Design of Secure Processor Architectures

Jakub Szefer
Assistant Professor
Dept. of Electrical Engineering
Yale University

CHES 2019 – August 25, 2019

Slides and information available at: https://caslab.csl.yale.edu/tutorials/ches2019/

Outline

9:30 – 10:00 Secure Processor Architectures (30 min.)

- Secure Processor Architectures
- Memory Protections in Secure Processors
- Principles of Design of Secure Processors

10:10 – 11:20 Timing Channels: Attacks and Hardware Defenses (70 min.)

- Side and Covert Channels
- Timing Channels in Caches
- Timing Channels in Other Parts of Memory Hierarchy
- Secure Hardware Caches
- Secure Buffers, TLBs, and Directories

11:30 – 12:30 Transient Execution Attacks and Hardware Defenses (60 min.)

- Transient Execution Attacks
- Transient Attack Hardware Mitigation Techniques
- Transient Attacks and Secure Processors

Secure Processor Architectures

Secure Processor Architectures

Secure Processor Architectures extend a processor with hardware (and related software) features for protection of software

- Protected pieces of code and data are now commonly called Enclaves
  - But can be also Trusted Software Modules
- Focus on the main processor in the system
  - Others focus on co-processors, cryptographic accelerators, or security monitors
- Add more features to isolate secure software from other, untrusted software
  - Includes untrusted Operating System or Virtual Machines
  - Many also consider physical attacks on memory
- Isolation *should* cover all types of possible ways for information leaks
  - Architectural state
  - Micro-architectural state
  - Due to spatial or temporal sharing of hardware

Most recent threats, i.e. Spectre, etc.

Side and covert channel threats

Brief History of Secure Processor Architectures

Starting with a typical baseline processor, many secure architectures have been proposed

Starting in late 1990s or early 2000s, academics have shown increased interest in secure processor architectures:

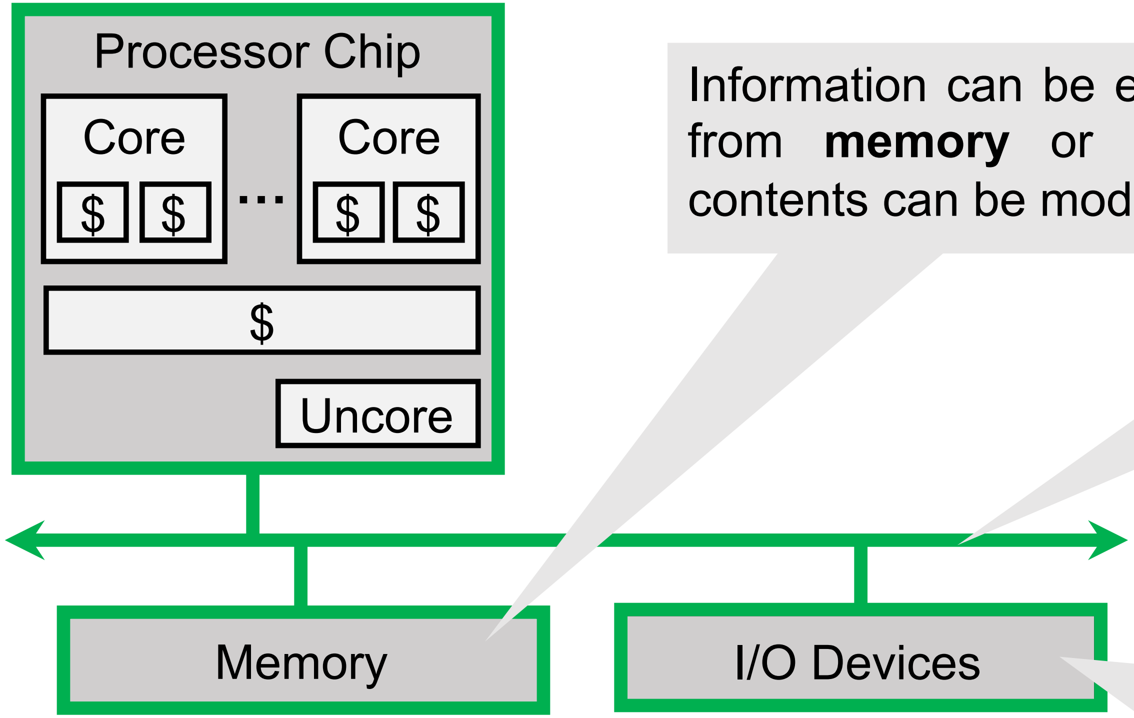
XOM (2000), AEGIS (2003), Secret-Protecting (2005), Bastion (2010),
NoHype (2010), HyperWall (2012), Phantom (2013), CHERI (2014), Sanctum (2016),
Keystone (about 2017), Ascend (2017), MI6 (2018)

