Sécurité des logiciels

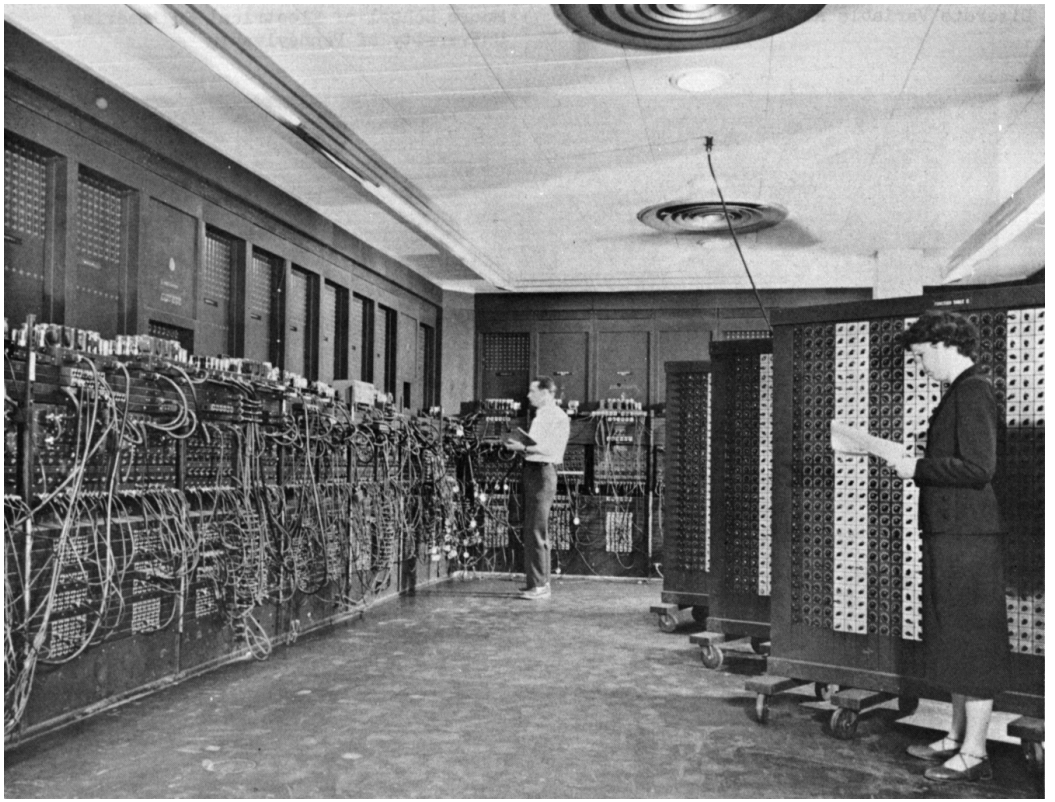
Side/Covert channels

Samuel Thibault <samuel.thibault@u-bordeaux.fr>

Isolation / communication

First computers

- No notion of “process”, “protection”, “Operating systems”
- Just computing
- Litterally feeding data by hand
- Security enforced by a lock on the door



First operating systems for mainframes

e.g. OS/360 in the 60's

- First time an OS is kept the same over several machines
- Keeps track of allocated resources
  - Reclaim them when the process is terminated
- But still mostly operated by hand
  - No communication

Operating systems for microcomputers

In 70's-80's, for the wide public

- CP/M / MS-DOS / PC-DOS
- Unix / Windows / Mac OS / GNU/Linux
- Multi-users, disk exchanges, ...

And development of ARPANET (soon to be the Internet)

In a word: **communication**

Communication

Computer Security Technology Planning Study (1972) worries about that:

"it replaces manual, easily visible controls with reliance upon logical and intangible program controls to keep separate data and programs belonging to different users"

Nowadays it is taken for granted

- Multiple levels of users in same computer
  - kernel, root, users, javascript vm
- Compartimentization to separate them
  - Kernel/User, processes, virtual memory, file access permissions, ...
- Channels between the levels
  - System calls, files, pipes, vm calls, ...

Isolation

Historically:

- Batch processing
  - Whole machine for yourself for a given time
- Processes
  - Whole virtual address space for yourself
  - CPU time-sharing
- User identifiers
  - File permissions
  - Process control
- Virtual machines
  - One step backwards
  - Guest thinks running on its own machine
  - Processes within it: *isolation nesting*
- Containers
  - One step in between
  - Not really complete virtual machines, not really simple processes

Communication

Historically:

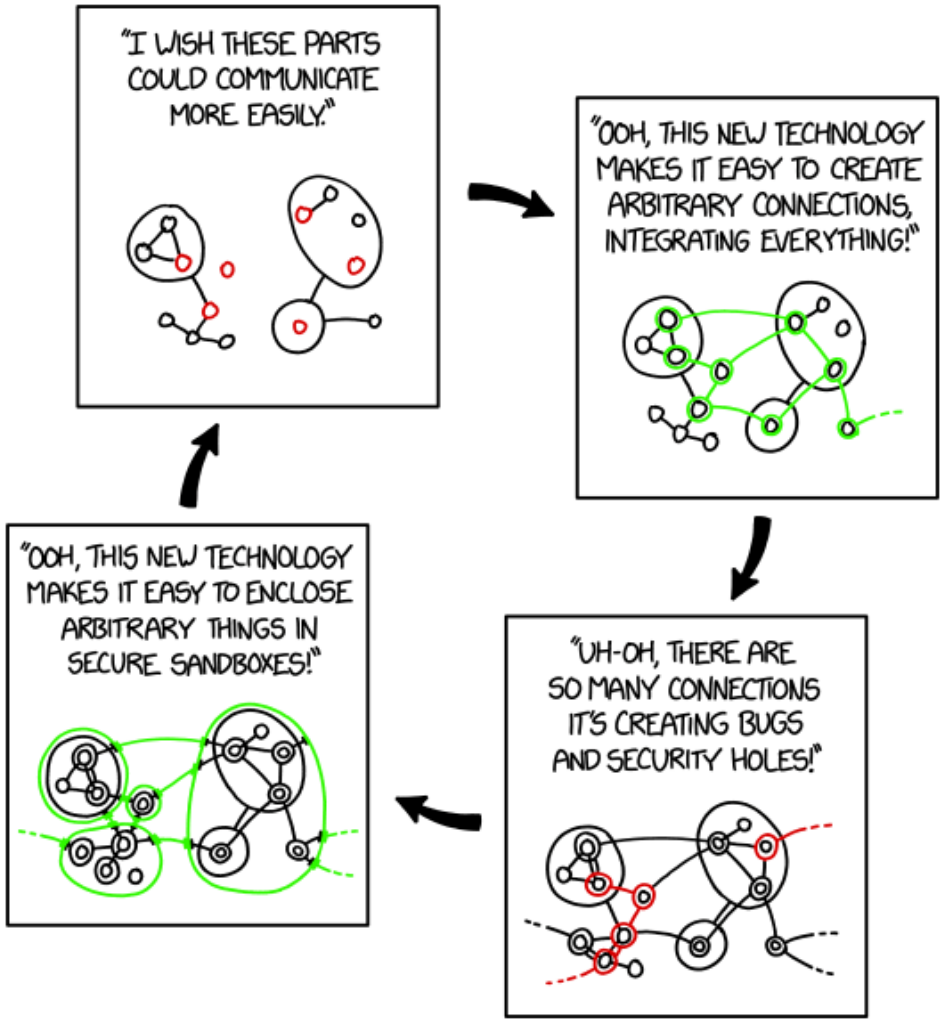
- Punch cards / printed paper
- Magnetic tapes
- Files
- Pipes
- Shared memory
- TCP/IP sockets
- VM channels

Communication

What harm could plugging this bring?

