Side-Channel Threats and Protections

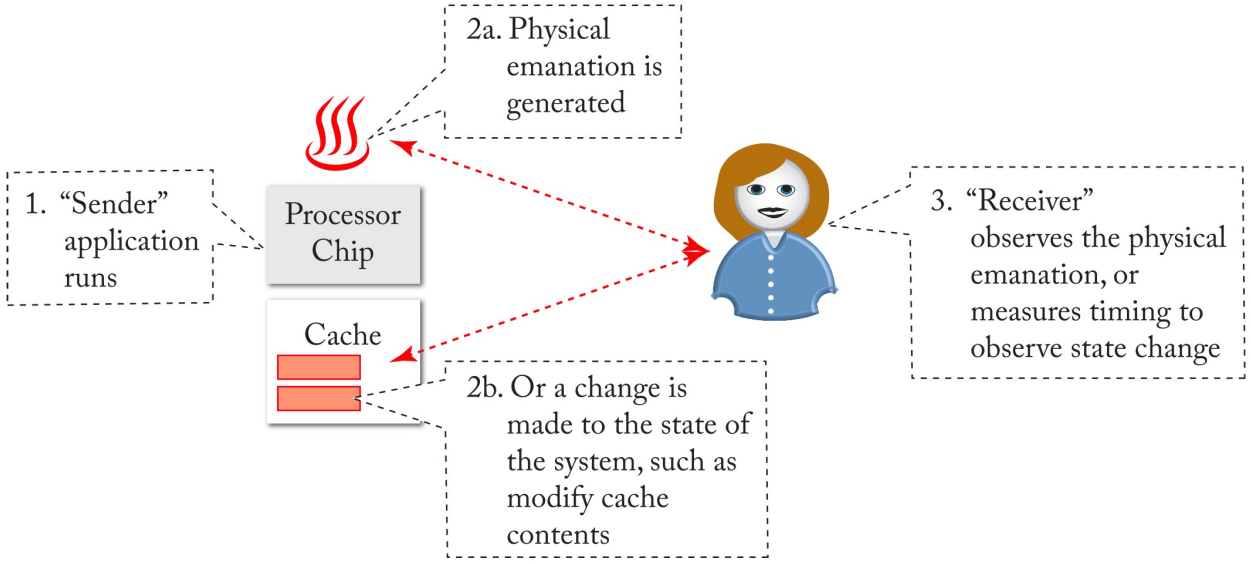
Chapter 8

[1] J. Szefer, “Principles of secure processor architecture design,” *Synth. Lect. Comput. Archit.*, vol. 13, no. 3, pp. 1–173, 2018.

https://caslab.csl.yale.edu/tutorials/asplos2021

Covert Channel

- **Intended** communication between sender and receiver via a medium not designed to be a communication channel
- Typically leverage observables changes in timing, power consumption, physical emanations (EM, acoustic, etc…)
- Cover channels are a means of communicating information (bypassing system protections)
  - E.g., **my program** access memory pattern generating radio signal https://arxiv.org/pdf/2012.06884.pdf



Side Channel

- **Unintended** information from "sender/**victim**" to the "receiver/**attacker**"

- Rather leaking of information is a side effect of how the hardware or software is implemented

- Sender is not aware of being observed
  - E.g., **any program** inside an Intel SGX enclave (sender) will **modify cache** content. Other program (receiver) can infer from that modification secret information(e.g., a secret key)

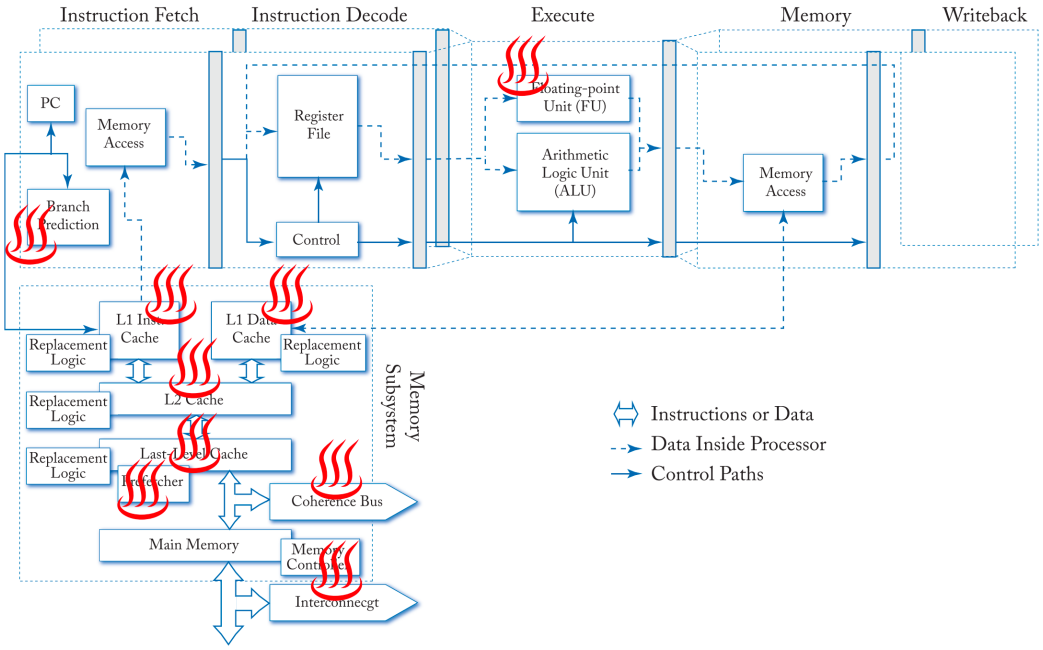
- Can also work in reverse: from **attacker** to **victim**
  - E.g., Fill the cache with garbage (some form of DDoS)
  - E.g., Tampering with DVFS (e.g., *Plundervolt* corrupts SGX state by playing with power scaling privileged interfaces)

Side and Cover Channels in Processors

- **Channels** are based on shared hardware between processes
- **Means to modulate channel** is highly optimized hardware with frequent slow and fast execution paths
- Mostly Logical
  - Software on software: both sender/victim or receiver/attacker are using **common microarchitectural features**
- The suppression of such channels, **will reduce (noticeably) performance** → mostly academic proposals, limited commercial use (performance/efficiency and security are at odds)
- True solution can be silencing the channel **without affecting performance or increasing the cost**: *still an open problem*

Processors Features and Information Leaks

- The sole **act of executing** an instruction can led to **a side or cover channel**
  - High performance requires predictions → predictions requires track the past → track the past requires to "modify" the internal state
  - Instantaneous effects (mainly optimizations) can be "externally" observed



Variable Instruction Execution Timing

- Some instructions (e.g., logic or simple arithmetic) are fast (1cycle), whereas others (e.g., floating point or complex arithmetic) are slow (>5cycles)

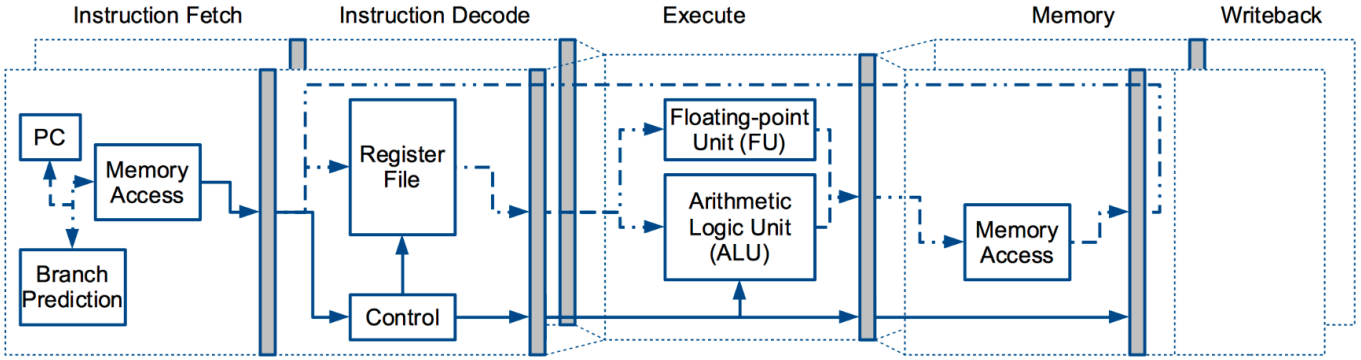
- Just by observing how fast/slow each instruction is executed, information (albeit weak at first glance) about the program may leak out.

