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: hardware enclaves, memory encryption (TME), RDRAND, and others.

• Some features can be exploited for attacks:
  – Speculative execution, transactional memory, ...
Intel SGX / TDX

An overview

Software Guard eXtensions (SGX)
Trust Domain eXtensions (TDX)

Also AMD SME and SEV
SGX / TDX: Goals

Extension to Intel processors that support:

• **Enclaves**: running code and memory isolated from the rest of system (code outside of enclave cannot read enclave memory)

• **Attestation**: prove to a remote system what code is running in the enclave

• **Minimum TCB**: only processor is trusted, nothing else!
  - RAM and peripherals are untrusted
  - ⇒ Memory controller must encrypt all writes to RAM (TME)
**Goal**: move data & VM to the cloud, without cloud seeing them in the clear

- Simple solution: encrypt data and VM, key stays with Client
- The problem: now cloud cannot search or compute on data
  ⇒ defeats the purpose of cloud computing
A solution: a HW enclave

Goal: no one can read the memory of the isolated VM; not the hypervisor, and not even a malicious admin.

How does this work? Will see in a minute.
Why enclaves 2: protecting keys

Storing a Web server HTTPS secret key:

secret key is only available in the clear inside an enclave

⇒ malware cannot extract the key
Intel SGX: how does it work?

An application defines part of itself as an enclave

Regular (non-isolated) memory

Process memory
How does it work?

An application defines part of itself as an enclave

Regular (non-isolated) memory

create enclave

Enclave

isolated memory in process memory space

Process memory
How does it work?

An application defines part of itself as an enclave

Regular (non-isolated) memory
- create enclave
- call TrustedFun
- code here cannot read enclave memory

Enclave
- enclave code runs using enclave data
- enclave memory: only readable by enclave code

Process memory
How does it work?

Part of process memory holds the enclave:

- Processor prevents access to cached enclave data outside of enclave.
- Enclave code and data are written encrypted to RAM (TME)
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

Enclave init code loaded as cleartext
When to send secret data to enclave: **attestation**

**The problem:** How does a remote system know when it can trust an enclave with its data?

**Remote Attestation (simplified):**

- Intel’s app enclave
- Intel’s quoting enclave
- remote server

```
E(pk, data)
```

`report`: contains hash of code

```
E(pk, data)
```

`pk`, `sk`

```
report = [pk, report]
```

```
cert = [pk, report]
```

```
data
```

```
validate cert
```

SGX Summary

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 main processor
  \[\Rightarrow\] Memory controller encrypts all writes to RAM

• Not suitable for legacy applications: must split app into parts
  • Requires lots of code rewriting ... not suitable for legacy apps
TDX Briefly: an easy-to-use enclalve

TDX: puts an entire VM in an enclave (e.g., an entire web server)

- Support for attestation and minimal TCB (as with SGX)
- Isolated VMs are managed by a new Intel TDX Module
  - The TDX module is implemented in signed code by Intel
  - It is loaded into an isolated region of physical memory
  - Creates, manages, and attests to isolated VMs

https://cdrdv2.intel.com/v1/dl/getContent/690419
One more example application

Data science on federated data:

Can we run analysis on $\text{union(}\text{dataset1}, \text{dataset2})$ ??

cryptographic solutions (e.g., MPC) work for simple computations
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

Cert includes hash of enclave code

E(pk, data1)
E(pk, data2)
result
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 \textit{sk} stored in Intel enclave on untrusted machines

What if attacker extracts \textit{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