- Software autorun
  - Now disabled
- Thunderbolt technology
  - Can basically read all RAM with DMA...

Isolation vs communication



From https://xkcd.com/2044/

Covert channels

Covert channels

"A covert channel is created by a sender process that modulates some condition (such as free space, availability of some service, wait time to execute) that can be detected by a receiving process"

I.e. a way to subvert something into communicating

- CPU % usage
- Memory usage
- File open date
- Network packets timing
  - TCP Port knocking
- LED blinking
  - Disk, keyboard

Side channels

Side channels

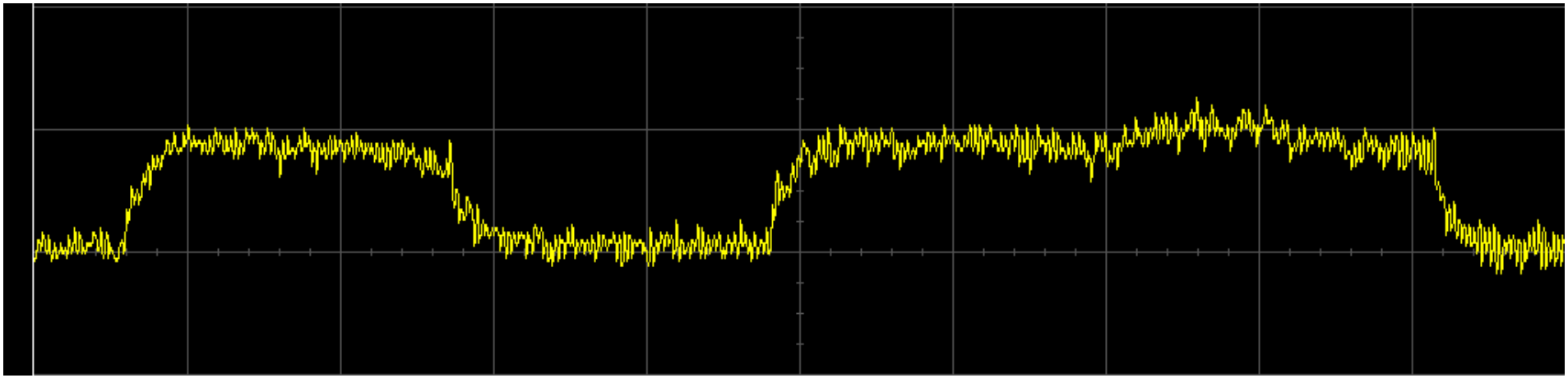
"a side-channel attack is any attack based on information gained from the implementation of a computer system, rather than weaknesses in the implemented algorithm itself"

Problem:

- **Visible** resources
- **Shared** resources

Visible resources

- Power monitoring

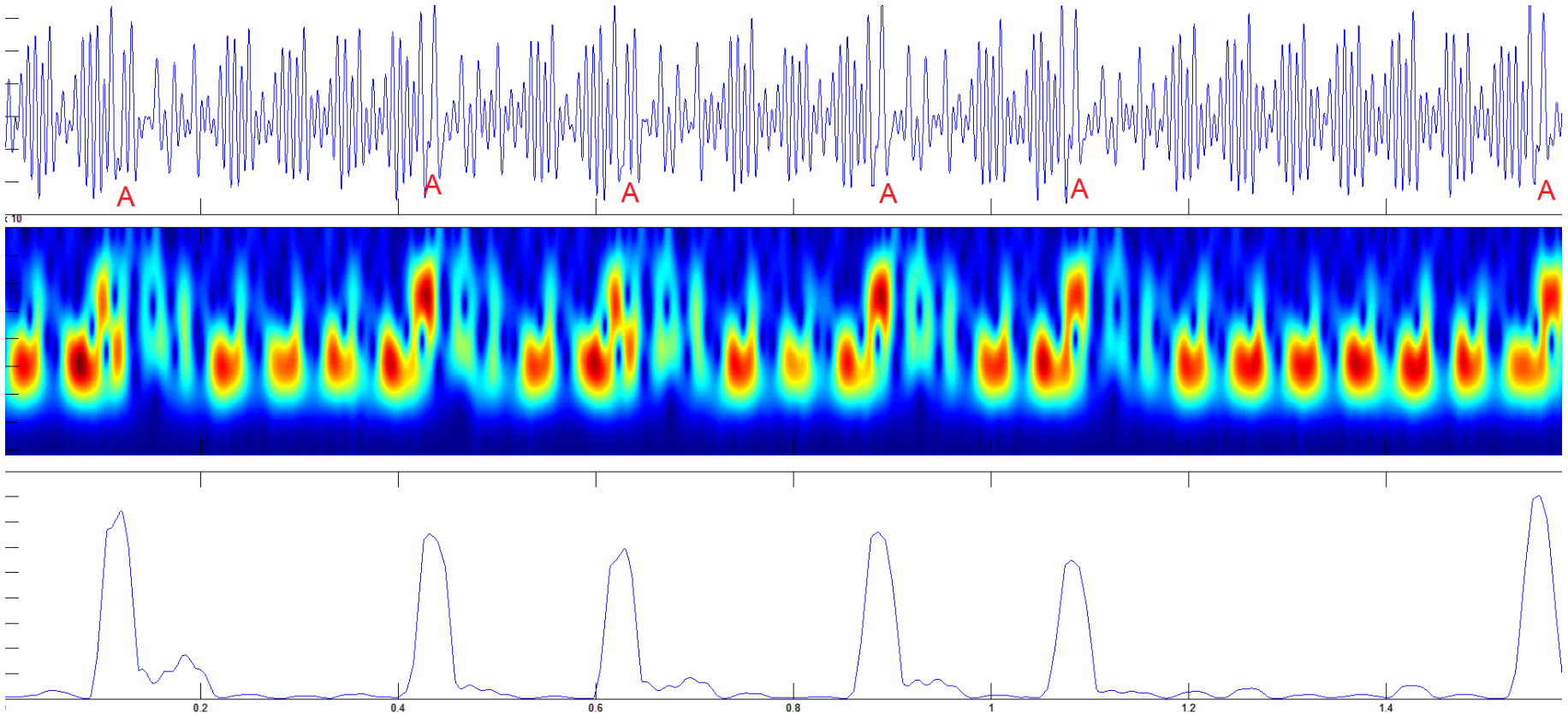


No multiplication

Multiplication

Visible resources

- Power monitoring
- Electromagnetic monitoring (TEMPEST)



Visible resources

- Power monitoring
- Electromagnetic monitoring (TEMPEST)
- Optical monitoring
- Accoustic monitoring

→ Use implementation whose behavior *does not depend on the data*

→ Or cipher data before processing it: *blinding*

  - Instead of deciphering y^e, decipher $y^e.r^e \rightarrow y.r$, eventually y

Visible resources

Data remanence

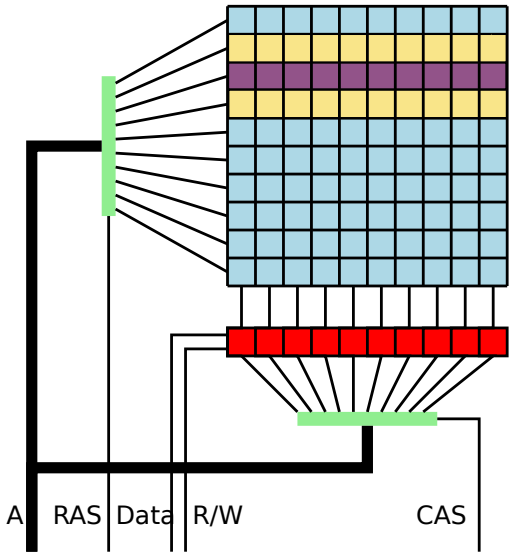
- “Erased” (?) files
- Cold boot attack
- Eject RAM

Metadata

- Communication size, source/destination, timing, ...

