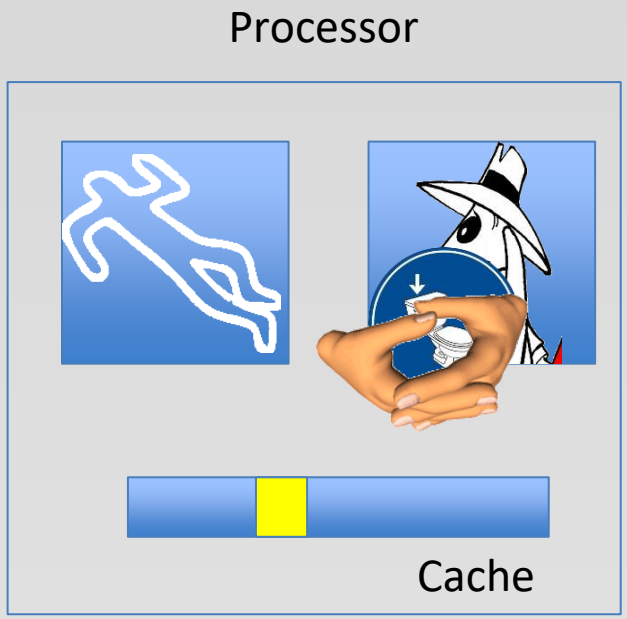
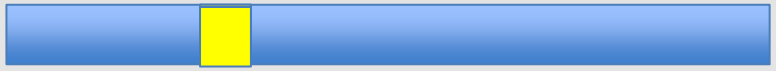
Sectre-type attacks

Covert Channels

FLUSH+RELOAD

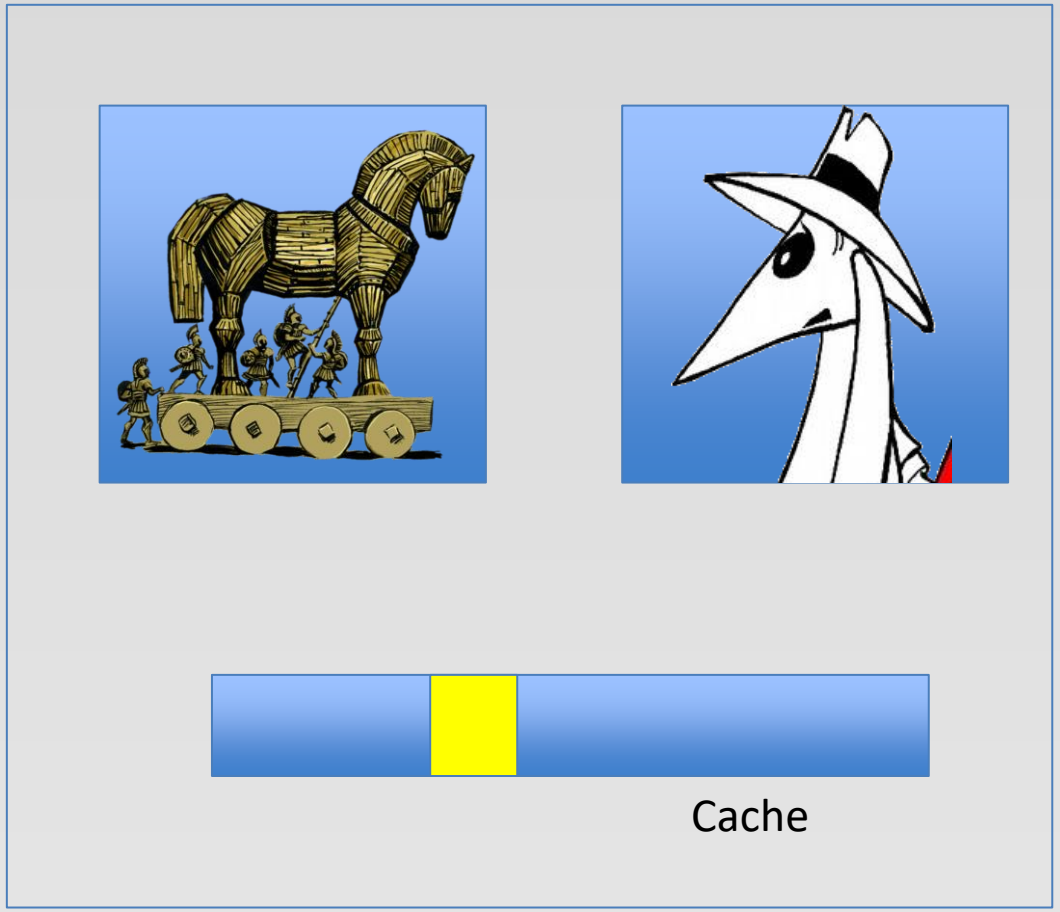
- Flush a memory line from the cache
- Wait a bit
- Measure time to load line
  - slow-> no access
  - fast-> access
- Repeat





Covert Channels

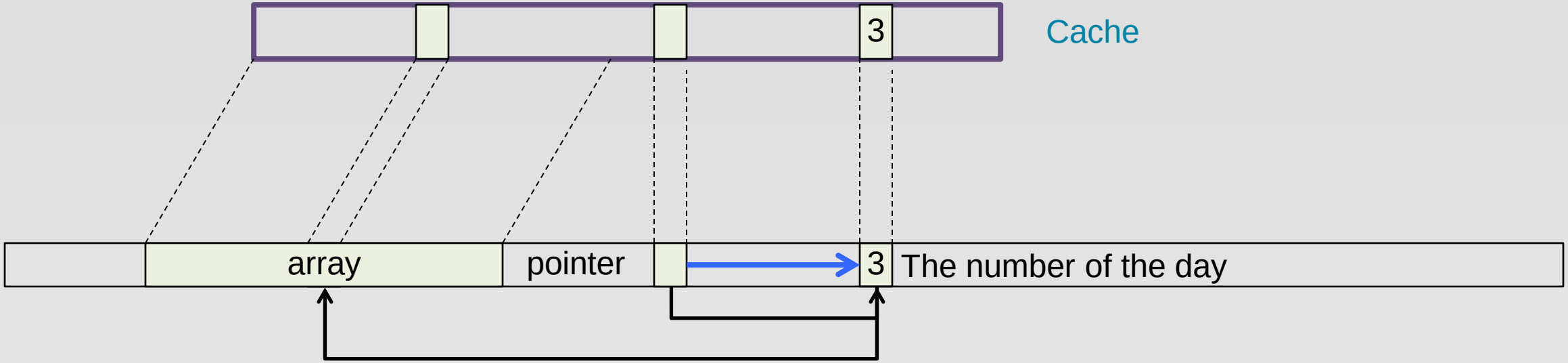
- Trojan in one protection domain
- Spy in another
- Communication via unintended channels
  - Not monitored



