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

Untrusted part

Process memory
How does it work?

An application defines part of itself as an enclave

Untrusted part

create enclave

Enclave

(isolated memory)

Process memory
How does it work?

An application defines part of itself as an enclave

- Untrusted part
  - create enclave
  - call TrustedFun

- Enclave
  - enclave code runs using enclave data
  - 67g35bd954bt

Process memory
How does it work?

An application defines part of itself as an enclave

Untrusted part
- create enclave
- call TrustedFun

Enclave
- enclave data only accessible to code in enclave
  - 67g35bd954bt

Process memory
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):

- Intel’s app enclave
- Intel’s quoting enclave
- pk, report
- pk, sk
- data
- E(pk, data)
- report: contains hash (code)

validate cert

cert = [pk, report]
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 $\text{union}(\text{dataset1, dataset2})$??

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 $\rightarrow$ caches
- Work during delays $\rightarrow$ speculative execution
Memory caches

(4-way associative)

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

CPU
Sends address, Receives data

Addr: 2A1C0700
Data: 9E C3 DA EE B7 D3

Addr: 132E1340
Data: AC 99 17 8F 44 09

Addr: 132E1340
Data: AC 99 17 8F 44 09

MEMORY CACHE

hash(addr) to map to cache set

Set | Addr | Cached Data ~64B
--- | --- | ---
0  | F0016280 31C6F4C0 339DD740 614F8480 | B5 F5 80 21 E3 2C 9A DA 59 11 48 2F C7 D7 A0 86 67 18 17 4C 59 B8 58 A7
1  | 71685100 132A4880 2A1C0700 C017E9C0 | 27 BD 5D 2E 84 29 30 B2 8F 27 05 9C 9E C3 DA EE B7 D9 D1 76 16 54 51 5B
2  | 311956C0 002D47C0 91507E80 55194040 | 0A 55 47 82 86 4E C4 15 4D 78 B5 C4 60 D0 2C DD 78 14 6F 66 E9 D0 11 43
3  | 9B27F8C0 8E771100 A001FB40 132E1340 | 84 A0 7F C7 4E BC 3B 08 20 0C DB 58 29 D9 F5 6A 72 50 AC 99 17 8F 44 09
4  | 661E8980 BA0CD840 89B92C00 090F9C40 | 35 11 4A E0 2E F1 B0 FC 5A 20 D0 7F 1C 50 A4 F8 EB 6F BB 71 ED 16 07 1F

Address: 132E1340
Data: AC 99 17 8F 44 09

MAIN MEMORY
Big, slow e.g. 16GB SDRAM

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:**

```plaintext
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 (?)
Speculative Execution

CPU regularly performs incorrect calculations, then deletes mistakes

Architectural Guarantee

Register values eventually match result of in-order execution

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

```c
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

Before attack:

- Train branch predictor to expect if() is true (e.g. call with $x < \text{array1\_size}$)
- Evict $\text{array1\_size}$ and $\text{array2}[\cdot]$ from cache

```c
if (x < array1_size)
    y = array2[array1[x]*4096];
```
Conditional branch (Variant 1) attack

```c
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

| Memory at array1 base: 8 bytes of data (value doesn’t matter) |
| Memory at array1 base+1000: F1 98 CC 90... (something secret) |

- Uncached
- Cached

Contents don’t matter only care about cache status
Conditional branch (Variant 1) attack

```c
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
Conditional branch (Variant 1) attack

```c
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...

```javascript
if (index < simpleByteArray.length) {
  index = simpleByteArray[index | 0];
  index = (((index * TABLE1_STRIDE)|0) & (TABLE1_BYTES-1))|0;
  localJunk ^= probeTable[index|0]|0;
}
```

- `index` will be in-bounds on training passes, and out-of-bounds on attack passes
- JIT thinks this check ensures `index < length`, so it omits bounds check in next line. Separate code evicts `length` for attack passes
- Do the out-of-bounds read on attack passes!
- 4096 bytes = memory page size
- Keeps the JIT from adding unwanted bounds checks on the next line
- Leak out-of-bounds read result into cache state!
- “|0” is a JS optimizer trick (makes result an integer)
- Need to use the result so the operations aren’t optimized away

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. \texttt{jmp[rax]}

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

Attack steps:

- \textbf{Mistrain} branch prediction so speculative execution will go to gadget
- \textbf{Evict} address [rax] from cache to cause speculative execution
- \textbf{Execute} victim so it runs gadget speculatively
- \textbf{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

```c
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

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

**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

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