- **Memory access instructions** might be much serious (with abysmal differences between instructions , e.g., L1 hit vs LLC miss)

- **Solution: Constant time software**
  - **Painfully slow**

Functional Unit Contention

- Functional units within processor are re-used or shared to save on area and cost of the processor resulting in varying program execution: Simultaneous Multithreading (SMT)
  - Contention of multiple threads are reflected in the timing (i.e. some form of information leaks)



- This is also observed in memory (later)
- **Solution: Temporal spatial multiplexing**
  - **Painfully costly**

Stateful Functional Units

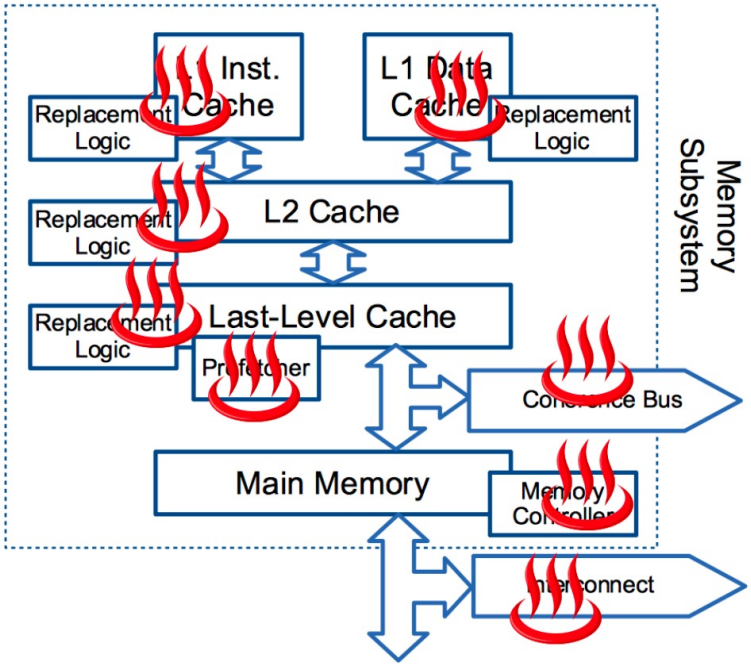
- Many functional units inside the processor keep some history of past execution and use the information for prediction purposes.
  - Execution time or other output may depend on the state of the functional unit
  - If functional unit is shared, other programs can guess the state (and thus the history)
  - E.g., caches, branch predictor, load-store queues, prefetcher, etc.

- **Solution: Flushing state will erase the history.**
  - **Painfully slow**

Memory Hierarchy: Components

- Memory hierarchy aims to improve system performance by **hiding memory access latency** (creating fast and slow executions paths); and parts of the hierarchy area a shared resource

- Cache replacement logic
  - Inclusive caches
  - Non-inclusive caches
  - Exclusive caches
- Prefetcher logic
  - Also, speculative instruction fetching from processor core
- Memory controller
- Interconnect
- Coherence controllers



Memory Hierarchy (cont…)

- Caches
  - Facts
    - Solution for Memory/Bandwidth Wall problems
    - Multilevel (3 or 4)
    - Usually shared in 3 and upwards (last level called LLC)
    - Highly variable access time (1-2cycles L1 to 100-1000cycles DRAM)
  - Leakage
    - Unavoidable interference between isolated contexts (process, VM, etc…)
    - Easy to detect when some data is in the cache or not
    - Use insertion and replacement algorithm knowledge to "extract" isolated piece of information from a context
  - **Solutions**
    - **No cache**
    - **Randomize insertion/replacement**

Memory Hierarchy (cont…)

- Prefetcher
  - Facts
    - Solution for Memory/Bandwidth Wall problems
    - Bring in advance data that the processor **will need soon**
    - Detects predictable patterns by using prediction
    - Multiple prefetchers (usually LLC, L2)
    - Fairly complex
    - State is shared across execution contexts
  - Leakage
    - Cross-Influence isolated context by wrongfully inducing the prefetcher to do something that he didn't want to do
    - Might be possible to detect access pattern of others
  - **Solutions**
    - **Disable prefetcher**

Memory Hierarchy (cont.…)

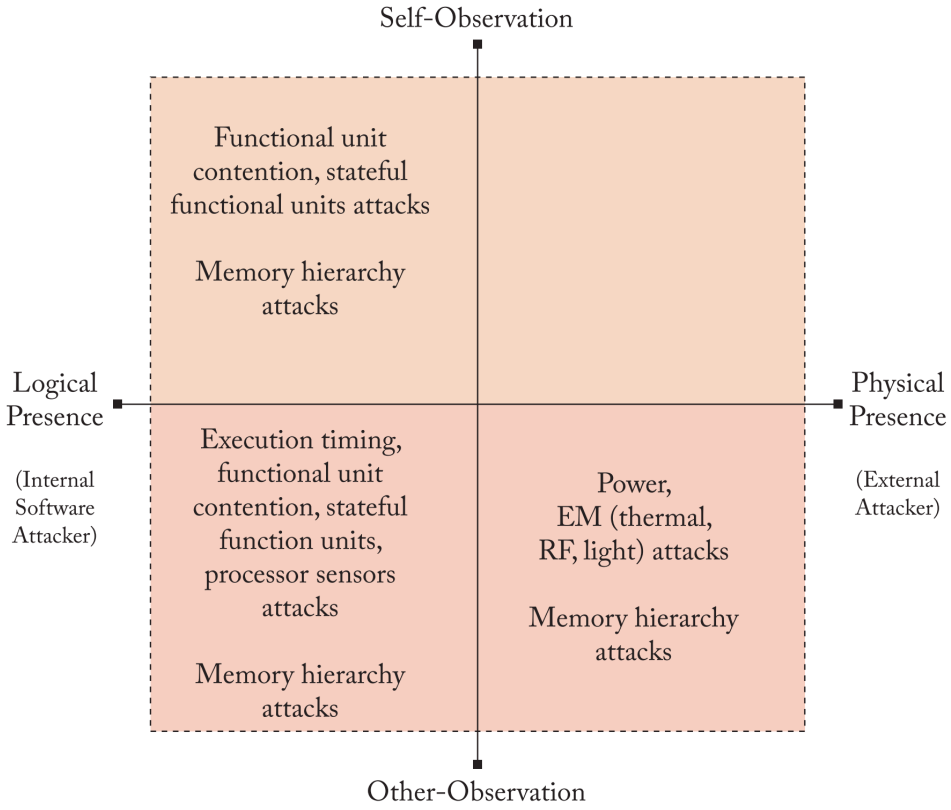
- DRAM
  - Facts
    - Uses a complex arbitration process at the memory controller
    - Open page operations goes first (use as much as possible the row buffer before close it)
    - Controller might reorganize the memory operations accordingly
  - Leakage
    - Easy to "fabricate" contention in the isolated contexts (that are using the same memory controller)
  - **Solutions**
    - **Improve memory controller fairness (use policies more advanced than open page)**

Memory Hierarchy (cont…)

- Interconnects/Coherence protocol
  - Facts
    - Todays interconnects are based on Point-to-Point networks
    - Cache coherent non-Uniform memory architecture
    - Coherence protocols might introduce a wide variability for latency of memory operations
  - Leakage
    - Use disparate access times from local to remote DRAM
    - Atomic operations might stall interconnect the one isolated context can infer certain information about other
  - **Solutions**
    - **???**

Side and Cover Channel Classifications

- Left-hand are our concern
- Top only requires to observe the behavior
- Bottom involves some action of the attacker on the shared resources



Side and Cover Channel Classifications

- Two broad categories
  - **Classical**: does not require speculation
  - **Speculative**: are based in speculative execution

- Main difference is victim is not in control about what instructions are executed (i.e., some are executed speculatively)