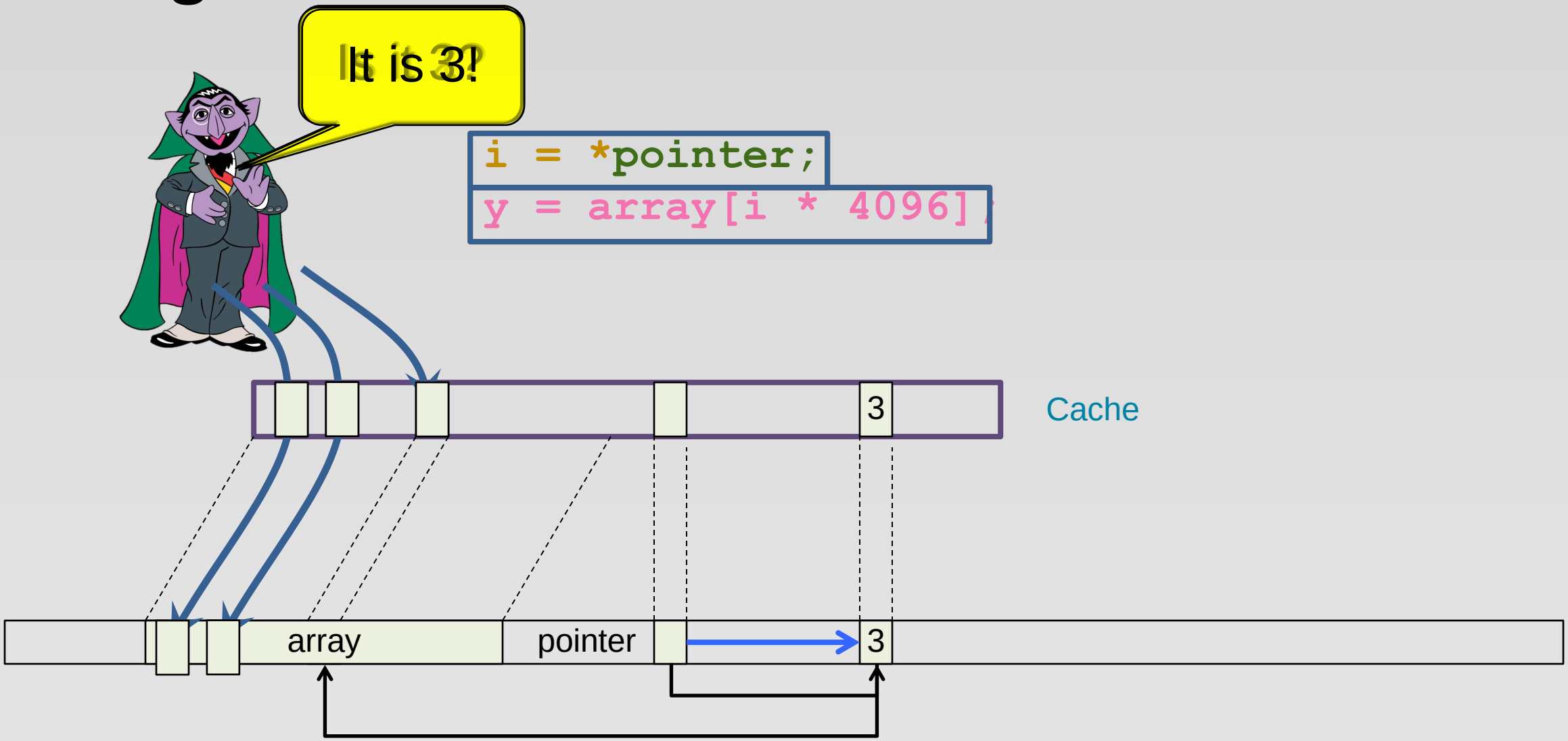
Covert Channels

Using a Covert Channel

```
i = *pointer;
y = array[i * 4096];
```



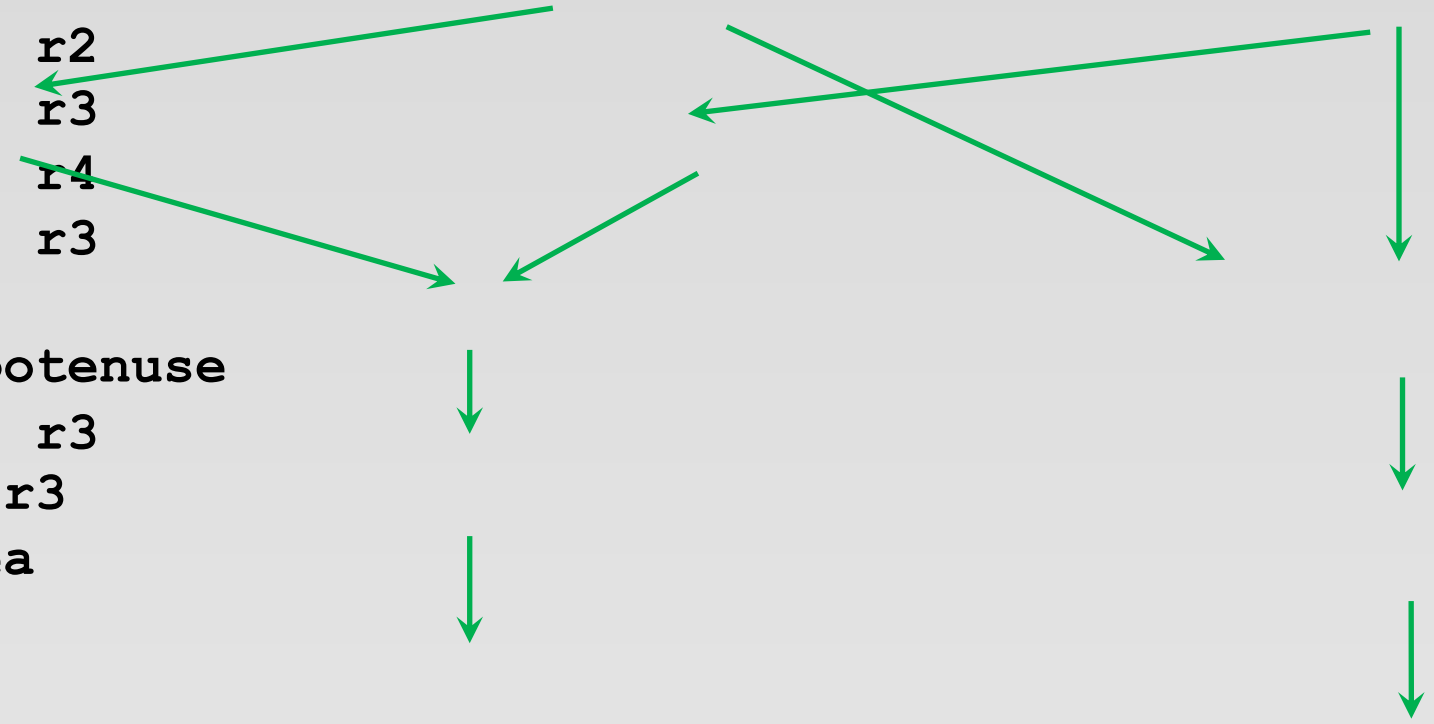
Using a Covert Channel



Speculative Execution

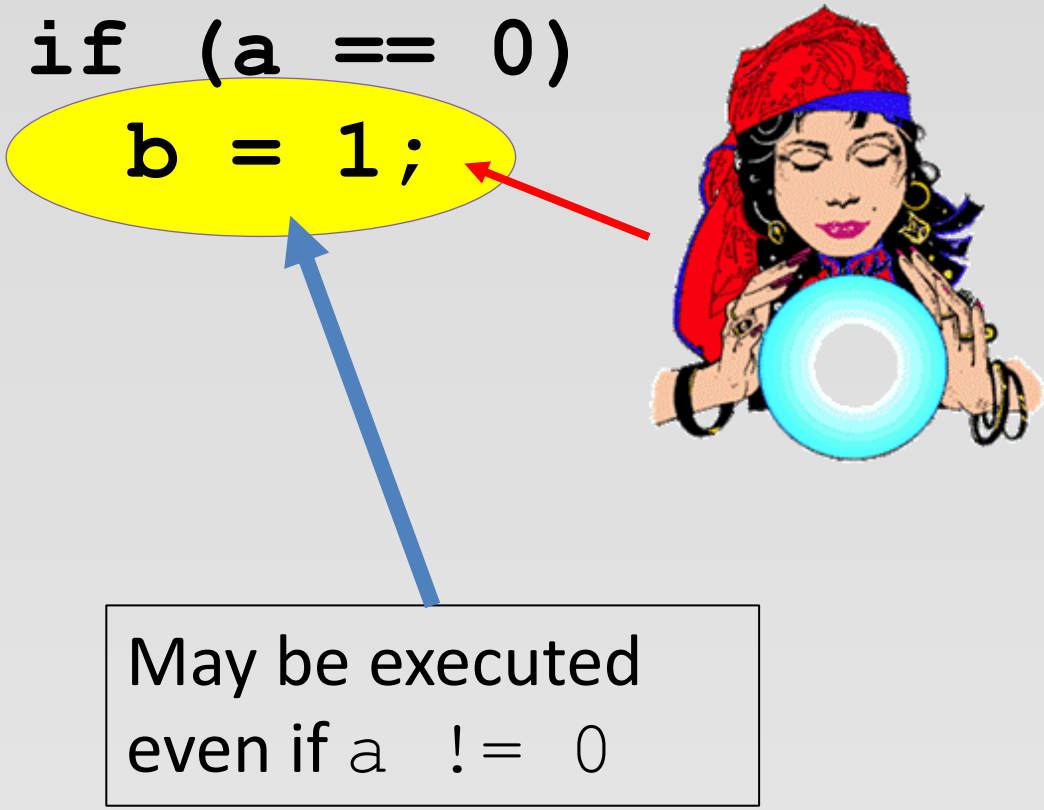
Recap – Out-of-Order execution

```
load  base, r1
load  height, r2
mul   r1, r1, r3
mul   r2, r2, r4
add   r3, r4, r3
sqrt  r3, r3
store r3, hypotenuse
mul   r1, r2, r3
div   r3, 2, r3
store r3, area
```



Speculative execution

- Data-dependent branches introduce dependencies on all younger instructions
- Speculating branch outcome reduces dependency
- … but introduces potentially incorrect execution