Perturbate resources

- Differential fault analysis
  - Perturbate with rays
  - Check how errors happen, can reveal algorithm details
- Software-initiated fault attacks
  - Row hammer: just reading data!



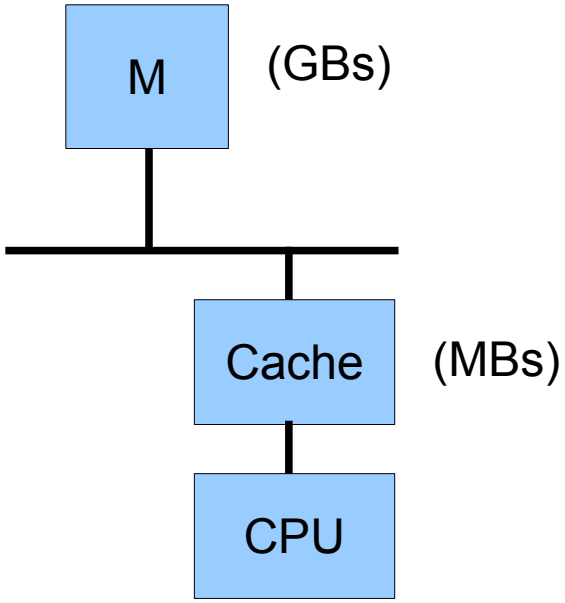
Shared resources: caches

Memory hierarchy

- Memory: ~100ns latency
- Cache: ~10ns latency

Data transparently duplicated in the cache

- No semantic difference
- Timing difference
- *Micro-architectural state*



Shared resources: caches

Memory hierarchy

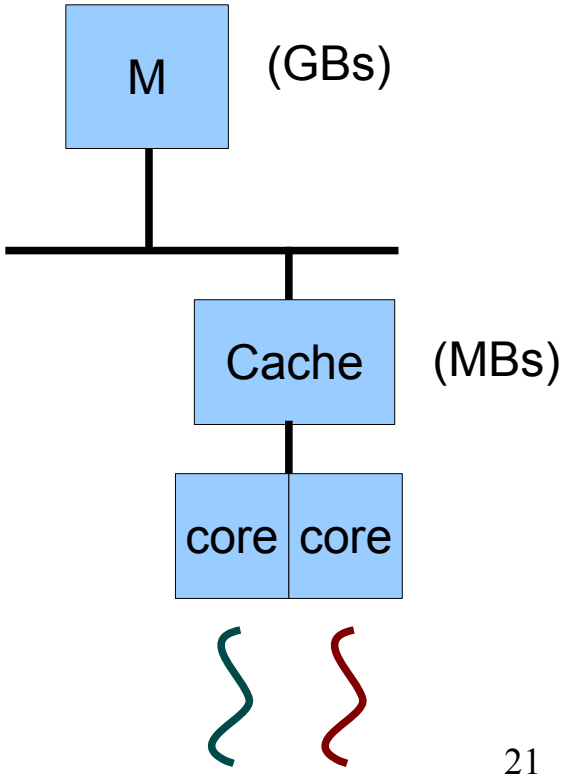
- Memory: ~100ns latency
- Cache: ~10ns latency

Data transparently duplicated in the cache

- No semantic difference
- Timing difference
- *Micro-architectural state*

Shared between cores!

Allows to observe behavior of other program



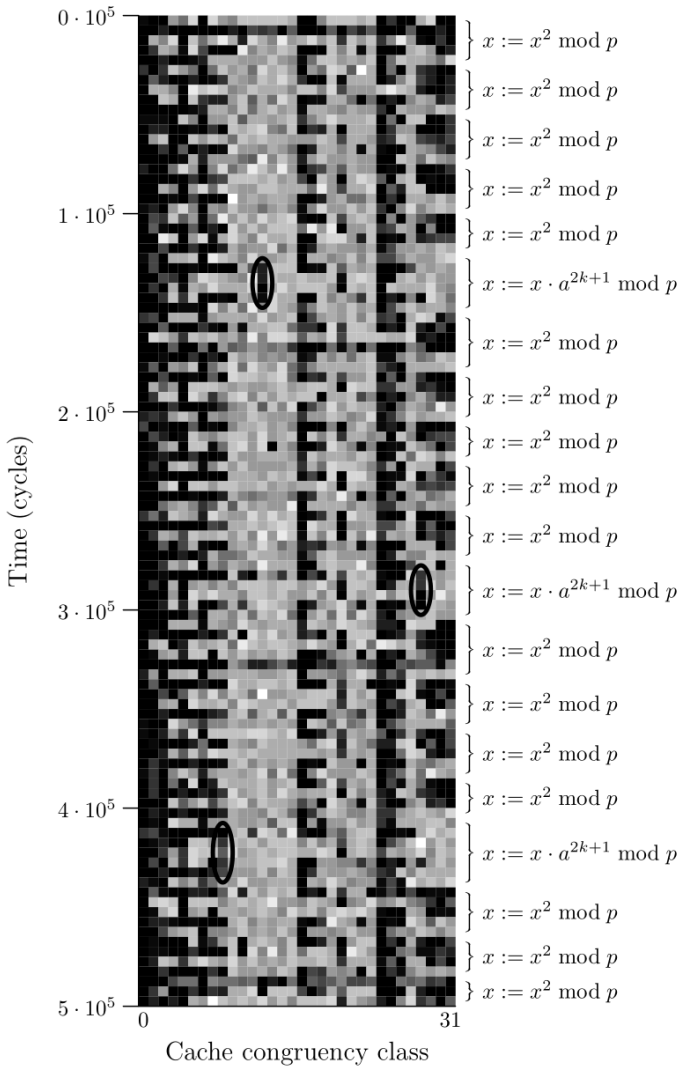
Shared resources: caches

Cache attacks!

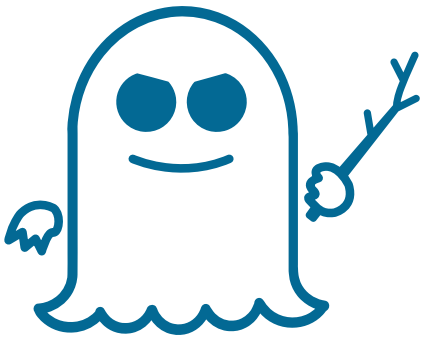
- Allow to observe the program behavior

Cache missing for fun and profit

More generally, monitor accesses to ciphering tables, etc.



But there is much worse on this...



