Securing Processor Architectures

Jakub Szefer
Associate Professor
Dept. of Electrical Engineering
Yale University

Tutorial co-located with ASPLOS 2021 – April 15, 2021

Logistics and Schedule

All times are in the Pacific Time (PT) time zone.

11:00am – 1:00pm
Tutorial – Topics: Information leaks in processors and transient execution attacks

1:00pm – 1:12pm
Brainstorming and shuffle rooms

1:12pm – 1:15pm
Break

1:15pm – 2:30pm
Tutorial – Topics: Trusted Execution Environments

2:30pm – 2:42pm
Brainstorming and shuffle rooms

2:42pm – 3:00pm
Wrap-up and conclusion

Need for Securing Processor Architectures

- Computer systems have become pervasive today
- Attacks on such systems affect privacy and security of users
- **Processor hardware security** focuses on analyzing attacks and defenses which originate in the hardware

- Need to understand the attacks to develop better defenses
- Hardware defenses can tackle the root-causes of the attacks
- Threats keep evolving so research and defenses need to keep up

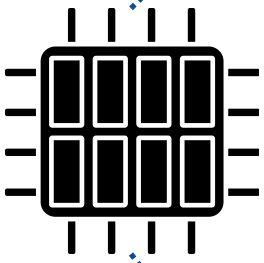
Outline

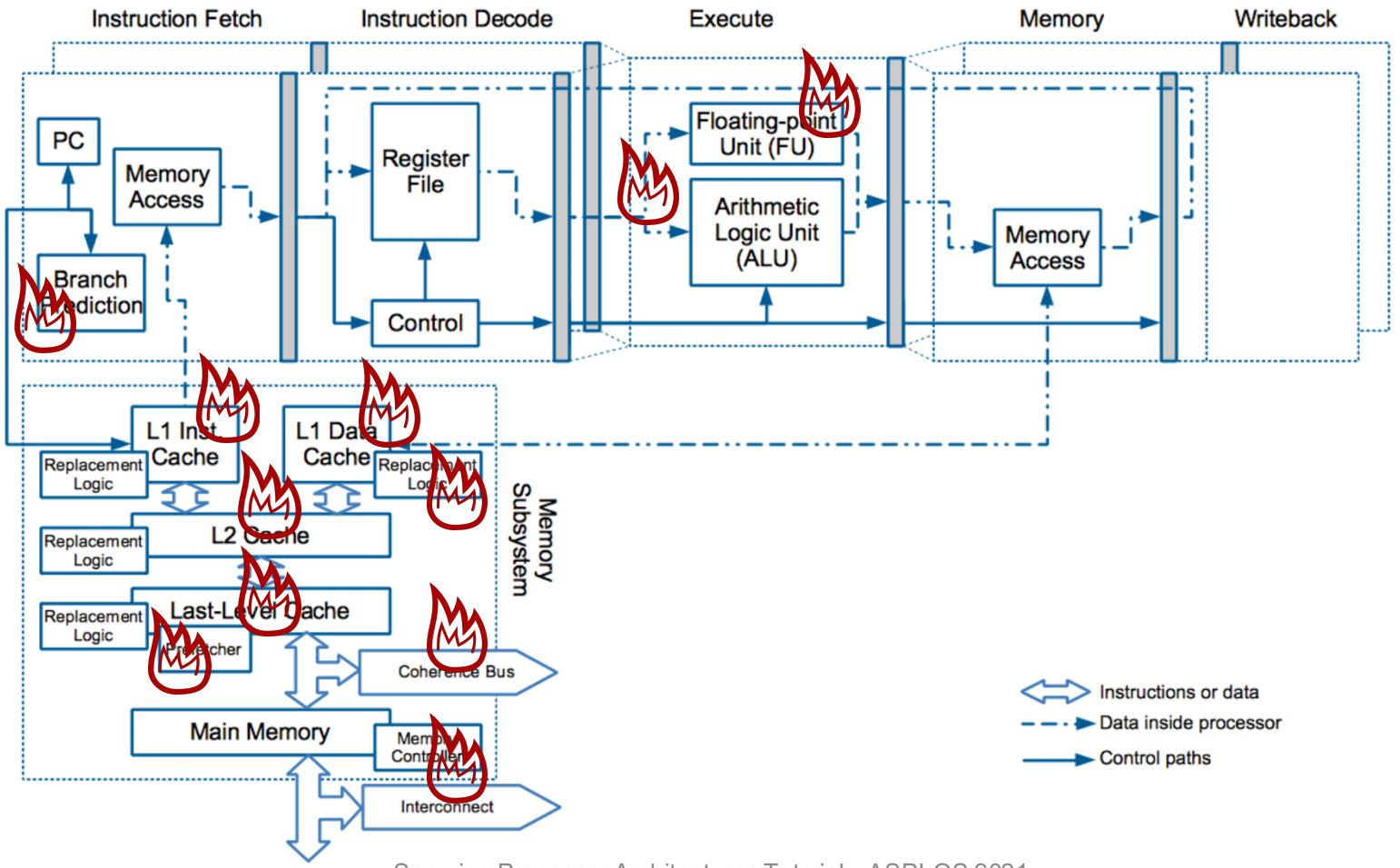
- **Information Leaks in Processors**
  - Side and Covert Channels in Processors
  - Side Channel Attacks on ML Algorithms
  - Other Attacks: Power and Energy based Attacks
  - Hardware Defense for Side and Covert Channels
- **Transient Execution Attacks in Processors**
  - Transient Execution and Attacks on Processors
  - Hardware Defenses for Transient Execution Attacks
- **Design of Trusted Execution Environments**
- **Wrap-up and Conclusion**

Side and Covert Channels in Processors

Information Leaks in Processors

Many components of a modern processor pipeline can contribute to information leaks





Focus of this talk is on (remote) timing and power information leaks

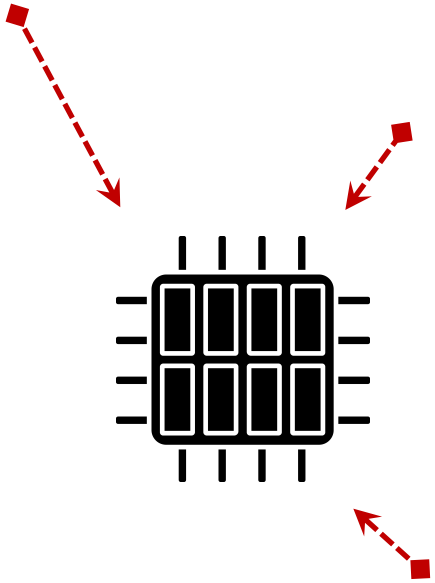
Information Leaks with and without Physical Access

This talk:

Remote Attacks
(don't require physical access)

Timing channels don't require measurement equipment, only attacker can run code on victim (not even always needed, c.f. AVX-based channel used in NetSpectre) and have network connection.

Power channels possible with access to proper performance counters



Physical Access Attacks
(require physical connection)

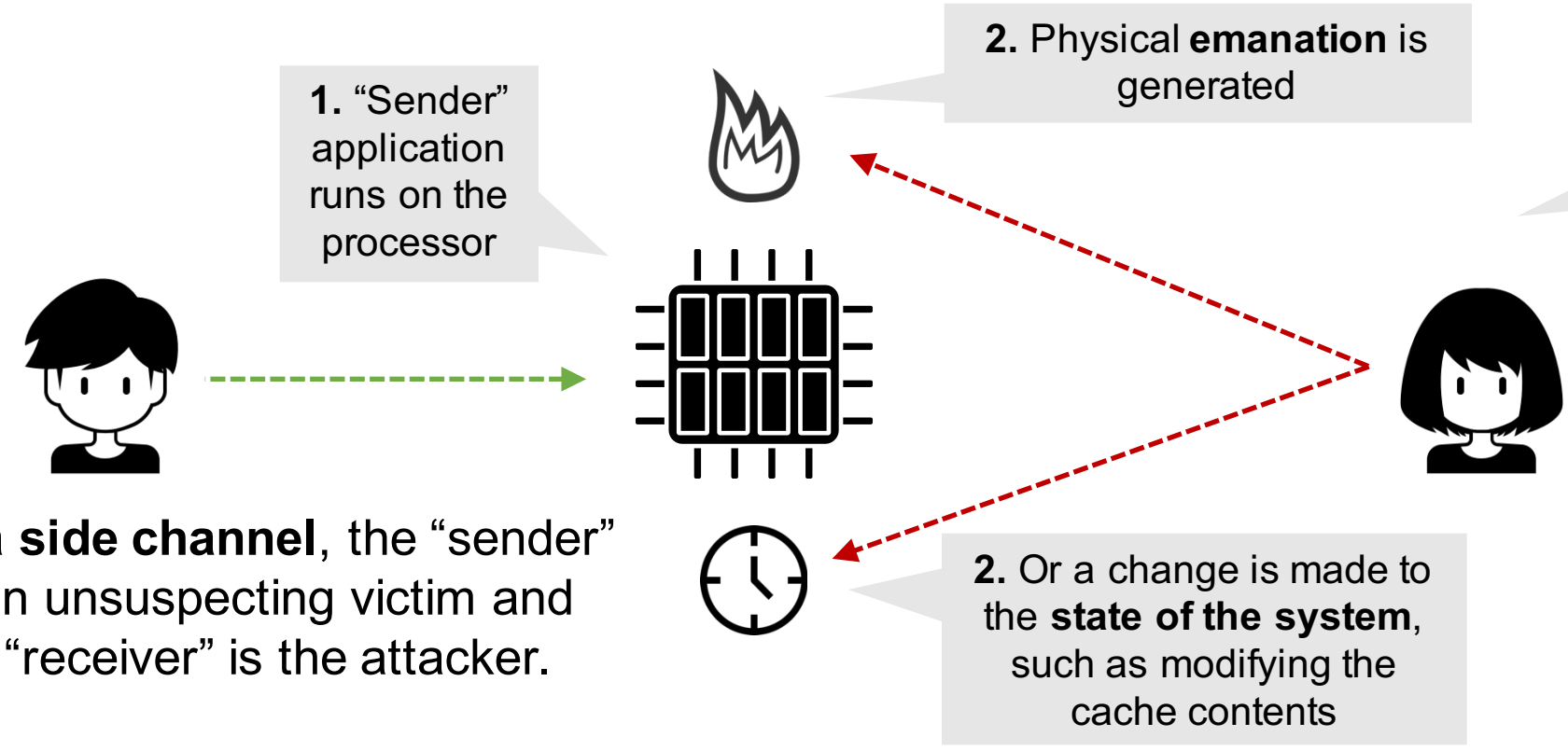
Power channels require physical connection to measure the power consumed by the CPU (assuming there is no on-chip sensor that can be abused to get power measurements).

Emanations-based Attacks
(within emanation's signal range)

Thermal, **acoustic**, and **EM** emanation based channels allow for remote signal collection, but depend on strength of the transmitter and type of emanation.

Information Leaks and 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.



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

Means for transmitting information:

- Timing
- Power
- Thermal emanations
- Electro-magnetic (EM) emanations
- Acoustic emanations

Interference Between Attacker and Victim

- Many of the information leakage attacks are based on interference
- Typically an information leak is from victim to attacker
  - Goal is to extract some information from victim

Victim's operation "sends" information to attacker

Attacker obtains information via the side channel

- But interference can also affect the victim
  - Attacker's behavior can "send" some information and modulate victim's behavior

Victim's operation depends on the processor state set by the attacker

Attacker modulates some processor state that affects the victim

Sources of Timing and Power Side Channels

Five source of timing and power channels that can lead to attacks:

1. **Instructions have Different Execution Timing or Power** – Execution of different instructions takes different amount of time or power
2. **An Instruction's Variable Timing or Power** – Execution of a specific instruction takes different time or power depending on the state of the unit
3. **Functional Unit Contention** – Sharing of hardware leads to contention, affecting timing and power that is used
4. **Prediction Units** – Prediction and misprediction affects execution timing and power
5. **Memory Hierarchy** – Caching creates fast and slow execution paths, leading to timing or power differences

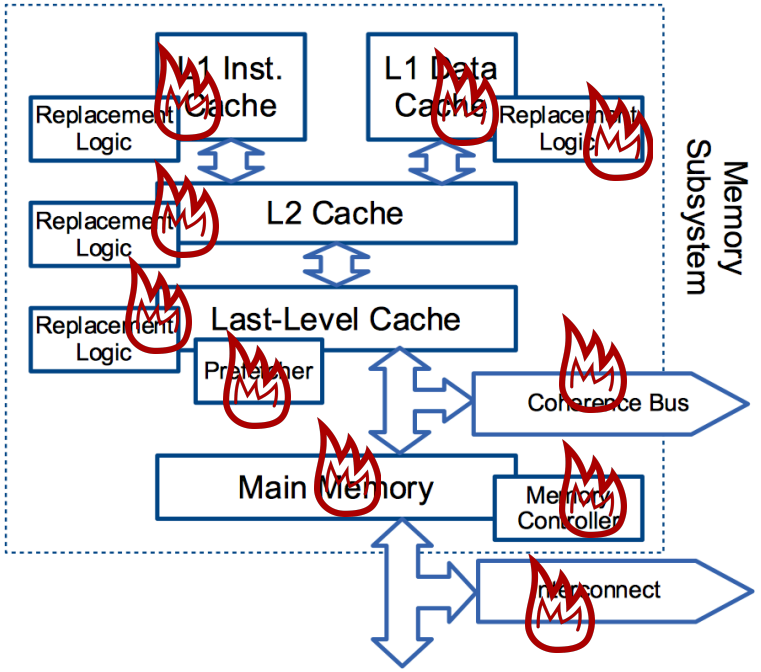
Side and covert channels are orthogonal to transient execution attacks

