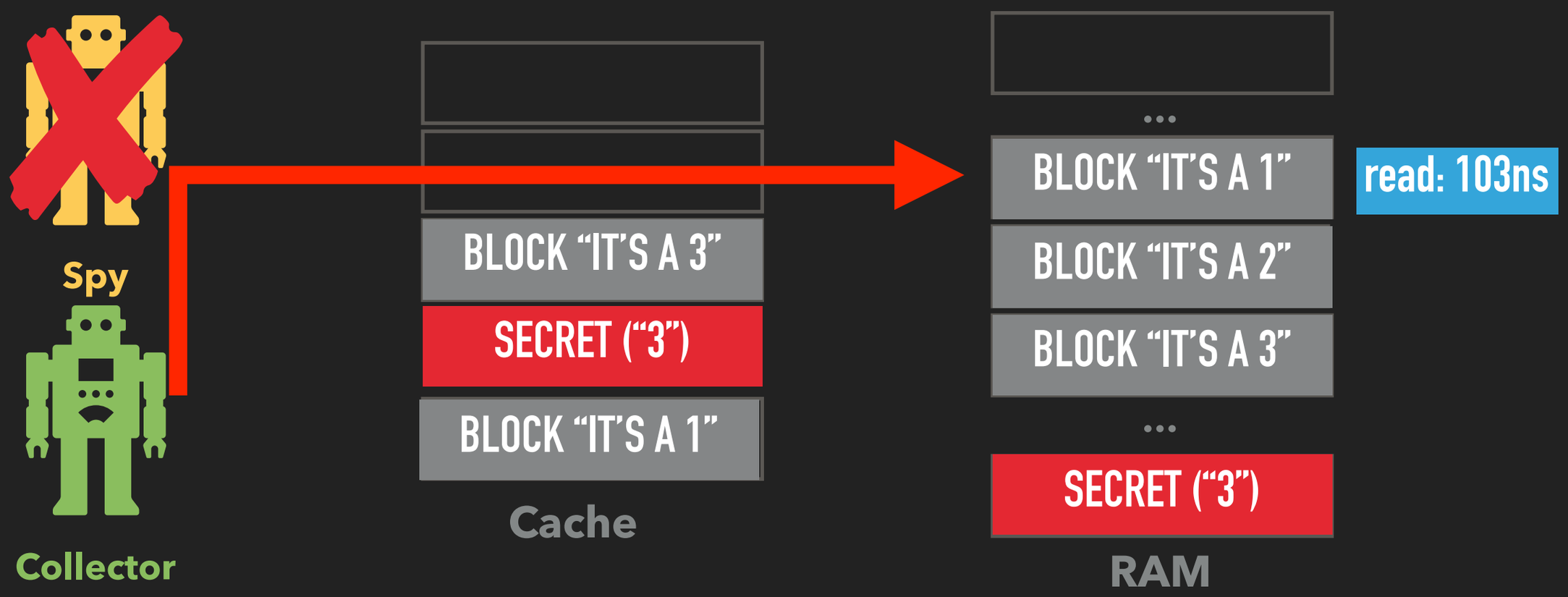
Cache



RAM

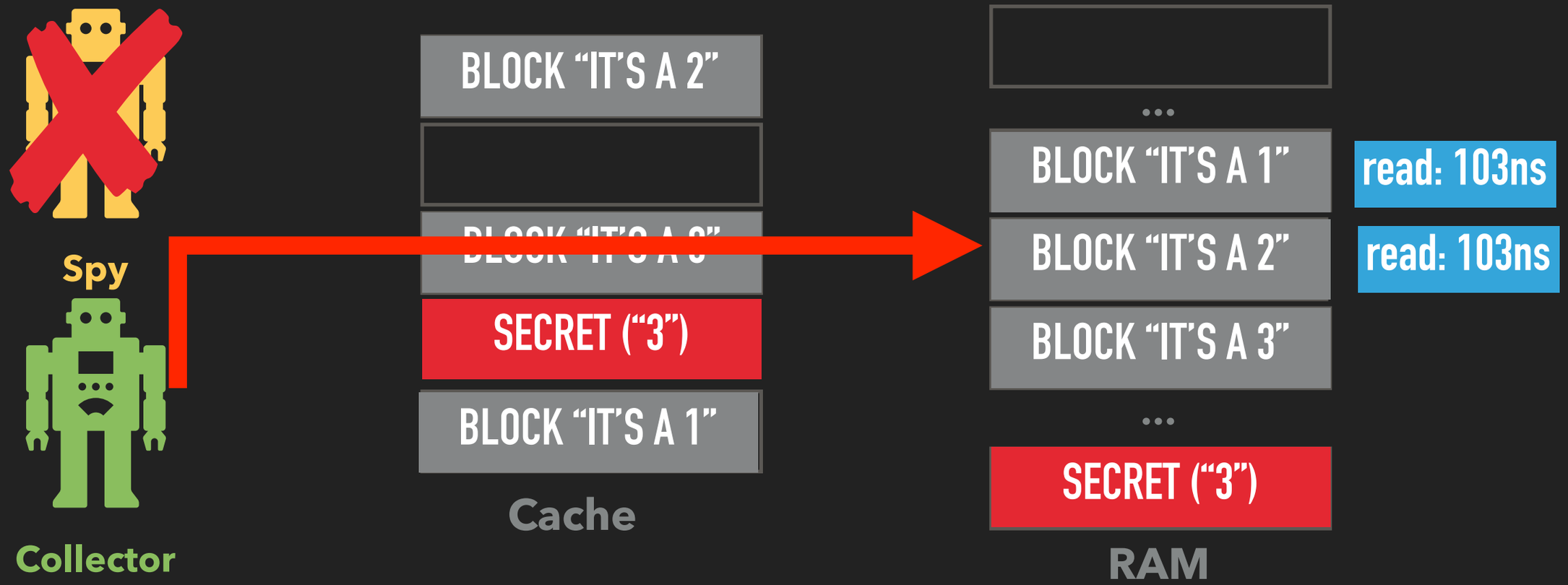
1. **Spy** will read the **secret**
2. Depending on the **value**, **Spy** will cache a grey block
3. CPU detects **Spys** access validation and terminates **Spy**
4. **Collector** now reads all grey blocks and stops the time

