Processor Architecture Security

Part 4: Transient Execution Attacks and Mitigations

Jakub Szefer
Assistant Professor
Dept. of Electrical Engineering
Yale University

(These slides include some prior slides by Jakub Szefer and Wenjie Xiong from HOST 2019 Tutorial)

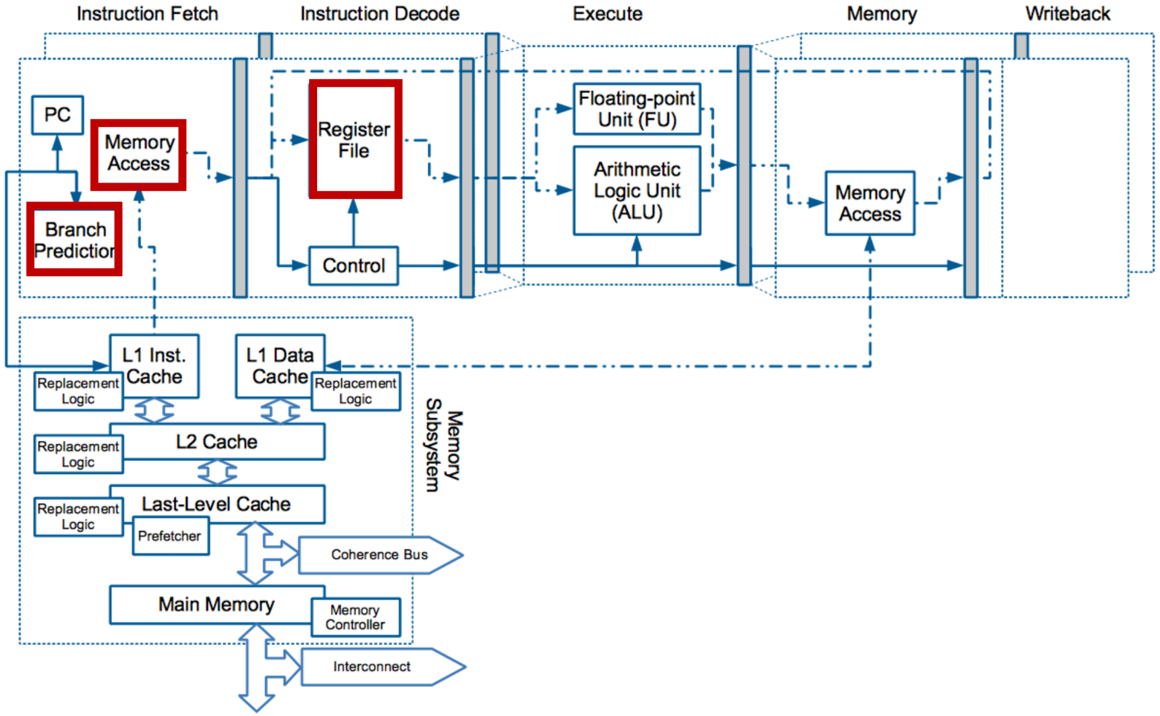
ACACES 2019 – July 14th - 20th, 2019

Slides and information available at: https://caslab.csl.yale.edu/tutorials/acaces2019/

Prediction and Speculation in Modern CPUs

Prediction is one of the six key features of modern processor

- 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 architectural 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 + Covert or Side Channel

Example: Spectre (v1) – Bounds Check Bypass

Example of Spectre variant 1 attack:

Victim code:

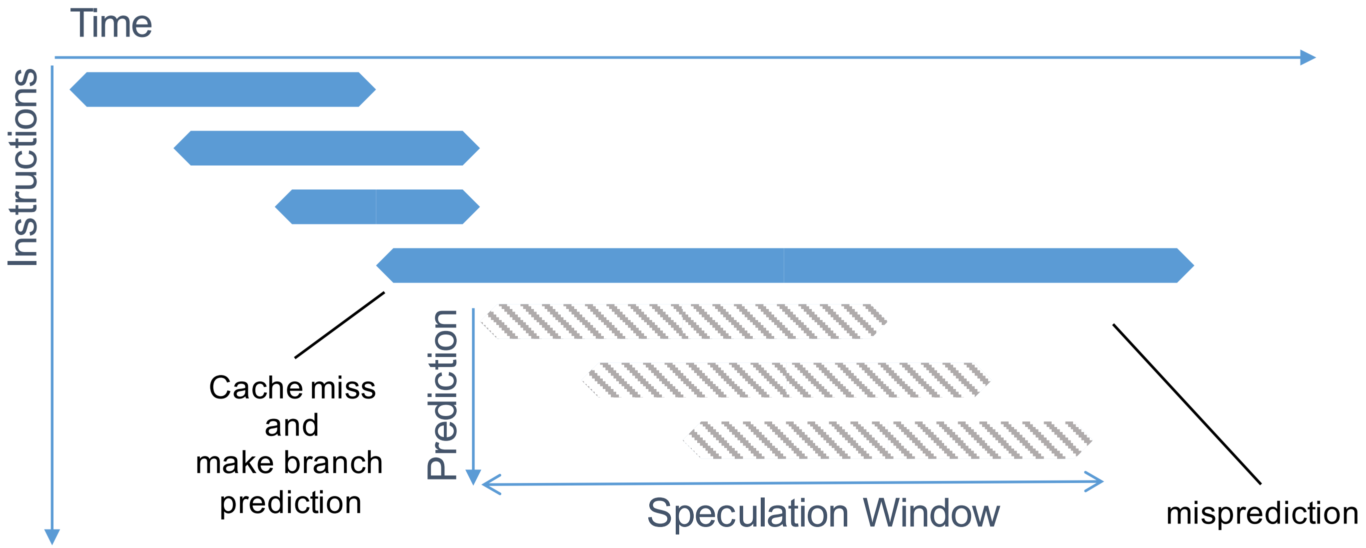
```
struct array *arr1 = ...;
struct array *arr2 = ...;
unsigned long offset = ...;
if (offset < arr1->len) {
  unsigned char value = arr1->data[offset];
  unsigned long index = value;
  unsigned char value2 = arr2->data[index];
...
```

Probe array (side channel)

Controlled by the attacker

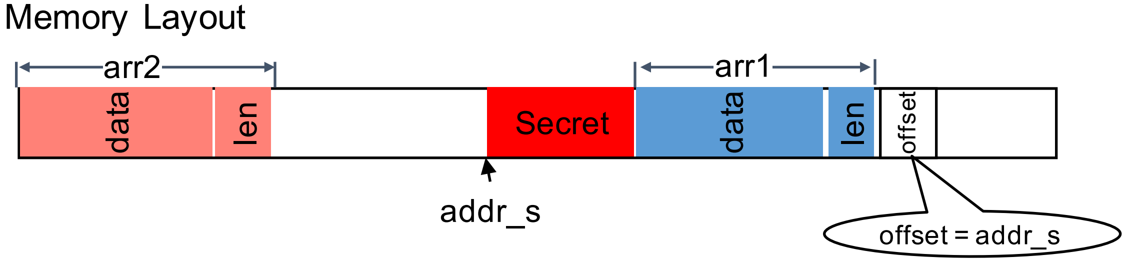
arr1->len is not in cache

change the cache state

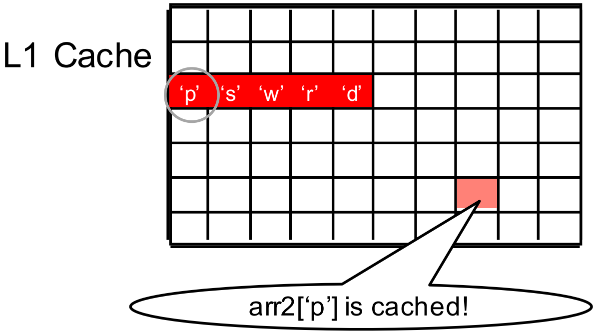


Cache miss and make branch prediction

misprediction



offset = addr_s



arr2['p'] is cached!