Transient attacks = transient execution + covert channel

Memory Hierarchy

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

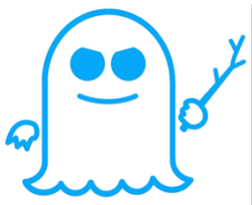
- **Caches**
  - Inclusive caches, Non-inclusive caches, Exclusive caches
  - Different cache levels: L1, L2, LLC
- **Cache Replacement Logic**
- **Load, Store, and Other Buffers**
- **TLBs**
- **Directories**
- **Prefetches**
- **Coherence Bus and Coherence State**
- **Memory Controller and Interconnect**



Importance of Cache Timing Attacks

- There is renewed interest in timing attacks due to Transient Execution Attacks
- Most of them use **transient executions and** leverage **cache timing attacks**
- Variants using cache timing attacks (side or covert channels):

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

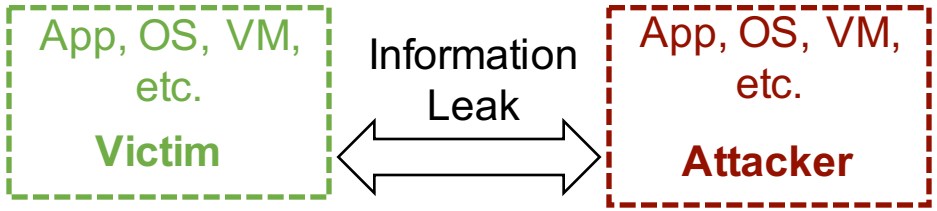


NetSpectre, Foreshadow, SGXSpectre, or SGXPectre
SpectrePrime and MeltdownPrime (both use Prime+Probe instead of original Flush+Reload cache attack)

And more…

Cache Timing Attacks

- **Attacker and Victim**
  - Victim (holds security critical data)
  - Attacker (attempts to learn the data)



- **Attack requirement**
  - Attacker has ability to monitor timing of cache operations made by the victim or by self
  - Can control or trigger victim to do some operations using sensitive data

- **Use of instructions which have timing differences**
  - Memory accesses: load, store
  - Data invalidation: different flushes (clflush, etc.), cache coherence

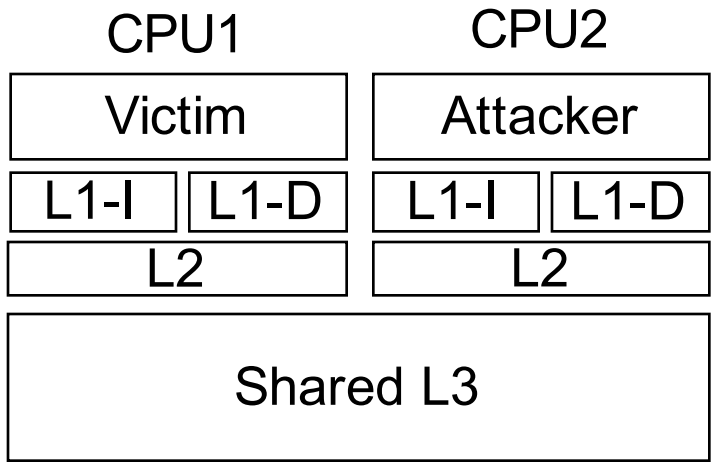
- **Side-channel attack vs. covert-channel attack**
  - Side channel: victim is not cooperating
  - Covert channel: victim (sender) works with attacker – easier to realize and higher bandwidth

- **Many known attacks:** Prime+Probe, Flush+Reload, Evict+Time, or Cache Collision Attack

Prime-Probe Attacks

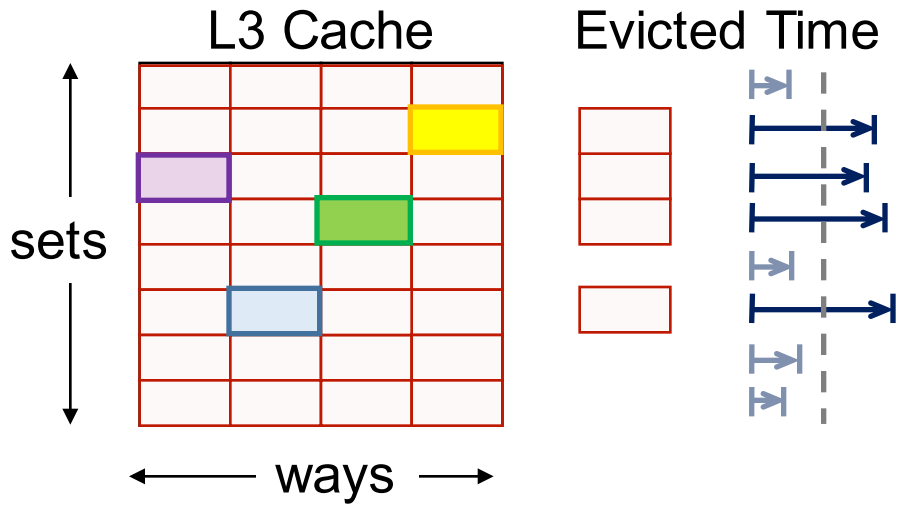
Osvik, D. A., Shamir, A., & Tromer, E, "Cache attacks and countermeasures: the case of AES". 2006.

2- Victim accesses critical data



1- Attacker primes each cache set

3- Attacker probes each cache set (measure time)

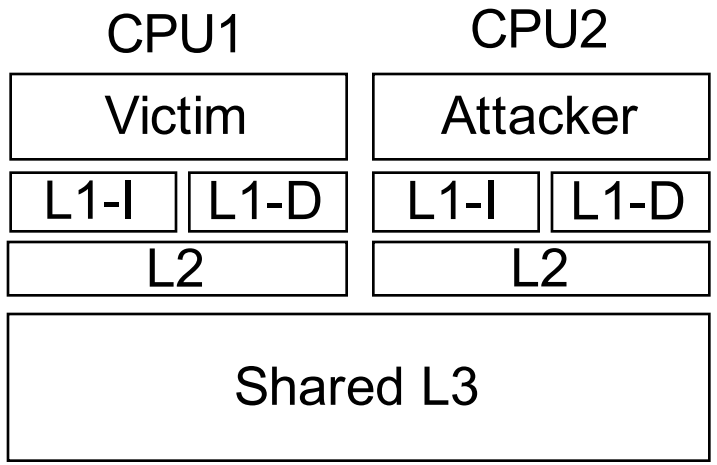


Data sharing is not needed

Flush-Reload Attack

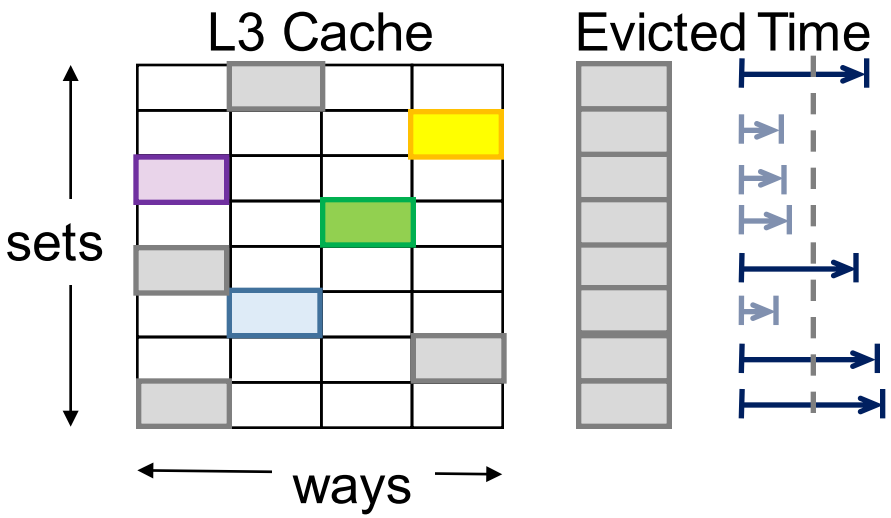
Yarom, Y., & Falkner, K. "FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack", 2014.

2- Victim accesses critical data



1- Attacker flushes each line in the cache

3- Attacker reloads critical data by running specific process (measure time)



Data sharing is needed

A Three-Step Model for Cache Timing Attack Modeling

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

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 Model

$Step1$	⇝	$Step2$	⇝	$Step3$ (fast/slow)
The initial state of the cache block set by a memory operation		Memory operation alters the state of the cache		Final memory operations and timing observation (fast/slow)

- Analyzed possible states of the cache block + used cache three-step simulator and reduction rules derive all the effective vulnerabilities

- **There are 88 possible cache timing attack types**

Discovering New Cache Attack Types

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

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].
(2) Cache Internal Collision attack [4].
(3) Bernstein's attack [2].
(4) Prime + Probe attack [31,33], Alias-driven attack [16].
(5) Flush + Reload attack [50,49], Evict + Reload attack [15].
(6) SpectrePrime, MeltdownPrime attack [41].

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].

Cache Related Attack: LRU Timing Attacks

Wenjie Xiong and Jakub Szefer, "Leaking Information Through Cache LRU States", 2019

- Cache replacement policy has been shown to be a source of timing attacks
- Many caches use variant of Least Recently Used (LRU) policy
  - **Update LRU state** on miss and **also on a cache hit**
  - Different variants exist, True LRU, Tree LRU, Bit LRU

- LRU timing attacks leverage LRU state update on both hit or miss
  - After filing cache set, even on a hit, LRU will be updated, which determines which cache line gets evicted
  - More stealthy attacks based on hits
  - Affect secure caches, such as PL cache

- High-bandwidth and work with Spectre-like attacks

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

Cache Related Attacks: TLB Timing Attacks

Ben Gras et al. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks" USENIX Security Symposium, 2018.

- Existing practical attacks have been demonstrated against TLBs, e.g., TLBleed attack on RSA

```
1.  void _gcry_mpi_powm (gcry_mpi_y_ res,
                         gcry_mpi_t base, gcry_mpi_t expom gcry_mpi_t_mod)
2.  {
3.        mpi_ptr_t rp, xp; /* pointers to MPI data */
4.        mpi_ptr_t tp;
5.        ...
6.        for(;;) {
7.            /* For every exponent bit in expo*/
8.            _gcry_mpih_sqr_n_basecase(xp, rp);
9.            if(secret_exponent || e_bit_is1) {
10.               /* unconditional multiply if exponent is
11.                * secret to mitigate FLUSH+RELOAD
12.                */
13.               _gcry_mpih_mul(xp, rp);
14.           }
15.           if(e_bit_is1) {
16.               /*e bit is 1, use the result*/
17.               tp = rp; rp = xp; xp = tp;
18.               rsize = xsize;
19.           }
20.       }
21. }
```

Existing TLBleed work can extract the cryptographic key from the RSA public-key algorithm* with a 92% success rate.

* modular exponentiation function of RSA from Libgcrypt 1.8.2: https://gnupg.org/ftp/gcrypt/libgcrypt/

Timing Channels due to Other Components

- **Cache Replacement Logic** – LRU states can be abused for a timing channel, especially cache hits modify the LRU state, no misses are required
- **TLBs** – Translation Look-aside Buffers are types of caches with similar vulnerabilities
- **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 recent Micro-architectural Data Sampling attacks
- **Coherence Bus and Coherence State** – different coherence state of a cache line may affect timing, such as flushing or upgrading state
- **Memory Controller and Interconnect** – memory and interconnect are shared resources vulnerable to contention channels

Side Channel Attacks on ML Algorithms

Attacks on Machine Learning

- Caches and other structures can leak information about secrets or user programs
  - Information leak depends on the application

- Microarchtiecture attacks on Machine Learning have emerged recently as important class of attacks:

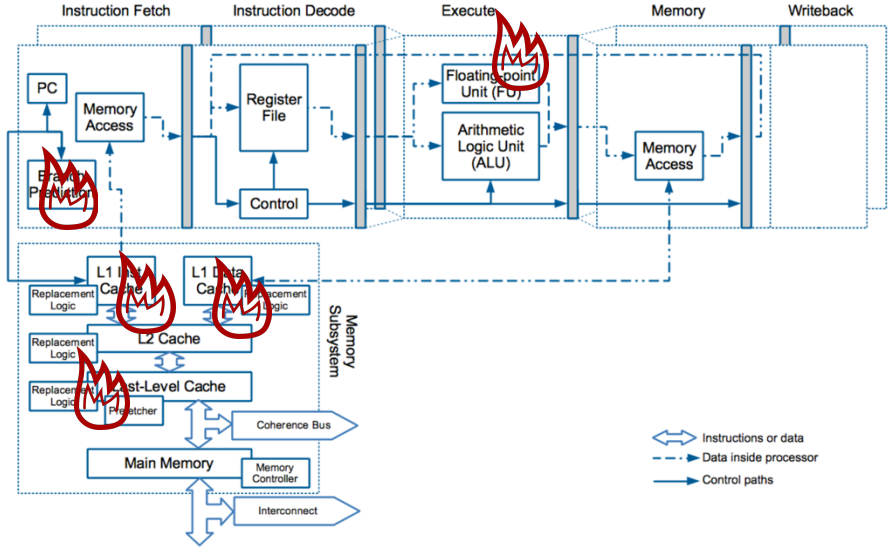
Secret or private information in Machine Learning (typically Deep Neural Networks)

- *Number of layers*
- *Layer parameters*
- *Weights and biases*
- *Inputs*

Many more attack targets

Secret or private information in Cryptography

- *Secret key*



- Most attacks target inference that is run on the CPU
  - ML parameters are considered sensitive or secret intellectual property

Example Attacks on Machine Learning Algorithms

Numerous researchers have recently explored ways to abuse timing or power related information to leak secrets from machine learning algorithms running on processors.