Commercial processor architectures have also included security features:

LPAR in IBM mainframes (1970s), Security Processor Vault in Cell Broadband Engine (2000s),
ARM TrustZone (2000s), Intel TXT & TPM module (2000s), Intel SGX (mid 2010s),
AMD SEV (late 2010s)

Baseline (Unsecure) Processor Hardware

Typical computer system with no secure components nor secure processor architectures considers all the components as trusted:



Information can be extracted from **memory** or memory contents can be modified.

Snooping on the **system bus** is possible to extract information.

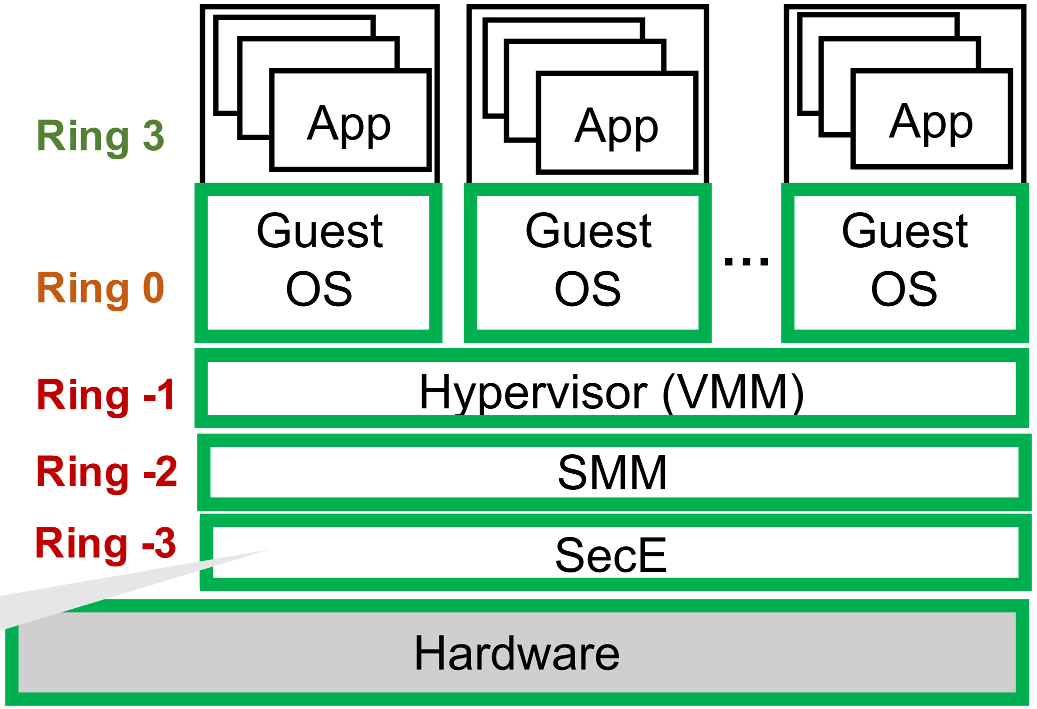
Compromised or malicious **devices** can attack other components of the system.

Baseline (Unsecure) Processor Software

The hardware is most privileged as it is the lowest level in the system.