- Solution: Speculate on branch outcomes, but allow squashing of incorrect execution



Implementing speculative execution

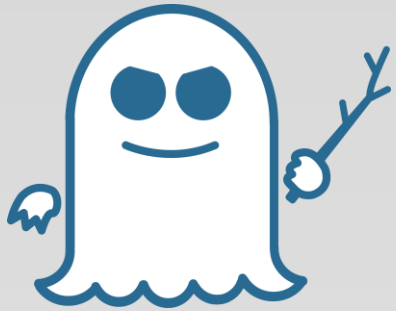
- Core idea:
  - Retire instructions in-order
  - No architecturally-visible state changes before instruction retires
- Also works for other cases
  - Memory ordering
  - Traps

Microarchitectural State changes are not reverted!

Spectre

Spectre (Variant 1)

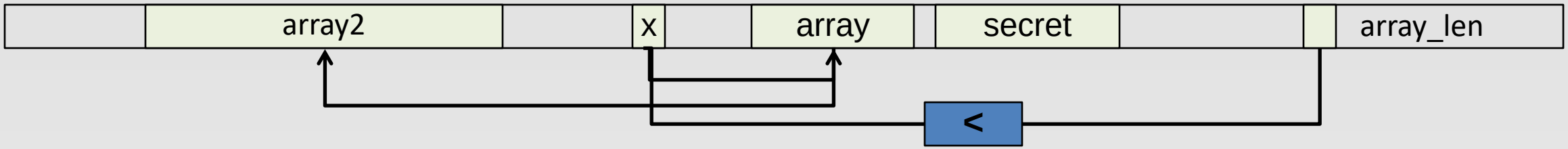
Victim



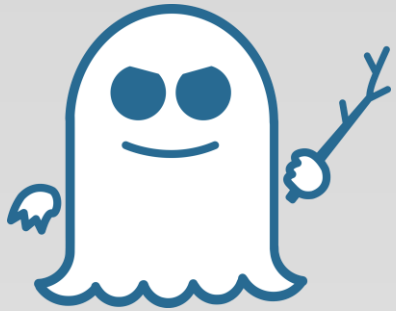
Attacker

Branch not taken!

```
if (x < array_len) {
    i = array[x];
    y = array2[i * 4096];
}
```



Spectre (Variant 1)