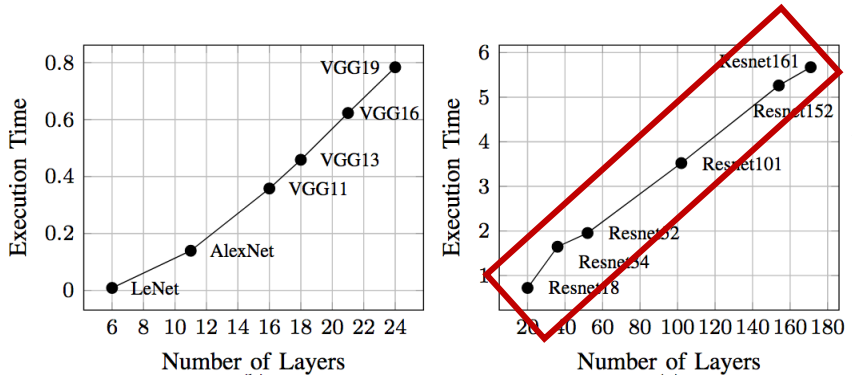
Attacks requiring physical access and measurements

Attacks not requiring physical access

	Platform	Channel	Remote Attack	Target			Target Algorithm
				Arch.	Para.	Data	
Xiang, et al.	CPU (Arm)	Power	N	✓	×	×	AlexNet, ResNet
"CSI NN"	CPU (Arm)	EM, Power	N	✓	✓‡	◯	4-layer, 7-layer CNN
Takatoi, et al.	Micro-controller (RISC-based)	EM	N	✓††	×	×	Learn activation function
Alam, et al.	CPU (Xeon)	HPC	N	×	×	✓△	CNN
Duddu, et al.	CPU (Xeon)	Timing	Y	✓†	×	×	ResNet, VGG
"Cache Telepathy", DeepRecon	CPU (Xeon)	Cache	Y	✓	×	×	ResNet, VGG
"GANRED"	CPU (Intel i7)	Cache	Y	✓	×	×	AlexNet, VGG
Cheng, et al.	CPU (ARM)	Floating Point Unit	Y	✓	✓	×	4-layer DNN

Simple Timing Attacks and Floating Point Attacks on ML

- Vasisht et al. use simple execution timing which can reveal information about ML architectures
  - Can reveal some sensitive information such as number of layers



Timing correlates to number of layers

Duddu, Vasisht, et al. "Stealing neural networks via timing side channels." *arXiv preprint arXiv:1812.11720* (2018).

- Cheng et al. use floating-point timing side channel to learn Machine Learning network's parameters (weights and biases)
  - Target IEEE-754 single-precision
  - Target subnormal floating-point numbers which are less frequent
  - Subnormal operations are executed in software

IEEE SINGLE PRECISION FLOATING POINT NUMBER TYPES

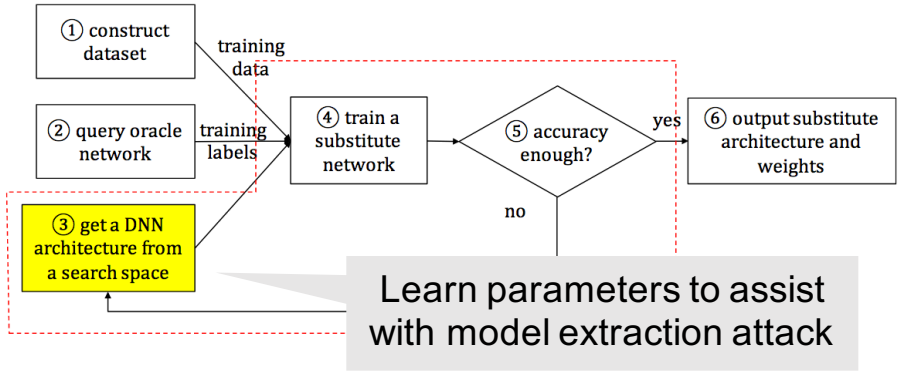
Type	Exponent	Mantissa	Formula
normal	[1, 254]	any	$(-1)^S \cdot 2^{E-127} \cdot 1.M$
zero	zero	zero	0
subnormal	zero	non-zero	$(-1)^S \cdot 2^{E-126} \cdot 0.M$
Infinities	255	zero	Inf
Not-a-Number	255	non-zero	NaN

Emulated in software = slower execution

Gongye, Cheng, Yunsi Fei, and Thomas Wahl. "Reverse-engineering deep neural networks using floating-point timing side-channels." 2020 57th ACM/IEEE Design Automation Conference (DAC). IEEE, 2020.

From Timing and Cache Attacks on ML

- Yan et al. observe that DNN architecture parameters determine the number of GEMM (Generalized Matrix Multiply) calls and the dimensions of the matrices
  - Cache side channels can reveal GEMM calls
  - Reduce the search space of DNN model architectures by learning GEMM usage
  - Use Flush+Reload and Prime+Probe attacks



Yan, Mengjia, et al. "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures." *29th USENIX Security Symposium (USENIX Security 20)*. 2020.

- Hong et al. recover the DNN structure by monitoring function calls through a Flush+Reload cache channel.

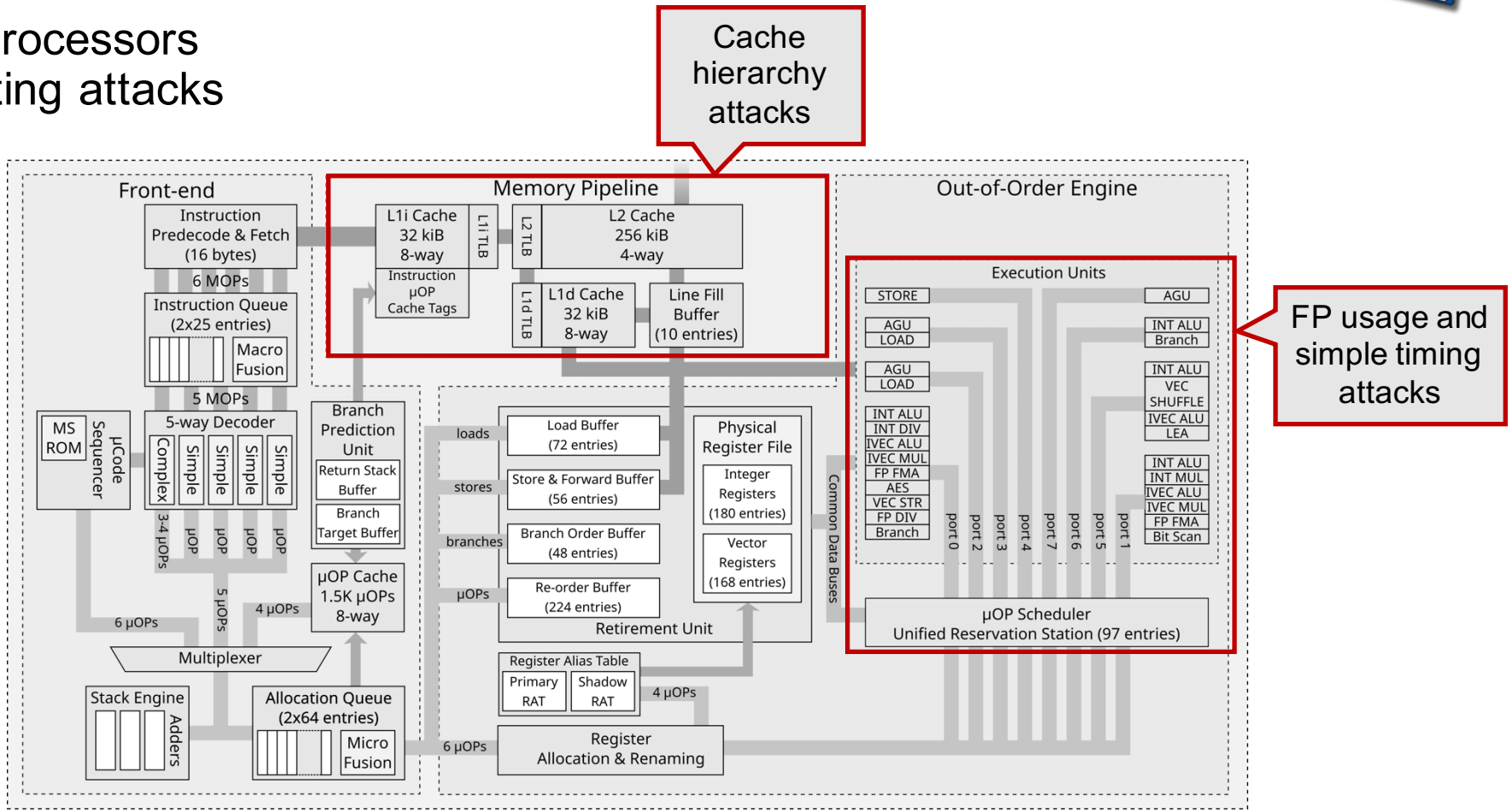
Hong, Sanghyun, et al. "Security analysis of deep neural networks operating in the presence of cache side-channel attacks." arXiv preprint arXiv:1810.03487 (2018).

- Liu et al. present a generative adversarial network (GAN)-based Reverse Engineering attack on DNNs
  - Use discriminator compares the cache side-channel information of victim vs. attacker DNN

Liu, Yuntao, and Ankur Srivastava. "GANRED: GAN-based Reverse Engineering of DNNs via Cache Side-Channel." Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop. 2020.

Microarchitecture and Attacks on ML

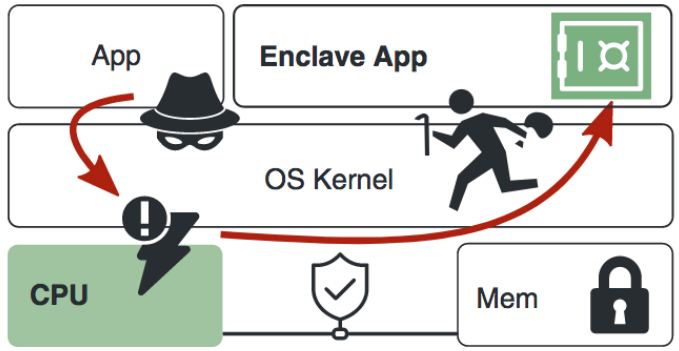
- Performance features of processors have been abused in existing attacks
- Area and power savings of simpler FP lead to attacks
- Power and performance savings of caches lead to attacks
- Many more attack targets when considering ML algorithms



Other Attacks: Power and Energy based Attacks

Abusing DVFS Interfaces to Inject Faults

- Plundervolt, by Murdock et al., bypasses Intel SGX's memory encryption engine protection boundaries by abusing an undocumented voltage scaling interface to induce predictable faults
  - Must be run with administrative privileges



Undervolting causes faults in computations

Murdock, Kit, et al. "Plundervolt: How a little bit of undervolting can create a lot of trouble." IEEE S&P 18.5 (2020): 28-37.

- VoltJockey, by Qiu et al., manipulates voltages in Arm processors to generate faults in TrustZone
  - Requires access to DVFS interface

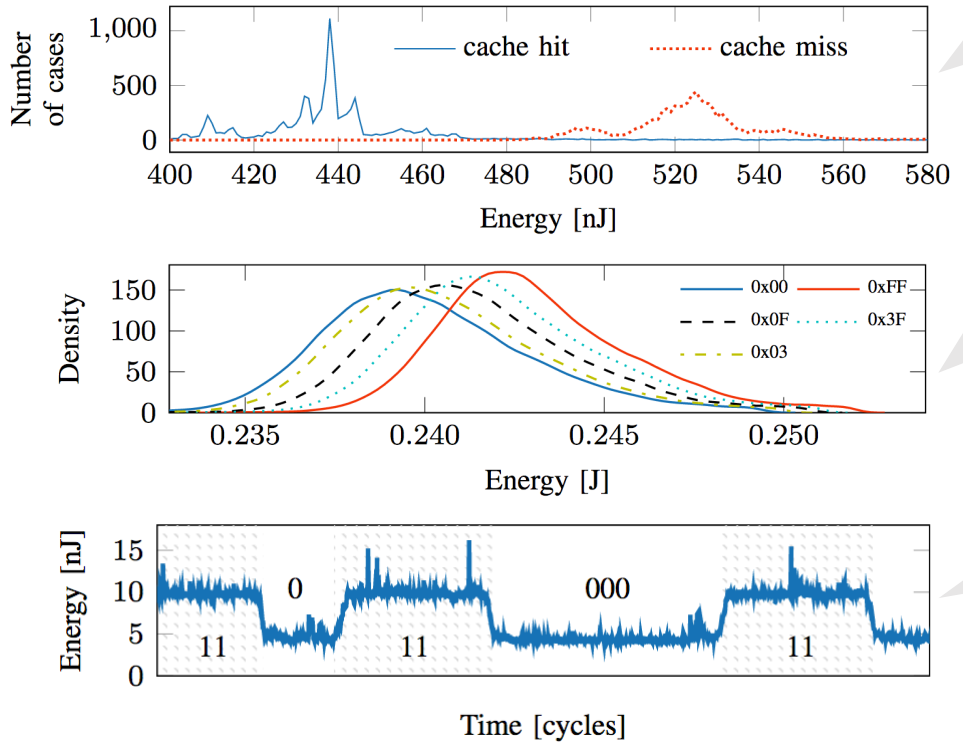
Qiu, Pengfei, et al. "VoltJockey: Breaching TrustZone by software-controlled voltage manipulation over multi-core frequencies." Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019.