The attacker can then check if arr2[X] is in the cache. If so, secret = X

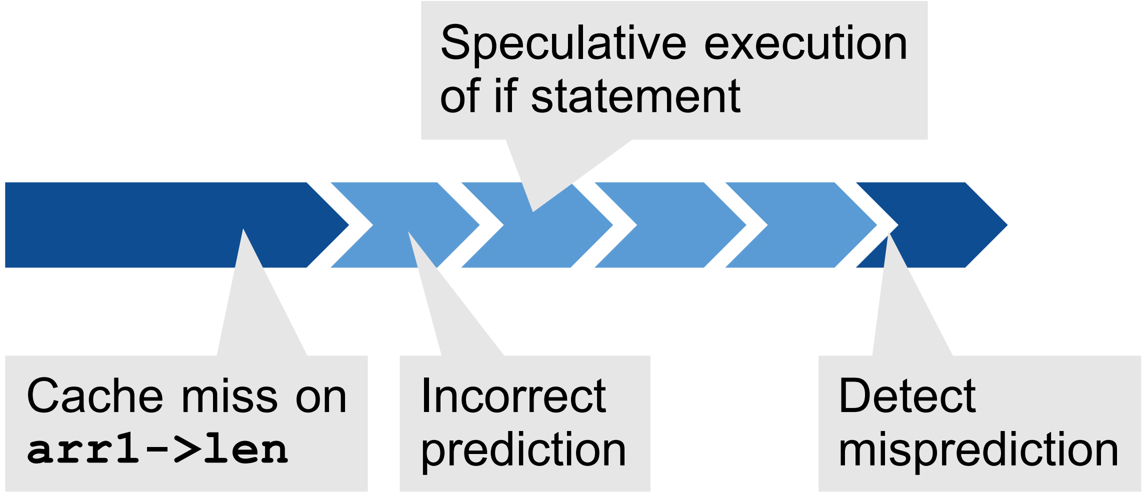
Transient Execution – due to Prediction

***transient** (adjective):* lasting only for a short time; impermanent

- 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 cleansup 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];
  ...
}
```



Speculative execution of if statement

Cache miss on **`arr1->len`**

Incorrect prediction

Detect misprediction

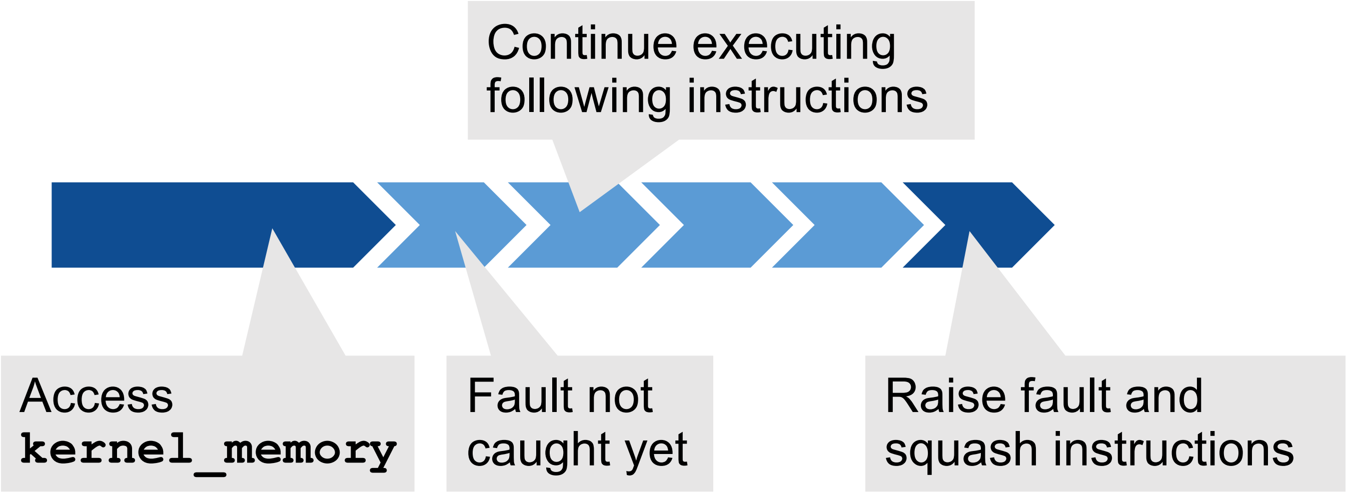
Transient Execution – due to Faults

***transient** (adjective):* lasting only for a short time; impermanent

- 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 cleansup 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];
...
```



Continue executing following instructions

Access **`kernel_memory`**

Fault not caught yet

Raise fault and squash instructions

Speculative or Transient Execution Threats

Speculation causes transient execution to exist in modern processors

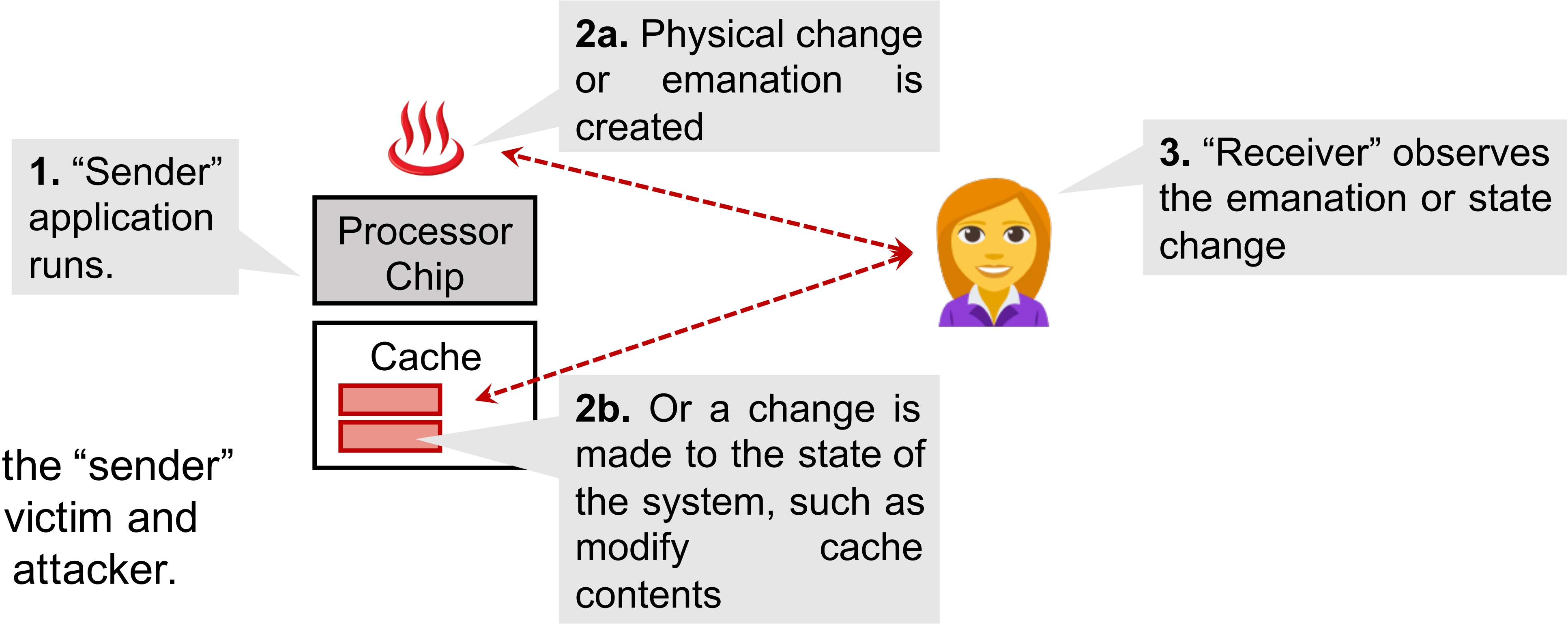
- During transient execution, processor state is modified
- If state (architectural or micro-architectural) is not properly cleaned up when mispredicted instructions are squashed, sensitive data can be leaked out

Attacks based on transient execution have two parts:

1. Leverage speculation to execute some code transiently,
   which modifies processor state based on some secret value
2. Use a side or covert channel to extract the information from the processor state

Side and Covert Channels

A **covert channel** is an intentional communication between a sender and a receiver via a medium not designed to be a communication channel.



1. "Sender" application runs.

2a. Physical change or emanation is created

2b. Or a change is made to the state of the system, such as modify cache contents

3. "Receiver" observes the emanation or state change

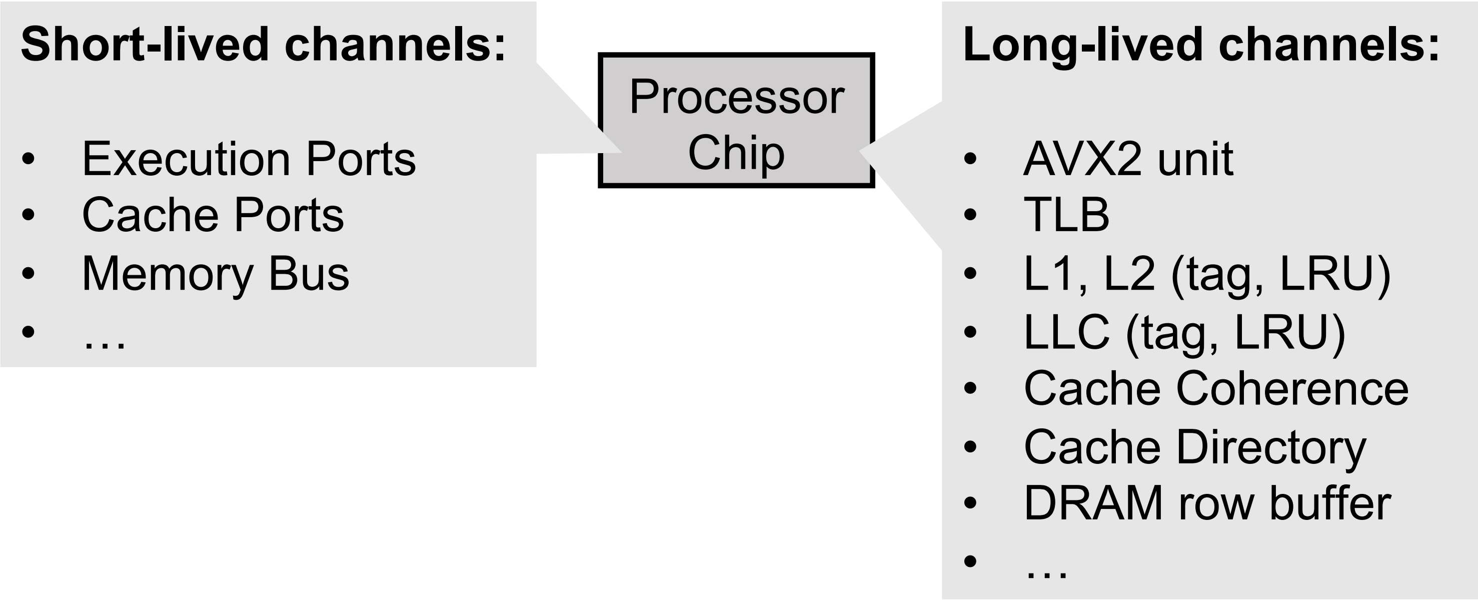
In a **side channel**, the "sender" in an unsuspecting victim and the "receiver" is the attacker.

Emoji Image:
https://www.emojione.com/emoji/2668
https://www.emojione.com/emoji/1f469-1f4bc

Side and Covert Channels

The channels can be **short-lived** or **long-lived** channels:

- Short-lived channels hold the state for a (relatively) short time and eventually data is lost, these are typically **contention-based** channels that require concurrent execution of the victim and the attacker
- Long-lived channels hold the state for a (relatively) long time



Short-lived channels:

- Execution Ports
- Cache Ports
- Memory Bus
- …

Processor Chip

Long-lived channels:

- AVX2 unit
- TLB
- L1, L2 (tag, LRU)
- LLC (tag, LRU)
- Cache Coherence
- Cache Directory
- DRAM row buffer
- …

Covert channels not (yet) explored in transient attacks:

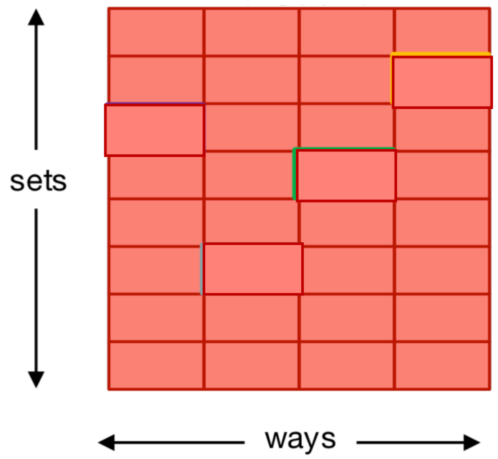
- Random Number Generators
- …

Spectre

Spectre vulnerability can be used to break isolation between different applications.