- Root cause **is the same**:
  - State of functional units are modified by the victim and observed by the attacker (via **timing changes**)

- Defending speculative attacks defends classical attacks but **not other way around** (e.g., cache partitioning don't prevent pattern observation)

Timing Side-channel Attacks (Classical)

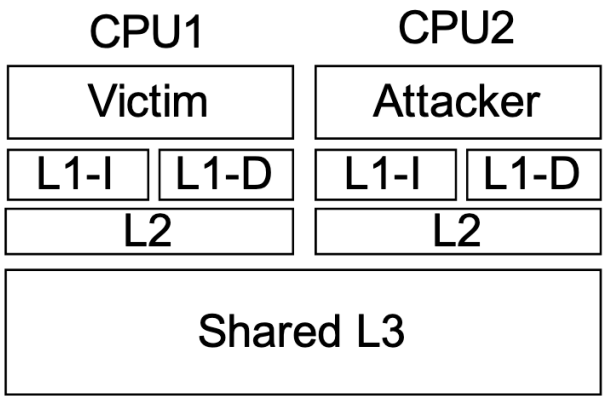
https://caslab.csl.yale.edu/tutorials/asplos2021/

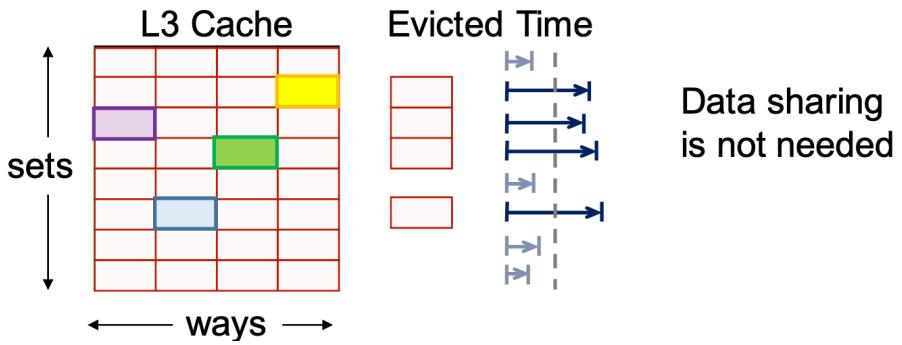
Cache Side-channel

- Victim holds secure data / Attacker wants to learn information about it
- Attacker can:
  - **Measure time** in his operations (or the operations made by the victim)
  - Can control or **trigger** the victim to do something with sensitive data
- Use of instructions on memory which might have timing differences
  - Memory accesses, data invalidations, etc..
- Many approaches
  - Prime + probe
  - Flush + reload
  - …

Example: Prime-Probe Attack

1. Attacker primes all cache sets (priming phase)
2. Victim access critical data
3. Attacker measure access time to his own data (probe phase)



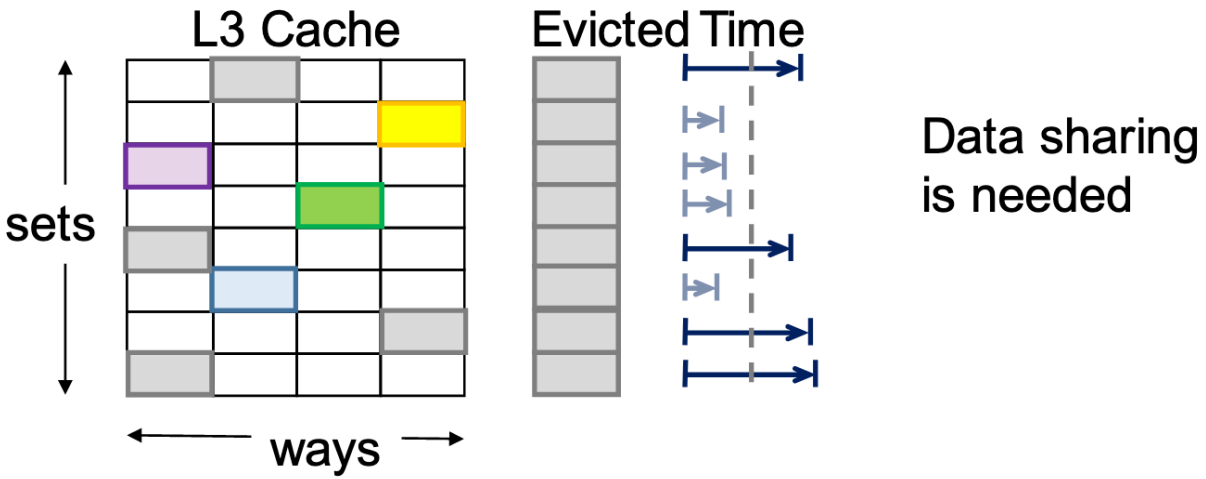


Osvik, D. A., Shamir, A., & Tromer, E, “Cache attacks and countermeasures: the case of AES”. 2006.

A context can discover the access pattern of others to infer the S-Box in AES (and thus, the data being encrypted).

Example: Flush-reload Attack

1. Attacker flushes each line in the cache
2. Victim accesses critical data
3. Attacker reloads critical data by running specific process (measure time)

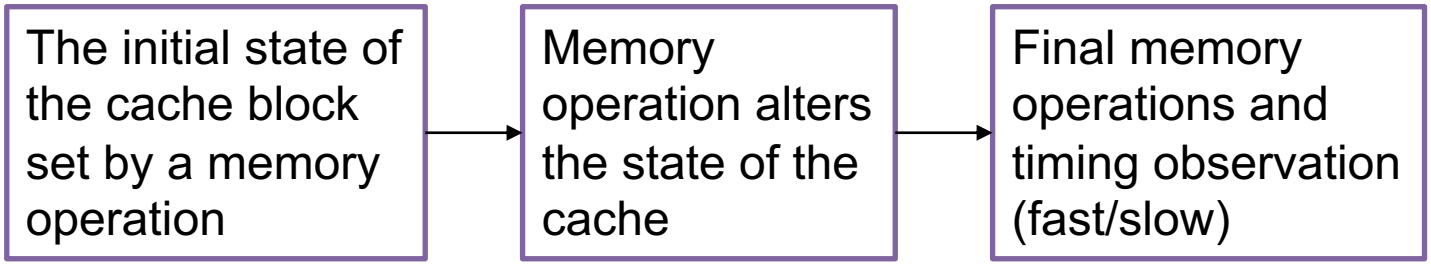


Yarom, Y., & Falkner, K. “FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack”, 2014.

Tied to a particular GnuPG implementation (applied across VM)

A three-Step Model for Cache Timing Attack Modeling

- Observation:
  - All the existing cache timing attacks equivalent to three memory operations → **three-step model**
  - Cache replacement policy the same to each cache block → focus on one cache block
- The Three-Step Single-Cache-Block-Access Mode



