# Processor security

# The processor


Part of the trusted computing base (TCB):

- but is optimized for performance,
  … security may be secondary

Processor design and security:

- Important security features, such as hardware enclaves
- Some features can be exploited for attacks:
  - Speculative execution, transactional memory, …
  - An active area of research!

# Intel SGX

## An overview

# SGX: Goals

Extension to Intel processors that support:

- **Enclaves**: running code and memory isolated from the rest of system
- **Attestation**: prove to local/remote system what code is running in enclave
- **Minimum TCB**: only processor is trusted
  nothing else: DRAM and peripherals are untrusted
  ⇒ all writes to memory are encrypted

# Applications

**Server side**:

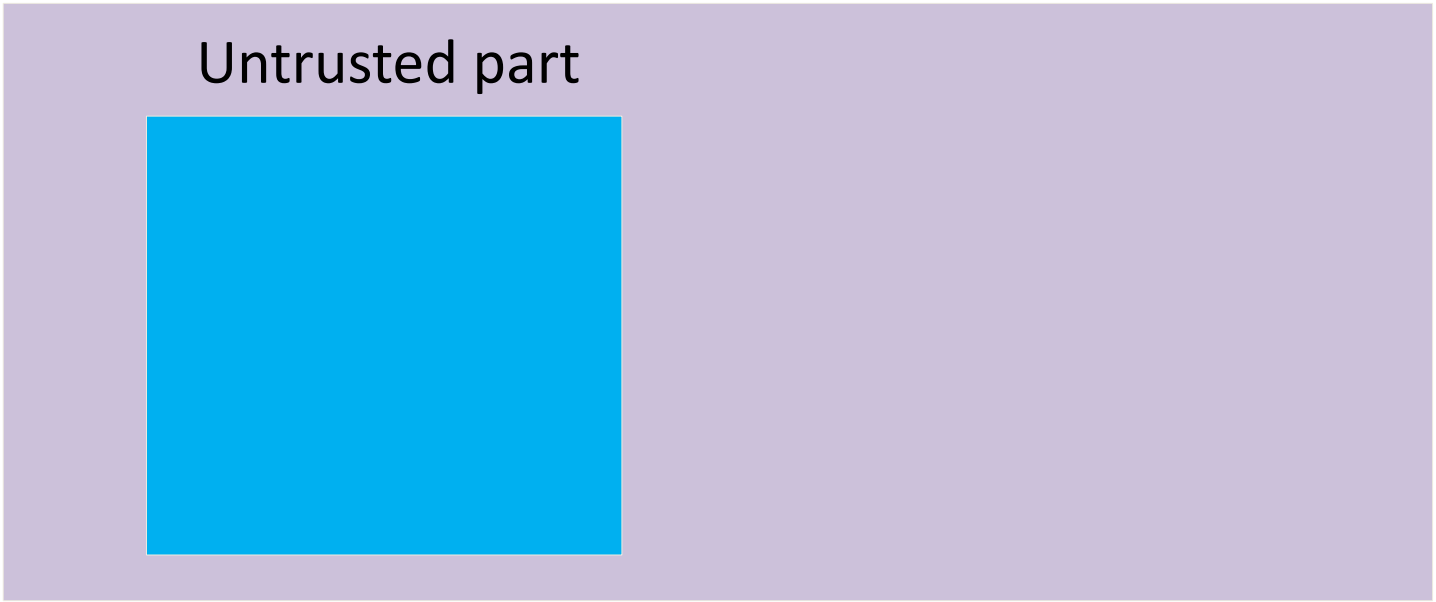
- Storing a Web server HTTPS secret key:
  secret key only opened inside an enclave
  ⇒ malware cannot get the key
- Running a private job in the cloud: job runs in enclave
  Cloud admin cannot get code or data of job

**Client side**:

- Hide anti-virus (AV) signatures:
  AV signatures are only opened inside an enclave
  not exposed to adversary in the clear

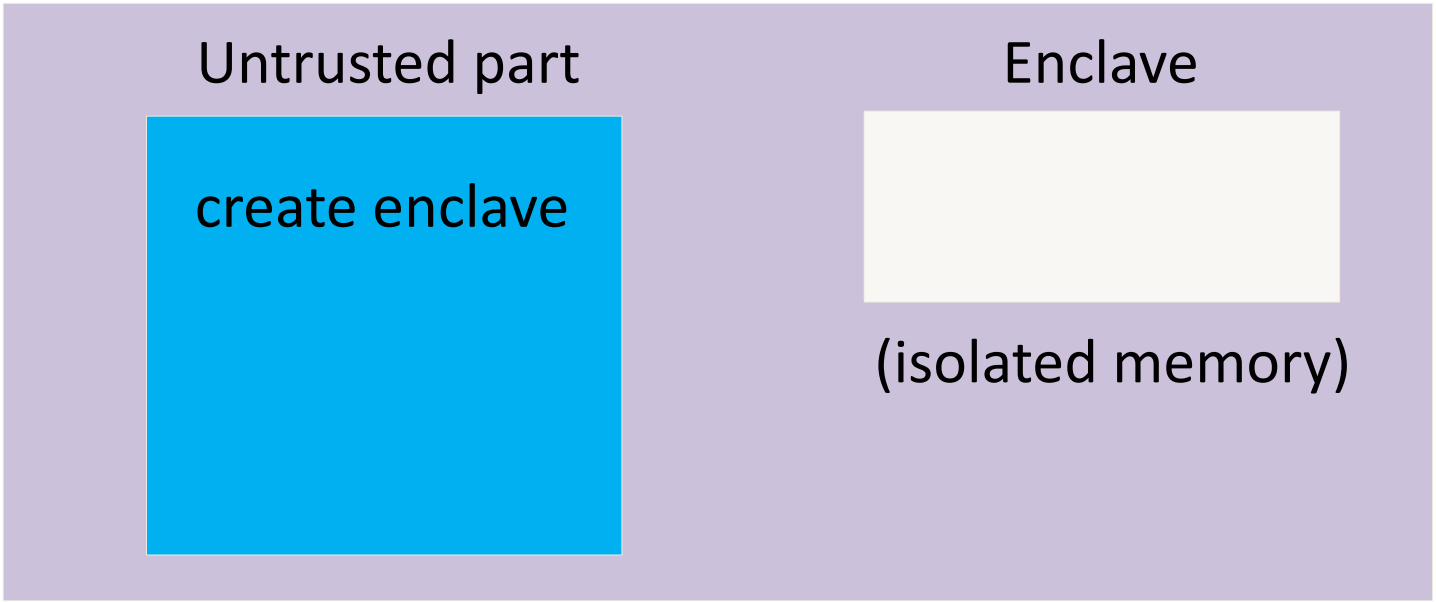
# Intel SGX: how does it work?