1. Attacker "trains" branch predictor
2. If statement in example is executed (predicted true)
3. Secret data from array1 is used as index to array2
4. Cache state is modified
5. Branch is resolved, processor cleans up the state, **but** data is left in cache

```
if (x < array1_size)
  y = array2[array1[x] * 256];
```

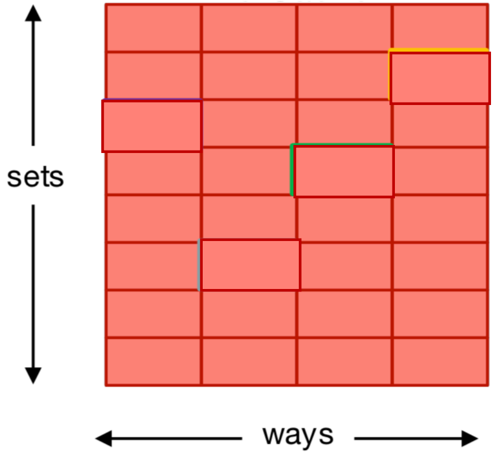


Meltdown

Meltdown vulnerability can be used to break isolation between user applications and the operating system.

1. Attempt to read data from kernel memory (mapped into address space of application)
2. Before an exception is raised, following instructions are speculatively executed
3. Exception is raised, however…
4. Cache state is modified
5. Processor cleans up the state, **but** data is left in cache

```
raise_exception();
access(probe_array[data * 4096]);
```



Spectre, Meltdown, and Their Variants

- Most Spectre & Meltdown attacks and their variants use transient execution
- Many use cache timing channels to extract the secrets

Different Spectre and Meltdown attack variants:

Variant 1:	Bounds Check Bypass (BCB)	Spectre
Variant 1.1:	Bounds Check Bypass Store (BCBS)	Spectre-NG
Variant 1.2:	Read-only protection bypass (RPB)	Spectre
Variant 2:	Branch Target Injection (BTI)	Spectre
Variant 3:	Rogue Data Cache Load (RDCL)	Meltdown
Variant 3a:	Rogue System Register Read (RSRR)	Spectre-NG
Variant 4:	Speculative Store Bypass (SSB)	Spectre-NG
(none)	LazyFP State Restore	Spectre-NG 3
Variant 5:	Return Mispredict	SpectreRSB

- Others: NetSpectre, Foreshadow, SMoTher, SGXSpectre, or SGXPectre
- SpectrePrime and MeltdownPrime (both use Prime+Probe instead of original Flush+Reload cache attack)

NetSpectre is a Spectre Variant 1 done over the network with Evict+Reload, also with AVX covert channel

Foreshadow is Meltdown type attack that targets Intel SGX, **Foreshadow-NG** targets OS, VM, VMM, SMM; all steal data from L1 cache

SMoTher is Spectre variant that uses port-contention in SMT processors to leak information from a victim process

SGXSpectre is Spectre Variant 1 or 2 where code outside SGX Enclave can influence the branch behavior

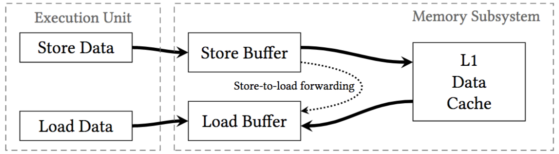
SGXPectre is also Spectre Variant 1 or 2 where code outside SGX Enclave can influence the branch behavior

More Spectre and Meltdown Variants

Micro-architectural Data Sampling (**MDS**) vulnerabilities:

- **Fallout** – *Store Buffers*

Meltdown-type attack which "exploits an optimization that we call Write Transient Forwarding (WTF), which incorrectly passes values from memory writes to subsequent memory reads" through the store and load buffers

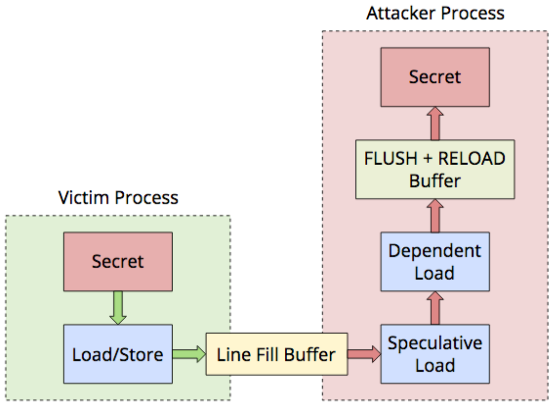


- **RIDL (Rogue In-Flight Data Load)** and **ZombieLoad** – *Line-Fill Buffers* and *Load Ports*

Meltdown-type attacks where "faulting load instructions (i.e., loads that have to be re-issued for either architectural or micro-architectural reasons) may transiently dereference unauthorized destinations previously brought into the buffers by the current or a sibling logical CPU."

RIDL exploits the fact that "if the load and store instructions are ambiguous, the processor can speculatively store-to-load forward the data from the store buffer to the load buffer."

ZombieLoad exploits the fact "that the fill buffer is accessible by all logical CPUs of a physical CPU core and that it does not distinguish between processes or privilege levels."



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**), …

Types of prediction:
- **Data prediction**
- **Address prediction**
- **Value prediction**

- **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, which is forwarded without checking permissions, load-fill buffer (**LFB**), or store-to-load forwarding (**STL**)

Variants:

- Targeting SGX
- Using non-cache based channels

Attack Components

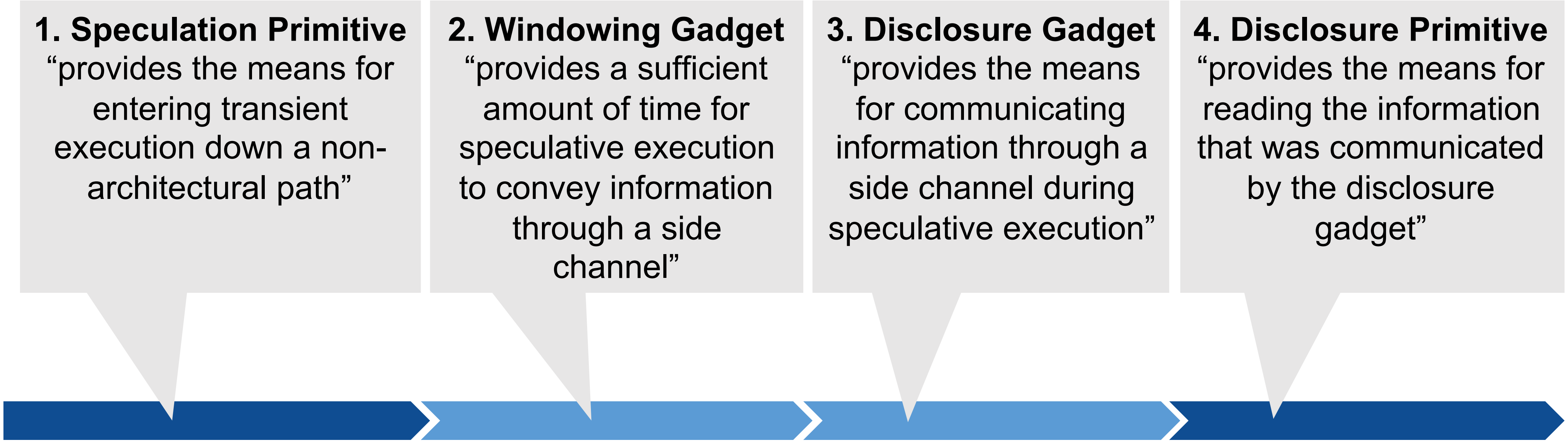
Microsoft, https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/

Attacks leveraging transient execution 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”

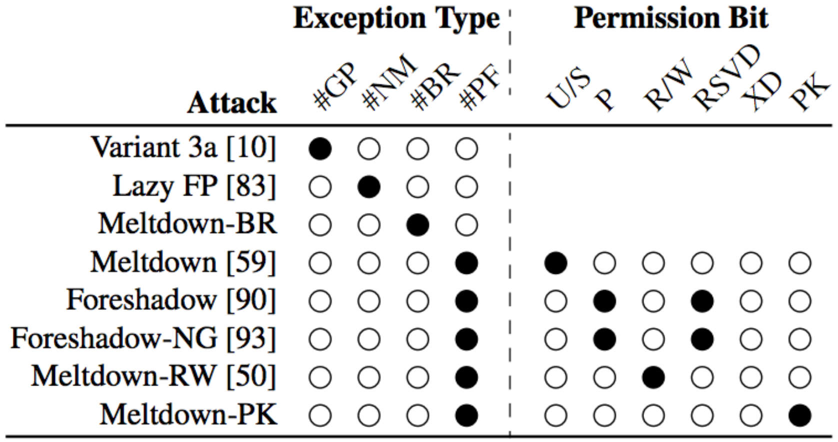
4. Disclosure Primitive “provides the means for reading the information that was communicated by the disclosure gadget”

Speculation Primitives

1. Speculation Primitive

C. Canella, et al., "A Systematic Evaluation of Transient Execution Attacks and Defenses", 2018

- **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 (RSB) SpectreRSB (V5)
  - Memory disambiguation prediction Speculative Store Bypass (V4)
- **Meltdown-type**: transient execution following a CPU exception



	Exception Type				Permission Bit					
Attack	#GP	#NM	#BR	#PF	U/S	P	R/W	RSVD	XD	PK
Variant 3a [10]	●	○	○	○						
Lazy FP [83]	○	●	○	○						
Meltdown-BR	○	○	●	○						
Meltdown [59]	○	○	○	●	●	○	○	○	○	○
Foreshadow [90]	○	○	○	●	○	●	○	●	○	○
Foreshadow-NG [93]	○	○	○	●	○	●	○	●	○	○
Meltdown-RW [50]	○	○	○	●	○	○	●	○	○	○
Meltdown-PK	○	○	○	●	○	○	○	○	○	●