  - There are 88 possible cache timing attack types

Attack Strategy	Vulnerability Type			Macro Type	Attack
	Step 1	*Step 2*	*Step 3*		
Cache Internal Collision	A^{inv}	V_u	V_a (fast)	IH	(2)
	V^{inv}	V_u	V_a (fast)	IH	(2)
	A_d	V_u	V_a (fast)	IH	(2)
	V_d	V_u	V_a (fast)	IH	(2)
	$A_{a^{alias}}$	V_u	V_a (fast)	IH	(2)
	$V_{a^{alias}}$	V_u	V_a (fast)	IH	(2)
	A_a^{inv}	V_u	V_a (fast)	IH	(2)
	V_a^{inv}	V_u	V_a (fast)	IH	(2)
Flush + Reload	A_a^{inv}	V_u	A_a (fast)	EH	(5)
	V_a^{inv}	V_u	A_a (fast)	EH	(5)
	A^{inv}	V_u	A_a (fast)	EH	(5)
	V^{inv}	V_u	A_a (fast)	EH	(5)
	A_d	V_u	A_a (fast)	EH	(5)
	V_d	V_u	A_a (fast)	EH	(5)
	$A_{a^{alias}}$	V_u	A_a (fast)	EH	(5)
	$V_{a^{alias}}$	V_u	A_a (fast)	EH	(5)
Reload + Time	V_u^{inv}	A_a	V_u (fast)	EH	new
	V_u^{inv}	V_a	V_u (fast)	IH	new
Flush + Probe	A_a	V_u^{inv}	A_a (slow)	EM	(6)
	A_a	V_u^{inv}	V_a (slow)	IM	new
	V_a	V_u^{inv}	A_a (slow)	EM	new
	V_a	V_u^{inv}	V_a (slow)	IM	new
Evict + Time	V_u	A_d	V_u (slow)	EM	(1)
	V_u	A_a	V_u (slow)	EM	(1)
Prime + Probe	A_d	V_u	A_d (slow)	EM	(4)
	A_a	V_u	A_a (slow)	EM	(4)
Bernstein's Attack	V_u	V_a	V_u (slow)	IM	(3)
	V_u	V_d	V_u (slow)	IM	(3)
	V_d	V_u	V_d (slow)	IM	(3)
	V_a	V_u	V_a (slow)	IM	(3)
Evict + Probe	V_d	V_u	A_d (slow)	EM	new
	V_a	V_u	A_a (slow)	EM	new
Prime + Time	A_d	V_u	V_d (slow)	IM	new
	A_a	V_u	V_a (slow)	IM	new
Flush + Time	V_u	A_a^{inv}	V_u (slow)	EM	new
	V_u	V_a^{inv}	V_u (slow)	IM	new

(1) Evict + Time attack [31]

Attack Strategy	Vulnerability Type			Macro Type	Attack
	Step 1	*Step 2*	*Step 3*		
Cache Internal Collision Invalidation	A^{inv}	V_u	V_a^{inv} (slow)	IH	new
	V^{inv}	V_u	V_a^{inv} (slow)	IH	new
	A_d	V_u	V_a^{inv} (slow)	IH	new
	V_d	V_u	V_a^{inv} (slow)	IH	new
	$A_{a^{alias}}$	V_u	V_a^{inv} (slow)	IH	new
	$V_{a^{alias}}$	V_u	V_a^{inv} (slow)	IH	new
Flush + Flush	A_a^{inv}	V_u	V_a^{inv} (slow)	IH	(1)
	V_a^{inv}	V_u	V_a^{inv} (slow)	IH	(1)
	A_a^{inv}	V_u	A_a^{inv} (slow)	EH	(1)
	V_a^{inv}	V_u	A_a^{inv} (slow)	EH	(1)
Flush + Reload Invalidation	A^{inv}	V_u	A_a^{inv} (slow)	EH	new
	V^{inv}	V_u	A_a^{inv} (slow)	EH	new
	A_d	V_u	A_a^{inv} (slow)	EH	new
	V_d	V_u	A_a^{inv} (slow)	EH	new
	$A_{a^{alias}}$	V_u	A_a^{inv} (slow)	EH	new
	$V_{a^{alias}}$	V_u	A_a^{inv} (slow)	EH	new
Reload + Time Invalidation	V_u^{inv}	A_a	V_u^{inv} (slow)	EH	new
	V_u^{inv}	V_a	V_u^{inv} (slow)	IH	new
Flush + Probe Invalidation	A_a	V_u^{inv}	A_a^{inv} (fast)	EM	new
	A_a	V_u^{inv}	V_a^{inv} (fast)	IM	new
	V_a	V_u^{inv}	A_a^{inv} (fast)	EM	new
	V_a	V_u^{inv}	V_a^{inv} (fast)	IM	new
Evict + Time Invalidation	V_u	A_d	V_u^{inv} (fast)	EM	new
	V_u	A_a	V_u^{inv} (fast)	EM	new
Prime + Probe Invalidation	A_d	V_u	A_d^{inv} (fast)	EM	new
	A_a	V_u	A_a^{inv} (fast)	EM	new
Bernstein's Invalidation Attack	V_u	V_a	V_u^{inv} (fast)	IM	new
	V_u	V_d	V_u^{inv} (fast)	IM	new
	V_d	V_u	V_d^{inv} (fast)	IM	new
	V_a	V_u	V_a^{inv} (fast)	IM	new
Evict + Probe Invalidation	V_d	V_u	A_d^{inv} (fast)	EM	new
	V_a	V_u	A_a^{inv} (fast)	EM	new
Prime + Time Invalidation	A_d	V_u	V_d^{inv} (fast)	IM	new
	A_a	V_u	V_a^{inv} (fast)	IM	new
Flush + Time Invalidation	V_u	A_a^{inv}	V_u^{inv} (fast)	EM	new
	V_u	V_a^{inv}	V_u^{inv} (fast)	IM	new

(1) Flush + Flush attack [14]

Deng, S., Xiong, W., Szefer, J., "Analysis of Secure Caches and Timing-Based Side-Channel Attacks", 2019

Timing attacks on other components on Mem. Hierarchy

- **Cache replacement logic**
  - LRU states can be abused for a timing channel, especially cache hits modify the LRU state, no misses are required
- **TLBs**
  - Have the same vulnerabilities of regular caches
- **Directories**
  - Directory used for tracking cache coherence state is a type of a cache as well
- **Prefetches**
  - Prefetchers leverage memory access history to eagerly fetch data and can create timing channels
- **Load**, **Store**, and **Other Buffers**
  - different buffers can forward data that is in-flight and not in caches, this is in addition to Micro-architectural Data Sampling attacks
- **Coherence Interconnect and Coherence State**
  - different coherence state of a cache line may affect timing, such as flushing or upgrading state
- **Memory Controller and Interconnect**–are shared resources vulnerable to contention channels

The Need for Hardware Secure Caches

- Software defenses are possible (e.g., "constant time" software)
  - But require **software developers to consider timing attacks**, and to ponder all possible attacks, (if new attack is demonstrated previously written secure software may no longer be secure).
- Root cause of timing attacks are caches themselves
  - Caches by design have **timing differences** (hit vs. miss, slow vs. fast flush)
  - Correctly functioning caches can leak critical secrets like encryption keys when the cache is shared between victim and attacker
- Need to consider about different levels for the cache hierarchy, different kinds of caches, and cache-like structures
  - Secure processor architectures also are affected by timing attacks on caches
  - E.g., Intel SGX is vulnerable to cache attacks and some Spectre variants
  - E.g., cache timing side-channel attacks are possible in ARM TrustZone
  - Secure processors must have secure caches

Secure Cache Techniques

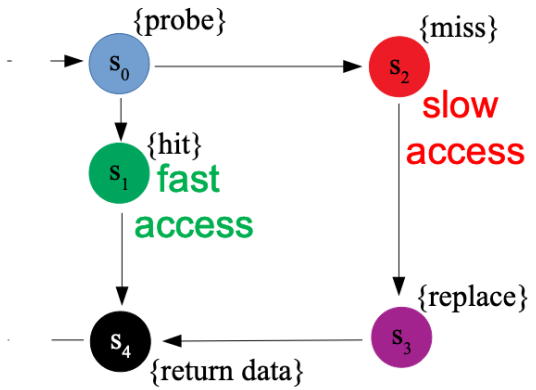
- Numerous **academic proposals** have introduced various secure cache architectures that aim to defend against diverse cache-based side channels.