- CLKscrew, by Tang et al., manipulates frequency settings in Arm to generate faults as well
  - Requires access to DVFS interface

Tang, Adrian, Simha Sethumadhavan, and Salvatore Stolfo. "CLKSCREW: exposing the perils of security-oblivious energy management." 26th USENIX Security Symposium (USENIX Security 17). 2017.

Abusing Power Perf Counters to Leak Secrets

- Platypus, by Lipp et al. is a software-based power side-channel attack that abuses Intel's RAPL (Running Average Power Limit) interface to gather power consumption traces
  - Unprivileged attack, but fixed with new software restrictions in `powercap` framework
  - Privileged attack still possible
- RAPL interface has a bandwidth of 20 kHz, but still good enough to measure different processor activities
  - Cache activity
  - Different instructions
  - Create covert channel
  - Extract secrets from Intel's SGX



RAPL measurement to distinguish cache hit vs miss

`imul` instructions with different operands Hamming weight

Covert channel example

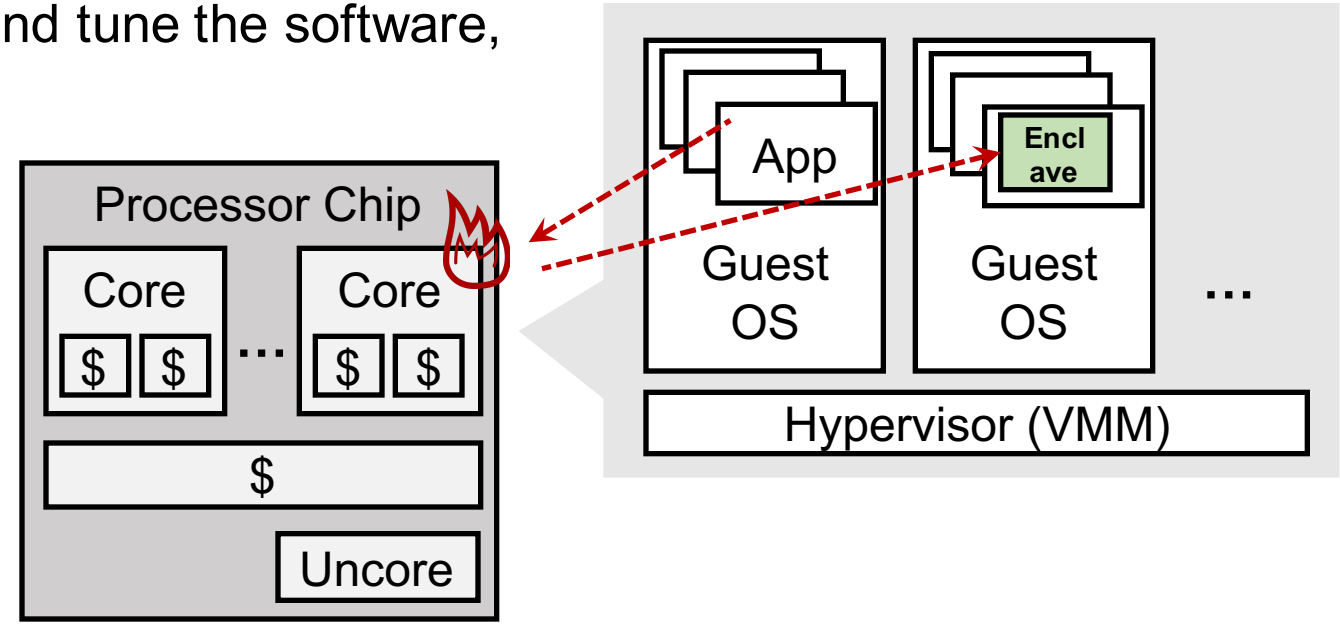
Lipp, Moritz, et al. "PLATYPUS: Software-based Power Side-Channel Attacks on x86." IEEE Symposium on Security and Privacy (SP). 2021.

User-level Control vs. Security

- Giving users access to DVFS interfaces or power and energy performance counters can help measure and tune the software, but introduces new security issues

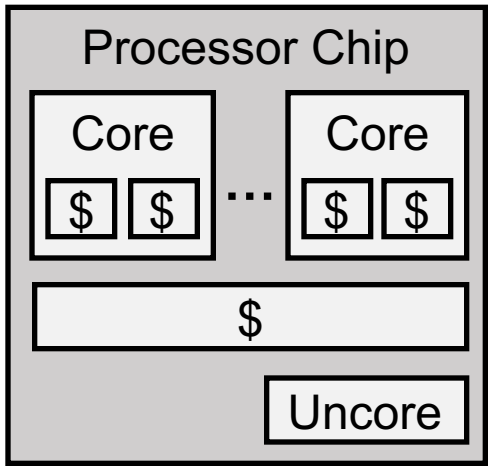
- DVFS interfaces need to be designed to prevent users abusing them
  - User-level attacks
  - Privileged-level attacks, e.g. on SGX

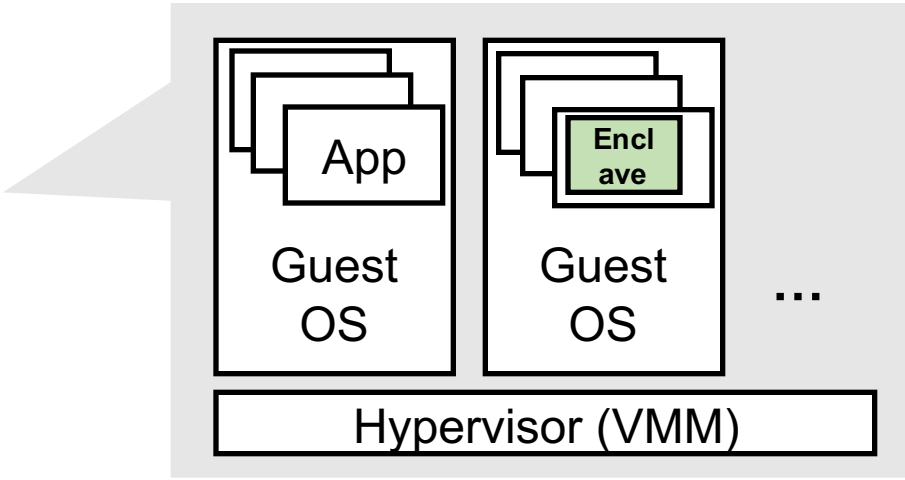
- Performance counters for power and energy can leak secrets as well
  - User-level attacks
  - Privileged-level attacks, e.g. on SGX as well



Providing Protections with a Trusted Processor Chip

Key to most secure processor architecture designs is the idea of **trusted processor chip** as the security wherein the protections are provided.





Hardware Defenses for Side and Covert Channels

Timing Channels due to Memory Hierarchy

- **Caches** – state of the cache affects timing of operations: cache hit, cache miss, flush timing
- **Cache Replacement Logic** – LRU states can be abused for a timing channel, especially cache hits modify the LRU state, no misses are required
- **TLBs** – Translation Look-aside Buffers are types of caches with similar vulnerabilities
- **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 recent Micro-architectural Data Sampling attacks
- **Coherence Bus and Coherence State** – different coherence state of a cache line may affect timing, such as flushing or upgrading state
- **Memory Controller and Interconnect** – memory and interconnect are shared resources vulnerable to contention channels

Motivation for Design of Hardware Secure Caches

- Software defenses are possible (e.g. page coloring or “constant time” software)
  - But require software writers to consider timing attacks, and to consider 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

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

- Numerous academic proposals have presented different secure cache architectures that aim to defend against different cache-based side channels.
- To-date there are approximately 18 secure cache proposals
- They share many similar, key techniques

Secure Cache 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

Goal of all secure caches is to minimize interference between victim and attacker or within victim themselves

Different Types of Interference Between Cache Accesses

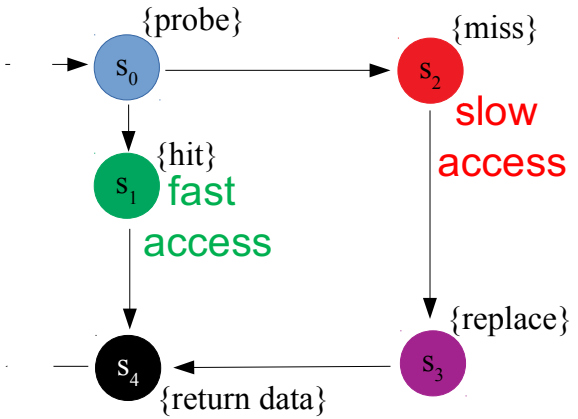
Deng, S., Xiong, W., Szefer, J., "Analysis of Secure Caches and Timing-Based Side-Channel Attacks", 2019

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

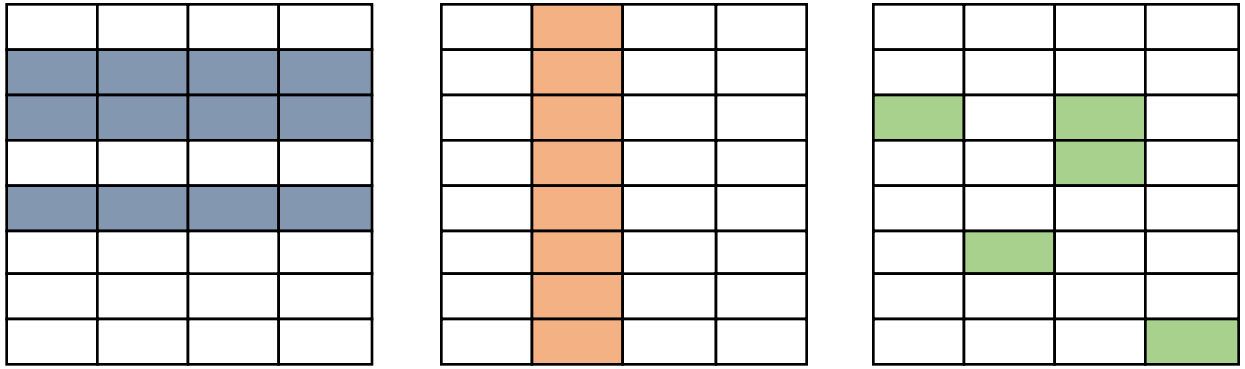
Memory reuse conditions

- 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)



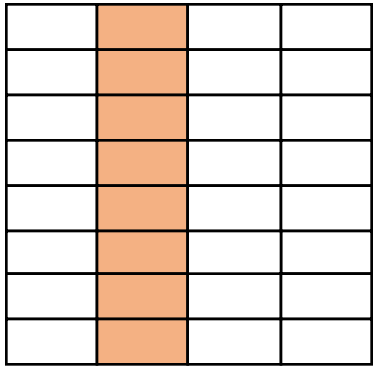
Partitioning

- **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



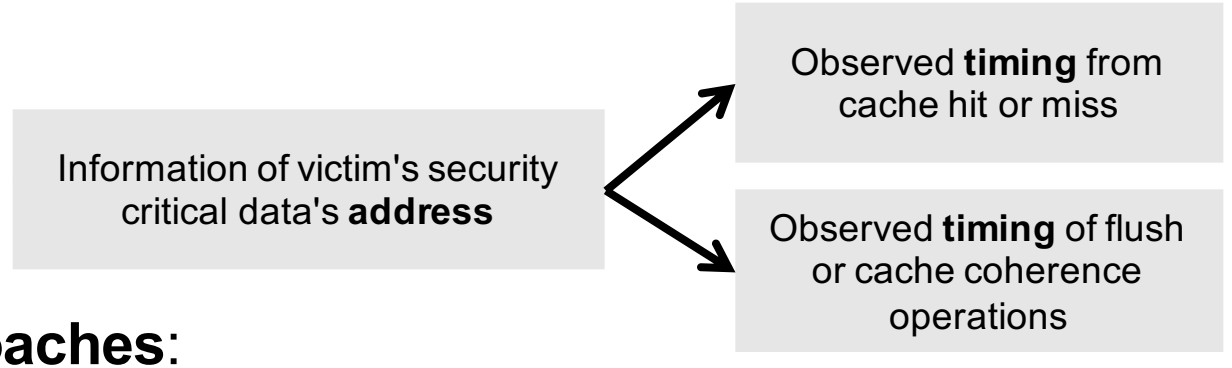
Partitioning (cont.)