GP:	general protection fault
NM:	device not available
BR:	bound range exceeded
PF:	page fault
U/S:	user / supervisor
P:	present
R/W:	read / write
RSVD:	reserved bit
XD:	execute disable
PK:	memory-protection keys (PKU)

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 (RSB) -- 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;
```

Store “slowly”

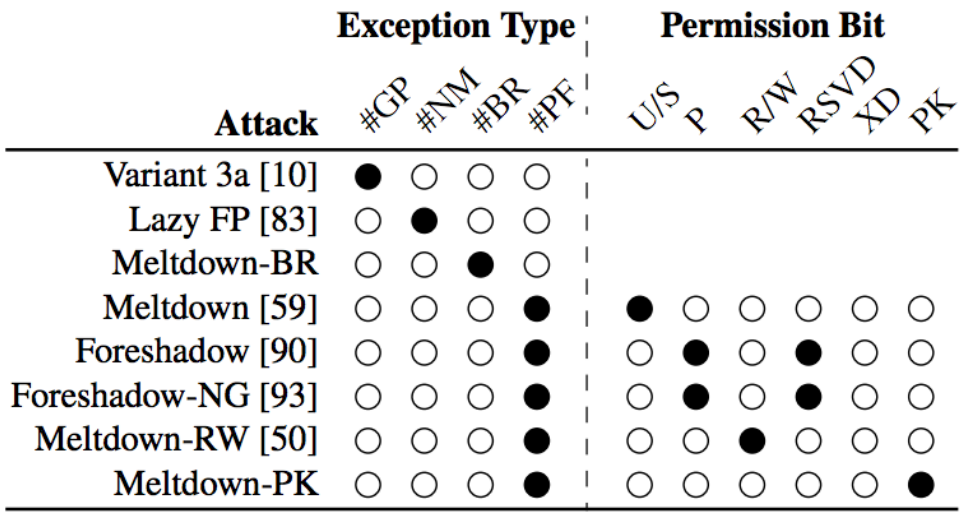
```
value2 = arr2[(*ptr)<<12];
```

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

Speculation Primitives – Sample Code

C. Canella, et al., "A Systematic Evaluation of Transient Execution Attacks and Defenses", 2018

Meltdown-type: transient execution following a CPU exception



	Exception Type				Permission Bit					
Attack	#GP	#NM	#BR	#PF	U/S	P	R/W	RSVD	XD	PK
Variant 3a [10]	●	○	○	○						
Lazy FP [83]	○	●	○	○						
Meltdown-BR	○	○	●	○						
Meltdown [59]	○	○	○	●	●	○	○	○	○	○
Foreshadow [90]	○	○	○	●	○	●	○	●	○	○
Foreshadow-NG [93]	○	○	○	●	○	●	○	●	○	○
Meltdown-RW [50]	○	○	○	●	○	○	●	○	○	○
Meltdown-PK	○	○	○	●	○	○	○	○	○	●

GP:	general protection fault
NM:	device not available
BR:	bound range exceeded
PF:	page fault
U/S:	user/supervisor
P:	present
R/W:	read/write
RSVD:	reserved bit
XD:	execute disable
PK:	memory-protection keys (PKU)

```
(rcx = address lead to exception)
(rbx = probe array)
Retry:
mov al, byte [rcx]
shl rax, 0xc
jz retry
Mov rbx, qword [rbx + rax]
```

[M. Lipp et al., 2018]

Windowing Gadget

2. 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 :

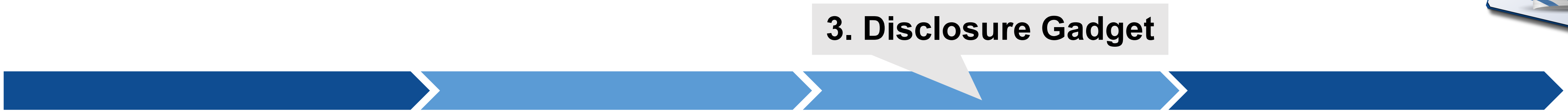
Memory access time determines how long it takes to resolve the branch

```
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:

speculative window size > disclosure gadget’s triger latency

Disclosure Gadget



1. Load the secret to register
2. Encode the secret into channel

} Transient execution

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();
  }
```

Spectre Variant2 (Branch Poisoning)
Cache side channel

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

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

Disclosure Primitives

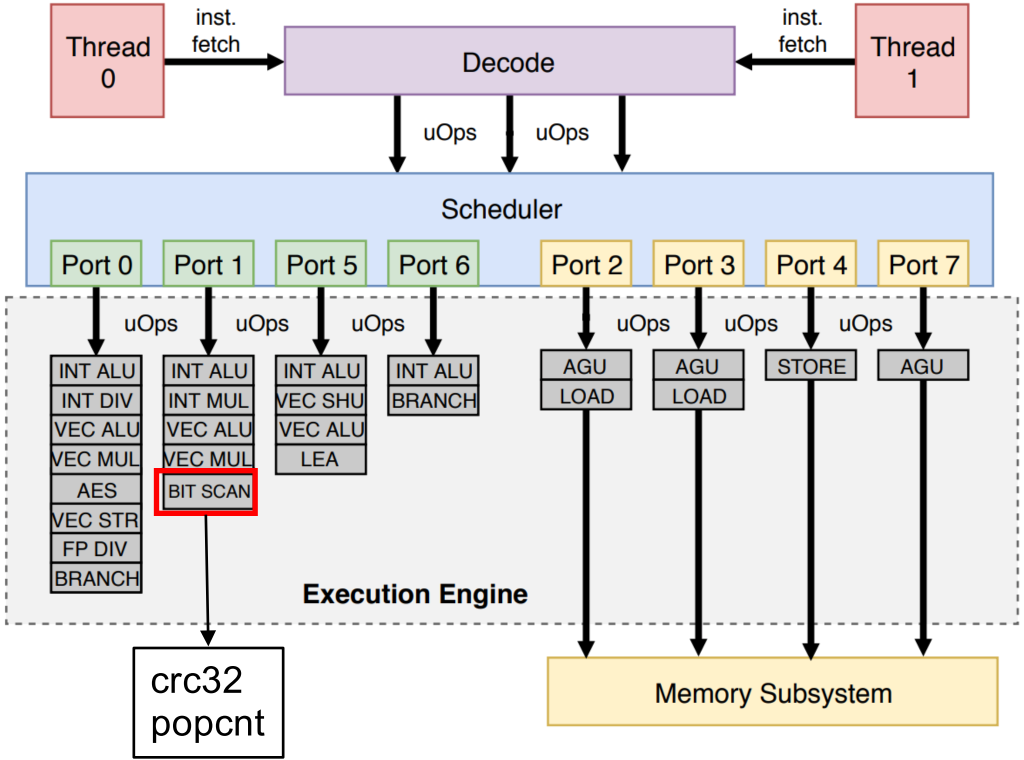
Two types of disclosure primitives:

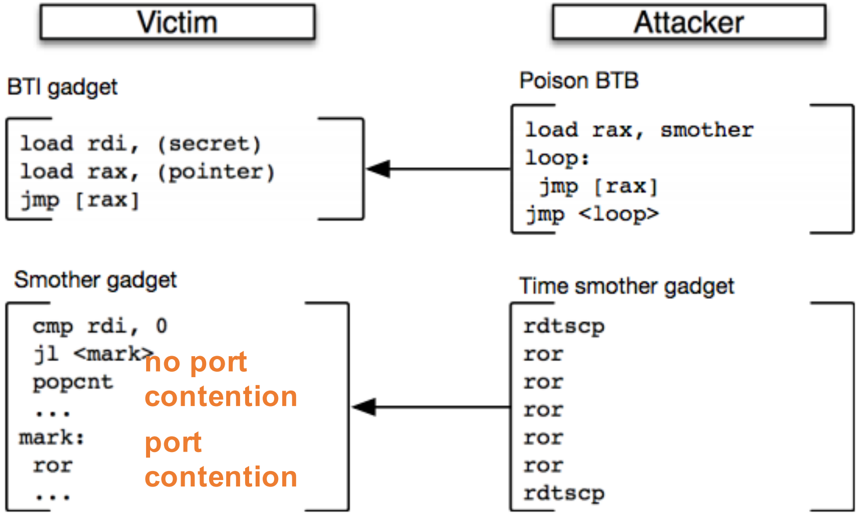
- **Transient channel** (hyper-threading / multi-core scenario):
    1. Share resource on the fly (e.g., bus, port, cache bank)
    2. or state change within speculative window (e.g., speculative buffer)
- **Permanent channel**:
    - Change the state of micro-architecture
    - The change remains even after the speculative window
    - Micro-architecture components to use:
        - D-Cache (L1, L2, L3) (Tag, replacement policy state, Coherence State, Directory), I-cache; TLB, AVX (power on/off), DRAM Rowbuffer, …
    - Encoding method:
        - Contention (e.g., cache Prime+Probe)
        - Reuse (e.g., cache Flush+Reload)

Disclosure Primitives – Port Contention

A. Bhattacharyya, et al., "SMoTherSpectre: exploiting speculative execution through port contention", 2019
A. C. Aldaya, et al., "Port Contention for Fun and Profit", 2018

- Execution units and ports are shared between hyper-threads on the same core
- Port contention affect the timing of execution





BTI gadget

```
load rdi, (secret)
load rax, (pointer)
jmp [rax]
```

Poison BTB

```
load rax, smother
loop:
 jmp [rax]
jmp <loop>
```

Smother gadget

```
 cmp rdi, 0
 jl <mark>
 popcnt
 ...
mark:
 ror
 ...
```

Time smother gadget

```
rdtscp
ror
ror
ror
ror
ror
rdtscp
```

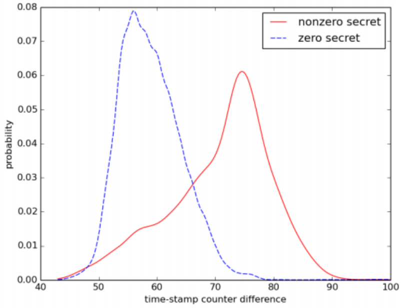


Fig. Probability density function for the timing of an attacker measuring ***crc32*** operations when running concurrently with a victim process that speculatively executes a branch which is conditional to the (secret) value of a register being zero.

Disclosure Primitives – Cache Coherence State

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

- 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;
      }
}
```

-- a write on the remote core makes the cache coherence state to be exclusive on the remote core.