- To-date there are ~20 secure cache proposals
- They share many similar key techniques

- Techniques
  - **Partitioning** – isolates the attacker and the victim
  - **Randomization** – randomizes address mapping or data brought into the cache
  - **Differentiating Sensitive Data** – allows fine-grain control of secure data

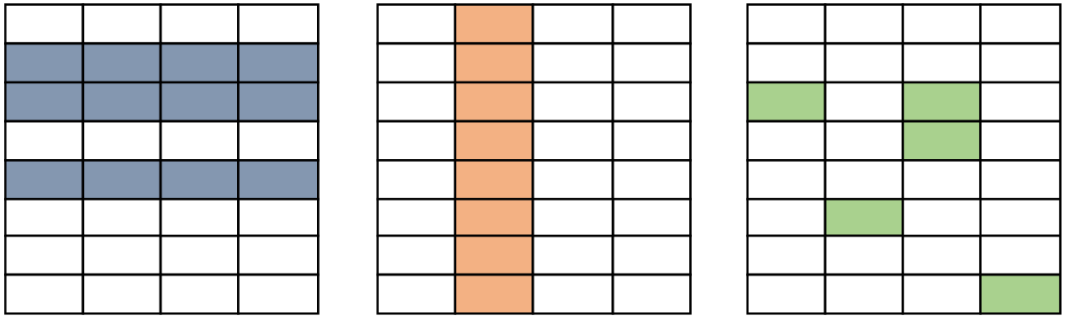
Types of inferences

- Where the interference happens
  - **External-interference** vulnerabilities
    - Interference (e.g., eviction of one party's data from the cache or observing hit of one party's data) happens between the attacker and the victim
  - **Internal-interference** vulnerabilities
    - Interference happens within the victim's process itself (e.g., access to secure enclave)
- Memory reuse conditions
  - **Hit-based** vulnerabilities
    - Cache hit (fast)
    - Invalidation of the data when the data is in the cache (slow)
  - **Miss-based** vulnerabilities
    - Cache miss (slow)
    - Invalidation of the data when the data is not in the cache (fast)



Partitioning

- **Goal**: limit the victim and the attacker to be able to only access a limited set of cache blocks
- **Partition among security levels**: High (higher security level) and Low (lower security level) or even more partitions are possible
- **Type**: Static partitioning vs. dynamic partitioning
- **Partitioning based on**:
  - Whether the memory access is victim's or attacker's
  - Where the access is to (e.g., to a sensitive or not sensitive memory region)
  - Whether the access is due to speculation or out-of-order load or store, or it is a normal operation
- **Partitioning granularity:**
  - Cache sets
  - Cache ways
  - Cache lines or block



Partitioning (cont)

- Partitioning usually targets external interference, but is **weak at defending internal interference**:
  - Interference between the attack and the victim **partition** becomes **impossible**, attacks based on these types of external interference will fail
  - Interference **within** victim itself is still **possible**
- **Wasteful** in terms of cache space and degrades system performance
  - Dynamic partitioning can help limit the negative performance and space impacts
    - At a cost of revealing some side-channel information when adjusting the partitioning size for each part
    - Does not help with internal interference
- **Partitioning** in hardware or software
  - Hardware partitioning
    - E.g., Intel Resource Director (Cache Allocation Technology) [RDT/CAT]
  - Software partitioning
    - E.g., page-coloring (restricts according page-color the ways of the that we can use)

Randomization

- Randomization aims to **inherently de-correlate the relationship** among the **address and the observed timing**

- Randomization approaches:
  - Randomize the **address to cache set mapping**
  - Random insertion
  - Random eviction
  - Random delay
- Goal: reduce the mutual information from the observed timing to 0
- Some limitations:
  - Requires a fast and secure random number generator, ability to predict the random behavior will defeat these technique
  - May need OS support or interface to specify range of memory locations being randomized; ...

Differentiating Sensitive Data

- Allows the **victim or management** software to explicitly **mark** a certain range of the victim's data that they **consider sensitive**.

- Can use new cache-specific instructions to protect the data and limit internal interference between victim's own data
  - E.g., it is possible to **disable victim's own flushing of victim's labeled data**, and therefore prevent vulnerabilities that leverage flushing
  - Has advantage in preventing internal interference

- Allows the designer to have stronger control over security critical data
  - **How to identify sensitive data** and whether this identification process is reliable are open
    research questions

- Independent of whether a cache uses partitioning or randomization

Proposals

- All academic
  - Not free (1%-10% performance, up to 5% power, up to 6% area)

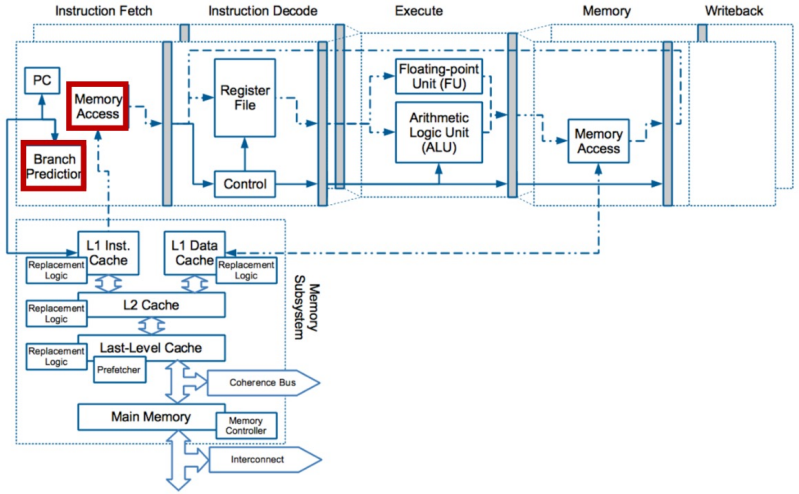
	SP	SecVerilog	SecDCP	NoMo	SHARP	Sanctum	CATalyst	RIC	PL	RP	Newcache	RF	CEASER	SCATTER	Non-det. cache
external miss-based attacks	✓	✓	~	✓	✓	✓	✓	✓	✓	✓	✓	X	✓	✓	O
internal miss-based attacks	X	X	X	X	X	X	✓	✓	X	X	✓	X	✓	✓	O
external hit-based attacks	X	✓	✓	X	X	✓	✓	X	X	✓	✓	✓	X	~	O
internal hit-based attacks	X	X	X	X	X	X	✓	X	X	X	X	✓	X	X	O

- Other components of mem. hierarchy??

Transient Execution Attacks

Prediction and Speculation in Modern CPUs

- Prediction is one of the six key features of modern processors
  - Instructions in a processor pipeline **have dependencies** on **prior instructions** which are in the pipeline and may **not have finished yet**
  - To keep pipeline as full as possible, prediction is needed if results of prior instruction are **not known yet**
  - Prediction can be done for:
    - Control Flow
    - Data dependencies
    - Actual data (also called value prediction)
  - **Not just branch prediction**: prefetcher, memory disambiguation, ...



Transient Execution Attacks

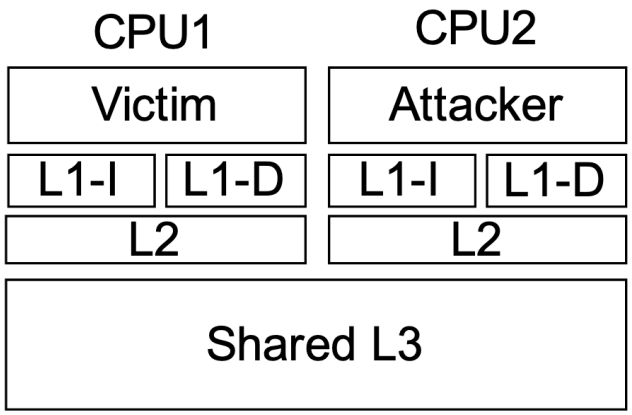
- **Spectre**, **Meltdown**, etc. leverage the instructions that are executed transiently:
  1. These transient instructions execute for a short time (e.g., due to mis-speculation),
  2. until processor computes that they are not needed, and
  3. the pipeline flush occurs, and it should discard any side effects of these instructions so
  4. architectural state remain as if they never executed, but ...