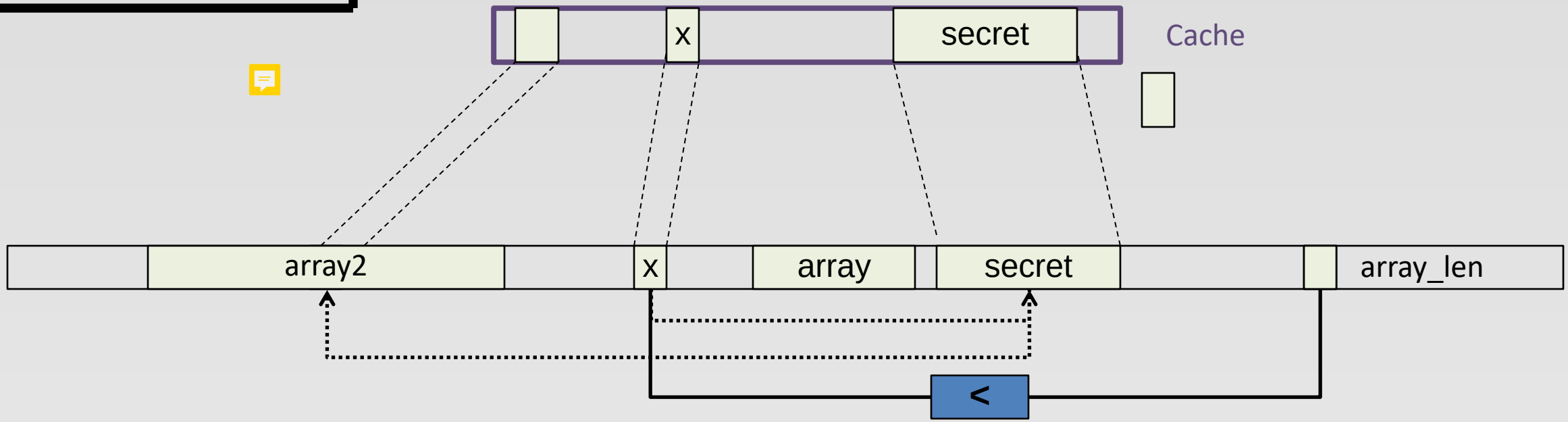


Attacker

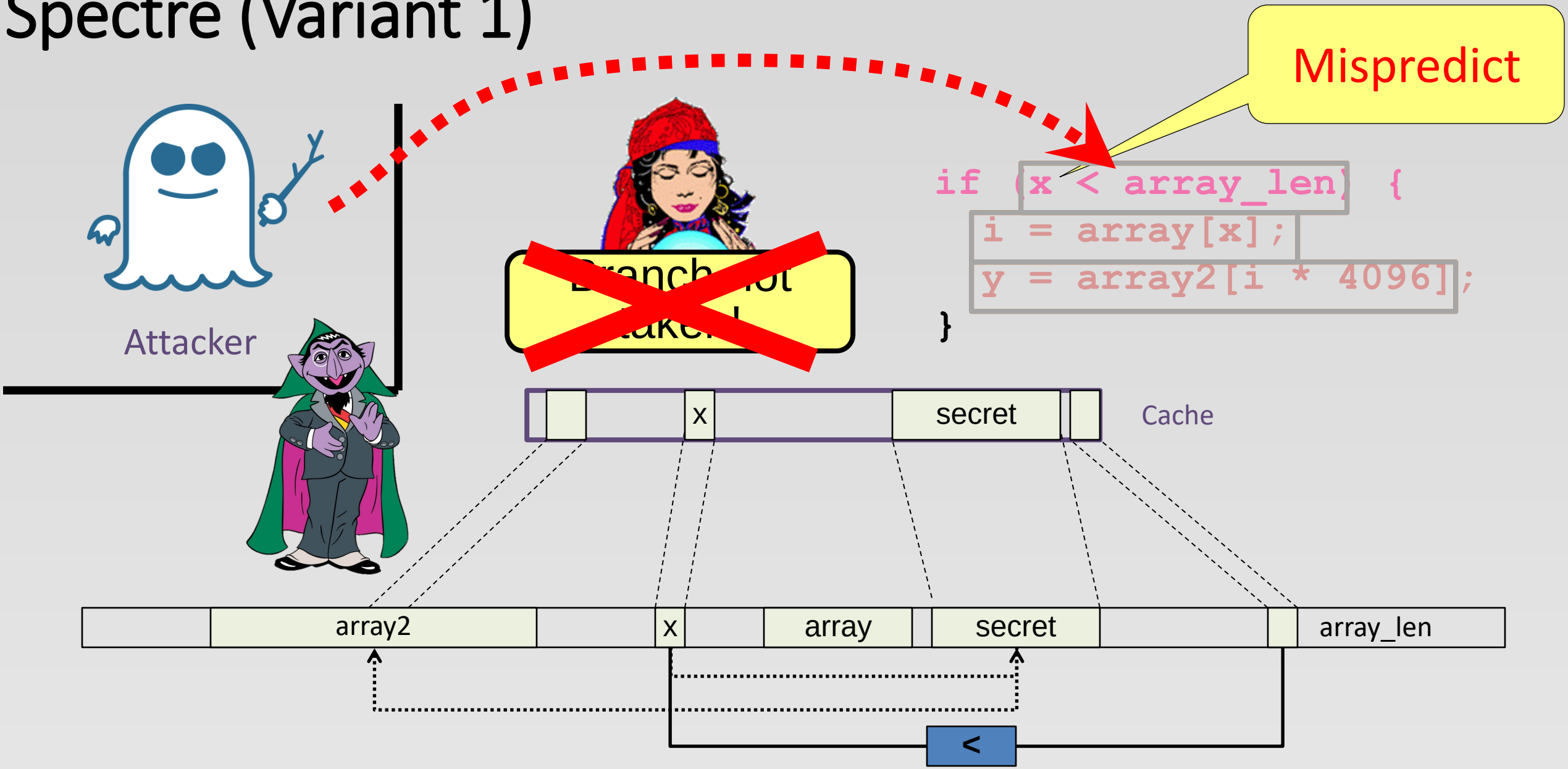
Branch not taken!

X is large

```
if (x < array_len) {
    i = array[x];
    y = array2[i * 4096];
}
```



Spectre (Variant 1)



Attack model

```
if (x < array_len) {
  i = array[x];
  y = array2[i * 4096];
}
```

Speculate

Access secret

Transmit secret

Receive secret

- Benign vs. malicious code
  - JavaScript, Linux eBPF

Defense - fences

- Prevent speculation using fences
- Automatically add a fence after a branch
- Need to protect both branches

```
if (x < array_len) {
  lfence()
  i = array[x];
  y = array2[i * 4096];
} else {
  lfence()
}
```

Overhead: ~400%

Array index masking

- Limits the offset for access for malicious code
  - Prevents access to secret
- Masks array indices before access
- Ensures index < 2*len

- Only protects arrays

```
if (x < len) {
  i = array[x]& array_mask]
  y = array2[i * 4096]& array2_mask]
}
```

Speculative load hardening (SLH)

- Track speculation status
- Poison data loaded from memory if under speculation
- Need to protect both branches
- Does not protect secrets in registers
- Supported in LLVM

```
mask = 0
if (x < len) {
  mask = (x < len) ? mask : -1
  i = array[x] | mask
  y = array2[i * 4096] | mask
} else {
  mask = (x < len) ? -1 : mask
}
```