<table>
<thead>
<tr>
<th>Set</th>
<th>Addr</th>
<th>Cached Data ~64B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F0016280</td>
<td>B5 F5 80 21 E3 2C..</td>
</tr>
<tr>
<td></td>
<td>31C6F4C0</td>
<td>9A DA 59 11 48 F2..</td>
</tr>
<tr>
<td></td>
<td>339DD740</td>
<td>C7 D7 A0 86 67 18..</td>
</tr>
<tr>
<td></td>
<td>614F8480</td>
<td>17 4C 59 B8 58 A7..</td>
</tr>
<tr>
<td>1</td>
<td>71685100</td>
<td>27 BD 5D 2E 84 29..</td>
</tr>
<tr>
<td></td>
<td>132A4880</td>
<td>30 B2 8F 27 05 9C..</td>
</tr>
<tr>
<td></td>
<td>2A1C0700</td>
<td>9E C3 DA EE B7 D9..</td>
</tr>
<tr>
<td></td>
<td>C017E9C0</td>
<td>D1 76 16 54 51 5B..</td>
</tr>
<tr>
<td>2</td>
<td>311956C0</td>
<td>0A 55 47 B2 86 4E..</td>
</tr>
<tr>
<td></td>
<td>002D47C0</td>
<td>C4 15 4D 78 B5 C4..</td>
</tr>
<tr>
<td></td>
<td>91507E80</td>
<td>60 D0 2C DD 78 14..</td>
</tr>
<tr>
<td></td>
<td>55194040</td>
<td>DF 66 E9 D0 11 43..</td>
</tr>
<tr>
<td>3</td>
<td>9B27F8C0</td>
<td>84 A0 7F C7 4E BC..</td>
</tr>
<tr>
<td></td>
<td>8E771100</td>
<td>3B 0B 20 0C DB 58..</td>
</tr>
<tr>
<td></td>
<td>A001FB40</td>
<td>29 D9 F5 6A 72 50..</td>
</tr>
<tr>
<td></td>
<td>132E1340</td>
<td>AC 99 17 8F 44 09..</td>
</tr>
<tr>
<td>4</td>
<td>6618E980</td>
<td>35 11 4A E0 2E F1..</td>
</tr>
<tr>
<td></td>
<td>B0ACDB40</td>
<td>B0 FC 5A 20 D0 7F..</td>
</tr>
<tr>
<td></td>
<td>89B2C00</td>
<td>1C 50 A4 F8 EB 6F..</td>
</tr>
<tr>
<td></td>
<td>090F9C40</td>
<td>BB 71 ED 16 07 1F..</td>
</tr>
</tbody>
</table>

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

hash(addr)
to map to
cache set

hash(addr) to map to cache set

Fast

Slow

Fast

Fast

Evict to
make room

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:

```cpp
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

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]
... (Uncached)

Contents don’t matter
only care about cache status

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 )
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 \( \text{array2 base} + 09 \cdot 4096 \)
- Brings \( \text{array2}[09 \cdot 4096] \) into the cache
- Realize if() is false: discard speculative work

proceed to next instruction
Conditional branch (Variant 1) attack

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

Attacker calls victim with x=1000

**Attacker**: (another process or core)
- for i=0 to 255:
  - measure read time for array2[i*4096]
- When i=09 read is fast (cached), reveals secret byte !!
- Repeat with x=1001, 1002, ... (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;
}
```

4096 bytes = memory page size

“|0” is a JS optimizer trick (makes result an integer)

JIT thinks this check ensures `index < length`, so it omits bounds check in next line. Separate code evicts `length` for attack passes

Need to use the result so the operations aren’t optimized away

Leak out-of-bounds read result into cache state!

Can evict length and 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’

```
y = array2[array1[x]*4096];
```  

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

```c
if (x < array1_size) {
    y = array1[x];
    do_something_observable(y);
}
```
Variant 1 mitigation: Speculation stopping instruction (e.g. LFENCE)

- Idea: insert **LFENCE** on all vuln. code paths

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

**LFENCE**: stops speculative execution.
Variant 1 mitigation: Speculation stopping instruction (e.g. LFENCE)

Put LFENCES everywhere? ⇒ Abysmal performance

Insert by smart compiler? Must protect against all potentially-exploitable patterns
Supported in LLVM, along with other mitigations ⇒ protects all LLVM-based compilers

Transfer of blame (CPU -> SW): “you should have put an LFENCE there”
Mitigations: summary

Mitigations are non-trivial for all Spectre variants:

- Software must deal with microarchitectural complexity
- Mitigations are hard to test:
  - an active area of research (see Prof. Caroline Trippel’s work)

More ideas needed!
... but there is more

More speculative execution attacks:

- **Meltdown**
- Rogue inflight data load (**RIDL**) and **Fallout**
- **ZombieLoad**
- **Micro-op caches** (June 2020)
- **Pointer prefetching in Apple’s M1** (March 2024)

Enable reading unauthorized memory (client, cloud, SGX)

- Mitigating incurs significant performance costs
How to evaluate a processor?

Processors are measured by their performance on benchmarks:

• Processor vendors add many architectural features to speed-up benchmarks

• Until recently: security implications were secondary

⇒ lots of security issues found in last few years

... likely more will be found in coming years
THE END