These attacks exploit transient execution to encode secrets through microarchitectural side effects that can later be recovered by an attacker through a (most often timing based) observation at the architectural level

Transient Execution Attacks = Transient Execution + Cover or Side Attacks

Transient Execution Attack using Flush + reload Timing

- Attacker **flushes** each line in the cache of a data structure (e.g., evict an "array" from cache)
- Victim (**unwillingly**, e.g., via speculation) access to sensitive data
- Attacker uses a part of the sensitive data to access memory (during the **speculation window**) to access the **previously evicted "array"**
- The involved operation in memory is analyzed later by reloading the evicted array (measuring time, the **address of the hit** is part of the sensitive data exfiltrated)



Conditional branch (Variant 1) attack

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

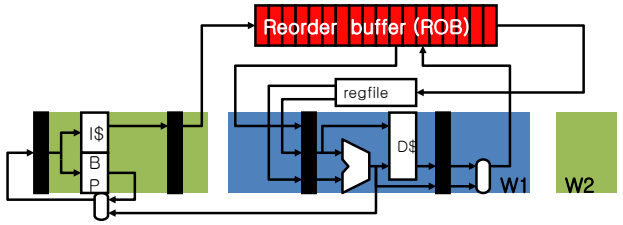
Assume code in kernel API, where unsigned int `x` comes from untrusted caller

Execution without speculation is safe

- CPU will not evaluate `array2[array1[x]*4096]` unless `x < array1_size`

What about with speculative execution? (i.e., guarantee that array1_size is not in cache and fool the BP to execute y=… speculatively with x out of bounds)

Example: Spectre Bounds Check Bypass Attack



Victim code:

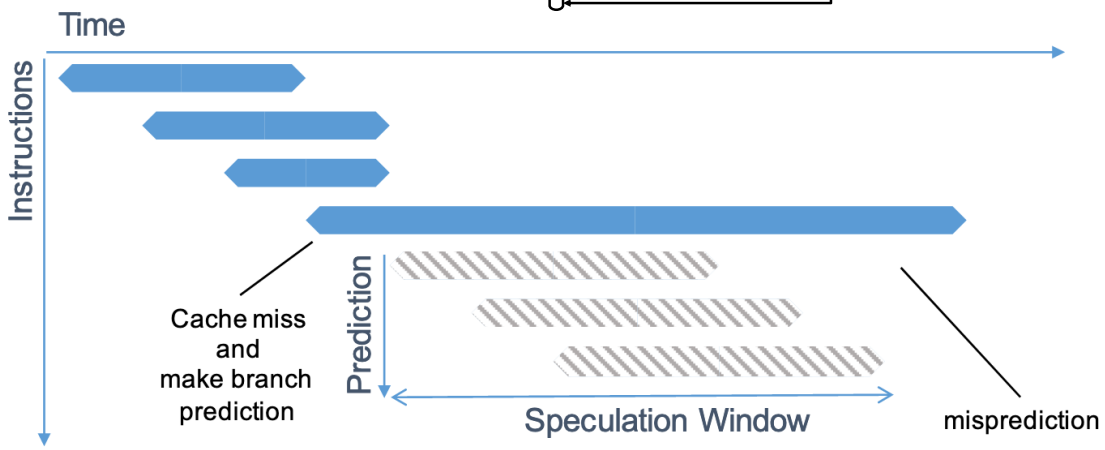
```
struct array *arr1 = ...;
struct array *arr2 = ...;
unsigned long offset = ...;
if (offset < arr1->len) {
  unsigned char value = arr1->data[offset];
  unsigned long index = value;
  unsigned char value2 = arr2->data[index * size];
...
```

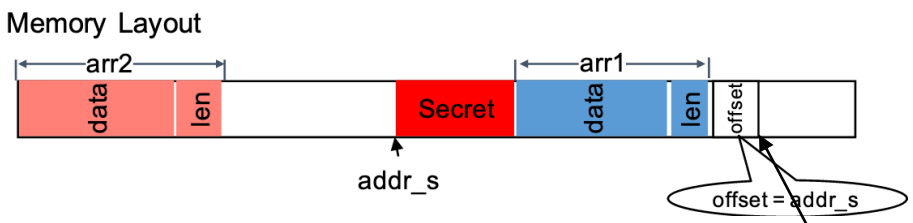
Probe array (side channel)