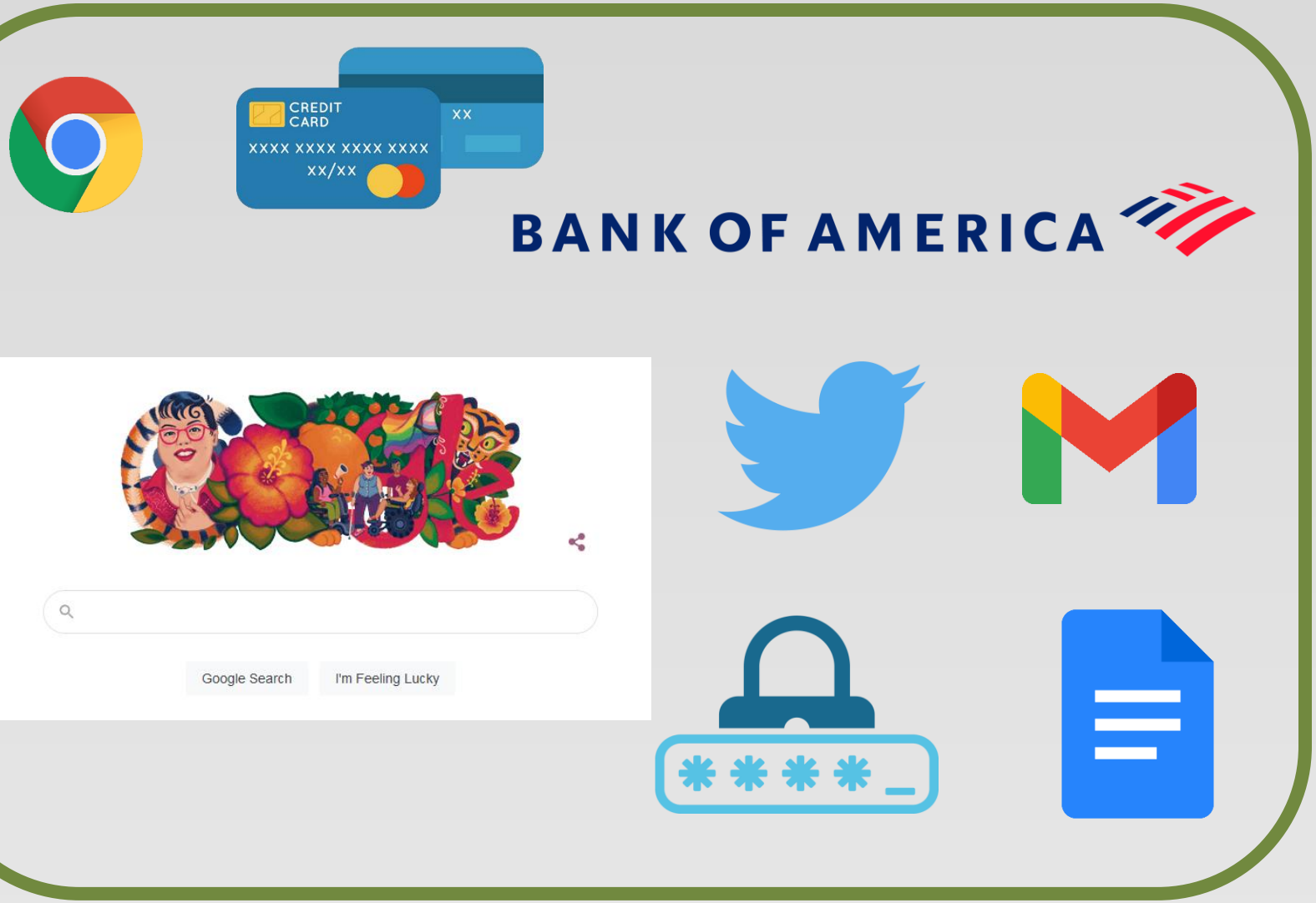
SLH – masking addresses

- No speculative access to secret-dependent address
  - Prevents covert-channel
- As in SLH, but protect the address
- Can leak through speculative secret-dependent branches
  - Port contention
  - Branch prediction

```
if (is_public(x)) {
  i = array[x];
}
```

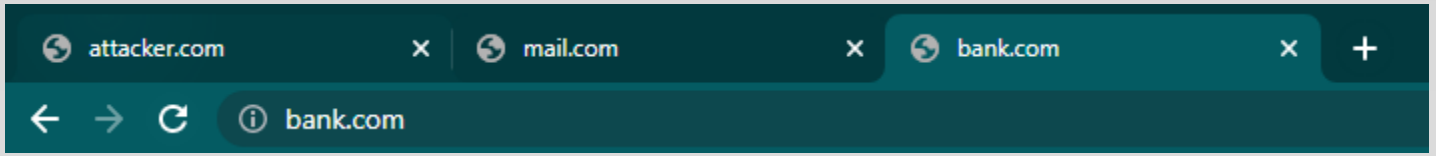
```
mask=0
if (is_public(x)) {
  mask = is_public(x)? mask : -1
  i = *((array + x) | mask) ;
} else {
  mask = is_public(x)? -1 : mask
}
```

Real-world issue

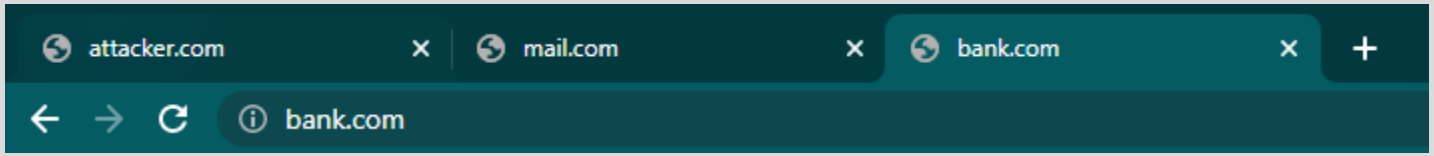


Browsers have secrets!

What happens if Spectre goes into the browser?



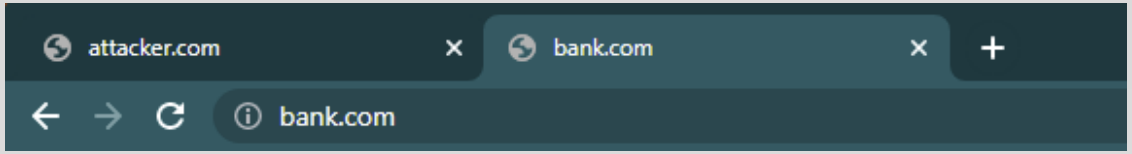
What happens if Spectre goes into the browser?

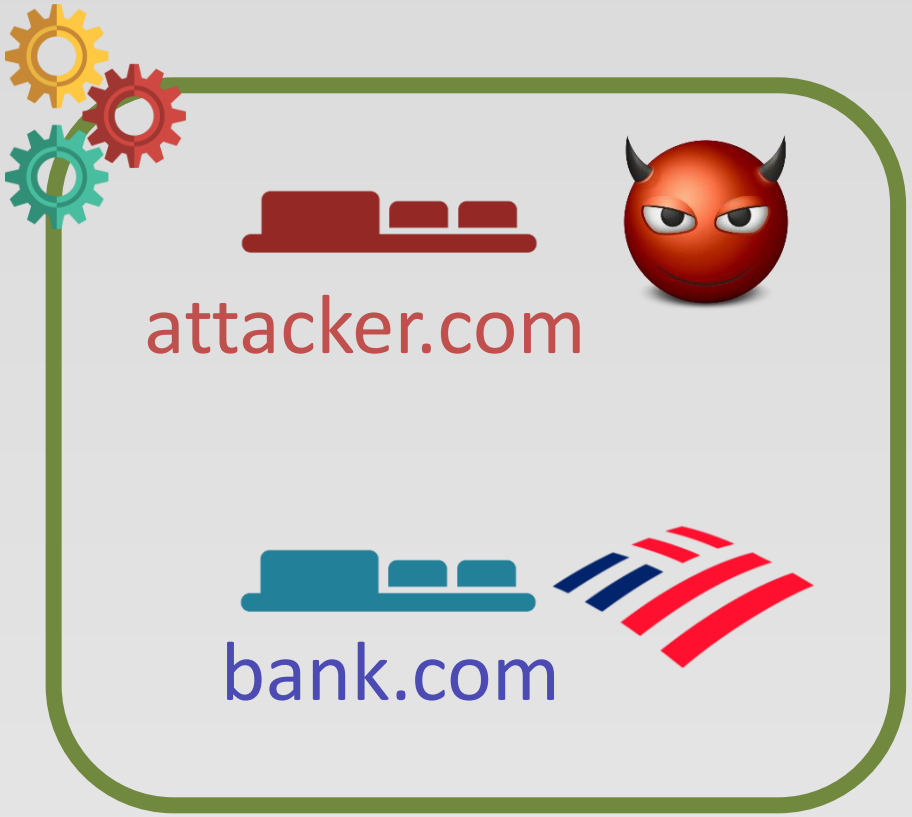


What are the countermeasures to overcome?