Spectre/Meltdown

2018 January 3rd

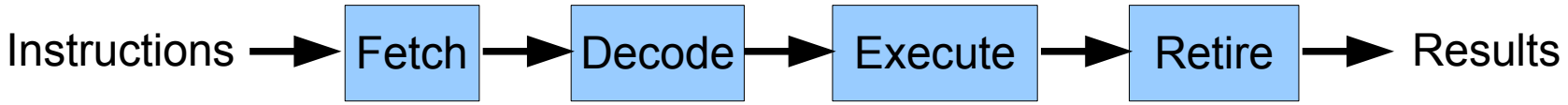
- Two **hardware vulnerabilities** disclosed for the price of one
- Discovered 6 months before, kept embargoed
- Introduced...
  - around 1995...

What happened??

Speculative execution

Nowadays processors optimize crazily

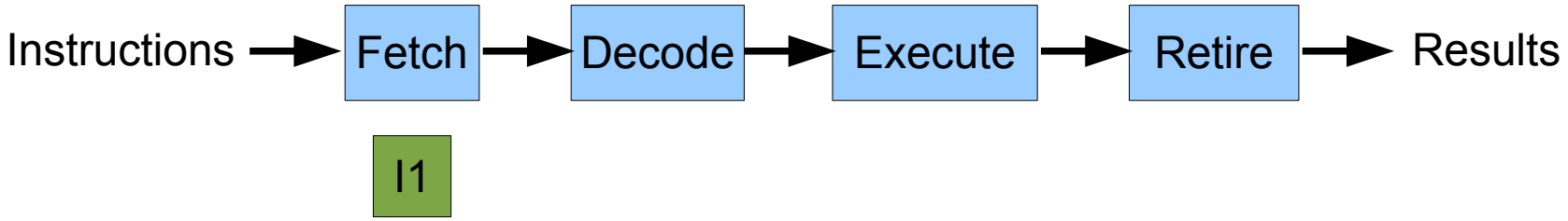
- Long pipeline



Speculative execution

Nowadays processors optimize crazily

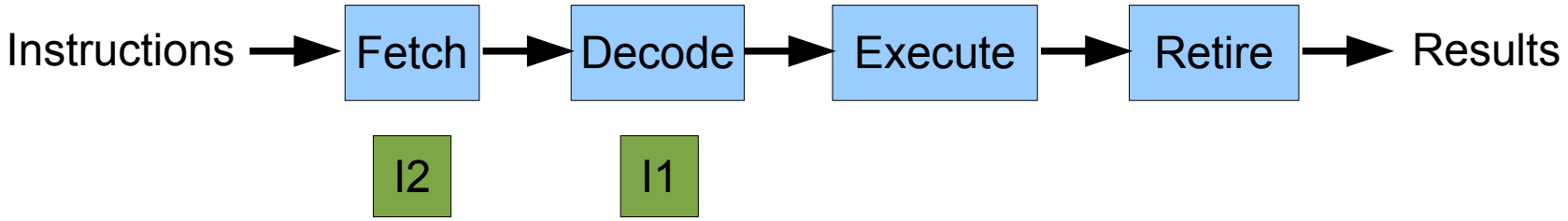
- Long pipeline



Speculative execution

Nowadays processors optimize crazily

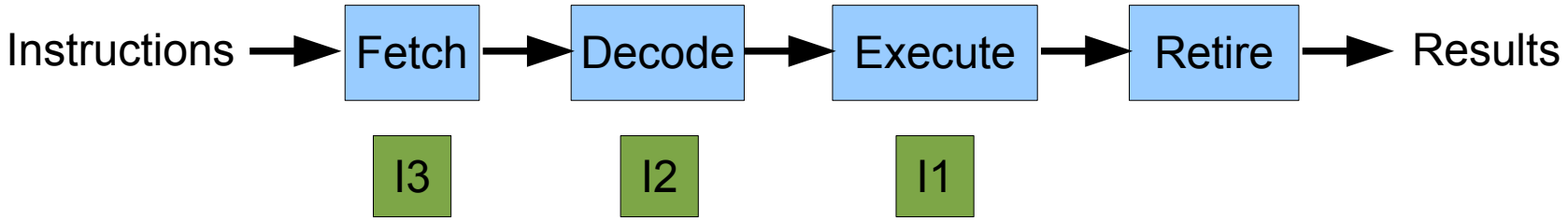
- Long pipeline



Speculative execution

Nowadays processors optimize crazily

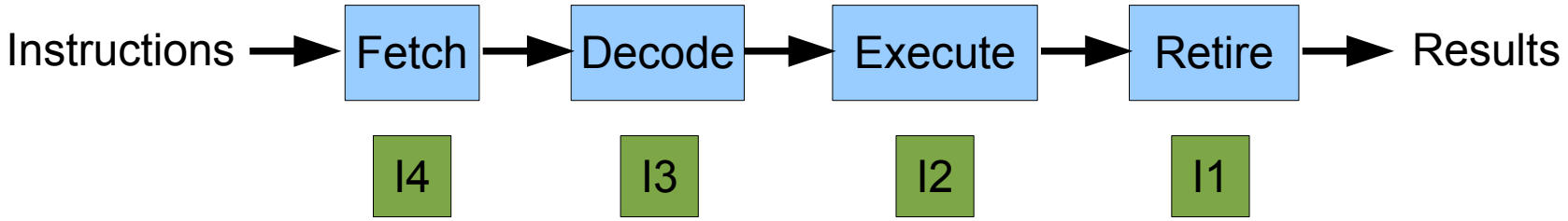
- Long pipeline



Speculative execution

Nowadays processors optimize crazily

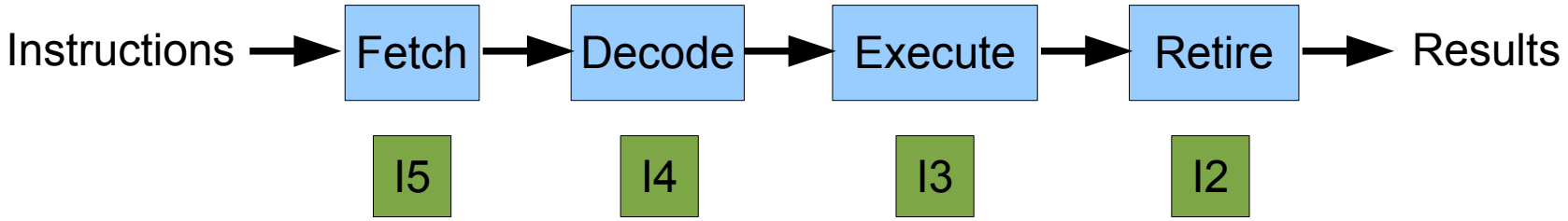
- Long pipeline



Speculative execution

Nowadays processors optimize crazily

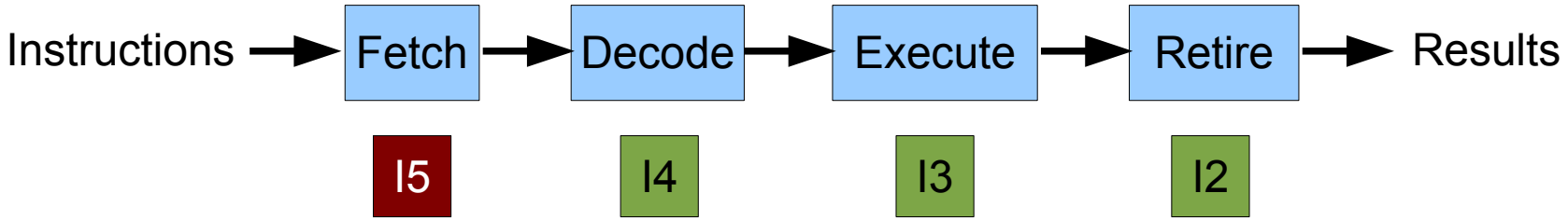
- Long pipeline



Speculative execution

Nowadays processors optimize crazily

- Long pipeline



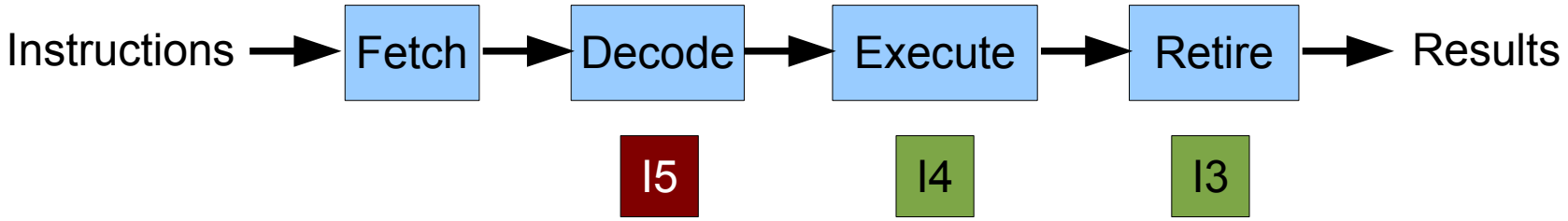
- But what about branches?
  - Would have to wait for result to know which instructions to load?