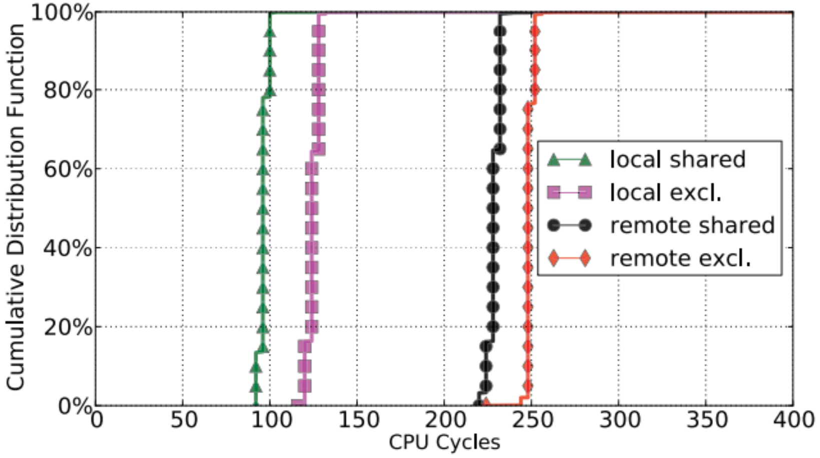


Fig. 2: Load operation latency in various (location, coherence state) combinations.

Disclosure Primitives – Directory in Non-Inclusive Cache

M. Yan, et al. “Attack Directories, Not Caches: Side-Channel Attacks in a Non-Inclusive World”, S&P 2019

- Similar to the caches, the directory structure in can be used as covert channel

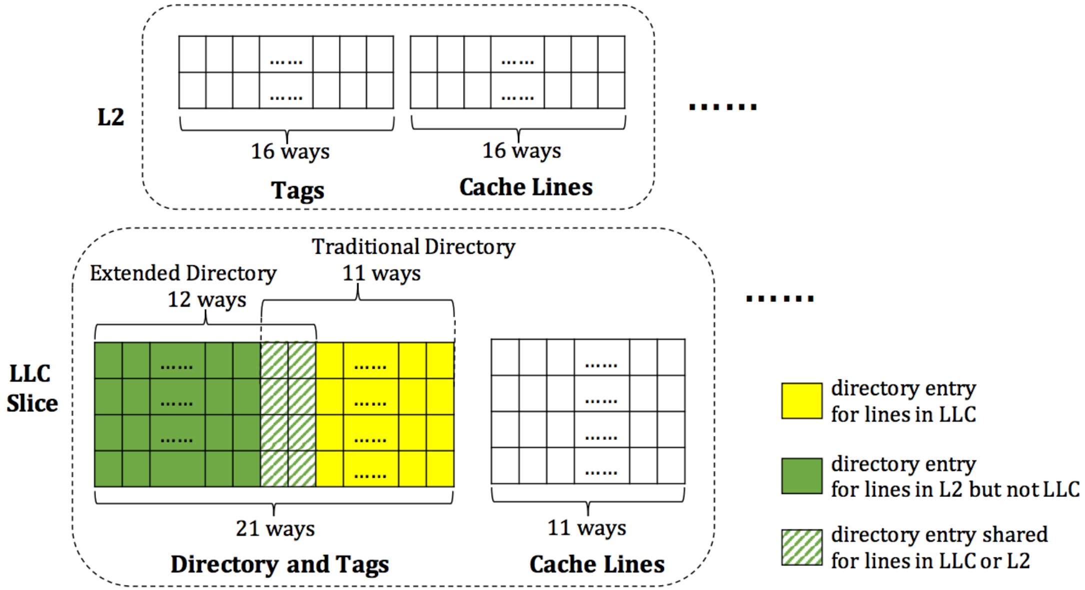


Fig. 9. Reverse engineered directory structure.

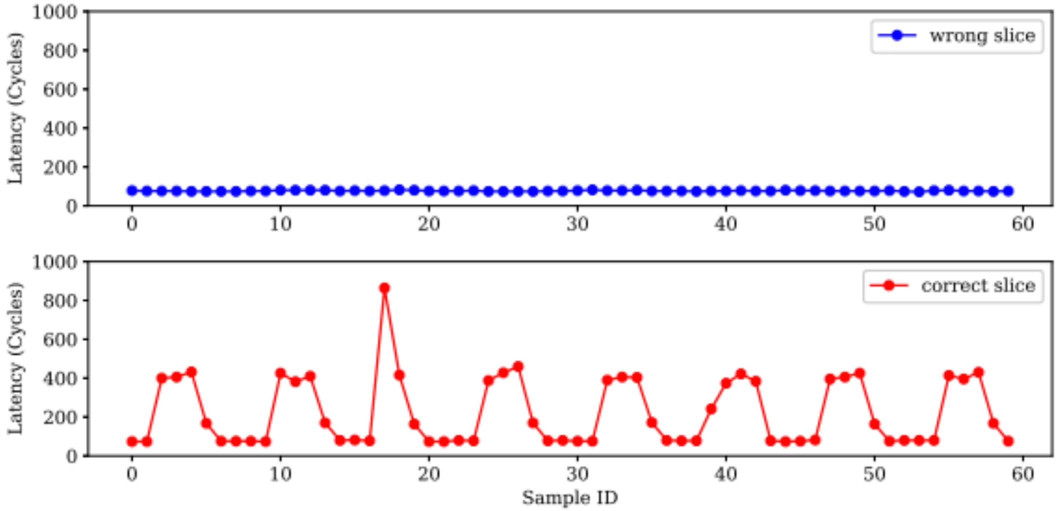


Fig. 13. The upper plot shows receiver’s access latencies on a slice not being used for the covert channel, while the lower one shows the one used in the covert channel. Sender transmits sequence “101010...”.

Disclosure Primitives - AVX Unit States

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

- 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

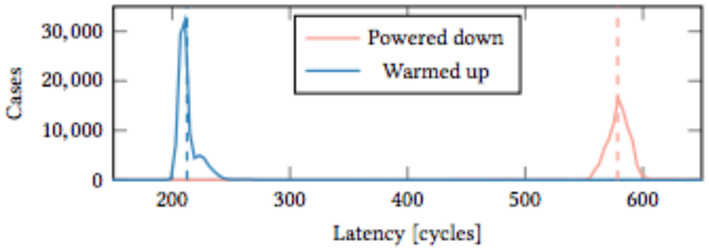


Figure 5: Differences in the execution time for AVX2 instructions (Intel i5-6200U). If the AVX2 unit is inactive (powered down), executing a 256-bit instruction takes on average 366 cycles longer than on an active AVX2 unit. The average values are shown as dashed vertical lines.

Gadget:

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

Attack “Parameters”

1. **Ability to affect speculation primitive**
   - Can the attacker affect predictor state?
2. **Speculative window size**
   - The delay from prediction to when misprediction is detected
3. **Disclosure gadget’s latency (encoding time)**
   - Amount of time needed to extract secret information and put into micro-architectural state
4. **Time reference resolution**
   - How accurate is reference clock
5. **Extraction window size** or **Disclosure primitive latency**
   - Amount of time when data can be extracted
6. **Retention time of channel**
   - How long the channel will keep the secret. e.g., AVX channel, 0.5~1ms

Bandwidth of the channel: How fast data can be transmitted? High-bandwidth is about 100bps

In-thread, Cross-thread, or Cross-processor: Do attacker and victim share same thread, are on sibling threads in SMT, or can be on separate processors?

Necessary (but not sufficient) success conditions:

speculative window size > disclosure gadget’s latency
retention time of channel > disclosure prim. latency

Disclosure Gadget Latency

Common transient execution attacks leverage some form of cache-based timing attacks:

1. Disclosure gadget modifies cache state
2. Disclosure primitive uses cache timing to find out how the state changed

Whole disclosure gadget has to fit into speculation window:

- E.g. cache Flush+Reload attack requires to fetch data from main memory, thus window has to be bigger than about 300 cycles
- E.g. Foreshadow attack requires fetch from L1 cache, so few cycles window is enough

Cache and Memory Access Latencies

L1	*1 cycle*
L2	*10 cycles*
L3	*50 cycles*
Memory	*~300 cycles*

Transient Attacks Categorization

A categorization of transient attacks has been proposed by Canella, et al.:

- Attacks depend on prediction of faults
- No attacks found to depend on traps and aborts