- There is a linear relationship between protection ring and privilege (lower ring is more privileged)

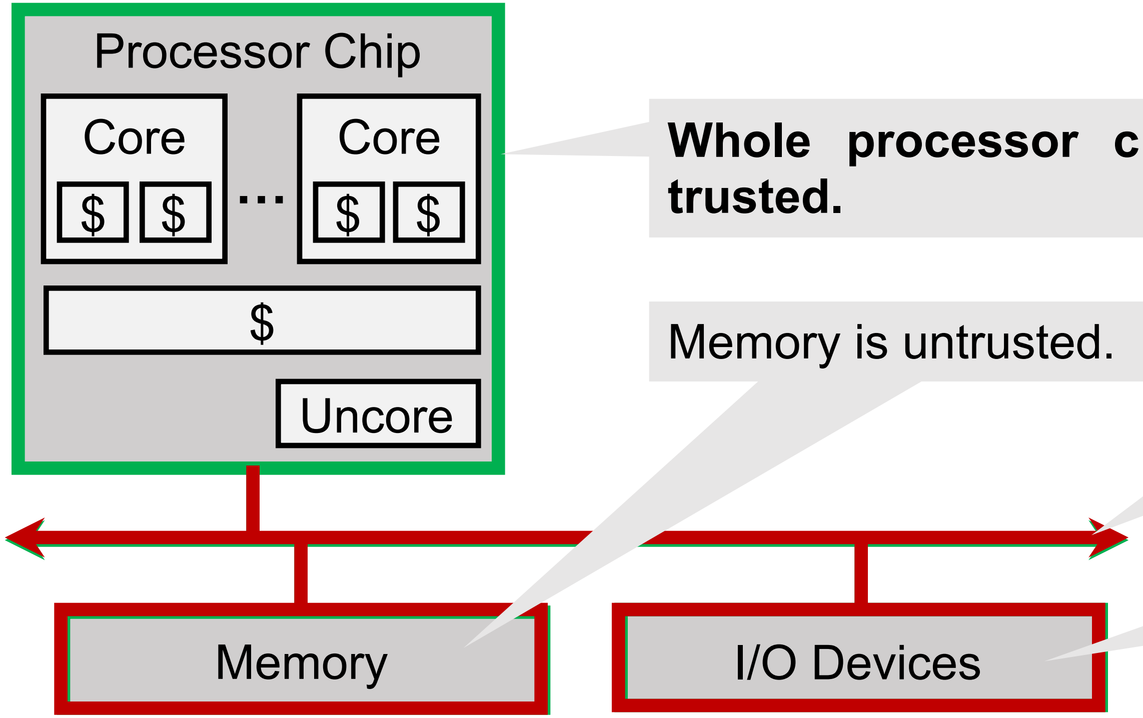
- Each component **trusts** all the software "below" it



Security Engine (SecE) can be something like Intel's ME or AMD's PSP.

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.



Whole processor chip is trusted.

Memory is untrusted.

System bus is untrusted.

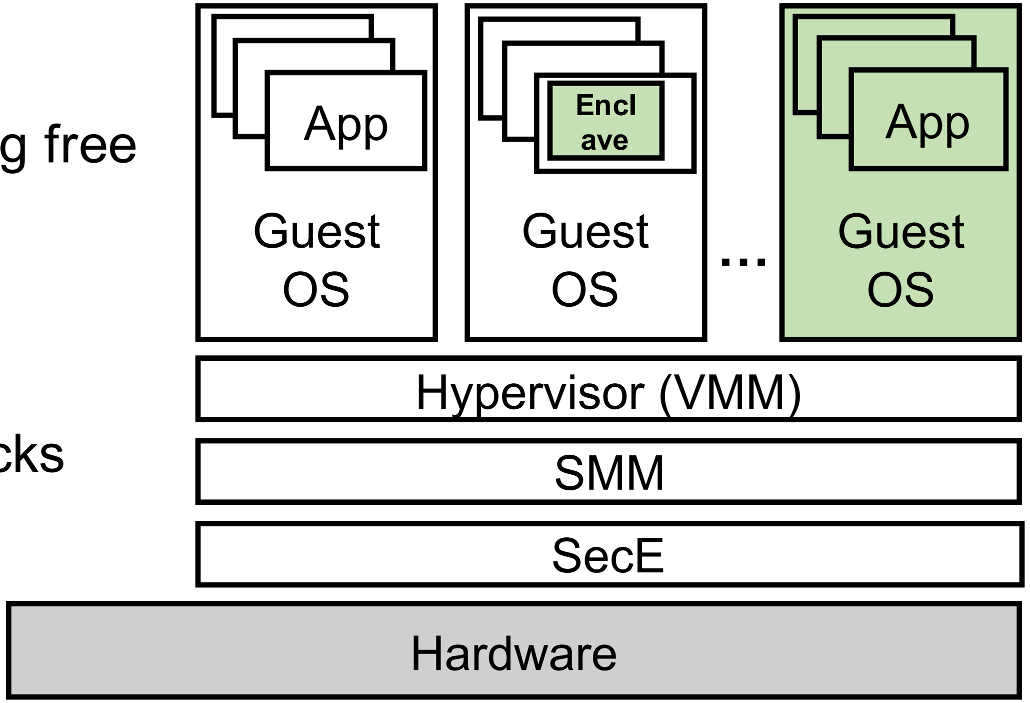
Devices are untrusted.