```
if (t[x] > 0) {
  foo(x);
} else {
  bar(x);
}
```

Speculative execution

Nowadays processors optimize crazily

- Long pipeline



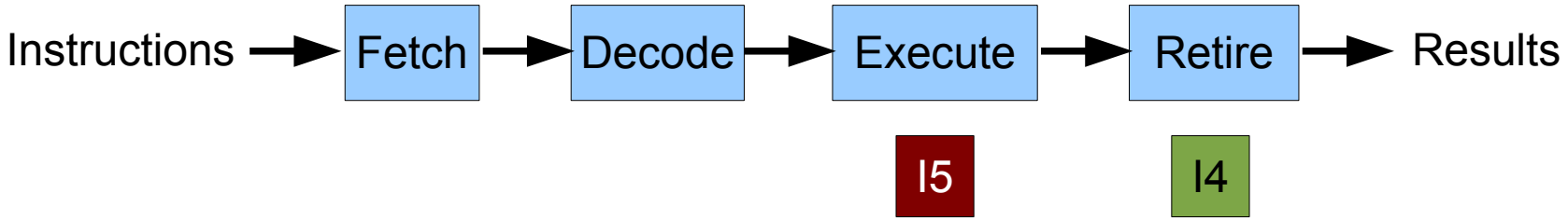
- But what about branches?
  - Would have to wait for result to know which instructions to load?

```
if (t[x] > 0) {
  foo(x);
} else {
  bar(x);
}
```

Speculative execution

Nowadays processors optimize crazily

- Long pipeline



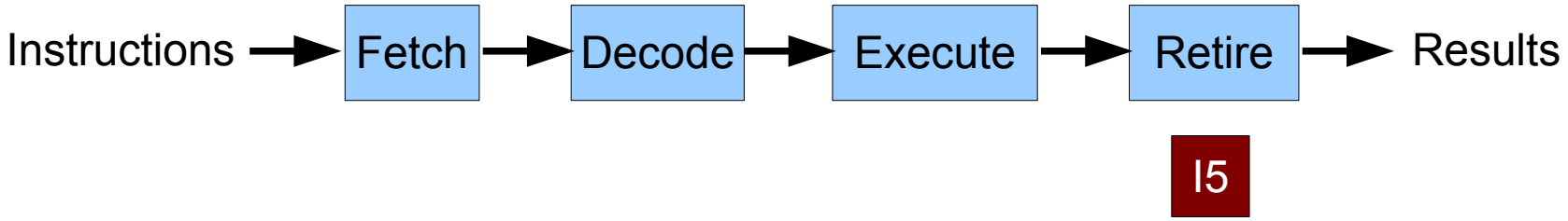
- But what about branches?
  - Would have to wait for result to know which instructions to load?

```
if (t[x] > 0) {
  foo(x);
} else {
  bar(x);
}
```

Speculative execution

Nowadays processors optimize crazily

- Long pipeline



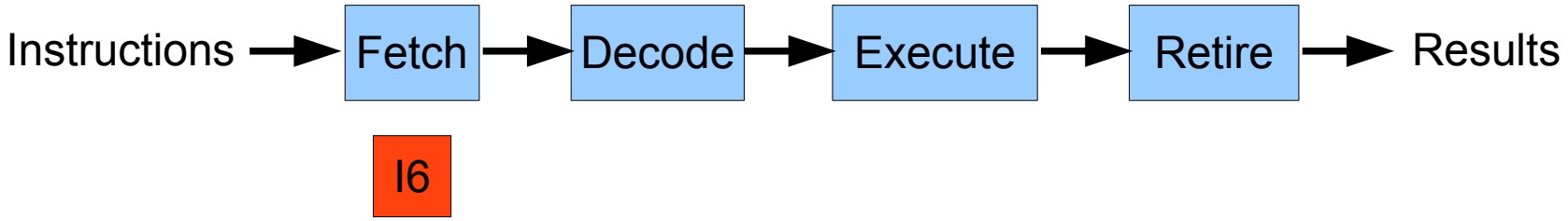
- But what about branches?
  - Would have to wait for result to know which instructions to load?

```
if (t[x] > 0) {
  foo(x);
} else {
  bar(x);
}
```

Speculative execution

Nowadays processors optimize crazily

- Long pipeline



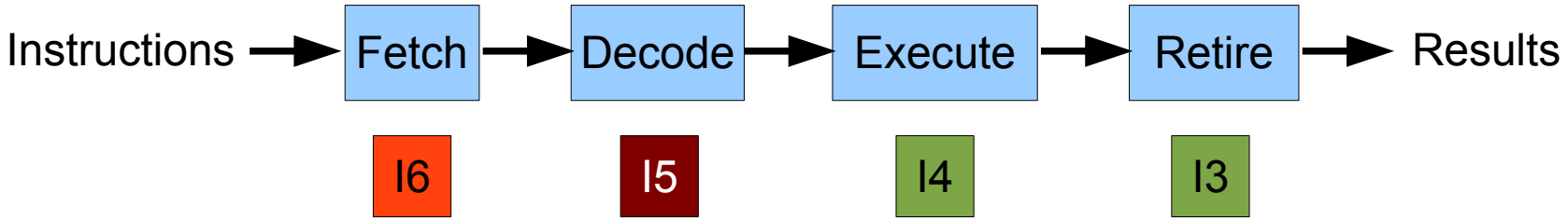
- But what about branches?
  - Would have to wait for result to know which instructions to load?

```
if (t[x] > 0) {
  foo(x);
} else {
  bar(x);
}
```