- **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
  - Software partitioning
    - E.g. page-coloring



Randomization

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

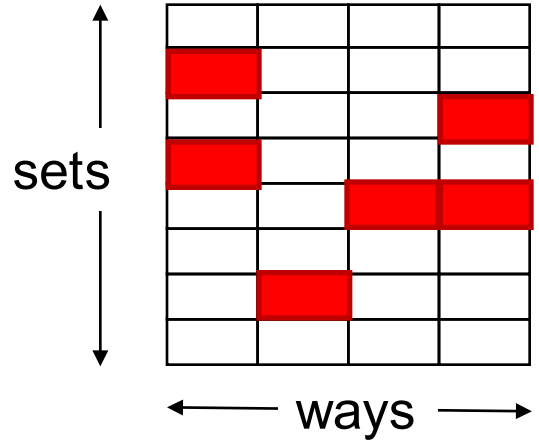


- **Randomization approaches**:
  - Randomize the address to cache set mapping
  - Random fill
  - Random eviction
  - Random delay
- **Goal**: reduce the mutual information from the observed timing to 0
- **Some limitations:** Requires a fast and secure random number generator, ability to predict the 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 label a certain range of the data of victim which they think are 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**

Set-associative cache



Secure Caches

Deng, Shuwen, Xiong, Wenjie, Szefer, Jakub, "Analysis of Secure Caches and Timing-Based Side-Channel Attacks", 2019

At least 18 different secure caches exist in literature, which use one or more of the below techniques to provide the enhanced security:

- **Partitioning-based caches**
  - Static Partition cache, SecVerilog cache, SecDCP cache, Non-Monopolizable (NoMo) cache, SHARP cache, Sanctum cache, MI6 cache, Invisispec cache, CATalyst cache, DAWG cache, RIC cache, Partition Locked cache
- **Randomization-based caches**
  - SHARP cache, Random Permutation cache, Newcache, Random Fill cache, CEASER cache, SCATTER cache, Non-deterministic cache
- **Differentiating sensitive data**
  - CATalyst cache, Partition Locked cache, Random Permutation cache, Newcache, Random Fill cache, CEASER cache, Non-deterministic cache

Secure Caches vs. Attacks

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

Effectiveness of the secure caches:

	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

CATalyst uses number of assumptions, such as pre-loading

Speculation-Related Secure Caches vs. Attacks

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

Effectiveness of the secure caches:

	MI6 cache		InivisiSpec cache		DAWG cache	
	Normal	Speculative	Normal	Speculative	Normal	Speculative
external miss-based attacks	✓	✓	X	✓	✓	✓
internal miss-based attacks	X	✓	X	✓	X	X
external hit-based attacks	✓	✓	X	✓	✓	✓
internal hit-based attacks	X	✓	X	✓	X	X

Secure Cache Performance

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

	SP*	SecVerilog	SecDCP	NoMo	SHARP	Sanctum	MI6	InvisiSpec	CATalyst	DAWG	RIC	PL	RP	Newcache	Random Fill	CEASER	SCATTER	Non Det.
Perf.	1%	-	12.5% better over SP cache	1.2% avr., 5% worst	3%-4%	-	-	reduce slowdown of Spectre from 74% to 21%	average slowdown of 0.7% for SPEC and 0.5% for PARSEC	L1 and L2 most 4%-7%	improves 10%	12%	0.3%, 1.2% worst	within the 10% range of the real miss rate	3.5%, 9% if setting the window size to be largest	1% for performance optimization	3.5% for perfor-manc e opti-miza-tion	7% with simple benchmarks
Pwr.	-	-	-	-	-	-	-	L1 0.56 mW, LLC 0.61 mW	-	-	-	-	average 1.5nj	<5% power	-	-		-
Area	-	-	-	-	-	-	-	L1-SB LLC-SB Area (mm2) 0.0174 0.0176	-	-	0.176%	-	-	-	-	-		-

Outline

- Information Leaks in Processors
  - Side and Covert Channels in Processors
  - Side Channel Attacks on ML Algorithms
  - Other Attacks: Power and Energy based Attacks
  - Hardware Defense for Side and Covert Channels

- **Transient Execution Attacks in Processors**
  - Transient Execution and Attacks on Processors
  - Hardware Defenses for Transient Execution Attacks

- **Design of Trusted Execution Environments**

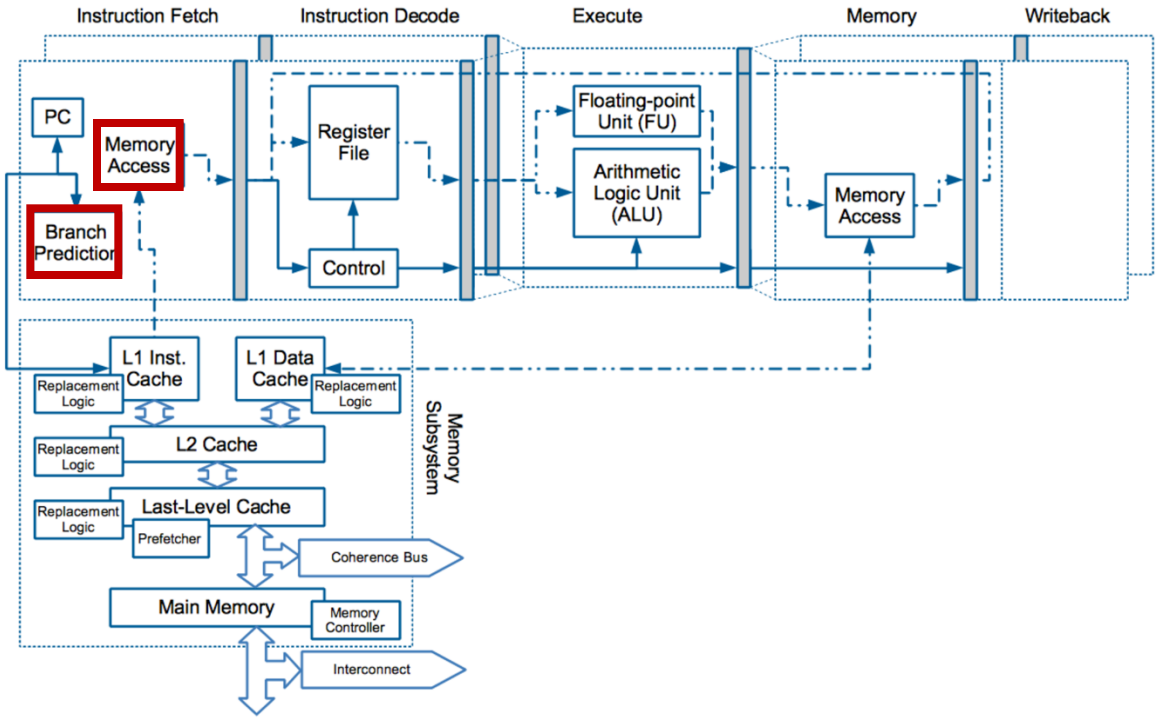
- **Wrap-up and Conclusion**

Transient Execution and Attacks on Processors

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)



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

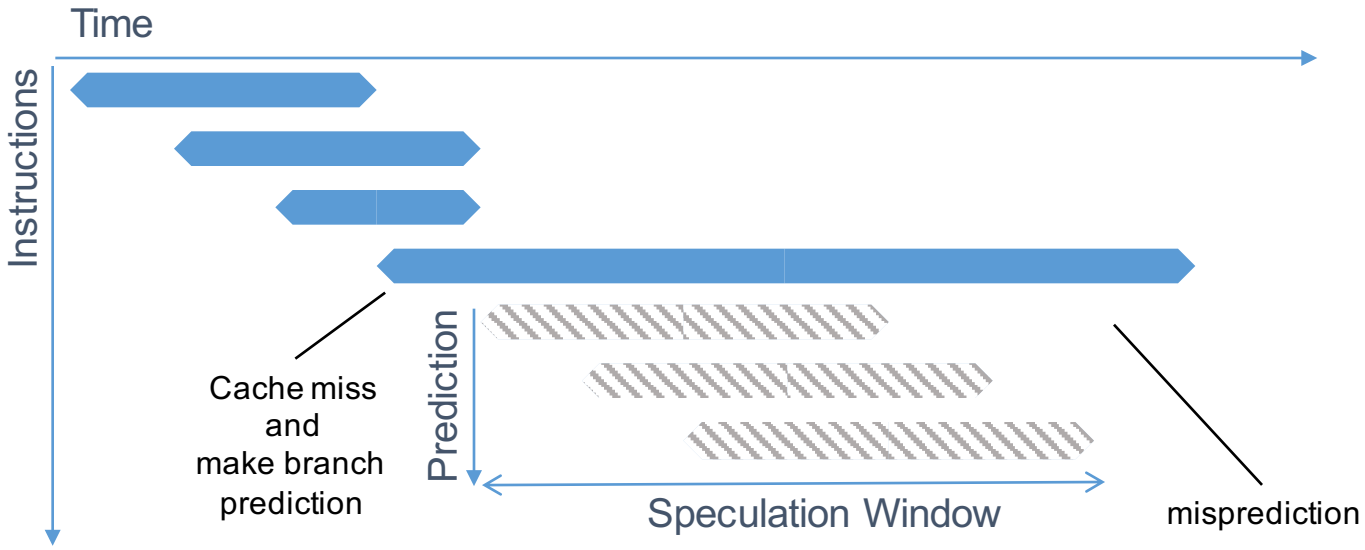
Transient Execution Attacks = Transient Execution + Covert or Side Channel

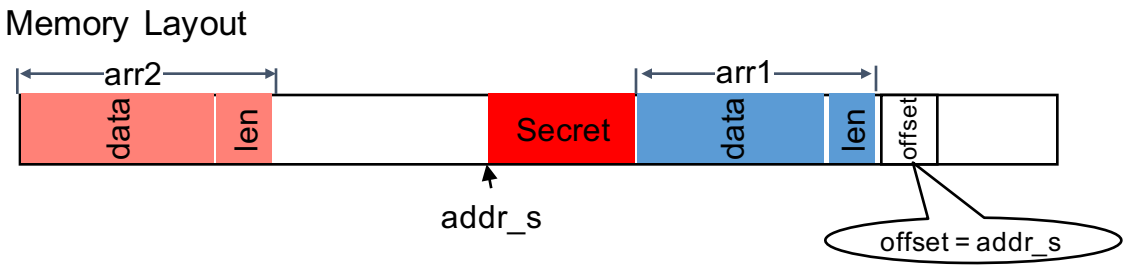
Example: Spectre Bounds Check Bypass Attack

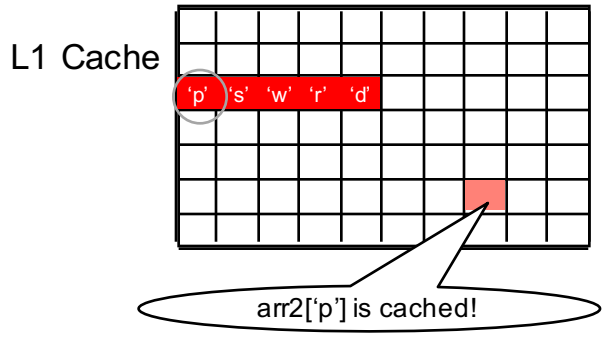
Example of Spectre variant 1 attack:

Victim code:

```
struct array *arr1 = ...;
struct array *arr2 = ...;                    Probe array (side channel)
unsigned long offset = ...;                  Controlled by the attacker
if (offset < arr1->len) {                    arr1->len is not in cache
  unsigned char value = arr1->data[offset];
  unsigned long index = value;
  unsigned char value2 = arr2->data[index * size];
...                                          change the cache state
```