MELTDOWN: THE ATTACK



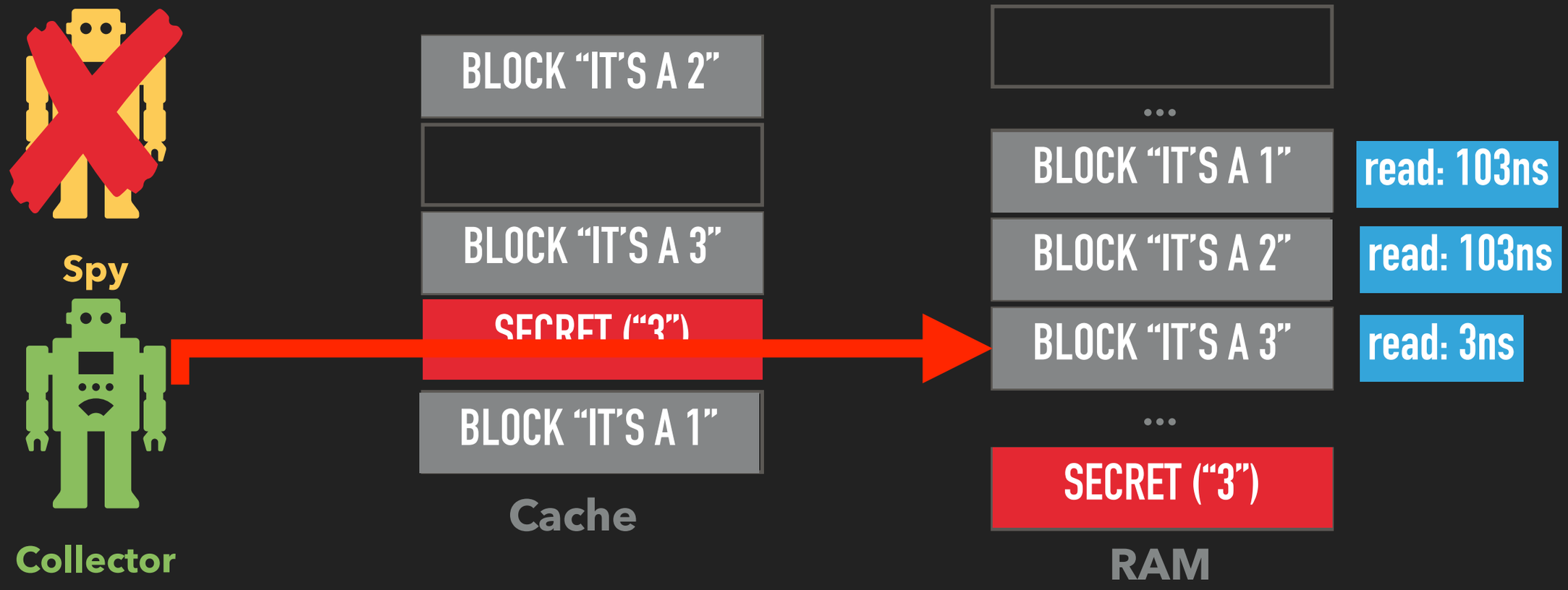
1. **Spy** will read the **secret**
2. Depending on the **value**, **Spy** will cache a grey block
3. CPU detects **Spys** access validation and terminates **Spy**
4. **Collector** now reads all grey blocks and stops the time

MELTDOWN: THE ATTACK



1. **Spy** will read the **secret**
2. Depending on the **value**, **Spy** will cache a grey block
3. CPU detects **Spys** access validation and terminates **Spy**
4. **Collector** now reads all grey blocks and stops the time