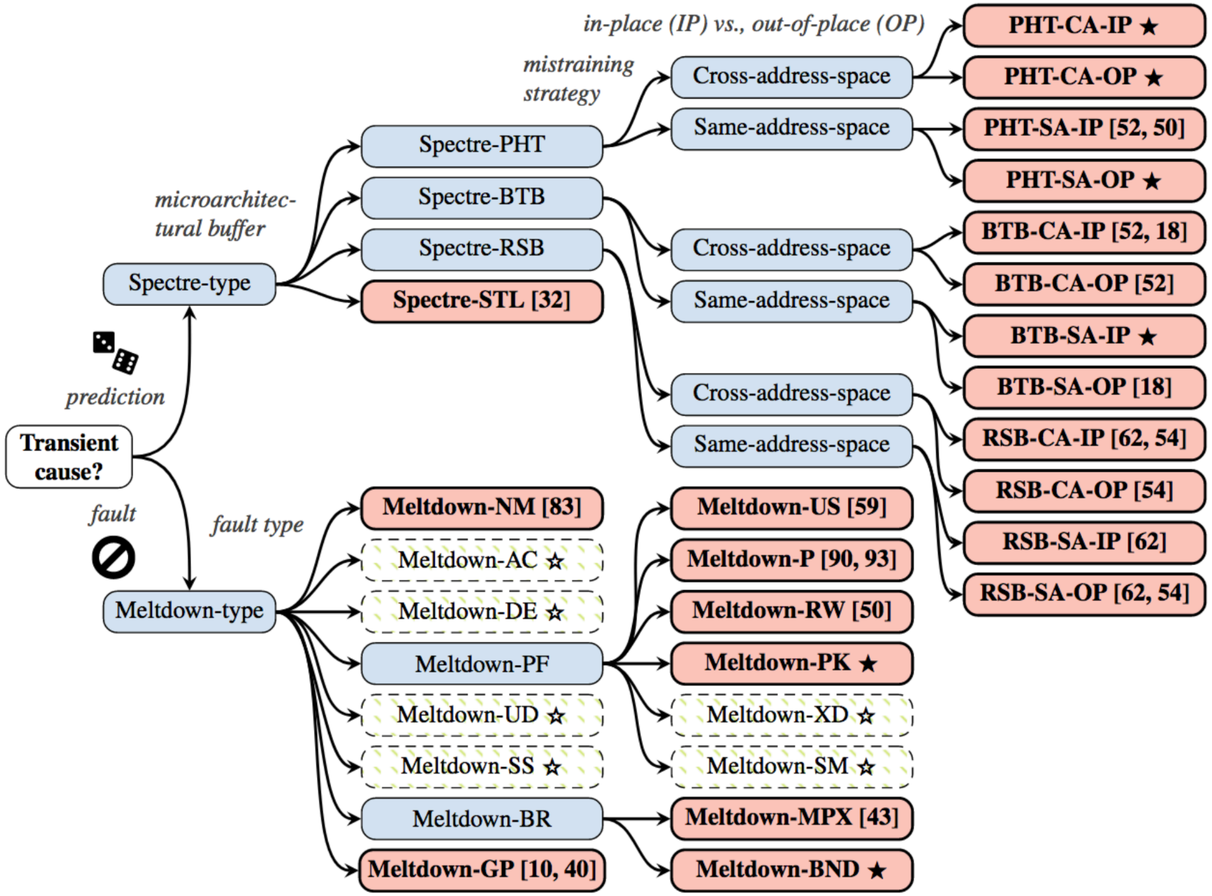


Image and reference:
“A Systematic Evaluation of Transient Execution Attacks and Defenses”, https://arxiv.org/pdf/1811.05441.pdf

Transient Attack Mitigation Techniques

Mitigation Techniques for Attacks due to Speculation

Transient Execution Attacks = Transient Execution + Covert or Side Channel

1. **Prevent or disable speculative execution** – addresses Speculation Primitives
   - Today there is no user interface for fine grain control of speculation; overheads unclear
2. **Limit attackers ability to influence predictor state** – addresses Speculation Primitives
   - Some proposals exist to add new instructions to minimize ability to affect branch predictor state, etc.
3. **Minimize attack window** – addresses Windowing Gadgets
   - Ultimately would have to improve performance of memory accesses, etc.
   - Not clear how to get exhaustive list of all possible windowing gadget types
4. **Track sensitive information** (information flow tracking) – addresses Disclosure Gadgets
   - Stop transient speculation and execution if sensitive data is touched
   - Users must define sensitive data
5. **Prevent timing channels** – addresses Disclosure Primitives
   - Add secure caches

Mitigation Techniques for Attacks due to Faults

Transient Execution Attacks = Transient Execution + Covert or Side Channel

1. **Evaluate fault conditions sooner**
   - Will impact performance, not always possible
2. **Limit access condition check races**
   - Don’t allow accesses to proceed until relevant access checks are finished

Mitigation Techniques for MDS

Transient Execution Attacks = Transient Execution + Covert or Side Channel

1. **Prevent Micro-architectural Data Sampling**
   - Will impact performance, not always possible

Mitigations in Micro-architecture: InvisiSpec

M. Yan, et al., “InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy”, 2018

- Focus on transient loads in disclosure gadgets
- Unsafe speculative load (USL)
    - The load is speculative and may be squashed
    - Which should not cause any micro-architecture state changes visible to the attackers
    - Speculative Buffer: a USL loads data into the speculative buffer (for performance), not into the local cache
- Visibility point of a load
    - After which the load can cause micro-architecture state changes visible to attackers
- Validation or Exposure:
    - Validation: the data in the speculative buffer might not be the latest, a validation is required to maintain memory consistency.
    - Exposure: some loads will not violate the memory consistency.
- Limitations: only for covert channels in caches

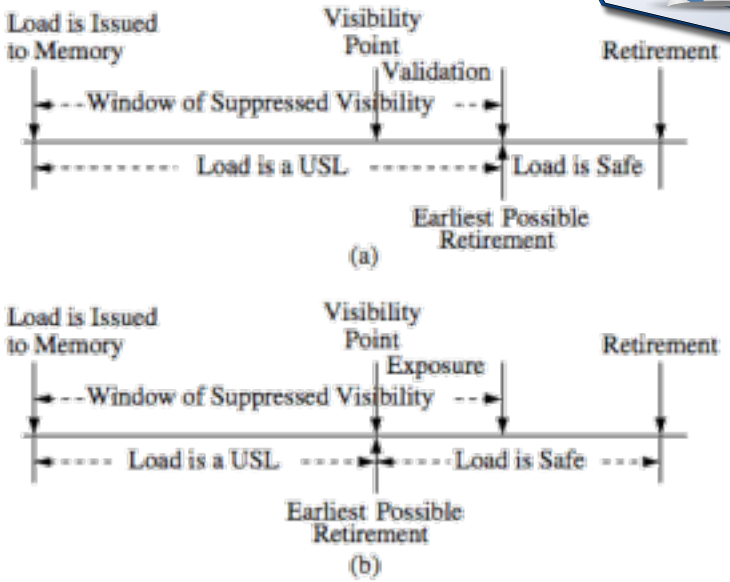
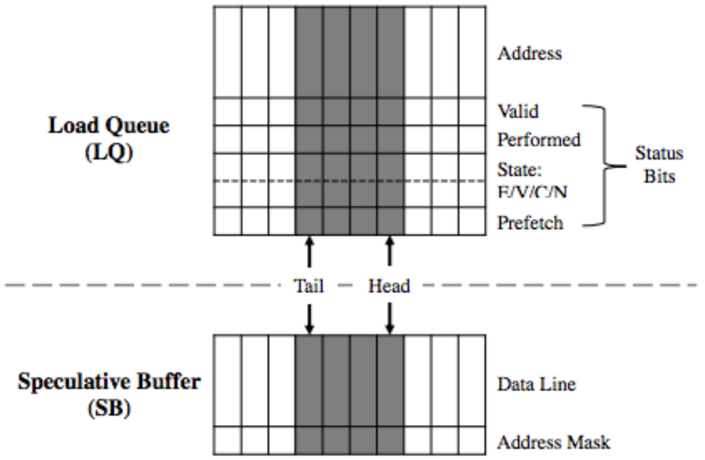


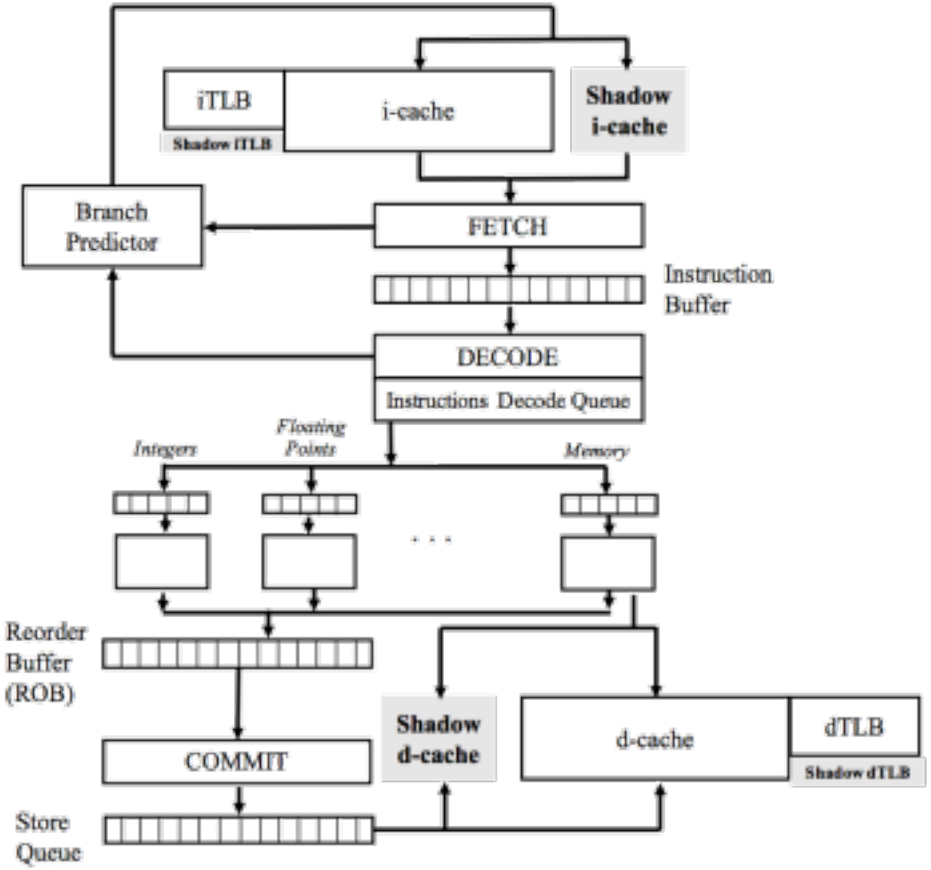
Fig. 2: Timeline of a USL with validation (a) and exposure (b).