TEE and TCB

The **Trusted Computing Base (TCB)** is the set of hardware and software that is responsible for realizing the TEE:

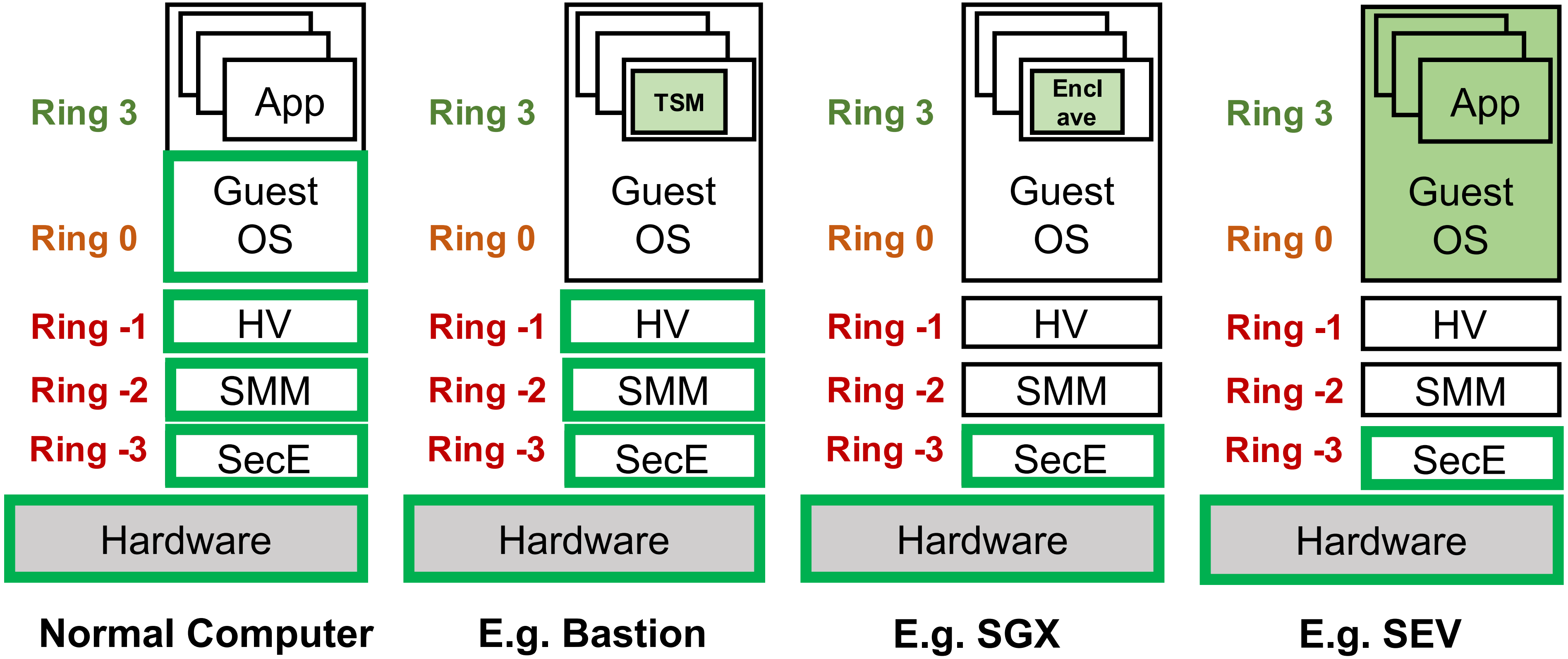
- TCB is trusted to correctly implement the protections
- Vulnerabilities or attacks on TCB nullify TEE protections
- TCB is trusted
- TCB may not be trustworthy, if is not verified or is not bug free