An application defines part of itself as an enclave



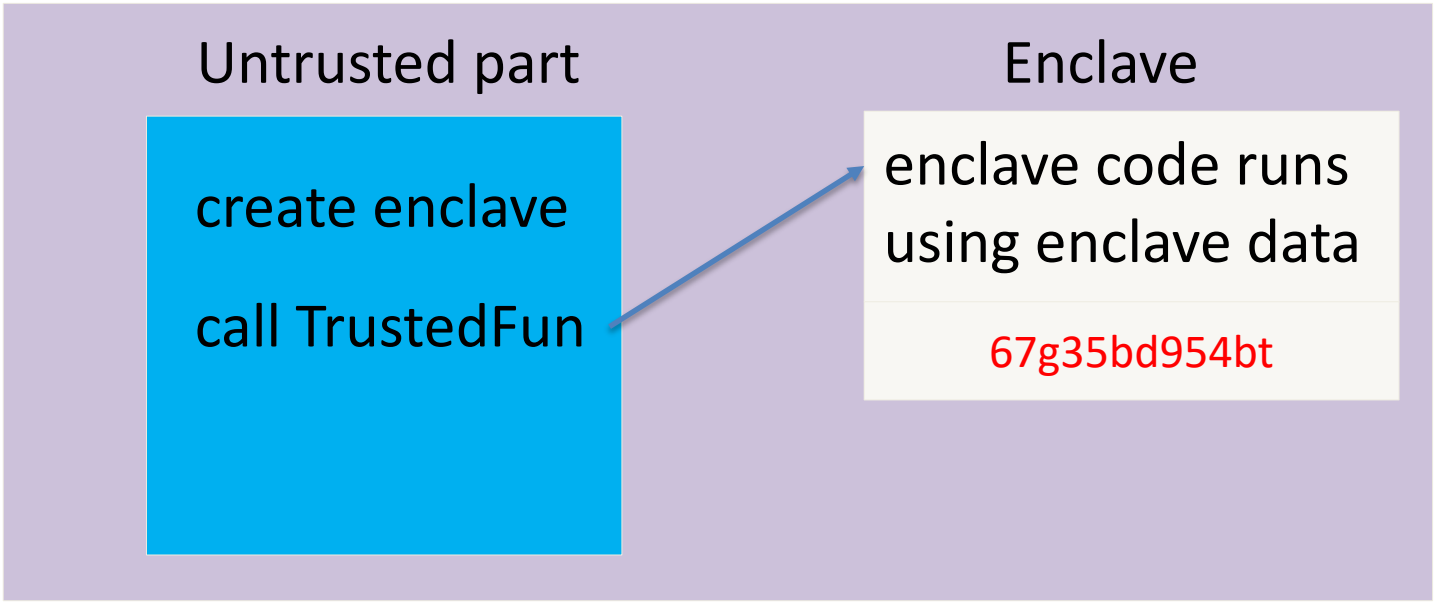
# How does it work?