Speculative execution

Nowadays processors optimize crazily

- Long pipeline



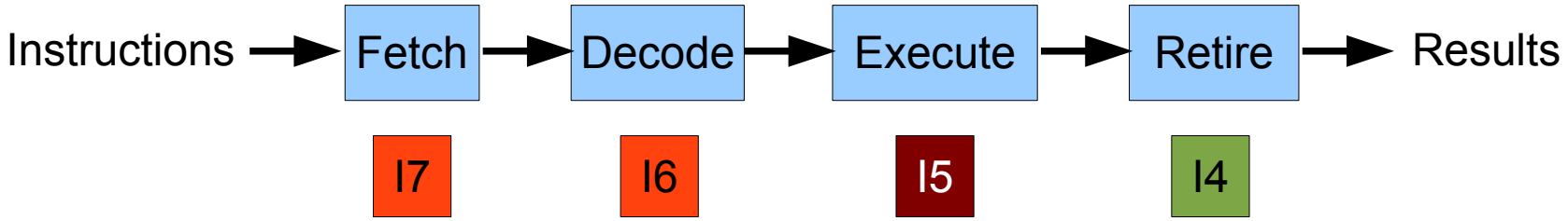
- But what about branches?
  - Would have to wait for result to know which instructions to load?
  - Rather **predict/speculate** which way to go

```
if (t[x] > 0) {
  foo(x);
} else {
  bar(x);
}
```

Speculative execution

Nowadays processors optimize crazily

- Long pipeline



- But what about branches?
  - Would have to wait for result to know which instructions to load?
  - Rather **predict/speculate** which way to go

```
if (t[x] > 0) {
  foo(x);
} else {
  bar(x);
}
```

Speculative execution

This speculation is essential for performance

- `for (i = 0; i < n; i++)`
  `t[i] = 0;`

  Don't want to wait for `i++` and `i<n` at each iteration!

- `if (i >= N)`
  `return -1;`

  Usually no error

Processor learns from experience

- "Most probably just like last time"
- *Branch prediction*

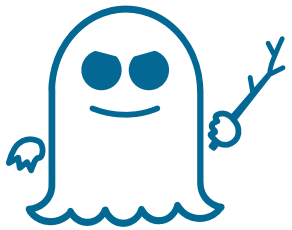
Speculative execution

What if the prediction got wrong?

- Processor cancels all effects
  - No memory/cache write
  - Restore register value
  - Get PC back to proper branch
- i.e. restore *architectural* state

But does not restore *micro-architectural* state

- What if speculated execution loaded a value in the cache?



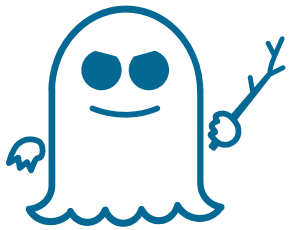
Spectre vulnerability

Vulnerable code e.g. in a system call:

```
static char T[N] = { ... };

void f(char *data, unsigned i) {
  if (i >= N)
    return -EINVAL;
  char val = T[i];
  return data[val*64];
}
```

Apparently checking that we are not overflowing T
And most often so indeed
But what if actually i ≥ N?



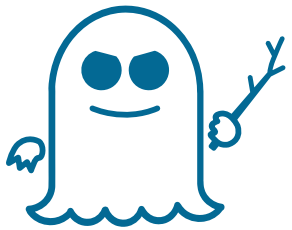
Spectre vulnerability

What if actually i ≥ N?

```
static char T[N] = { ... };

void f(char *data, unsigned i) {
  if (i >= N)
    return -EINVAL;
  char val = T[i];
  return data[val*64];
}
```

- Still predicts that we should proceed
- Reads T[i]



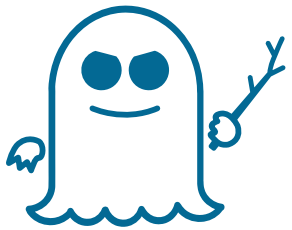
Spectre vulnerability

What if actually i ≥ N?

```
static char T[N] = { ... };

void f(char *data, unsigned i) {
  if (i >= N)
    return -EINVAL;
  char val = T[i];
  return data[val*64];
}
```

- Still predicts that we should proceed
- Reads T[i]
- And reads data[val*64]



Spectre vulnerability

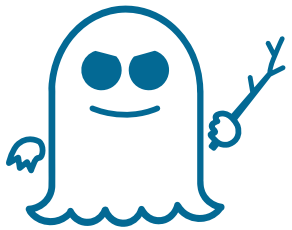
What if actually i >>> N?

```
static char T[N] = { ... };

void f(char *data, unsigned i) {
  if (i >= N)
    return -EINVAL;
  char val = T[i];
  return data[val*64];
}
```

The T[i] read is basically wherever you want!

- Can read whole memory of victim



Spectre vulnerability

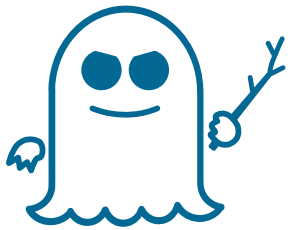
But hey, wait, processor rolls back before returning, doesn't it?

```
static char T[N] = { ... };

void f(char *data, unsigned i) {
  if (i >= N)
    return -EINVAL;
  char val = T[i];
  return data[val*64];
}
```

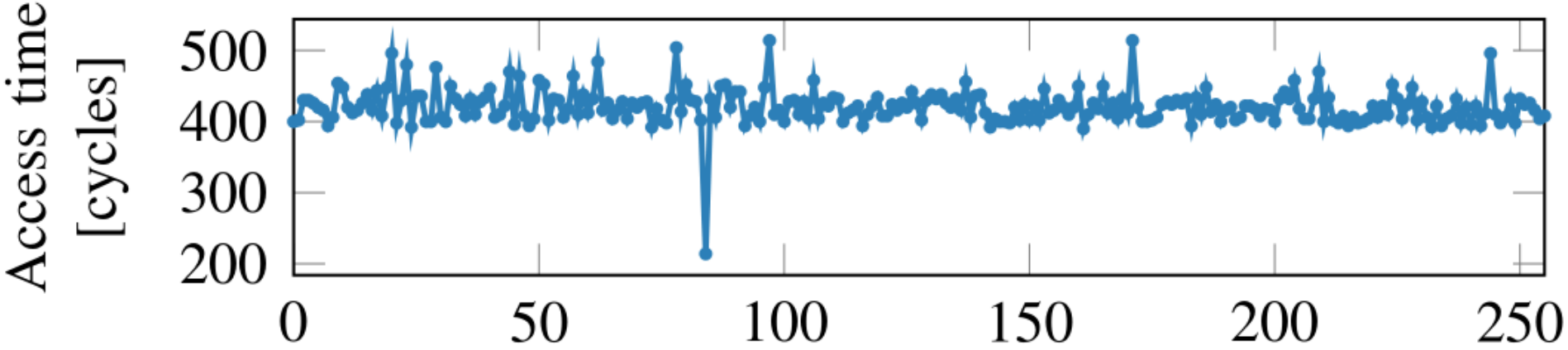
Yes, sure, but the leak is already done

- data[val*64] has been read, i.e. loaded into the cache
- and attacker can measure that
  - Having flushed the cache before calling f, then measuring access



Spectre vulnerability

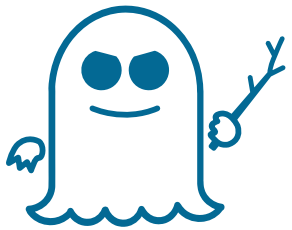
Attacker can measure that



That was 84!

Takes a bit of time

- Unoptimized version still achieves 10KB/s...