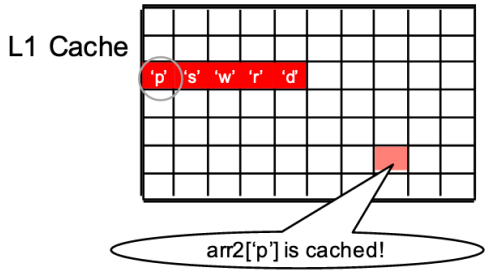
Controlled by the attacker

arr1->len is not in cache

change the cache state







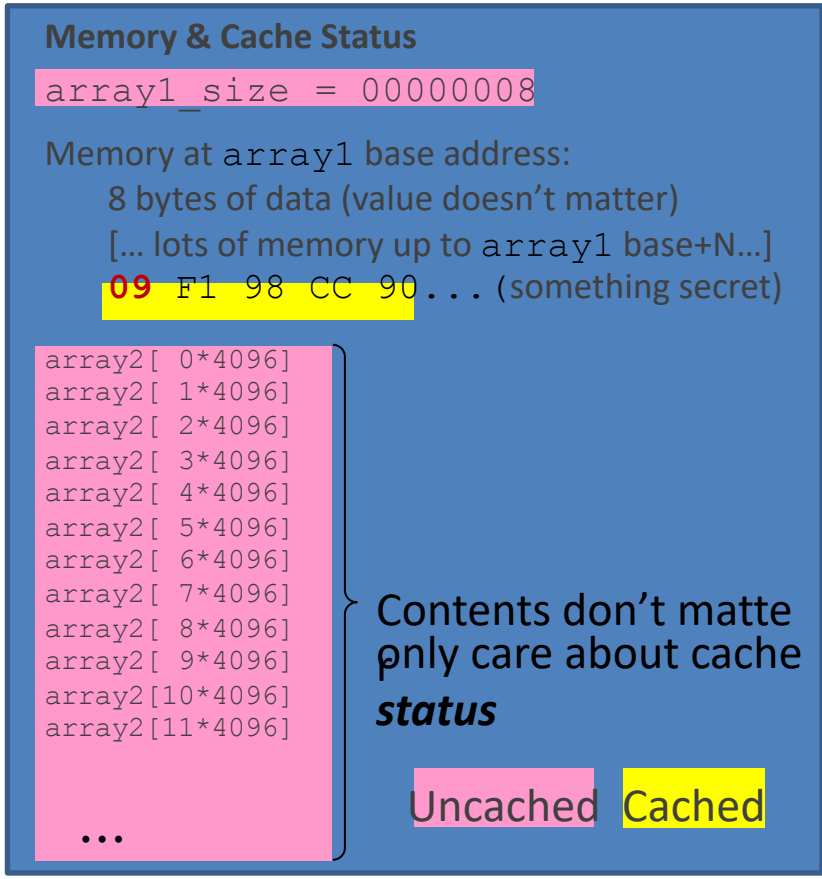
Secret

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



Conditional branch (Variant 1) attack

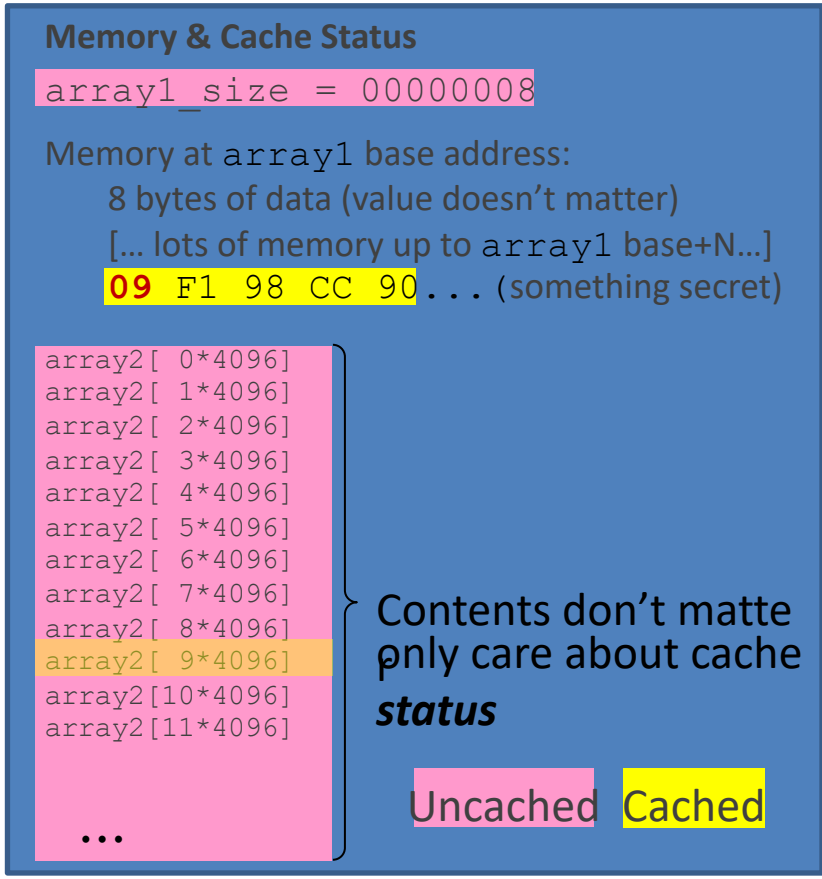
```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Attacker calls victim with `x`=N (where N > 8)

- Speculative exec while waiting for `array1_size`
  - Predict that if() is true
  - Read address (`array1` base + `x`) w/ out-of-bounds `x`
  - Read returns secret byte = **09** (fast – in cache)
  - Request memory at (`array2` base + **09***4096)
  - Brings `array2[09*4096]` into the cache
  - Realize if() is false: discard speculative work
- Finish operation & return to caller

Attacker measures read time for `array2[i*4096]`

- Read for `i`=**09** is fast (cached), revealing secret byte
- Repeat with many `x` (eg ~10KB/s)



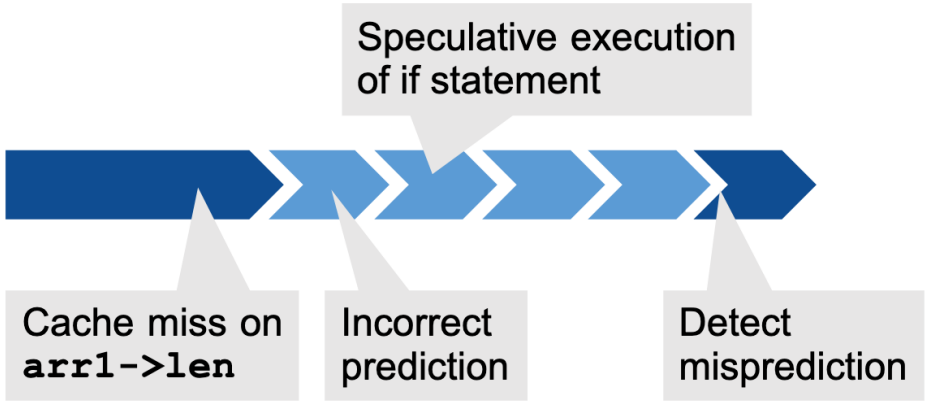
Demo

Transient Execution – due to Prediction

- Because of prediction, some instructions are executed transiently:
  1. Use prediction to begin execution of instruction with unresolved dependency
  2. Instruction executes for some amount of time, changing architectural and micro-architectural state
  3. Processor detects misprediction, squashes the instructions
  4. Processor cleans up architectural state and should cleanup all micro-architectural state

Spectre Variant 1 example:

```
if (offset < arr1->len) {
  unsigned char value = arr1->data[offset];
  unsigned long index = value;
  unsigned char value2 = arr2->data[index];
  ...
}
```

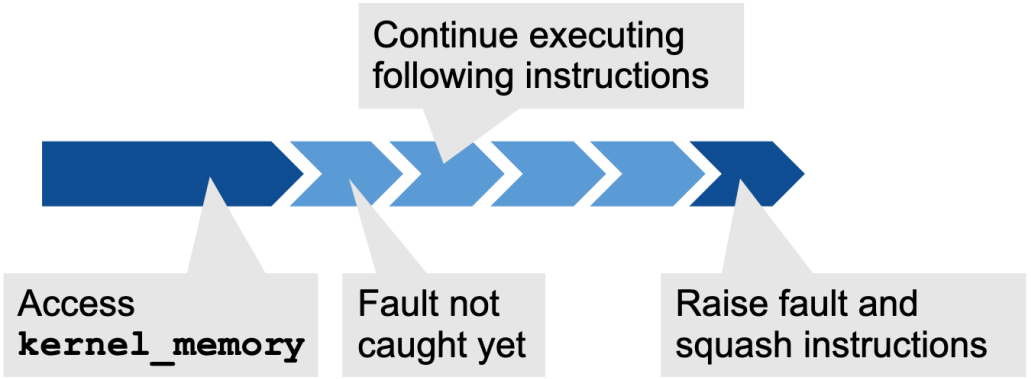


Transient Execution – due to Faults

- Because of faults, some instructions are executed transiently:
  1. Perform operation, such as memory load from forbidden memory address
  2. Fault is not immediately detected, continue execution of following instructions
  3. Processor detects fault, squashes the instructions
  4. Processor cleans up architectural state and should cleanup all micro-architectural state

Meltdown Variant 3 example:

```
...
kernel_memory  = *(uint8_t*)(kernel_address);
final_kernel_memory  = kernel_memory  * 4096;
dummy = probe_array[final_kernel_memory];
...
```



Classes of Attacks

- **Spectre** type
  - Attacks which leverage mis-prediction in the processor, pattern history table (PHT), branch target buffer (BTB), return stack buffer (RSB), store-to-load forwarding (STL), ...

- **Meltdown** type
  - Attacks which leverage exceptions, especially protection checks that are done in parallel to actual data Access

- **Micro-architectural Data Sampling (MDS)** type
  - Attacks which leverage in-flight data that is stored in fill and other buffers (speculatively), which is forwarded without checking permissions, load-fill buffer (LFB/MSHR), or store-to-load forwarding (STL)

Attack Components

- Transient execution attacks have 4 components

```
e.g. if (offset < arr1->len) {
       unsigned char value = arr1->data[offset];
       unsigned long index = value;
       unsigned char value2 = arr2->data[index];
     ...
```

Speculation Primitive `arr1->len` is not in cache → Windowing Gadget

Disclosure Gadget (cache Flush+Reload covert channel) → Disclosure Primitive

1. Speculation Primitive "provides the means for entering transient execution down a non-architectural path"

2. Windowing Gadget "provides a sufficient amount of time for speculative execution to convey information through a side channel"

3. Disclosure Gadget "provides the means for communicating information through a side channel during speculative execution"

4. Disclosure Primitive "provides the means for reading the information that was communicated by the disclosure gadget"

Speculation Primitive

- Speculation Primitives – Sample Code
  - Spectre-type: transient execution after a prediction
    - Branch prediction
      - Pattern History Table (PHT) -- Bounds Check bypass (V1)
      - Branch Target Buffer (BTB) -- Branch Target injection (V2)
      - Return Stack Buffer/Return Address Stack (RSB/RAS) -- SpectreRSB (V5)
    - Memory disambiguation prediction -- Speculative Store Bypass (V4)

Spectre Variant 1

```
struct array *arr1 = ...;
struct array *arr2 = ...;
unsigned long offset = ...;
if (offset < arr1->len) {
  unsigned char value = arr1->data[offset];
  unsigned long index = value;
  unsigned char value2 = arr2->data[index];
...
```

Spectre Variant 2

```
      (Attacker trains the BTB
      to jump to GADGET)

      jmp LEGITIMATE_TRGT
      ...
GADGET: mov r8, QWORD PTR[r15]
      lea rdi, [r8]
      ...
```

Spectre Variant 5

```
      (Attacker pollutes the RSB)
main: Call F1
      ...
F1:   ...
      ret
      ...
GADGET: mov r8, QWORD PTR[r15]
      lea rdi, [r8]
      ...
```

Spectre Variant 4

```
char sec[16] = ...;
char pub[16] = ...;
char arr2[0x200000] = ..
char * ptr = sec;
char **slow_ptr = *ptr;
clflush(slow_ptr)
*slow_ptr = pub;
value2 = arr2[(*ptr)<<12
```

Store "slowly"

Load the value at the same memory location "quickly". "ptr" will get a stale value.

Windowing Gadget

- Windowing gadget is used to create a "window" of time for transient instructions to execute while

  the processor resolves prediction or exception:

  - Loads from main memory
  - Chains of dependent instructions, e.g., floating point operations, AES

E.g.: Spectre v1 :

Memory access time determines how long it takes to resolve the branch