An application defines part of itself as an enclave



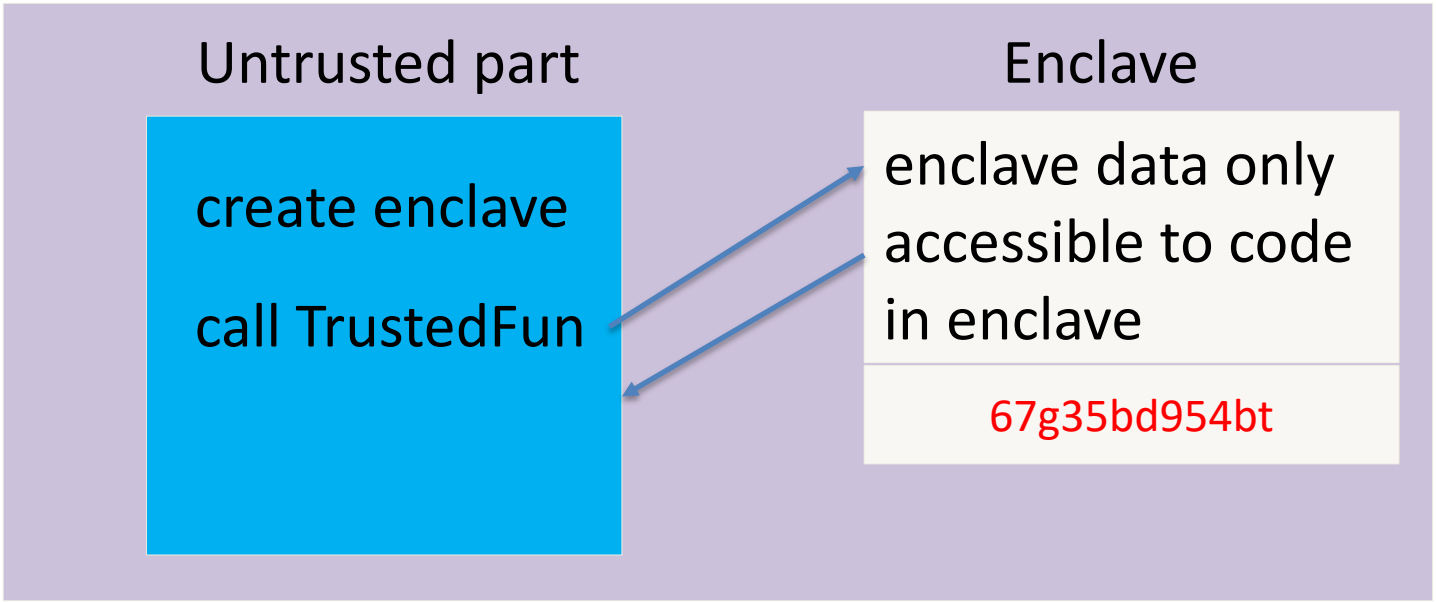
# How does it work?

An application defines part of itself as an enclave



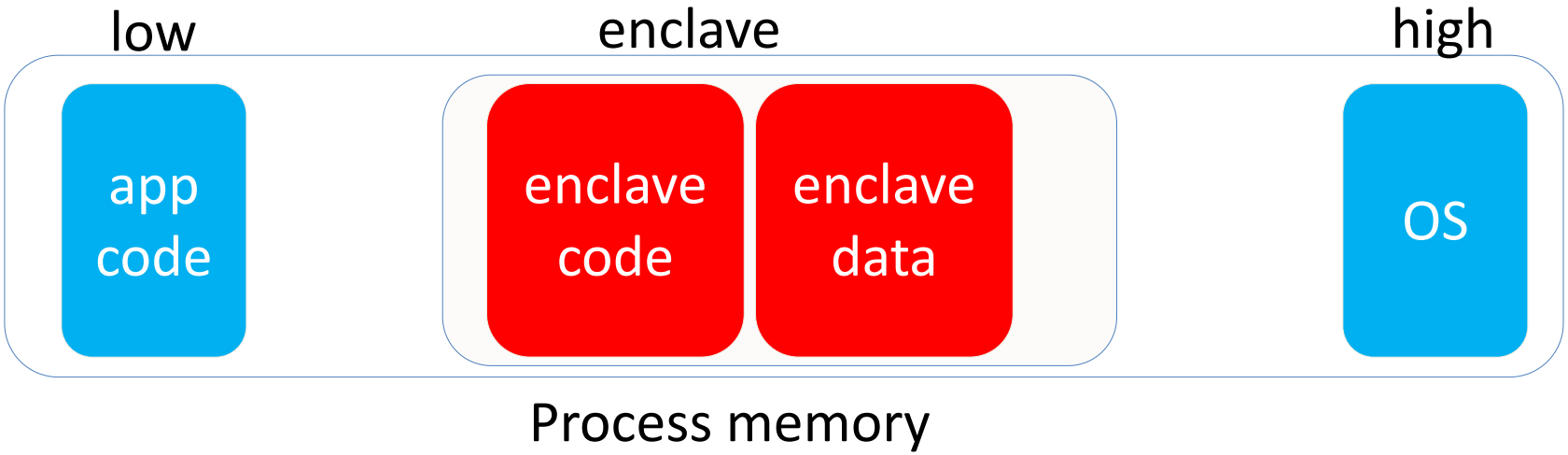
# How does it work?

An application defines part of itself as an enclave



# How does it work?

Part of process memory holds the enclave:




- Enclave code and data are stored encrypted in main memory
- Processor prevents access to cached enclave data outside of enclave.

# Creating an enclave: new instructions

- **ECREATE**: establish memory address for enclave
- **EADD**: copies memory pages into enclave
- **EEXTEND**: computes hash of enclave contents (256 bytes at a time)
- **EINIT**: verifies that hashed content is properly signed
  if so, initializes enclave (signature = RSA-3072)