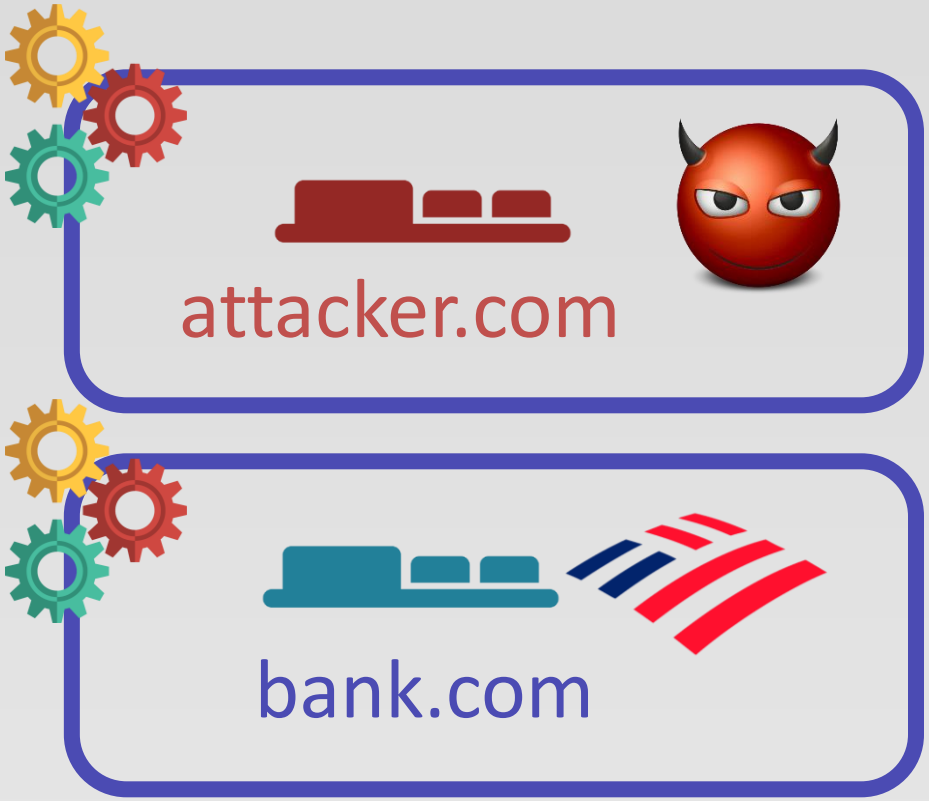
Site isolation

Site Isolation



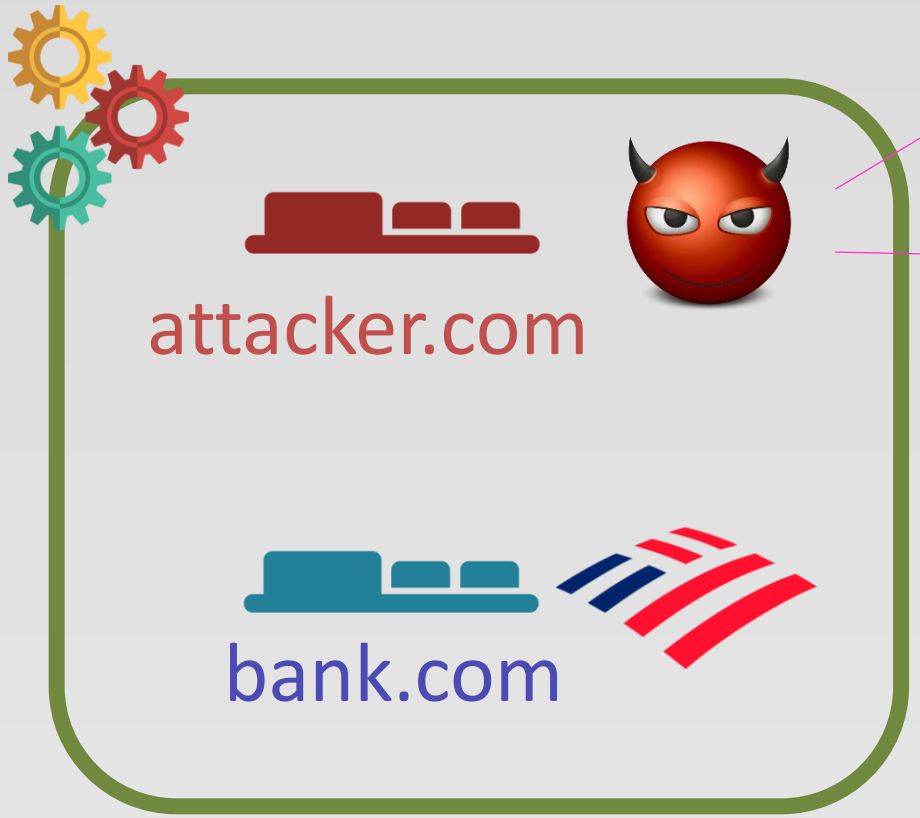


Ideal



Real

Safari and `window.open`?



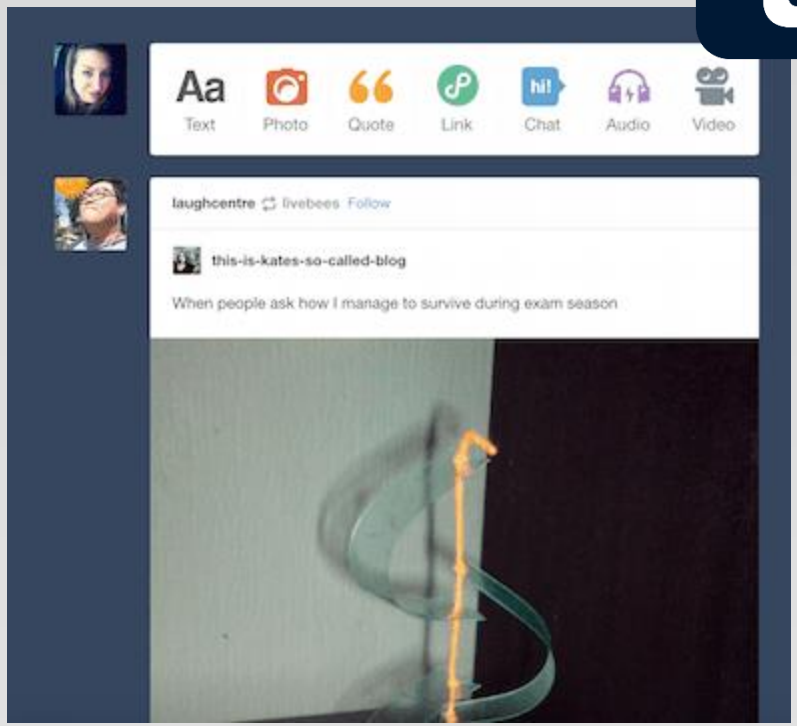
onclick="window.open('bank.com')"

Chrome: What happens to subdomains?