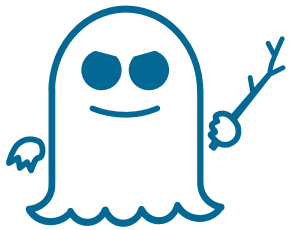


Spectre vulnerability

Sanitize your inputs....

```
static char T[N] = { ... };

void f(char *data, unsigned i) {
  if (i >= N)
    return -EINVAL;
// TODO: Something to flush execution prediction
  char val = T[i];
  return data[val*64];
}
```



Spectre vulnerability

Lessons learnt

All parties were right in their own model

- Speculative execution
- Cache
- Shared memory

It’s the combination which is wrong

- Cache effects in shared memory due to speculative execution

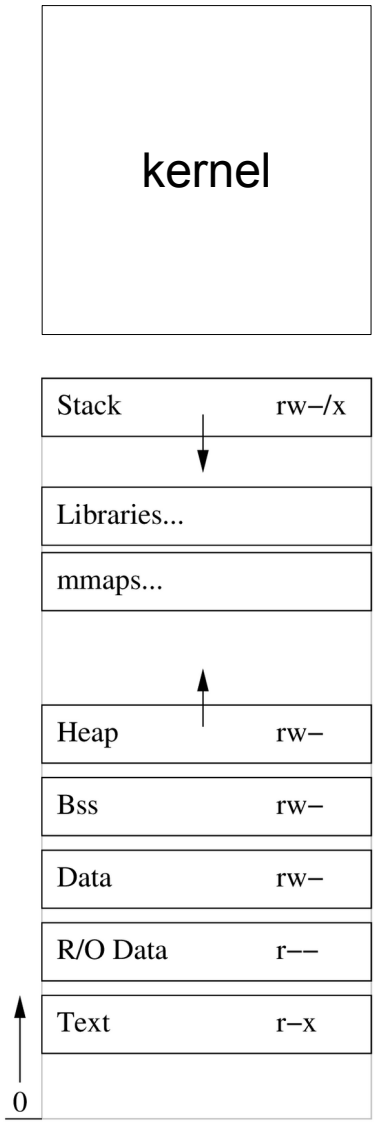
A problem of overall model...

Meltdown vulnerability

Wait, there is even much worse!

Kernel-land and User-land share addressing space

- But kernel space inaccessible to user-land
  - Protected by page flags

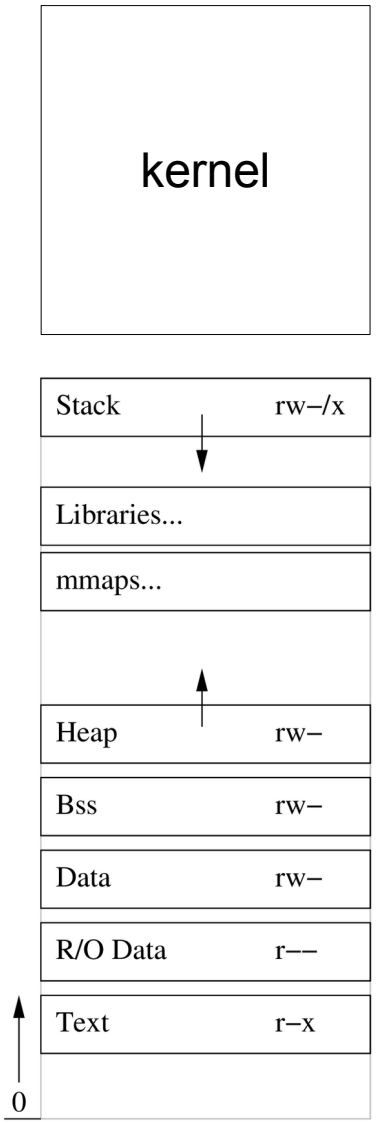


Meltdown vulnerability

Wait, there is even much worse!

```
// Address of net_secret in kernel
char *p = (char*) 0xffffffff825c8590;

char v = *p;
int x = data[v*64];
```



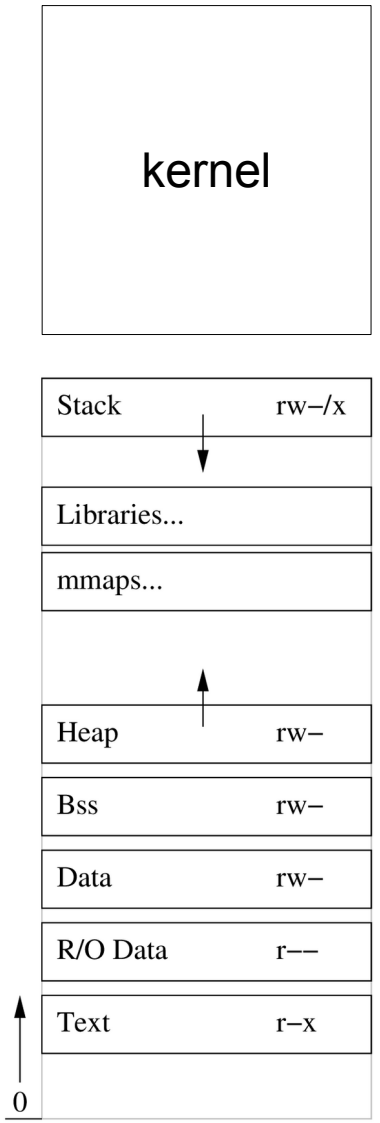
Meltdown vulnerability

Wait, there is even much worse!

```
// Address of net_secret in kernel
char *p = (char*) 0xffffffff825c8590;

char v = *p;
int x = data[v*64];
```

Reading the kernel variable speculated as being OK



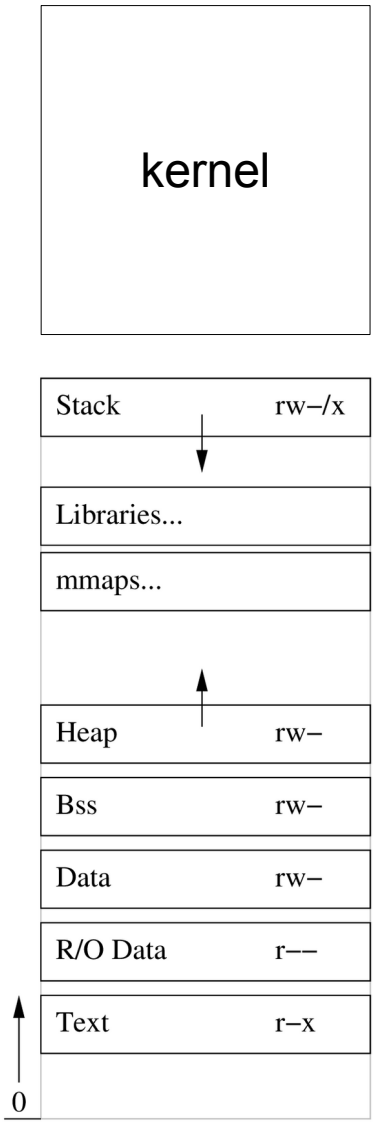
Meltdown vulnerability

Wait, there is even much worse!

```
// Address of net_secret in kernel
char *p = (char*) 0xffffffff825c8590;

char v = *p;
int x = data[v*64];
```

Reading the kernel variable speculated as being OK
Reading data according to the value