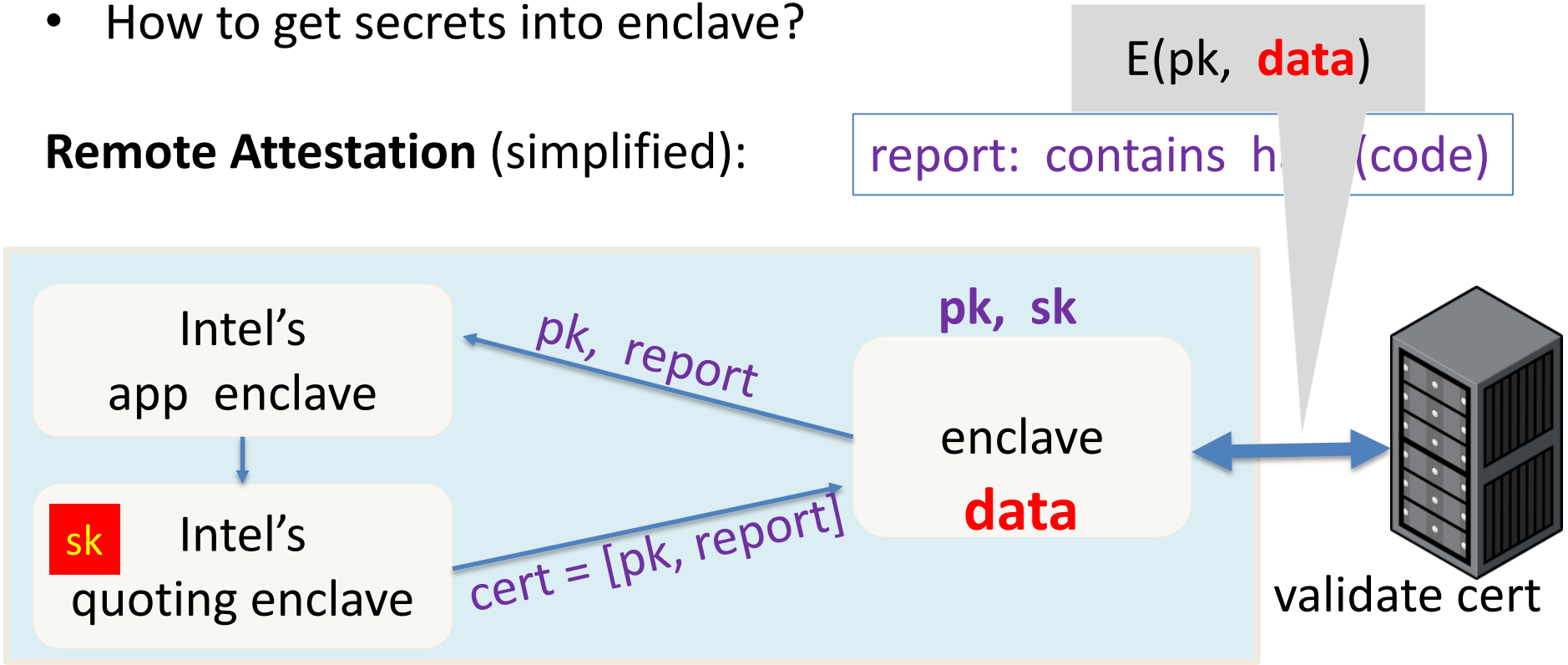
- **EENTER**: call a function inside enclave
- **EEXIT**: return from enclave

# Provisioning enclave with secrets: attestation

The problem: enclave memory is in the clear prior to activation (EINIT)

- How to get secrets into enclave?

**Remote Attestation** (simplified):



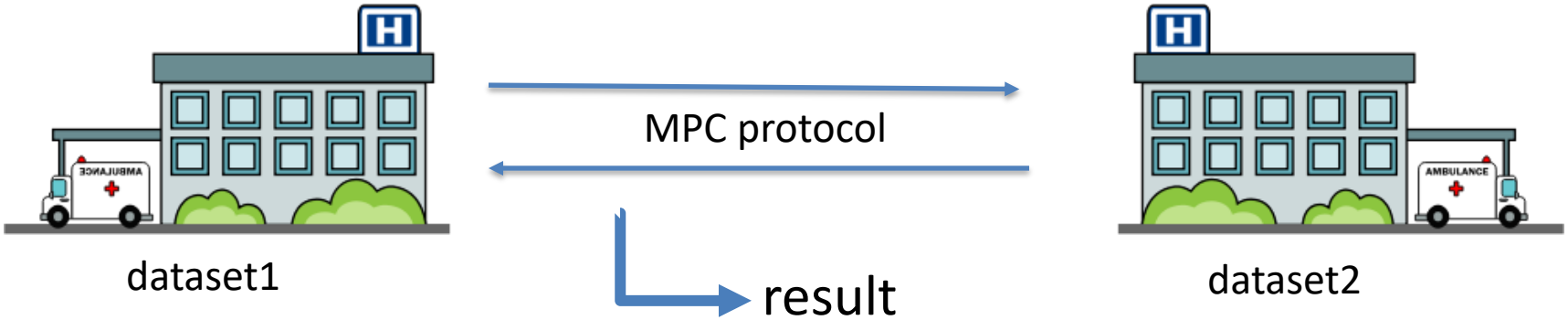
# Summary

SGX: an architecture for managing secret data

- Intended to process data that cannot be read by anyone, except for code running in enclave
- Attestation: proves what code is running in enclave
- Minimal TCB: nothing trusted except for x86 processor
- Not suitable for legacy applications

# An example application

Data science on federated data:

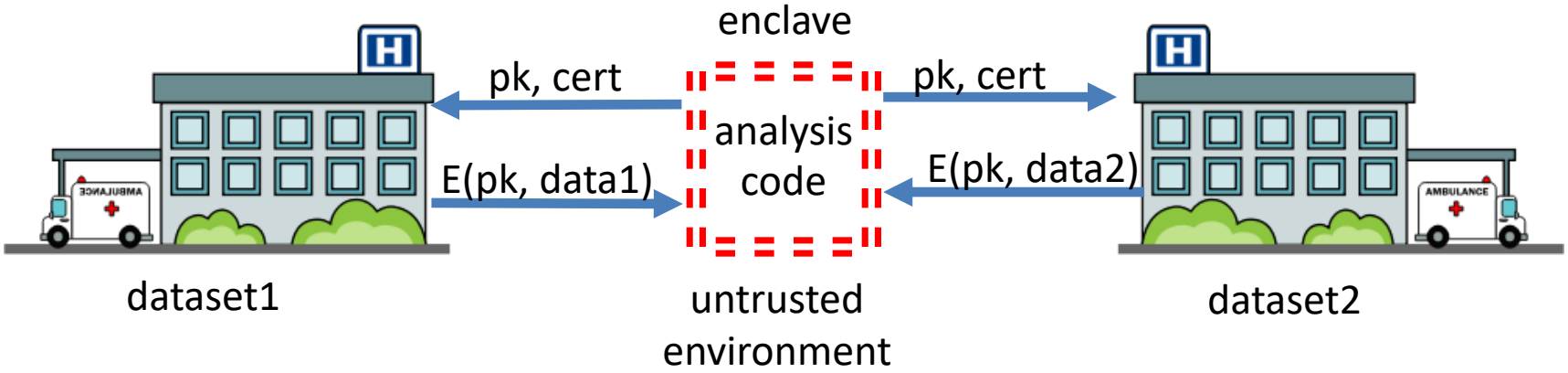



Can we run analysis on union(dataset1, dataset2) ??

For simple computations, can use multiparty computation (MPC)

# An example application

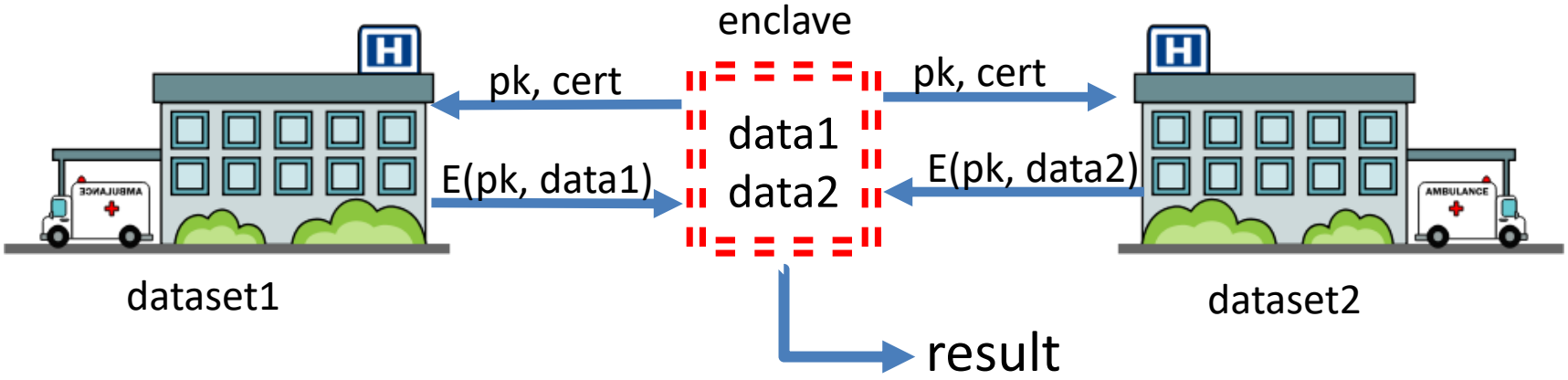
Data science on federated data:




For more complex analysis, can use (secure) hardware enclave

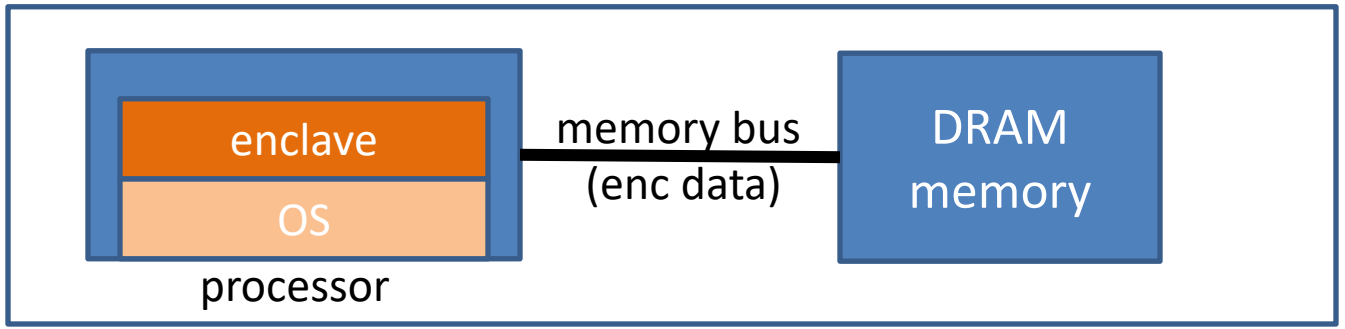
# An example application

Data science on federated data:




For more complex analysis, can use (secure) hardware enclave

# SGX insecurity: (1) side channels

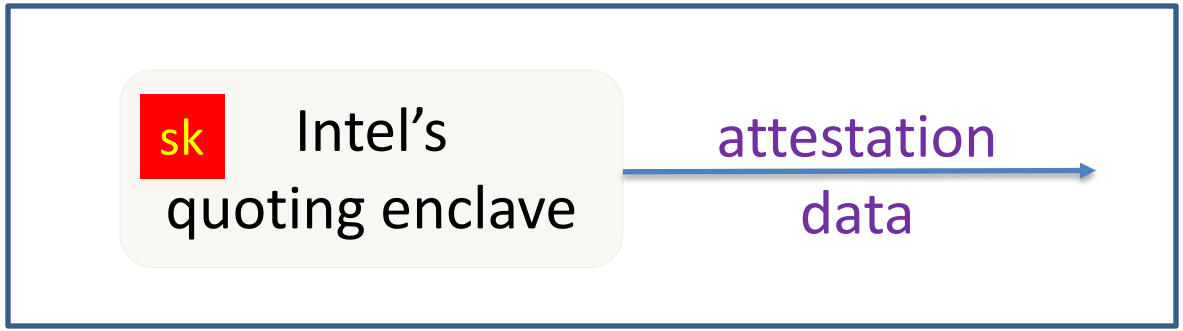



Attacker controls the OS. OS sees lots of side-channel info:

- Memory access patterns
- State of processor caches as enclave executes
- State of branch predictor

All can leak enclave data. Difficult to block.