Consolidated!

attacker-blog. **tumblr**.com

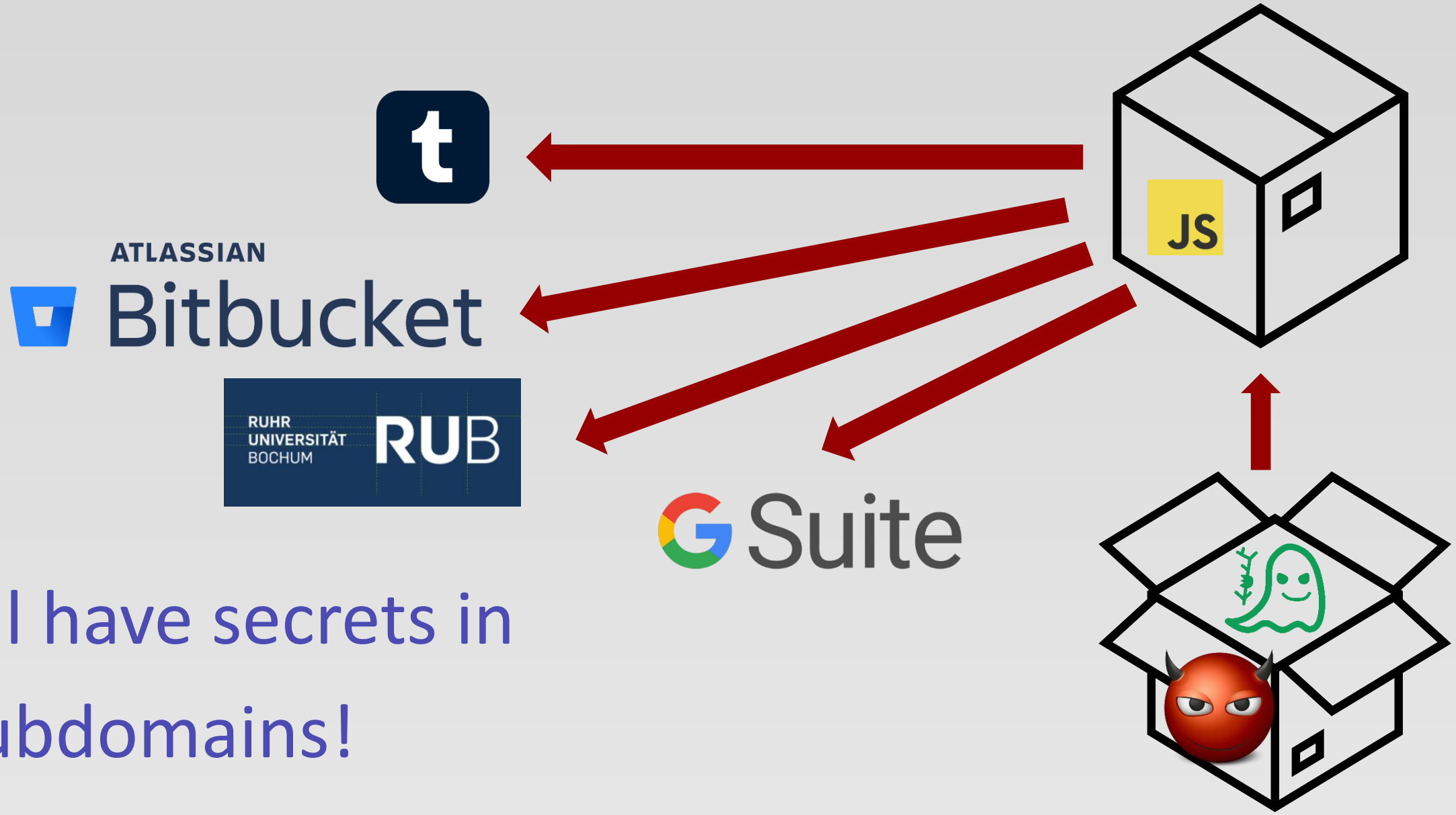
accounts. **tumblr**.com



foo.example.com — eTLD

bar.example.com — eTLD+1

Why is this a problem with some subdomains?



All have secrets in subdomains!

Now there are things to leak… what prevents us?

Site isolation

32-bit addressing

Why is textbook Spectre insufficient?

Spectre-v1 gadget

```
if (index < bounds)
{
    secret = *(base_ptr + index)
    return probe_array[secret]
}
```

Branch taken

$ hit

probe_array

64 bits in Chrome
NOT controllable by attacker

32 bits in Chrome
Controllable by attacker

base_ptr + index

Spectre array

32-bit addressable out
of bounds region (4 GB)

Secret from
other subdomain

base_ptr

Address space

How does array access work under the hood, anyway?

JavaScript

```
array[index]
```

Compiled by Chrome

```
if (argument type == TypedArray)
{
    . . .
    return *(base_ptr + index)
}
else
{
    throw exception
}
```

TypedArray

type = TypedArray
…
base_ptr
…

64 bits in Chrome
NOT controllable by attacker

32 bits in Chrome
Controllable by attacker

4 GB problem persists!

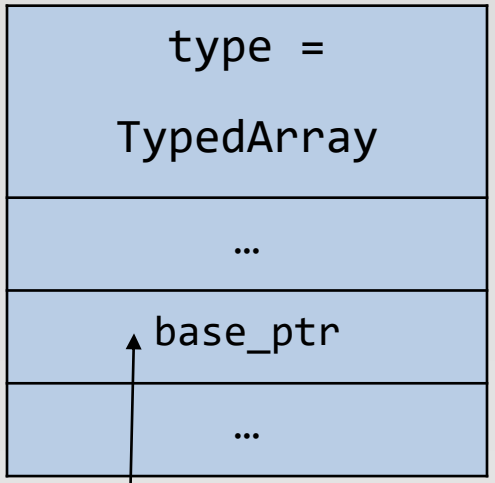
What if we provide a similar object… and try to index it?