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

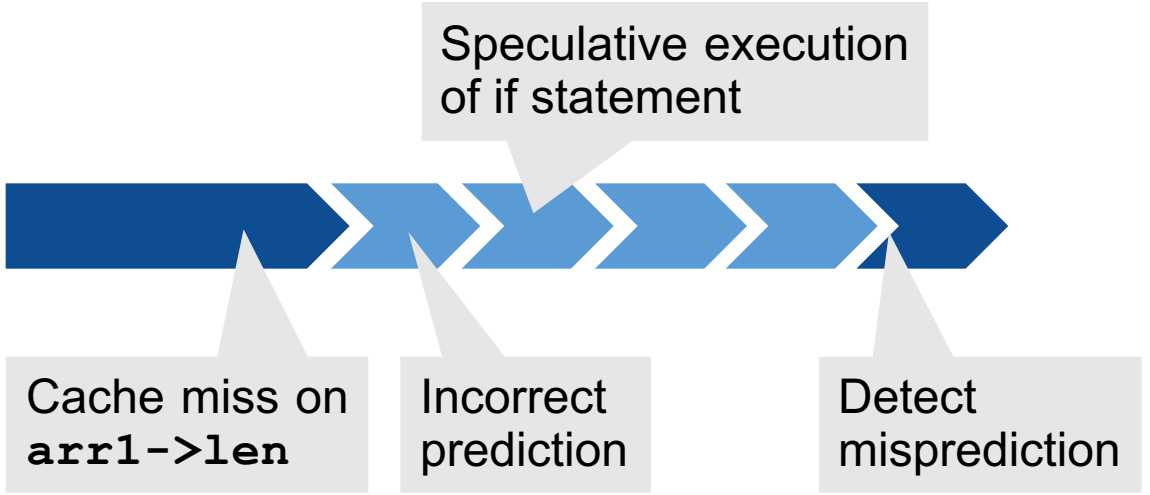
Transient Execution – due to Prediction

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 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];
  ...
}
```



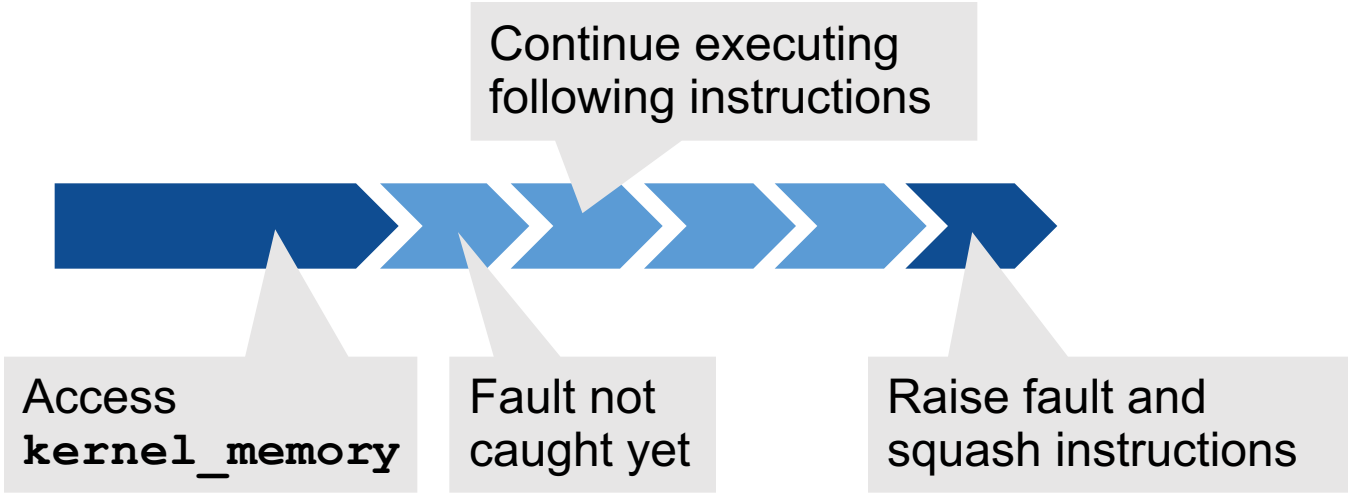
Transient Execution – due to Faults

***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 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];
...
```



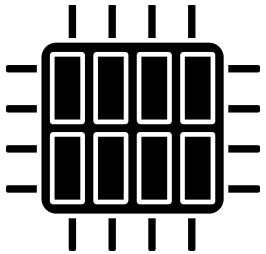
Covert Channels Usable for Transient Exec. Attacks

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
- …



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)
- AES or SHA instructions
- …

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)
- Spectre SWAPGS

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

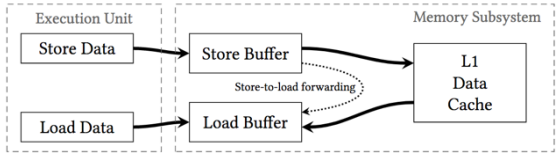
SGXPectre is also Spectre Variant 1 or 2 where code outside SGX Enclave can influence the branch behavior

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

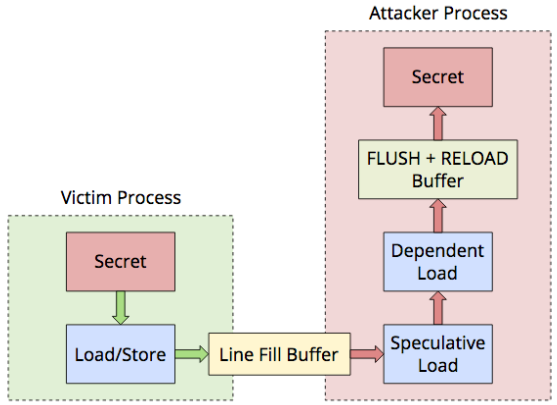


- **RIDL (Rogue In-Flight Data Load)** and **ZombieLoad** – *Line-Fill Buffers* and *Load Ports*

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/surpervisor
P:	present
R/W:	read/write
RSVD:	reserved bit
XD:	execute disable
PK:	memory-protection keys (PKU)

```
(rcx = address that leads 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:

windowing gadget’s latency > disclosure gadget’s trigger latency

Disclosure Gadget

3. 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]
        ...
```

More Disclosure Gadgets – SWAPGS

Bitdefender. "Bypassing KPTI Using the Speculative Behavior of the SWAPGS Instruction", Aug. 2019.

- Most recent disclosure gadget presented by researchers is the **`SWAPGS`** instruction on 64-bit Intel processors
- **`SWAPGS`** instruction
  - Kernel-level instruction, swap contents of IA32_GS_BASE with IA32_KERNEL_GS_BASE
  - GS points to per CPU data structures (user or kernel), IA32_GS_BASE can be updated by user-mode **`WRGSBASE`** instruction
- Disclosure gadgets with **`SWAPGS`** instruction
  - Scenario 1: SWAPGS not getting speculatively executed when it should
  - Scenario 2: SWAPGS getting speculatively executed when it shouldn't

```
1.  test byte ptr [nt!KiKvaShadow],1
2.  jne   skip_swapgs [4]
3.  swapgs
4.  mov   r10,qword ptr gs:[188h]
5.  mov   rcx,qword ptr gs:[188h]
6.  mov   rcx,qword ptr [rcx+220h]
7.  mov   rcx,qword ptr [rcx+830h]
8.  mov   qword ptr gs:[270h],rcx
```

Later use cache-based timing channel to lean information

Disclosure Primitives

4. Disclosure Primitive

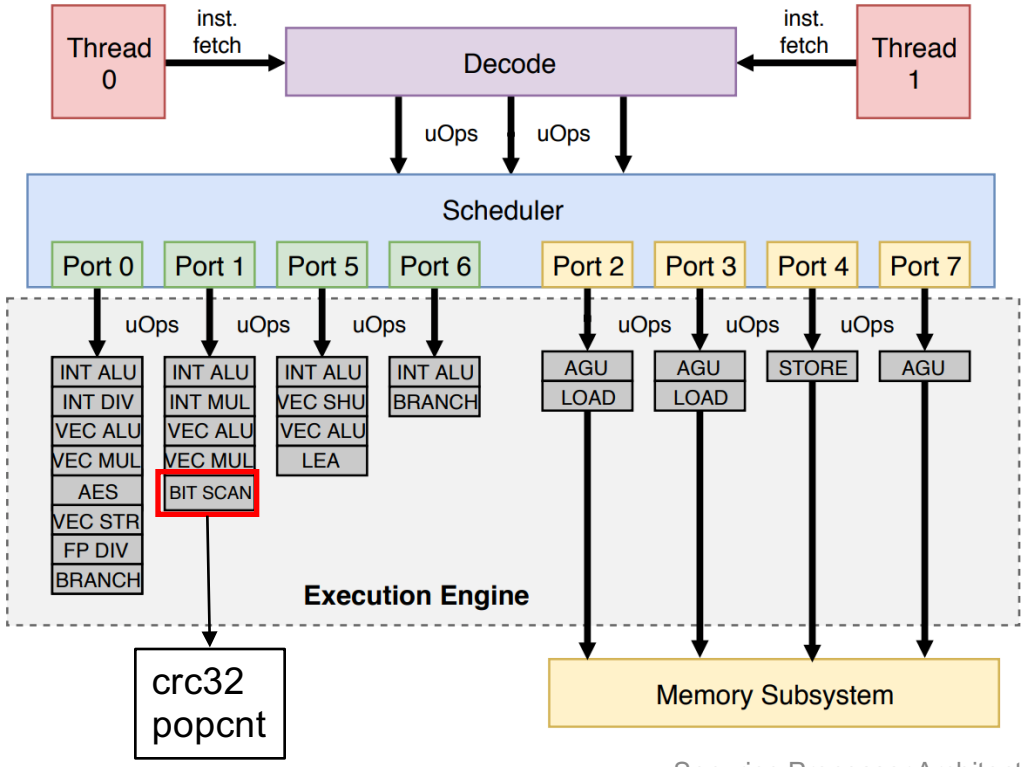
Two types of disclosure primitives:

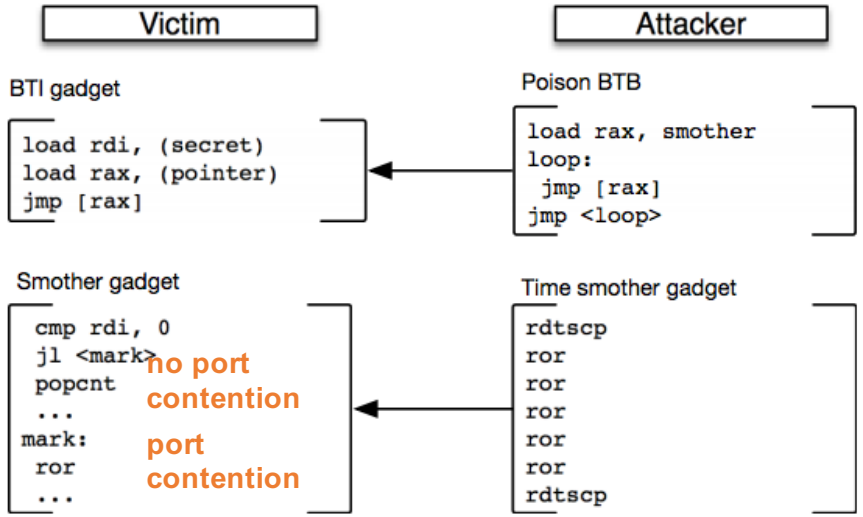
- **Short-lived** or **contention-based** (hyper-threading / multi-core scenario):
  1. Share resource on the fly (e.g., bus, port, cache bank)
  2. State change within speculative window (e.g., speculative buffer)
- **Long-lived channel**:
  - Change the state of micro-architecture
  - The change remains 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, 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





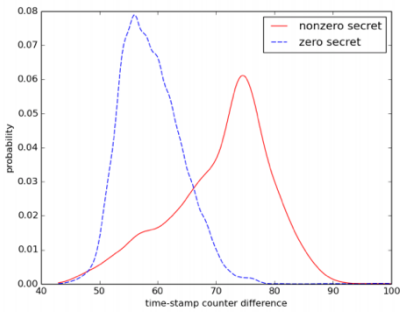


Fig. Probability density function for the timing of an attacker measuring ***crc32*** operations when running concurrently with a victim process that speculatively executes a branch which is conditional to the (secret) value of a register being zero.