# SGX insecurity: (2) extract quoting key




Attestation: proves to 3rd party what code is running in enclave

- Quoting **sk** stored in Intel enclave on untrusted machines

What if attacker extracts **sk** from some quoting enclave?

- Can attest to arbitrary non-enclave code

… see Foreshadow attack and Intel's response

# The Spectre attack

Speed vs. security in HW

[slides credit: Paul Kocher]

# Performance drives CPU purchases

Clock speed maxed out:

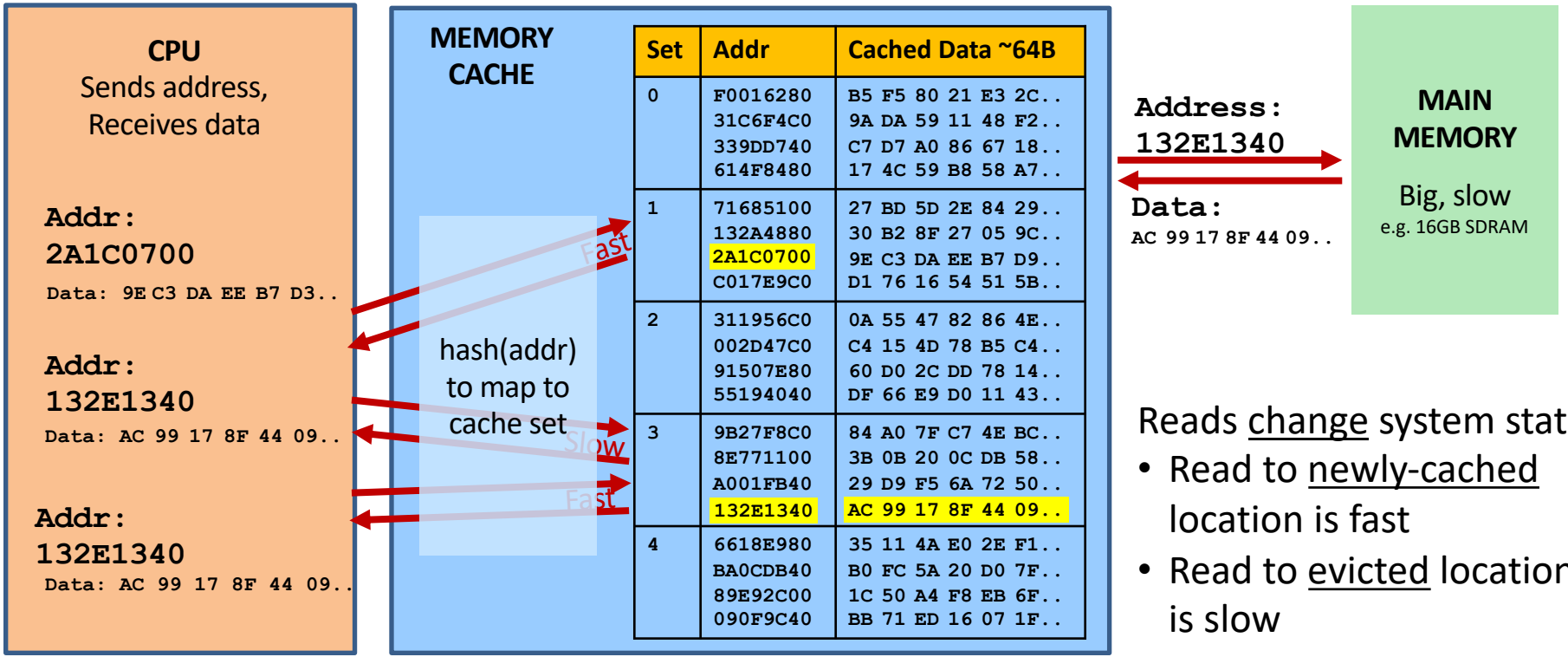
- Pentium 4 reached 3.8 GHz in 2004
- Memory latency is slow and not improving much

To gain performance, need to do more per cycle!

- Reduce memory delays ⟶ caches
- Work during delays ⟶ speculative execution

# Memory caches (4-way associative)

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory




Reads change system state:

- Read to newly-cached location is fast
- Read to evicted location is slow

# Speculative execution

CPUs can *guess* likely program path and do speculative execution

- Example:

```
if  (uncached_value == 1)     // load from memory
      a = compute(b)
```

- Branch predictor guesses if() is 'true'  (based on prior history)
- Starts executing *compute(b)* speculatively

- When value arrives from memory, check if guess was correct:
  - **Correct**:  Save speculative work ⇒ performance gain
  - **Incorrect**:  Discard speculative work ⇒ no harm (?)

**Architectural Guarantee**

Register values eventually match result of in-order execution

**Speculative Execution**

CPU regularly performs incorrect calculations, then deletes mistakes

Is making + discarding mistakes the same as in-order execution?

The processor executed instructions that were not supposed to run !!