Mitigations in Micro-architecture: SafeSpec

K. N. Khasawneh, et al., “SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation”, 2018

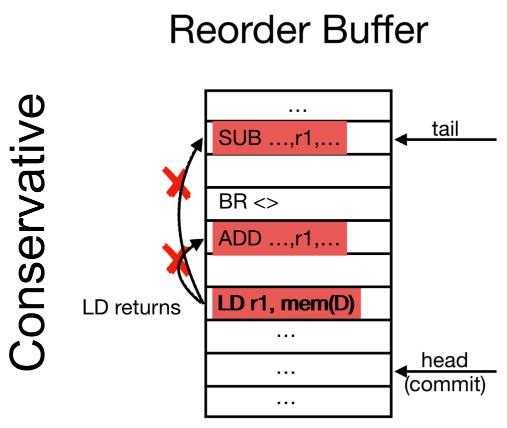
- Similar to InvisiSpec, shadow caches and TLBs are proposed to store the micro-architecture changes by speculative loads temporarily

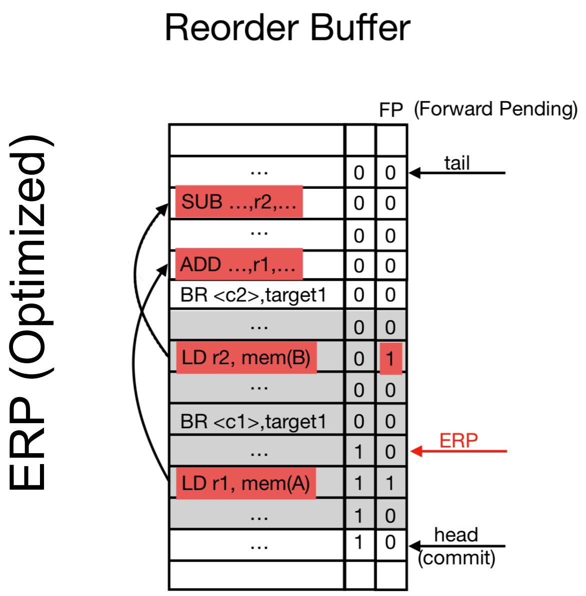


Mitigations in Micro-architecture: SpecShield

"WiP: Isolating Speculative Data to Prevent Transient Execution Attacks" Kristin Barber, et al., HASP 2019 Presentation

- Similar to other work key idea to **restrict speculative data use by dependent instructions**
- Approach:
  - Monitor speculative status of Load instructions
  - Forward data to dependents only when "safe"
- Two schemes:
  - **Conservative** – don't forward data from loads until they reach the head of the ROB
  - **Early Resolution Point** (Optimized) – all older branches have resolved *and* all older loads and stores have had addresses computed *and* No branch mis-predictions or memory-access exceptions





Mitigations in Micro-architecture: ConTExT

“ConTExT: Leakage-Free Transient Execution”, Michael Schwarz et al., arXiv 2019

- ConTExT (Considerate Transient Execution Technique) makes the proposal that **secrets can enter registers, but not transiently leave them**
- It mitigates the recently found MDS attacks on processor buffers, such as fill buffers:
  - Secret data is ‘tagged’ in memory using extra page table entry bits to indicate the secure data
  - Extra tag bits are added to registers to indicate they contain the secret data
- The tagged secret data cannot be used during transient execution

Mitigations in Micro-architecture: Conditional Speculation

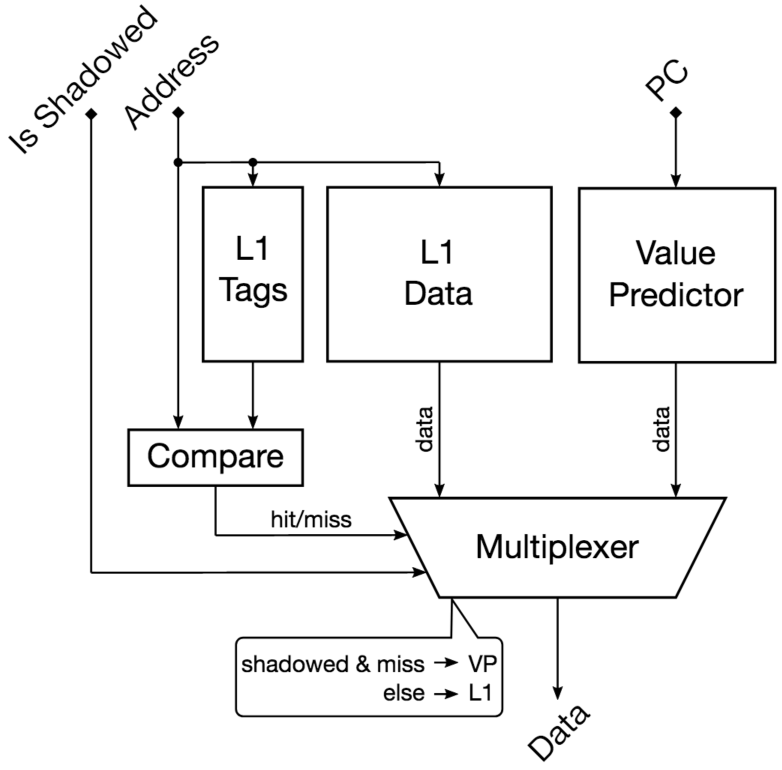
“Conditional Speculation: An Effective Approach to Safeguard Out-of-OrderExecution Against Spectre Attacks”, Peinan Li et al., HPCA 2019

- Introduces **security dependence**, a new dependence used to depict the speculative instructions which leak micro-architecture information
- **Security hazard detection** was introduced in the issue queue to identify suspected unsafe instructions with security dependence
- Performance filters:
  - **Cache-hit based Hazard Filter** targets at the speculative instructions which hit the cache – but there are now leaks via LRU!
  - **Trusted Page Buffer based Hazard Filter** targets at the attacks which use Flush+Reload type channels or other channels using shared page, others are assumed safe – but there are many other channels in the caches, and beyond!

Mitigations in Micro-architecture: EISE

"Efficient Invisible Speculative Execution throughSelective Delay and Value Prediction", Christos Sakalis, et al., ISCA 2019.

- Efficient Invisible Speculative Execution (**EISE** acronym added by course author) through selective delay and value prediction proposes to:
  a) (naïve) delay loads until they reach the head of ROB or (eager) until they will no longer be squashed
  b) allow only accesses that hit in the L1 data cache to proceed – LRU channel issues!
  c) prevent stalls by value predicting the loads that miss in the L1 – value prediction can leak data values as well, security of value prediction is not well studied

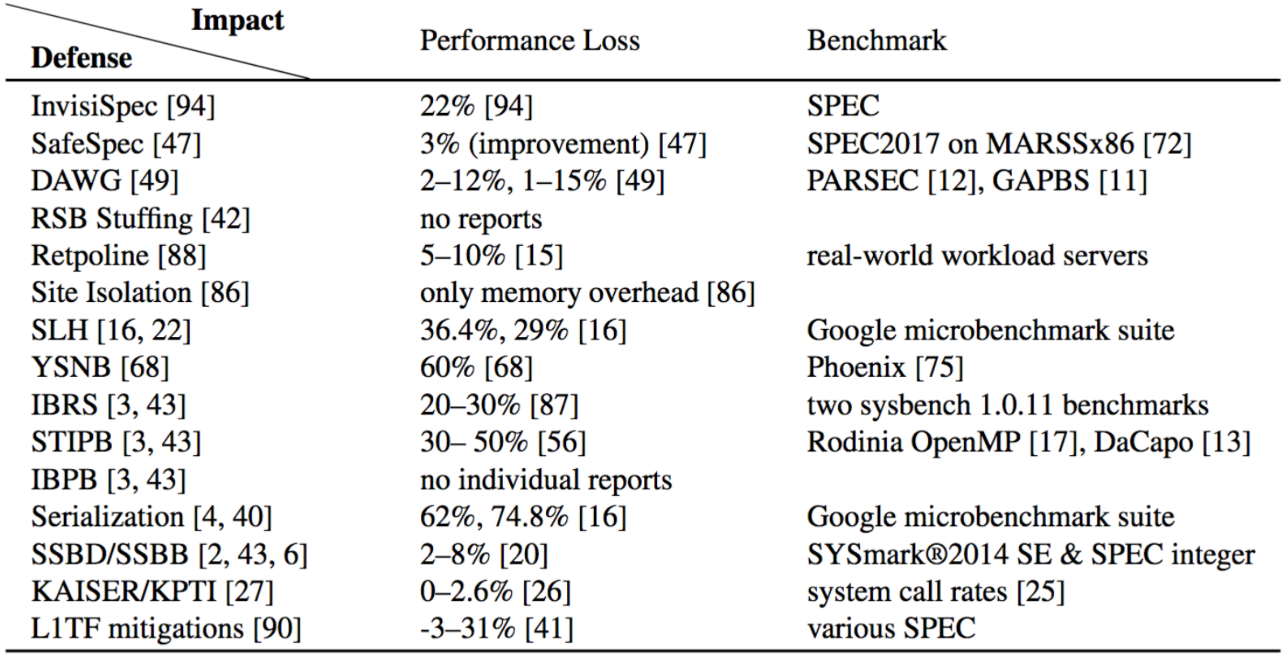


Mitigation Overheads

A summary of overheads has been compiled by Canella, et al.:

- No clear trend in mitigation overheads
  - From small negative to upwards of 80% overheads
- There exists lack of standard benchmarks and platforms for evaluation
- Overheads are application and micro-architecture specific

Ultimate mitigation: properly clean up all architectural and micro-architectural state following transient execution



Impact **Defense**	Performance Loss	Benchmark
InvisiSpec [94]	22% [94]	SPEC
SafeSpec [47]	3% (improvement) [47]	SPEC2017 on MARSSx86 [72]
DAWG [49]	2–12%, 1–15% [49]	PARSEC [12], GAPBS [11]
RSB Stuffing [42]	no reports	
Retpoline [88]	5–10% [15]	real-world workload servers
Site Isolation [86]	only memory overhead [86]	
SLH [16, 22]	36.4%, 29% [16]	Google microbenchmark suite
YSNB [68]	60% [68]	Phoenix [75]
IBRS [3, 43]	20–30% [87]	two sysbench 1.0.11 benchmarks
STIPB [3, 43]	30– 50% [56]	Rodinia OpenMP [17], DaCapo [13]
IBPB [3, 43]	no individual reports	
Serialization [4, 40]	62%, 74.8% [16]	Google microbenchmark suite
SSBD/SSBB [2, 43, 6]	2–8% [20]	SYSmark®2014 SE & SPEC integer
KAISER/KPTI [27]	0–2.6% [26]	system call rates [25]
L1TF mitigations [90]	-3–31% [41]	various SPEC

Image and reference:
“A Systematic Evaluation of Transient Execution Attacks and Defenses”, https://arxiv.org/pdf/1811.05441.pdf

Mitigation Overheads: Hardware-Only Schemes

- Performance overhead of hardware mitigations of at the micro-architecture level

	Performance Loss	Benchmark
Fence after each branch (software)	88%	SPEC2006
InvisiSpec [M. Yan, et al., 2018]	22%	SPEC2006
SafeSpec [K. N. Khasawneh, et al., 2018]	3% improvement (due to larger effective cache size)	SPEC2017
SpecShield [K. Barber, et al., 2019]	55% (conservative) 18% (ERP)	SPEC2006
ConTExT [M. Schwarz, et al., 2019]	71% (security critical applications) 1% (real-world workloads)	n/a
Conditional Speculation [P. Li, et al., 2019]	6% - 10% (when using their filters)	SPEC2006
EISE [C. Sakalis, et al., 2019]	74% naïve, 50% eager, 19% delay-on-miss, or 11% delay-on-miss + value prediction	SPEC2006