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;
        }
}
```

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

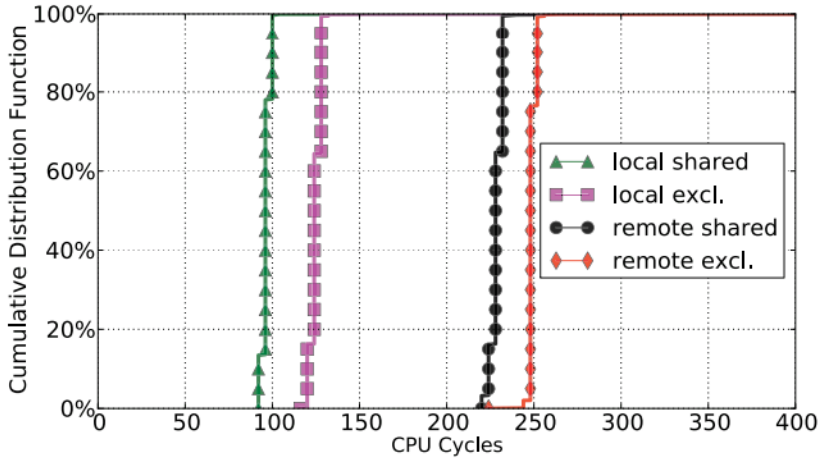


Fig. 2: Load operation latency in various (location, coherence state) combinations.

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

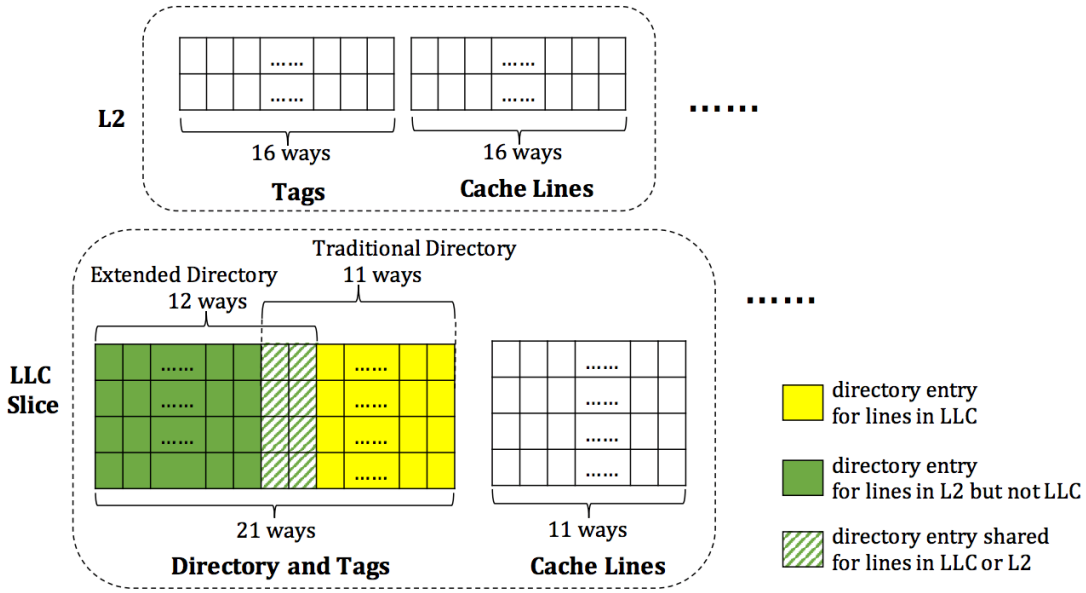


Fig. 9. Reverse engineered directory structure.

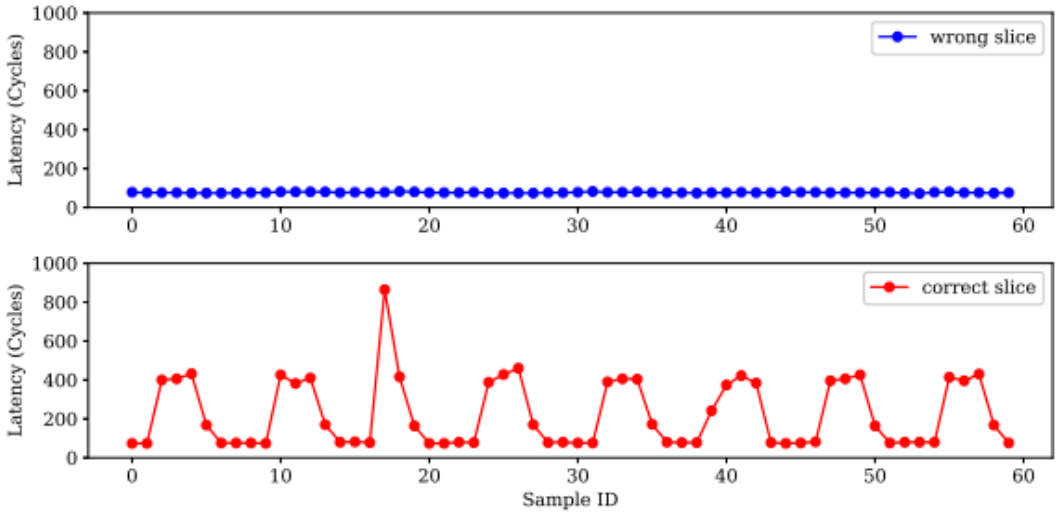


Fig. 13. The upper plot shows receiver's access latencies on a slice not being used for the covert channel, while the lower one shows the one used in the covert channel. Sender transmits sequence "101010...".

E.g. accessing LLC data creates directory entries, which may evict L2 entries (in the shared portion)

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

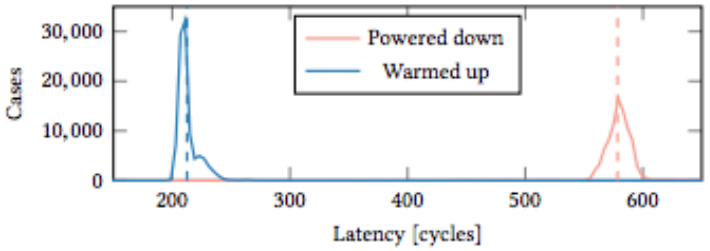


Figure 5: Differences in the execution time for AVX2 instructions (Intel i5-6200U). If the AVX2 unit is inactive (powered down), executing a 256-bit instruction takes on average 366 cycles longer than on an active AVX2 unit. The average values are shown as dashed vertical lines.

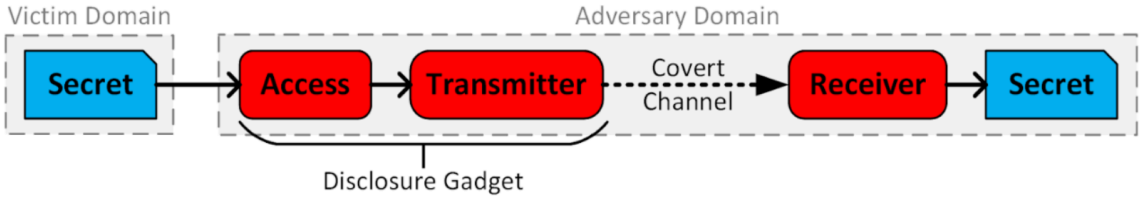
Gadget:

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

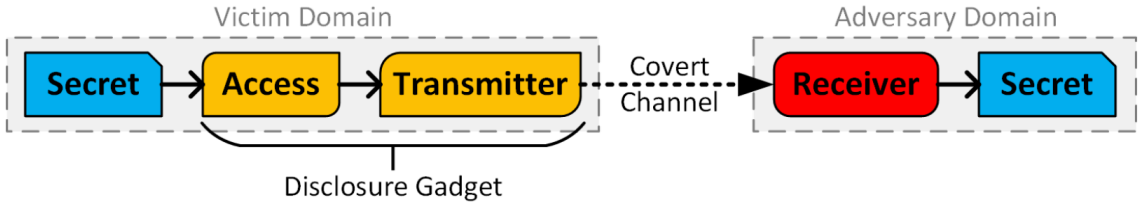
Alternative Speculative Execution Terminology from Intel

Source: https://software.intel.com/security-software-guidance/best-practices/refined-speculative-execution-terminology

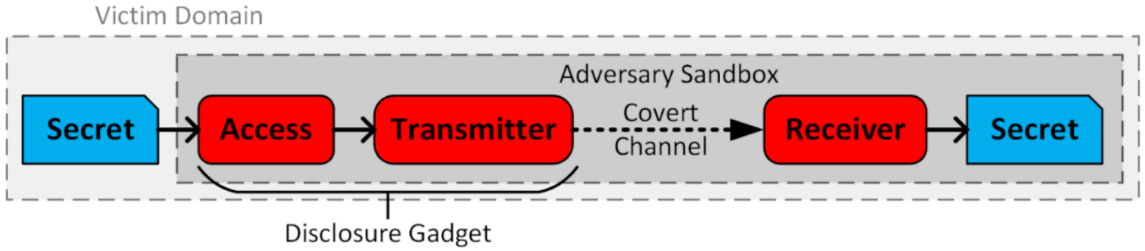
- **Domain-Bypass Attacks**
  - Meltdown, MDS, and newer MDS related attacks



- **Cross-Domain Attacks**
  - Spectre, LVI, and SWAPGS



- **In-Domain Attacks**
  - Spectre and LVI

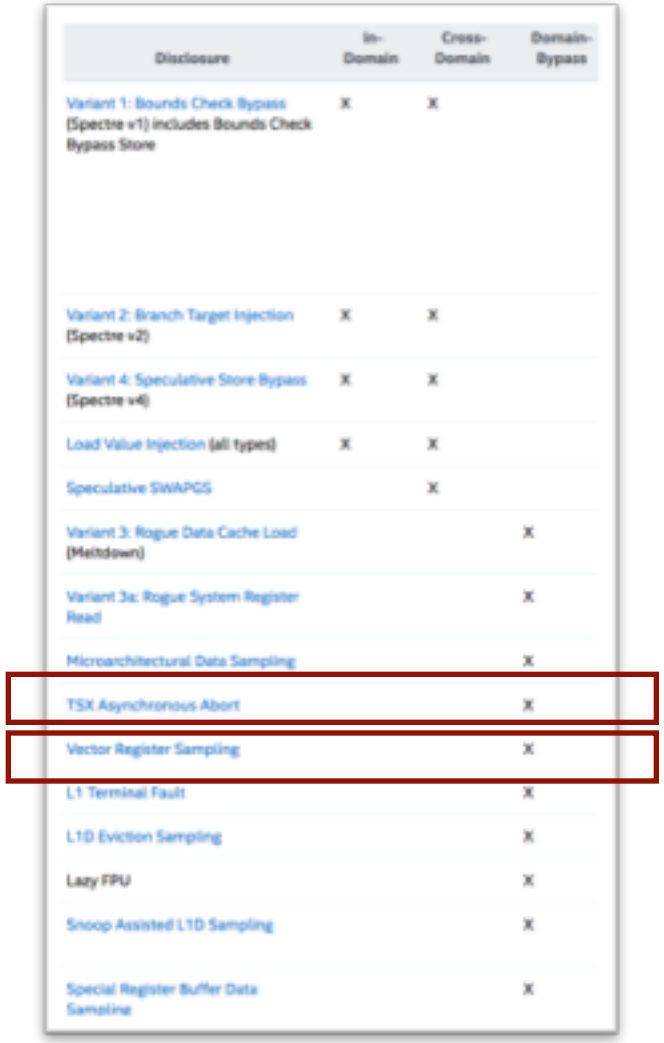


Recent Attacks (MDS Related)

Intel Best Practices for Security
https://software.intel.com/security-software-guidance/best-practices

- **TSX Asynchronous Abort**
  - In Intel Transactional Synchronization Extensions (TSX) an asynchronous abort takes place when a different thread accesses a cache line that is also used within the transactional region
  - Immediately after an uncompleted asynchronous abort, certain speculatively executed loads may read data from internal buffers and pass it to dependent operations. This can be then used to infer the value via a cache side channel attack.

- **Vector Register Sampling**
  - Partial data values previously read from a vector register on a physical core may be propagated into portions of the store buffer
  - Malicious software may be able to use vector register sampling to infer vector data used by previously run software, or to infer vector data used by software running on a sibling hyperthread



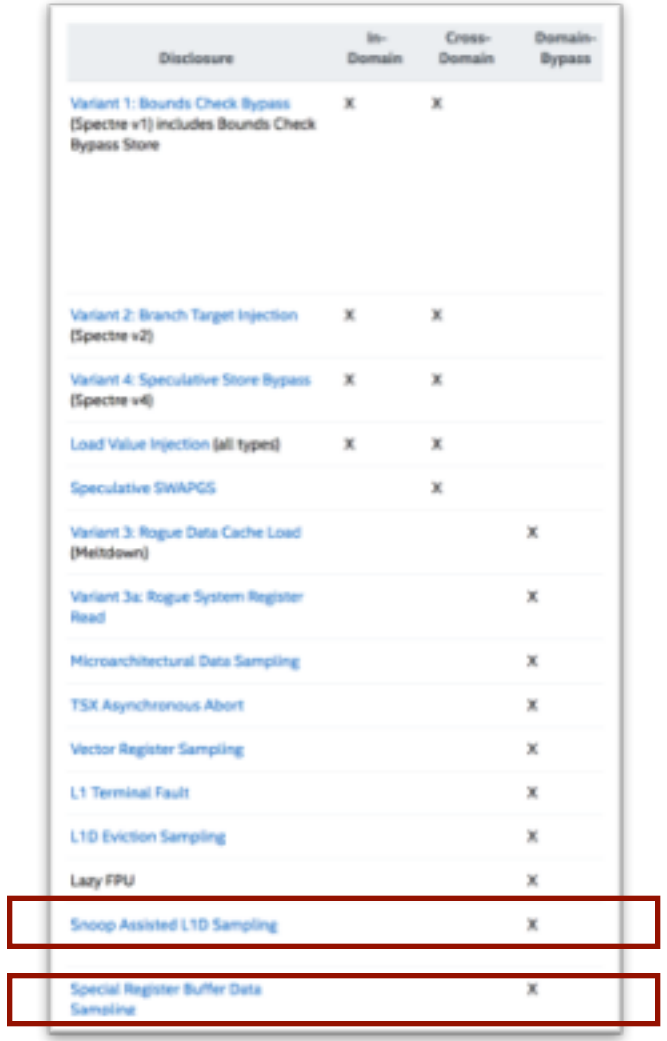
Disclosure	In-Domain	Cross-Domain	Domain-Bypass
Variant 1: Bounds Check Bypass (Spectre v1) includes Bounds Check Bypass Store	X	X	
Variant 2: Branch Target Injection (Spectre v2)	X	X	
Variant 4: Speculative Store Bypass (Spectre v4)	X	X	
Load Value Injection (all types)	X	X	
Speculative SWAPGS		X	
Variant 3: Rogue Data Cache Load (Meltdown)			X
Variant 3a: Rogue System Register Read			X
Microarchitectural Data Sampling			X
TSX Asynchronous Abort			X
Vector Register Sampling			X
L1 Terminal Fault			X
L1D Eviction Sampling			X
Lazy FPU			X
Snoop Assisted L1D Sampling			X
Special Register Buffer Data Sampling			X

Recent Attacks (MDS Related)

- **Snoop Assisted L1D Sampling**
  - Data in a modified cache line that is being returned in response to a snoop may also be forwarded to a faulting load operation to a different address that occurs simultaneously
  - Malicious adversary may infer modified data in the L1D cache

- **Special Register Buffer Data Sampling (SRBDS)**