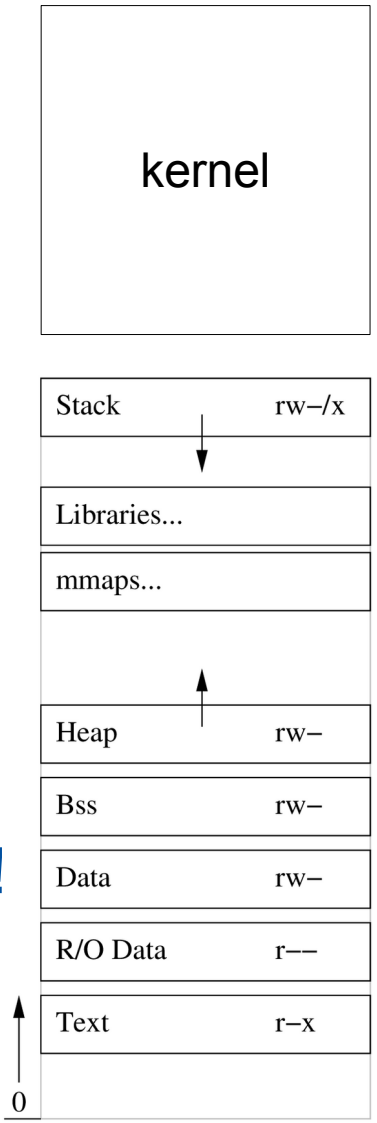
Meltdown vulnerability

Wait, there is even much worse!

```
// Address of net_secret in kernel
char *p = (char*) 0xffffffff825c8590;

char v = *p;
int x = data[v*64];
```

Reading the kernel variable speculated as being OK
Reading data according to the value
Eventually kernel raises SEGFAULT
But userland can still measure which data was loaded!



Meltdown vulnerability

Attacker does not even need to make system calls

- Just read the target variables...

Kernel has the whole memory mapped

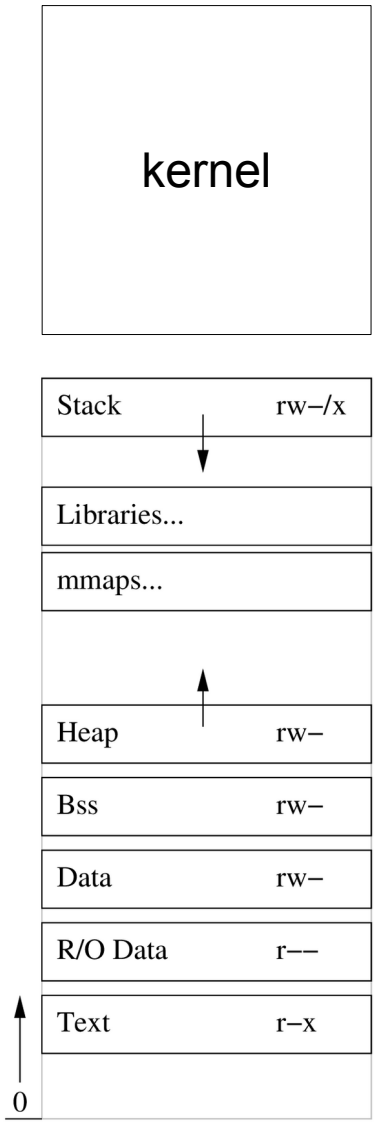
- Can read basically all memory

Real cause: bug in Intel processor

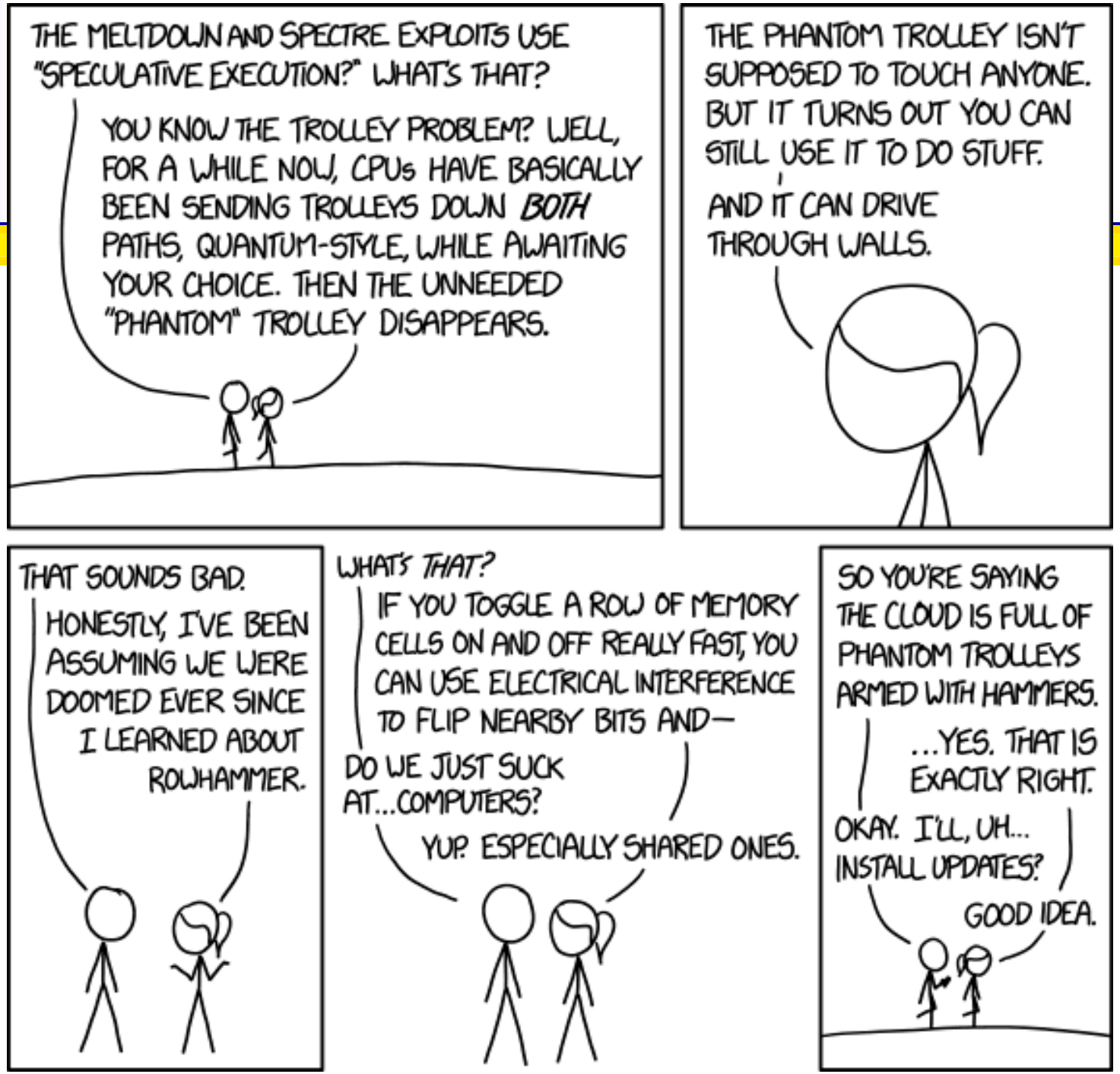
- Should have trapped the access error **before** loading the value, let alone use it!

Efficiency:

- 3.2KB/s - 503KB/s



Conclusion



From https://xkcd.com/1938/