Industry Perspective and Solutions

Solutions from industry are not covered in these slides

Likely or already implemented solutions:

- Architecture fixes for Meltdown type bugs
- Architectural fixes for MDS type attacks (new processors since 2019 already not vulnerable?)
- SGX related fixes (don't share speculative state between SGX and outside world)
- Mitigations for L1 cache related timing channels
- InvisiSpec-like Spectre solutions leveraging ROB information about instruction state

Unlikely or not coming soon solutions:

- New ISA that allows for deeper control of speculation
- New ISA that exposes micro-architectural state
- Tagging of secret data in hardware, information flow approaches

Available Today:

- **Disable or don't use SMT** so state can't be shared between two threads
- **Use simple processors such as RISC-V** that don't have performance enhancing features leading to these attacks
- **Don't run sensitive code on share hardware**

Transient Attacks and Secure Processors

Transient Execution Attacks on SGX: SgxPectre

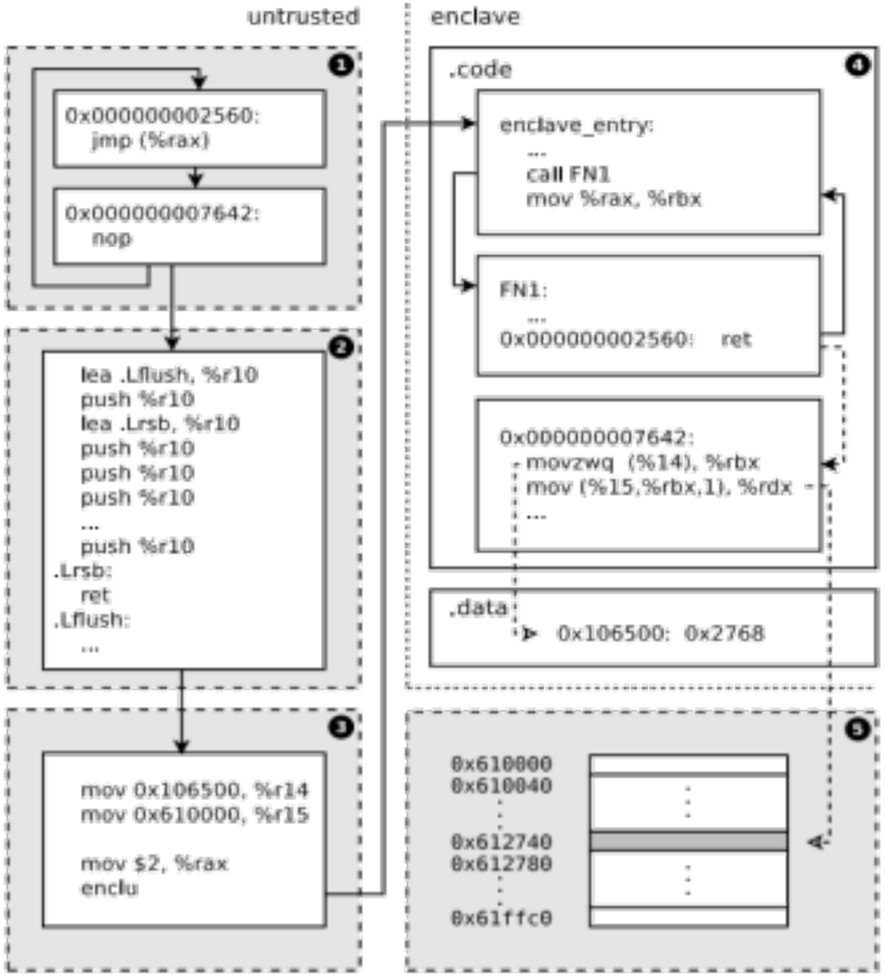
G. Chen, et al., "SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution", 2018

- Spectre can attack current secure architectures!
- E.g., Spectre v2 on SGX

1. Poison BTB
(Speculation Primitive)

2. Flush the victim's
branch target address
and deplete the RSB
(Windowing Gadget)

3. Set secret address
and probe array address



```
0x000000002560:
  jmp (%rax)
```

```
0x000000007642:
  nop
```

```
    lea .Lflush, %r10
    push %r10
    lea .Lrsb, %r10
    push %r10
    push %r10
    push %r10
    ...
    push %r10
.Lrsb:
    ret
.Lflush:
    ...
```

```
mov 0x106500, %r14
mov 0x610000, %r15

mov $2, %rax
enclu
```

```
enclave_entry:
  ...
  call FN1
  mov %rax, %rbx
```

```
FN1:
  ...
0x000000002560:   ret
```

```
0x000000007642:
  movzwq  (%14), %rbx
  mov (%15,%rbx,1), %rdx
  ...
```

```
0x106500:  0x2768
```

4. Execute victim code
(Disclosure Gadget)

5. Obtain secret from
covert channel
(Disclosure Primitive)

Figure 1: A simple example of SgxPectre Attacks. The gray blocks represent code or data outside the enclave. The white blocks represent enclave code or data.

Transient Execution Attacks on SGX: Foreshadow

J. Van Bulck, et al., “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”, 2018

- Meltdown-type attack can attack current secure architectures!

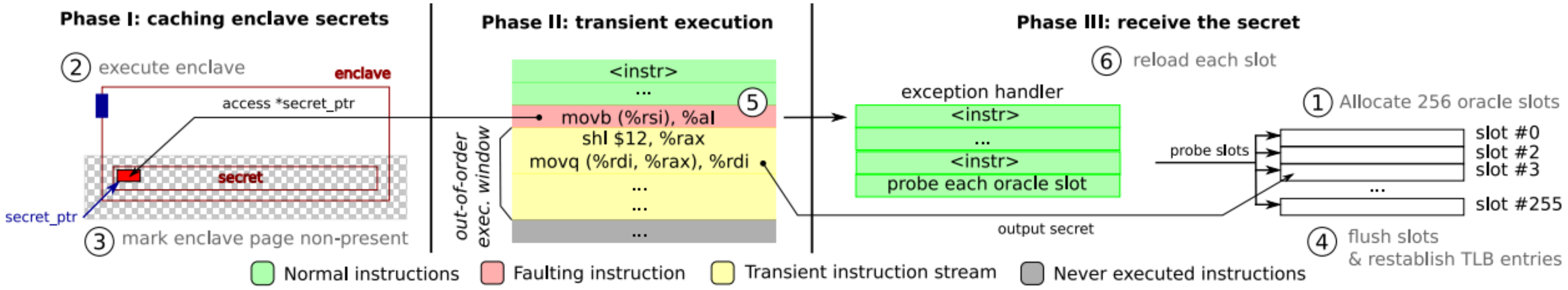


Figure 2: Basic overview of the Foreshadow attack to extract a single byte from an SGX enclave.

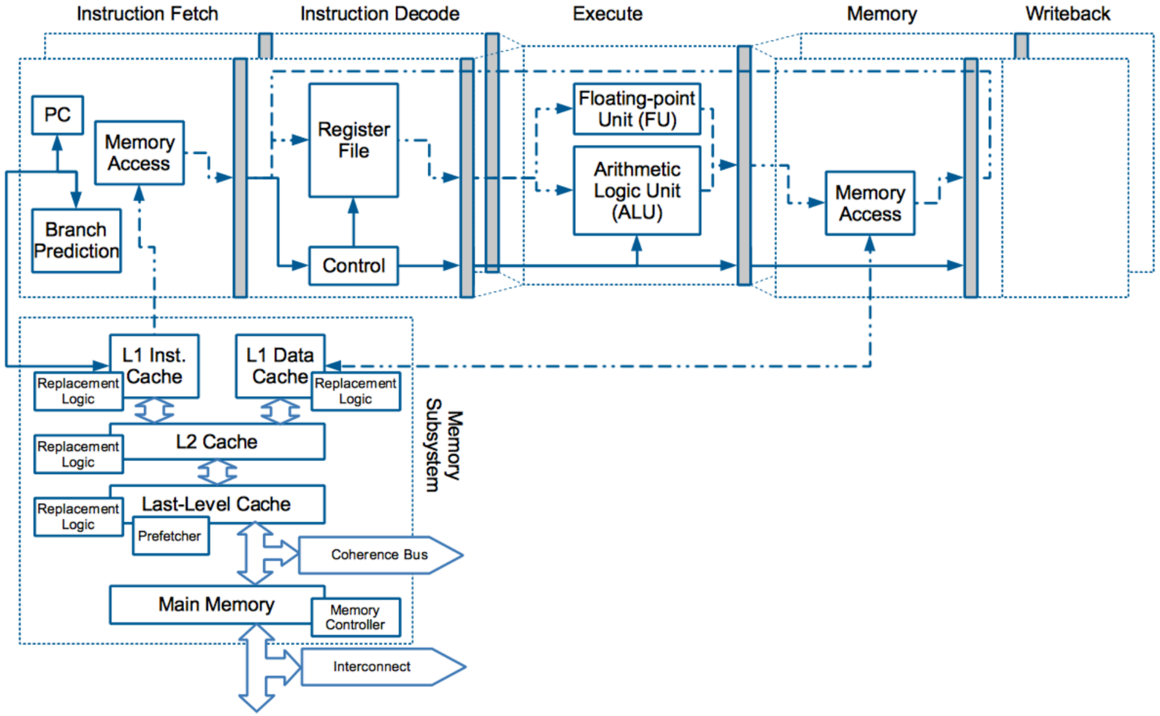
Summary

Prediction is one of the six key features of modern processor

- 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 however leads to transient execution



- **Contention during transient execution, or improperly cleaned up architectural or micro-architectural state after transient execution can lead to security attacks.**

Thank You!

Related reading…

*Jakub Szefer, "**Principles of Secure Processor Architecture Design**," in Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers,* October 2018.

https://caslab.csl.yale.edu/books/

Principles of Secure Processor Architecture Design

Jakub Szefer
Yale University

SYNTHESIS LECTURES ON COMPUTER *CTURE #43*

MORGAN & CLAYPOOL PUBLISHERS