The problem: instructions can have observable side-effects

# Conditional branch (Variant 1) attack

```
if (x < array1_size)
   y = array2[ array1[x]*4096 ];
```

Suppose `unsigned int x` comes from untrusted caller

Execution without speculation is safe:

`array2[array1[x]*4096]` not eval unless `x < array1_size`

What about with speculative execution?

# Conditional branch (Variant 1) attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

**Before attack:**

- Train branch predictor to expect if() is true (e.g. call with `x` < `array1_size`)
- Evict `array1_size` and `array2[]` from cache

## Memory & Cache Status

`array1_size = 00000008`

Memory at `array1` base:
8 bytes of data (value doesn't matter)

Memory at `array1` base+1000:
**`09`** `F1 98 CC 90...` (something secret)

```
array2[ 0*4096]
array2[ 1*4096]
array2[ 2*4096]
array2[ 3*4096]
array2[ 4*4096]
array2[ 5*4096]
array2[ 6*4096]
array2[ 7*4096]
array2[ 8*4096]
array2[ 9*4096]
array2[10*4096]
array2[11*4096]
  ...
```

Contents don't matter
only care about cache ***status***

Uncached Cached

# Conditional branch (Variant 1) attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Attacker calls victim with `x`=1000

Speculative exec while waiting for `array1_size`:

- Predict that if() is true
- Read address (`array1` base + `x`)
  (using out-of-bounds `x`=1000)
- Read returns secret byte = **09**
  (in cache ⇒ fast )

**Memory & Cache Status**

`array1_size = 00000008` ←

Memory at `array1` base:
8 bytes of data (value doesn't matter)

Memory at `array1` base+1000:
**09** F1 98 CC 90... (something secret)

```
array2[ 0*4096]
array2[ 1*4096]
array2[ 2*4096]
array2[ 3*4096]
array2[ 4*4096]
array2[ 5*4096]
array2[ 6*4096]
array2[ 7*4096]
array2[ 8*4096]
array2[ 9*4096]
array2[10*4096]
array2[11*4096]
  ...
```

Contents don't matter
only care about cache ***status***

Uncached Cached

# Conditional branch (Variant 1) attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Attacker calls victim with x=1000

Next:

- Request mem at (array2 base + 09*4096)
- Brings array2[09*4096] into the cache
- Realize if() is false: discard speculative work

Finish operation & return to caller

## Memory & Cache Status

array1_size = 00000008

Memory at array1 base:
8 bytes of data (value doesn't matter)

Memory at array1 base+1000:
09 F1 98 CC 90... (something secret)

array2[ 0*4096]
array2[ 1*4096]
array2[ 2*4096]
array2[ 3*4096]
array2[ 4*4096]
array2[ 5*4096]
array2[ 6*4096]
array2[ 7*4096]
array2[ 8*4096]
array2[ 9*4096]
array2[10*4096]
array2[11*4096]
...

Contents don't matter
only care about cache ***status***

Uncached Cached

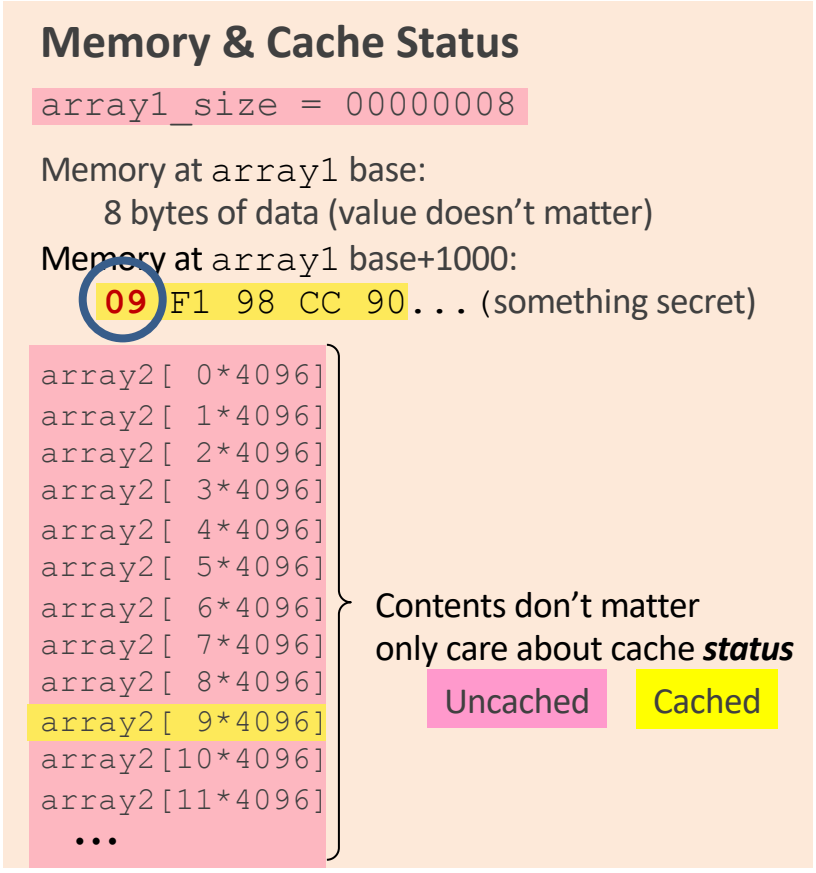
# Conditional branch (Variant 1) attack

```
if (x < array1_size)
   y = array2[array1[x]*4096];
```