```
if (offset < arr1->len) {
  unsigned char value = arr1->data[offset];
  unsigned long index = value;
  unsigned char value2 = arr2->data[index];
  ...
}
```

Necessary (but not sufficient) success condition:
windowing gadget's latency > disclosure gadget's trigger latency

Disclosure Gadget

- Whitin transient execution
  - Load the secret to register
  - Encode the secret into channel

The code pointed by the arrows is the disclosure gadget:

Spectre Variant1 (Bounds check)
Cache side channel

```
struct array *arr1 = ...;
struct array *arr2 = ...;
unsigned long offset = ...;
if (offset < arr1->len) {
  unsigned char value = arr1->data[offset];
  unsigned long index = value;
  unsigned char value2 = arr2->data[index];
...
```

AVX side channel

```
if(x < bitstream_length){
    if(bitstream[x])
        _mm256_instruction();
}
```

Disclosure Primitives Types

- **Short-lived** or contention-based channel:
  - Share resource on the fly (e.g., bus, port, cache bank)
  - State change within speculative window (e.g., speculative buffer)

- **Long-lived** channel:
  - Change the state of micro-architecture
  - The change remains even after the speculative window
  - Micro-architecture components to use:
    - D-Cache (L1, L2, L3) (Tag, replacement policy state, Coherence State, Directory), I-cache;TLB, DRAM Rowbuffer, ...
    - Encoding method:
      - Contention (e.g., cache Prime+Probe)
      - Reuse (e.g., cache Flush+Reload)

Disclosure Primitives

- Mem. access timing
- Port contention
- Coherence states
- AVX Unit states
- …

Disclosure Primitives - AVX Unit States

- To save power, the CPU can power down the upper half of the AVX2 unit which is used to perform operations on 256-bit registers
- The upper half of the unit is powered up as soon as an instruction is executed which uses 256-bit values
- If the unit is not used for more than 1 ms, it is powered down again

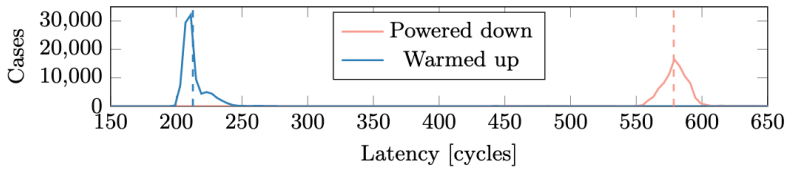


Fig. 5: If the AVX2 unit is inactive (powered down), executing an AVX2 instruction takes on average 366 cycles longer than on an active AVX2 unit (Intel i5-6200U). Average values shown as dashed lines.

Gadget:

```
if(x < bitstream_length){
    if(bitstream[x])
        _mm256_instruction();
}
```

1-bit of information extracted

M. Schwarz, et al., "NetSpectre: Read Arbitrary Memory over Network", 2018

Disclosure Primitives – Coherence State

- The coherence protocol may invalidate cache lines in sharer cores as a result of a speculative write access request even if the operation is eventually squashed

Gadget:

```
void victim_function(size_t x) {
        if (x < array1_size) {
                array2[array1[x] * 512] = 1;
        }
}
```

If `array2` is initially in shared state or exclusive state on attacker's core, after transient access it transitions to exclusive state on victim's core, changing timing of accesses on attacker's core

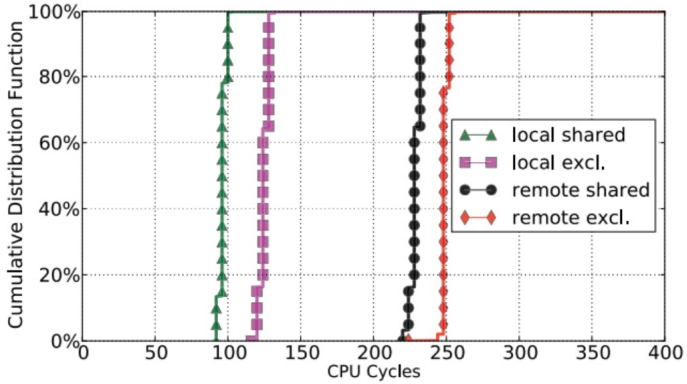


Fig. 2: Load operation latency in various (location, coherence state) combinations.

C. Trippel, et al., "MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols", 2018
F. Yao, et al., "Are Coherence Protocol States Vulnerable to Information Leakage?", 2018

Mitigations / Solutions

Serializing Instructions and Software Mitigations

- Serializing instructions can prevent speculative execution
  - LFENCE (x86) will stop **younger load instructions** from executing, even speculatively, before **older load instructions** have retired
  - CSDB (Consumption of Speculative Data Barrier) (Arm) instruction is a memory barrier that controls Speculative execution and data value prediction
- Insert the instructions in the code to help stop speculation
  - E.g., Microsoft's C/C++ Compiler uses **static analyzer** to select where to insert LFENCE instructions
  - E.g., LLVM includes Intel's patches for load-hardening mitigations that add LFENCE
- Large overheads for inserting serializing instructions, often cited overheads >50%, but depends on application and how and where instructions are inserted

Hardware Academic Solutions

- Invispec [M. Yan, et al., 2018]
  - Store speculative data aside (in shadow caches, TLB, etc…)
  - 22% performance loss SPEC2006
- Safespec [K. N. Khasawneh, et al., 2018]
  - Similar to Invispec
  - 3% due to larger cache sizes SPEC2017
- SpecShield [K. Barber, et al., 2019]
  - Restrict the use of speculative data for dependent instructions
  - 18%-55% performance loss SPEC2006
- Context [M. Schwarz, et al., 2019]
  - Focus on MDS
  - 1%-71% performance loss SPEC2006
- Conditional Speculation [P. Li, et al., 2019]
  - Identify potential unsafe instructions in issue queue and disable speculation
  - 6%-10% performance loss SPEC2006

Industry Solutions

- Hardware (x86) (Targets spectre V2)
  - **Indirect Branch Restricted Speculation** (IBRS): restricts speculation of indirect branches
  - Single Thread Indirect Branch Predictors (STIBP): Prevents indirect branch predictions from being controlled by the sibling hyperthread
  - Indirect Branch Predictor Barrier (IBPB): ensures that earlier code's behavior does not control later indirect branch predictions.

Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. Barberis, E.; Frigo, P.; Muench, M.; Bos, H.; and Giuffrida, C. In *USENIX Security*, August 2022.

Vendor	Model	μArch	BTI vulnerable	BHI vulnerable		BHI attack surface*	
x86-64				**IBRS**	**eIBRS**	**BTI in-place**	**BTI out-of-place**
Intel	Core i9-9900K	Coffee-Lake R	✓	✓	—	✗	✗
	Core i7-10700K	Comet-Lake	✓	✓	✓	✓	✓
	Core i7-11700	Rocket-Lake	✓	✓	✓	✓	✓
	Core i7-11800H	Tiger-Lake	✓	✓	✓	✓	✓
	Xeon Silver 4214	Cascade-Lake	✓	✓	✓	✓	✓
	Xeon Silver 4310	Ice-Lake	✓	✓	✓	✓	✓
AMD	Ryzen 5 5600X	Zen 3	✓	✗	—	✗	✗
	Epyc 7662	Zen 2	✓	✗	—	✗	✗
Arm				**Workaround_1**	**CSV2=1**		
Google	Tensor	Cortex A55	✗	—	—	✗‡	✗‡
	Tensor	Cortex A76	✓	—	✓	✓	✗
	Tensor	Cortex X1	✓	—	✓	✓	✗
Qualcomm	Snapdragon 855	Cortex A76	✓	✓	—	✓	✗

How serious is the problem?

- Available exfiltration data rate. **Above 100 bps** is considered a high bandwidth channel
- Target of many prevention mechanisms is to reduce the bandwidth

Spectre-like Attacks

		Intel	AMD
Hyper-Threaded	Algorithm 1	~500Kbps	~20Kbps
	Algorithm 2	~500Kbps	~20Kbps
Time-Sliced	Algorithm 1	~2bps	~0.2bps
	Algorithm 2	–	–

Solution: get rid of all

- Use disruptive technologies to get rid of all at once