  - When RDRAND, RDSEED and EGETKEY instructions are executed, the data is moved to the core through the special register mechanism that is susceptible to MDS attacks

Intel Best Practices for Security
https://software.intel.com/security-software-guidance/best-practices



Disclosure	In-Domain	Cross-Domain	Domain-Bypass
Variant 1: Bounds Check Bypass (Spectre v1) includes Bounds Check Bypass Store	X	X	
Variant 2: Branch Target Injection (Spectre v2)	X	X	
Variant 4: Speculative Store Bypass (Spectre v4)	X	X	
Load Value Injection (all types)	X	X	
Speculative SWAPGS		X	
Variant 3: Rogue Data Cache Load (Meltdown)			X
Variant 3a: Rogue System Register Read			X
Microarchitectural Data Sampling			X
TSX Asynchronous Abort			X
Vector Register Sampling			X
L1 Terminal Fault			X
L1D Eviction Sampling			X
Lazy FPU			X
Snoop Assisted L1D Sampling			X
Special Register Buffer Data Sampling			X

Hardware Defenses for Transient Execution Attacks

Serializing Instructions and Software Mitigations

- Serializing instructions can prevent speculative execution
  - **LFENCE** (**x86**) will stop younger instructions from executing, even speculatively, before older 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

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
   - Crate “secure” AVX, etc.

Mitigation Techniques for Attacks due to Faults and MDS

Transient Execution Attacks = Transient Execution + Covert or Side Channel

Due to Faults:

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

Due to MDS:

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

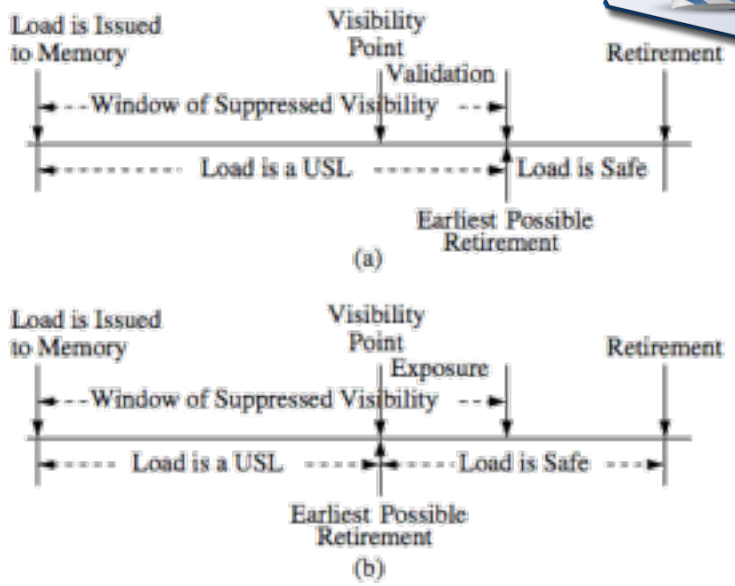
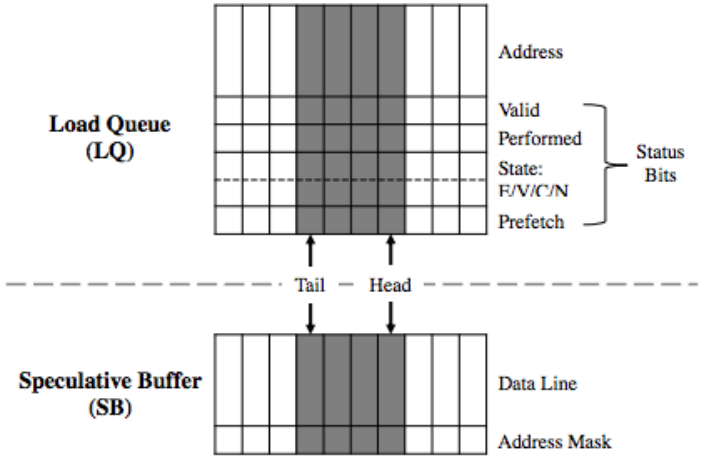


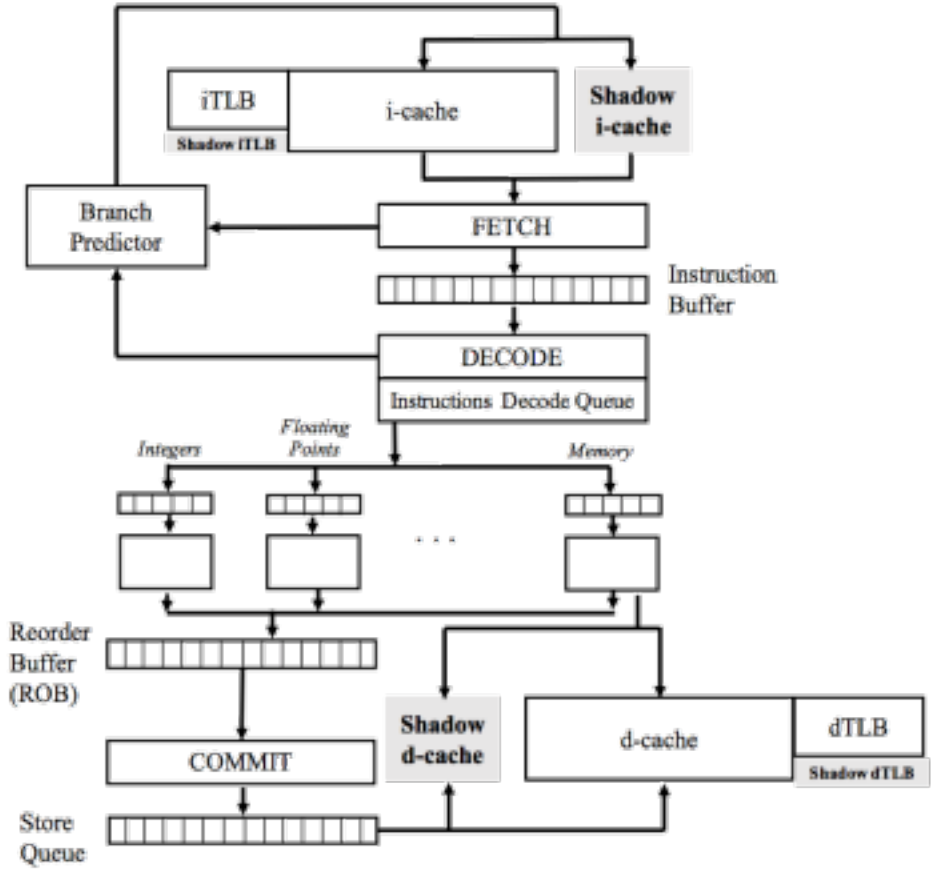
Fig. 2: Timeline of a USL with validation (a) and exposure (b).



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

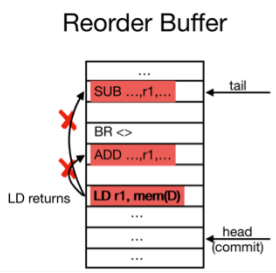


Mitigations in Micro-architecture: SpecShield

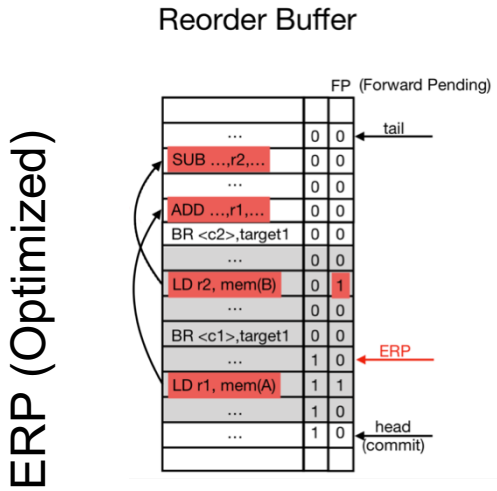
"WiP: Isolating Speculative Data to Prevent Transient Execution Attacks" Kristin Barber, et al., HASP 2019 Presentation

- 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* there are no branch miss-predictions or memory-access exceptions

Conservative



ERP (Optimized)



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

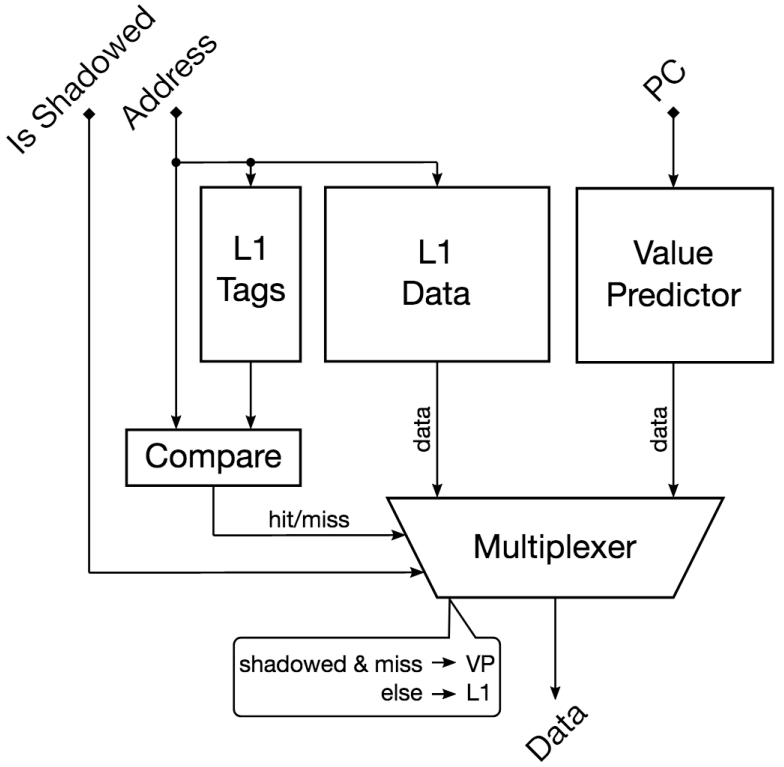
"Conditional Speculation: An Effective Approach to Safeguard Out-of-OrderExecution Against Spectre Attacks", Peinan Li et al., HPCA 2019

- 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 – have to be careful about 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

Mitigations in Micro-architecture: EISE

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

- Efficient Invisible Speculative Execution 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, similar to SpecShield and others
  b) allow only accesses that hit in the L1 data cache to proceed – but have to be careful about LRU channels
  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: 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

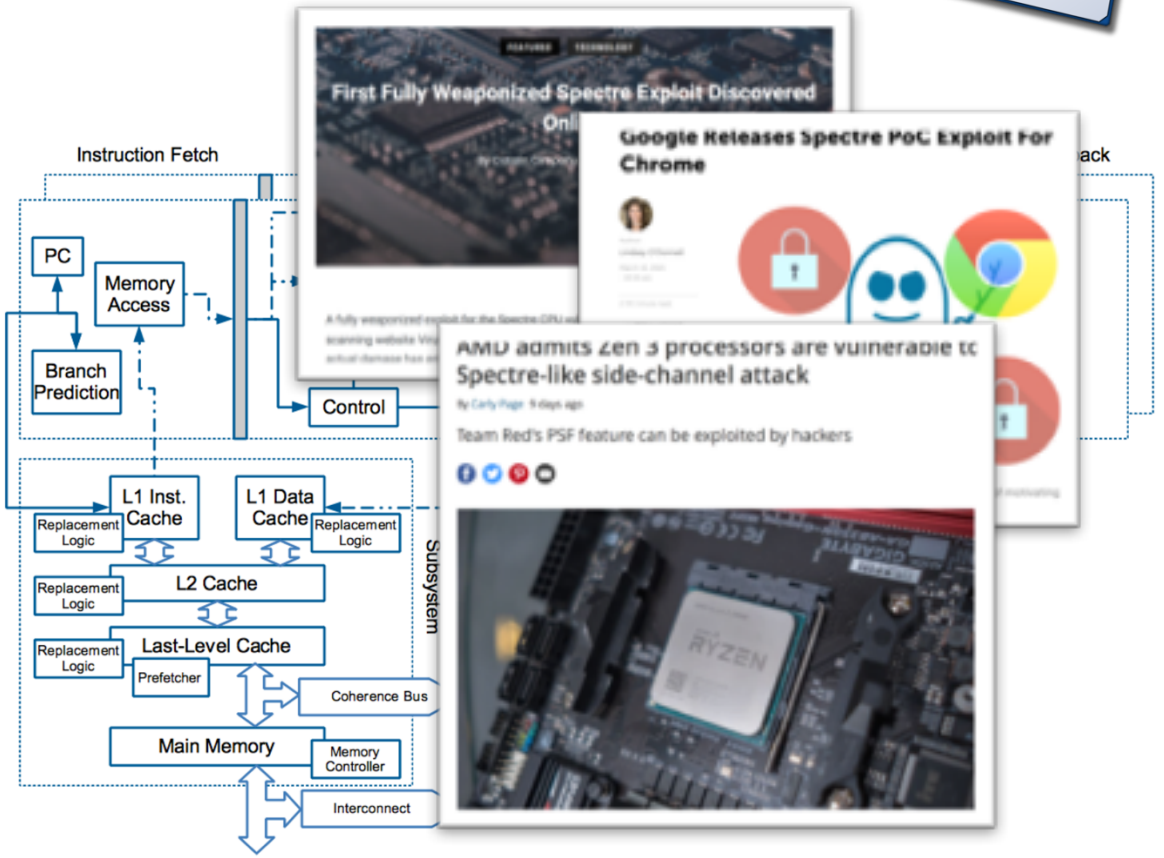
Industry Solutions and Mitigations

Hardware (x86)

- 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.

Microarchitectural Optimizations Still Pose Threats

- Threats keep evolving so research and defenses need to keep up
- “Hiden” microarchtiectural optimizations and operations expose new security threats
- “Security by obscurity” does not work
- There is need for new interface or contract between the hardware and software in regards to the microarchtiectural operation



End of First Part of the Tutorial

Jakub Szefer
Associate Professor
Dept. of Electrical Engineering
Yale University

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/