Attacker calls victim with `x`=1000

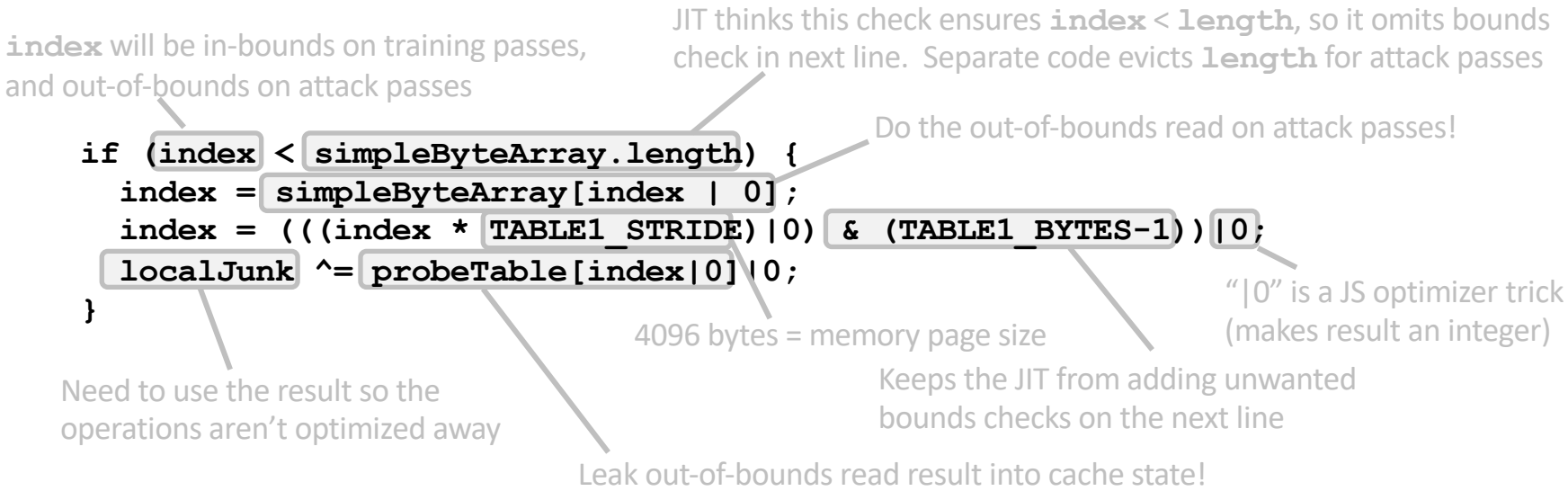
Attacker:

- measures read time for `array2[i*4096]`
- Read for `i`=**09** is fast (cached), reveals secret byte !!
- Repeat with many `x` (10KB/s)



# Violating JavaScript's sandbox

- Browsers run JavaScript from untrusted websites
  - JIT compiler inserts safety checks, including bounds checks on array accesses
- Speculative execution runs through safety checks…




Can evict length/probeTable from JavaScript (easy)

… then use timing to detect newly-cached location in probeTable

# Variant 2: indirect branches

Indirect branches: can go anywhere , e.g. `jmp[rax]`

- If destination is delayed, CPU guesses and proceeds speculatively
- Find an indirect jmp with attacker controlled register(s)
  … then cause mispredict to a useful ‘gadget’

Attack steps:

- **Mistrain** branch prediction so speculative execution will go to gadget
- **Evict** address [rax] from cache to cause speculative execution
- **Execute** victim so it runs gadget speculatively
- **Detect** change in cache state to determine memory data

# Non-mitigations

Can we prevent Spectre without a huge cost in performance?

**Idea 1:** fully restore cache state when speculation fails.

**Problem:** Insecure!

Speculative execution can have observable side effects beyond the cache state

```
if (x < array1_size) {
    y = array1[x];
    do_something_observable(y);
}
```

occupy a bus: detectable from another core, or cause EM radiation

# **Variant 1 mitigation:** Speculation stopping instruction (e.g. `LFENCE`)

- Idea: Software developers insert LFENCE on all vuln. code paths
- Claim: efficient, no performance impact on benchmarks software

| | |
|---|---|
| Insert LFENCEs manually?  | Often millions of control flow paths<br>Too confusing - speculation runs 188++ instructions, crosses modules<br>Too risky – miss one and attacker can read entire process memory |
| Put LFENCES everywhere?  | Abysmal performance - LFENCE is very slow<br>Not in binary libraries, compiler-created code patterns |
| Insert by smart compiler?  | Protect all potentially-exploitable patterns = too slow<br>– Compilers judged by performance, not security<br>Protect only known-bad bad patterns = unsafe<br>– Microsoft Visual C/C++ `/Qspectre` unsafe for 13 of 15 tests |

Transfer of blame (CPU -> SW): "you should have put an `LFENCE` there"

# Mitigations: Indirect branch variant

Remove all branches?

DOOM with no branches:

- One frame every ~7 hours

**A branchless DOOM**

This directory provides a branchless, mov-only version of the classic DOOM video game.


*DOOM, running with only mov instructions.*

This is thought to be entirely secure against the Meltdown and Spectre CPU vulnerabilities, which require speculative execution on branch instructions.

Oops!  Variant 4:  speculative store

# Mitigations: summary

Mitigations are messy for all Spectre variants:

- Software must deal with microarchitectural complexity
- Mitigations for all variants are really hard to test:
  - formal models beginning to appear.

**More ideas desperately needed !**

# ... but there is more

More speculative execution attacks:

- **Meltdown**
- Rogue inflight data load (**RIDL**) and **Fallout**
- **ZombieLoad**
- **Store-to-leak forwarding**

Enable reading unauthorized memory  (client, cloud, SGX)

- Mitigating incurs significant performance costs

THE END