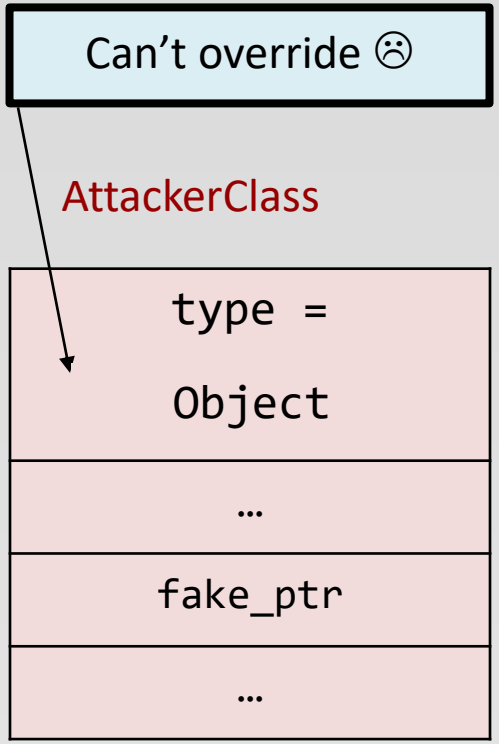
JavaScript

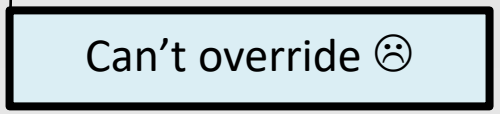
```
AttackerClass[index]
```

Compiled by Chrome

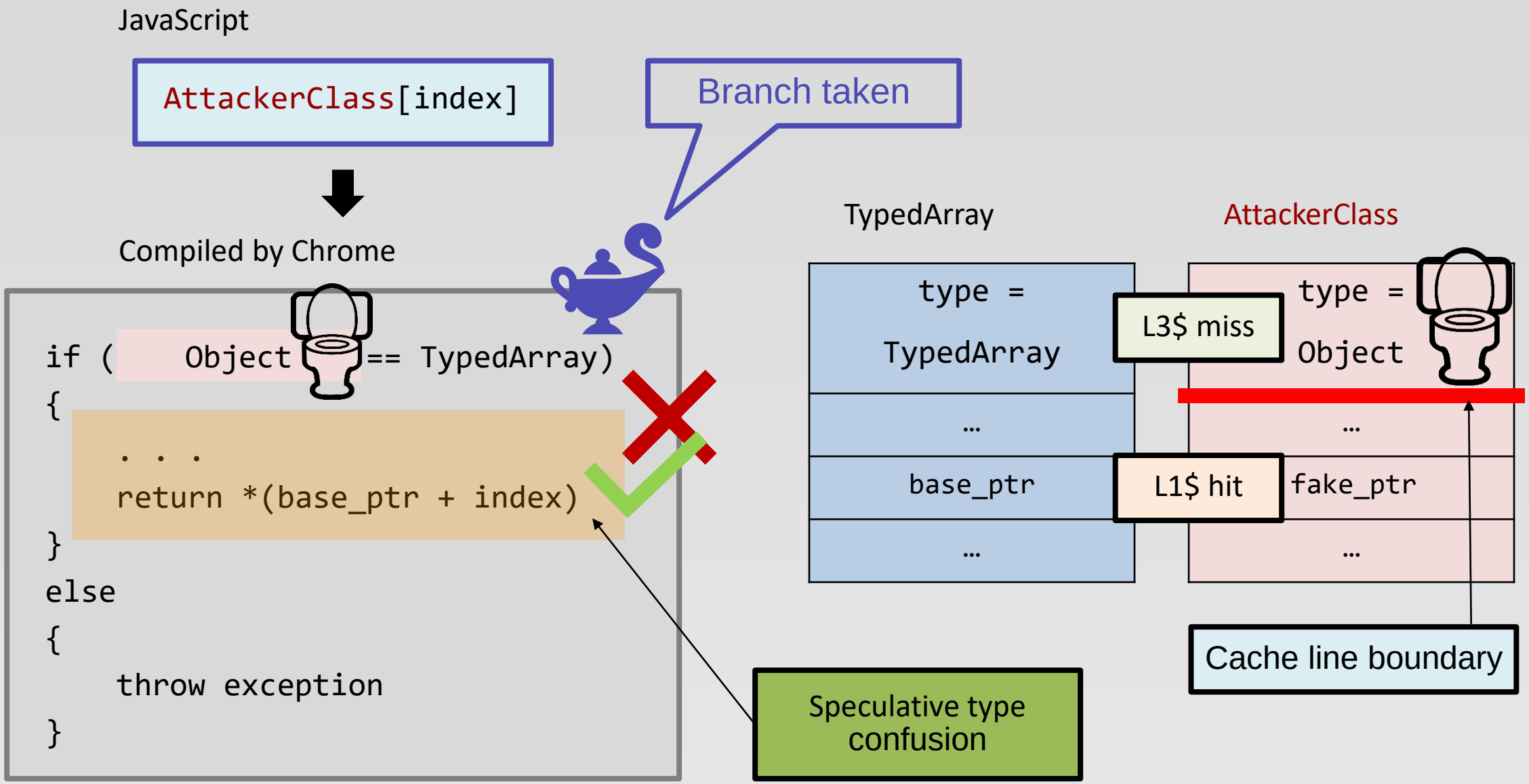
```
if (argument type == TypedArray)
{
    . . .
    return *(base_ptr + index)
}
else
{
    throw exception
}
```







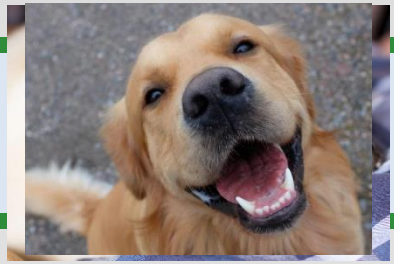
Type mismatch is the problem; what if we evict it from the cache?



Now we have arbitrary 64-bit reads. What's left?

Site isolation

32-bit addressing



Reliability

Hiding the exception with speculation

JavaScript

```
AttackerClass[index]
```

Compiled by Chrome

```
if (    Object      == TypedArray)
{
    . . .
    return *(base_ptr + index)
}
else
{
    throw exception
}
```

Under speculation

JavaScript

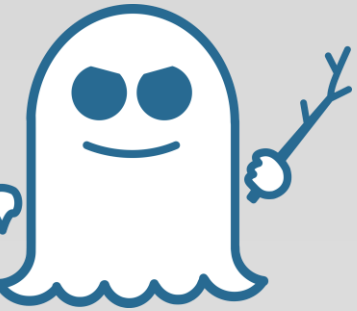
```
if (cond) {
    AttackerClass[index]
}
```

Compiled by Chrome

```
if (cond) {
    if (argument type == TypedArray)
    {
        . . .
        return *(base_ptr + index)
    }
    else
    {
        throw exception
    }
}
```

Under speculation

Spectre (Variant 2)



Attacker

Victim

RET

LEA RAX, GADGET
JMP RAX

MOV AX, [RBX]
MOV RCX, [8*RAX + RCX]

JMP RAX

Attacker's Address Space

Victim's Address Space

Spectre V2

- No restrictions on branch target
- Training from user to kernel space
- Aliasing allows training with different branches
  - eBPF trains the kernel

Defenses

- Retpolines: replace indirect branches with returns
- RET may also be predicted…

- Restrict cross-domain branches
- Invalidate branch history

```
JMP R11
```

```
  CALL SETUP
CAPTURE:
  PAUSE
  JMP CAPTURE
SETUP:
  MOV [RSP], R11
  RET
```

More variants

- Spectre-RSB – prediction of return instructions

- Spectre-V4 – prediction of memory ordering

Summary

- Speculative execution is a problem

- Next lecture: Meltdown-type attacks.
  - Read: Lipp et al. “Meltdown: Reading Kernel Memory from User Space”, USENIX Security 2018