MELTDOWN: THE ATTACK



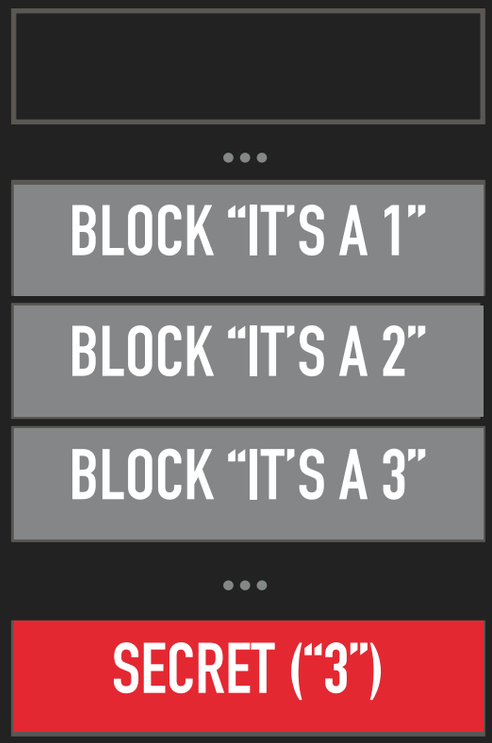
1. **Spy** will read the **secret**
2. Depending on the **value**, **Spy** will cache a grey block
3. CPU detects **Spys** access validation and terminates **Spy**
4. **Collector** now reads all grey blocks and stops the time

MELTDOWN: THE ATTACK

Spy

Collector

Cache



RAM

read: 103ns

read: 103ns

read: 3ns

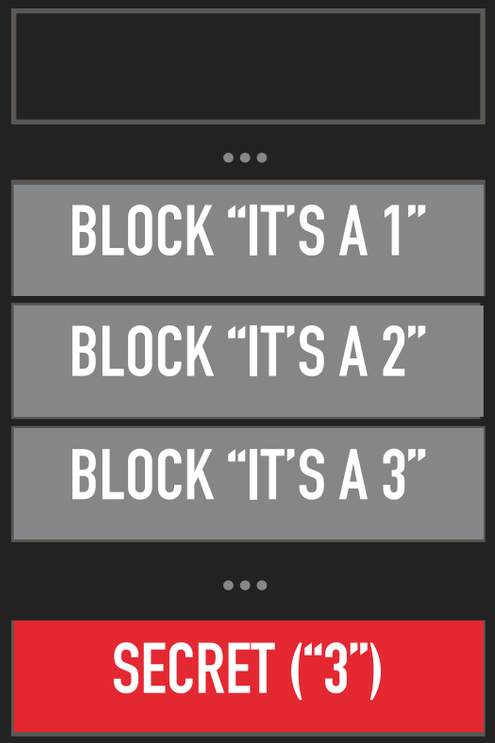
1. **Spy** will read the **secret**
2. Depending on the **value**, **Spy** will cache a grey block
3. CPU detects **Spys** access validation and terminates **Spy**
4. **Collector** now reads all grey blocks and stops the time
    1. Block “It’s a 3” will be the block read the fastest

MELTDOWN: THE ATTACK

Spy

Collector

Cache



RAM

read: 103ns

read: 103ns

read: 3ns

1. **Spy** will read the **secret**
2. Depending on the **value**, **Spy** will cache a grey block
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

MELTDOWN

Meltdown exploits two properties of modern CPUs

- Out of order *speculative* execution of OPs and µOPs
- Timing side channels for *speculatively executed* OPs

This allows an attacker to

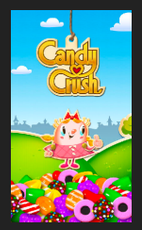
- Read all memory mapped[1] in a process
- This often includes all other processes memory
- This does NOT allow reading “outside of a VM”

[1] Virtual vs. physical memory is a subject for another time

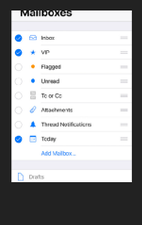
Q & A

https://github.com/neuhalje/presentation_meltdown_spectre

BOTH THE MELTDOWN AND SPECTRE LOGO ARE FREE TO USE, RIGHTS WAIVED VIA CCO. LOGOS ARE DESIGNED BY NATASCHA EIBL.
HTTPS://SPECTREATTACK.COM/

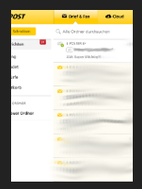


CANDY CRUSH LOGO FROM THE APP STORE © KING
HTTPS://DISCOVER.KING.COM/ABOUT/



APPLE MAIL SCREENSHOT © APPLE
HTTPS://SUPPORT.APPLE.COM/EN-GB/HT207213

SCREENSHOT © BUNDESNACHRICHTENDIENST
HTTPS://WWW.BND.BUND.DE/EN/_HOME/HOME_NODE.HTML



SCREENSHOT E-POST
HTTPS://PORTAL.E-POST.DE

ASSETS