The goal of **Trusted Execution Environments (TEEs)** is to provide protections for a piece of code and data from a range of software attacks and some hardware attacks



Breaking Linear Hierarchy of Protection Rings

Examples of architectures that do and don't have a linear relationship between privileges and protection ring level:

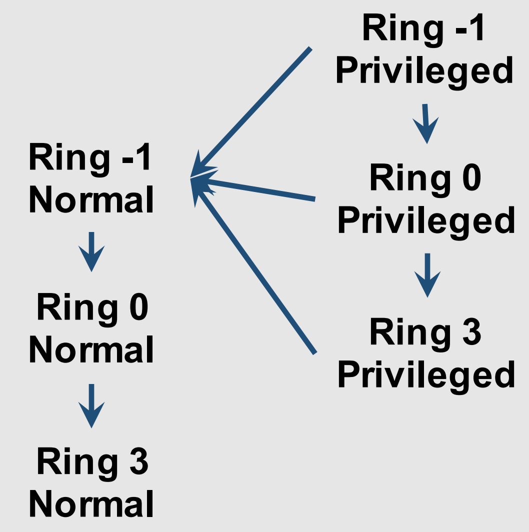


Adding Horizontal Privilege Separation

New privileges can be made orthogonal to existing protection rings.

- E.g. ARM’s TrustZone’s “normal” and “secure” worlds
- Need privilege level (ring number) and normal / secure privilege

Security levels from a lattice:



Ring -1 Privileged

Ring 0 Privileged

Ring 3 Privileged

Ring -1 Normal

Ring 0 Normal

Ring 3 Normal

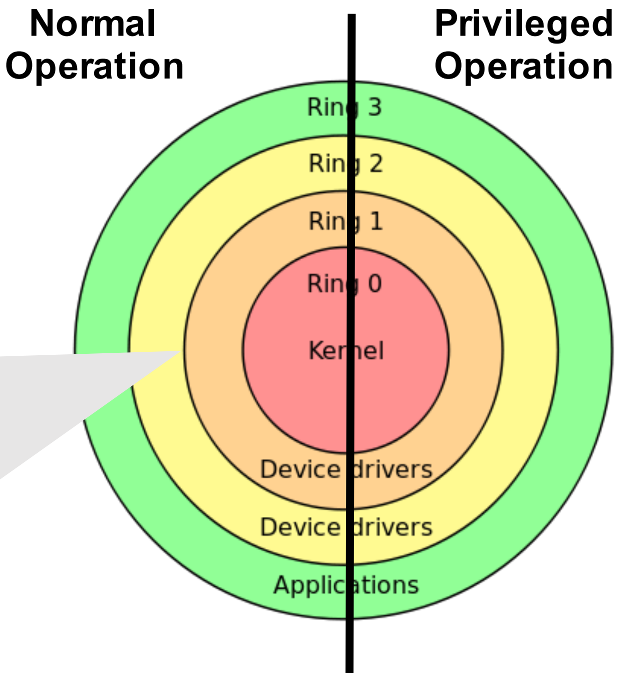
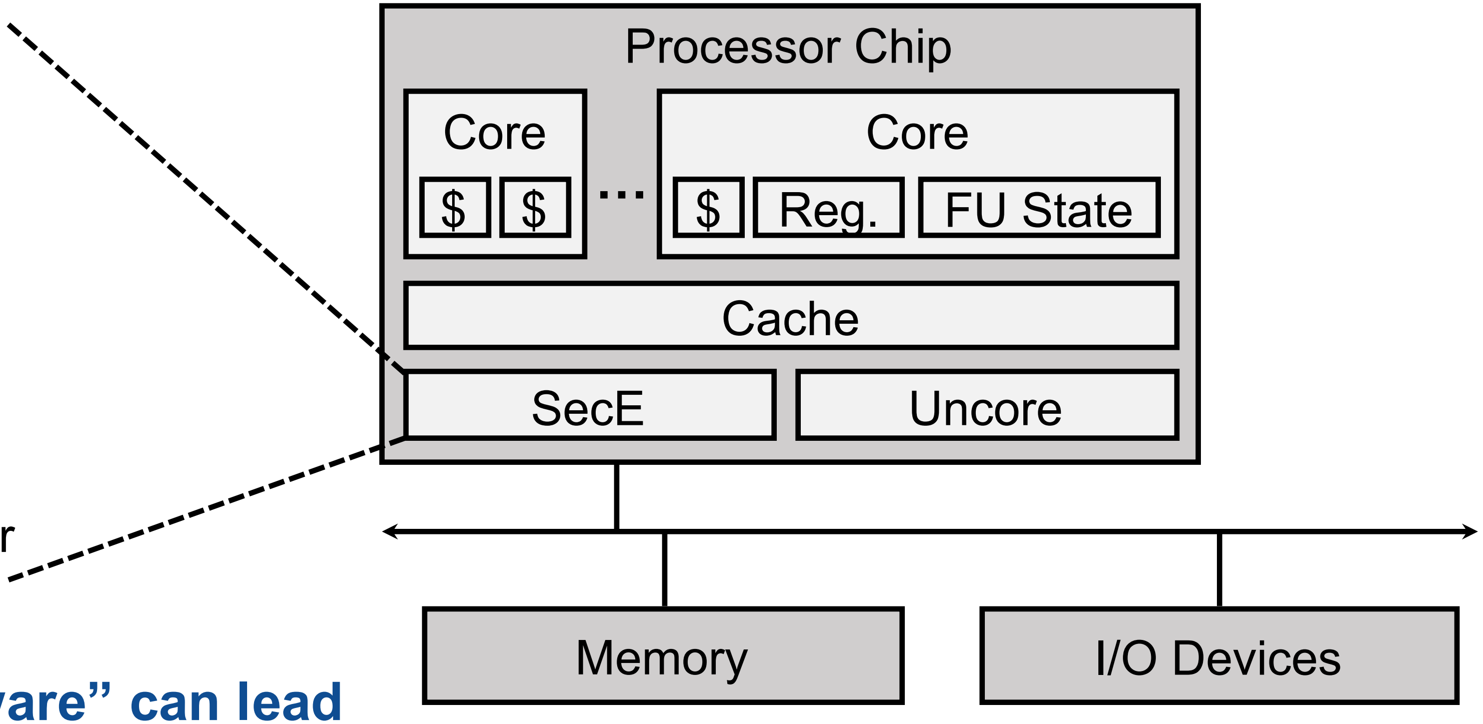


Image: https://commons.wikimedia.org/wiki/File:Priv_rings.svg

Hardware TCB as Circuits or Processors

Key parts of the hardware TCB can be implemented as dedicated circuits or as firmware or other code running on dedicated processor

- **Custom logic or hardware state machine:**
  - Most academic proposals
- **Code running on dedicated processor:**
  - Intel ME = ARC processor or Intel Quark processor
  - AMD PSP = ARM processor



Vulnerabilities in TCB "hardware" can lead to attacks that nullify the security protections offered by the system.

Protections Offered by Secure Processor Architectures

Security properties for the TEEs that secure processor architectures aim to provide:

- Confidentiality
- Integrity

Confidentiality is the prevention of the disclosure of secret or sensitive information to unauthorized users or entities.

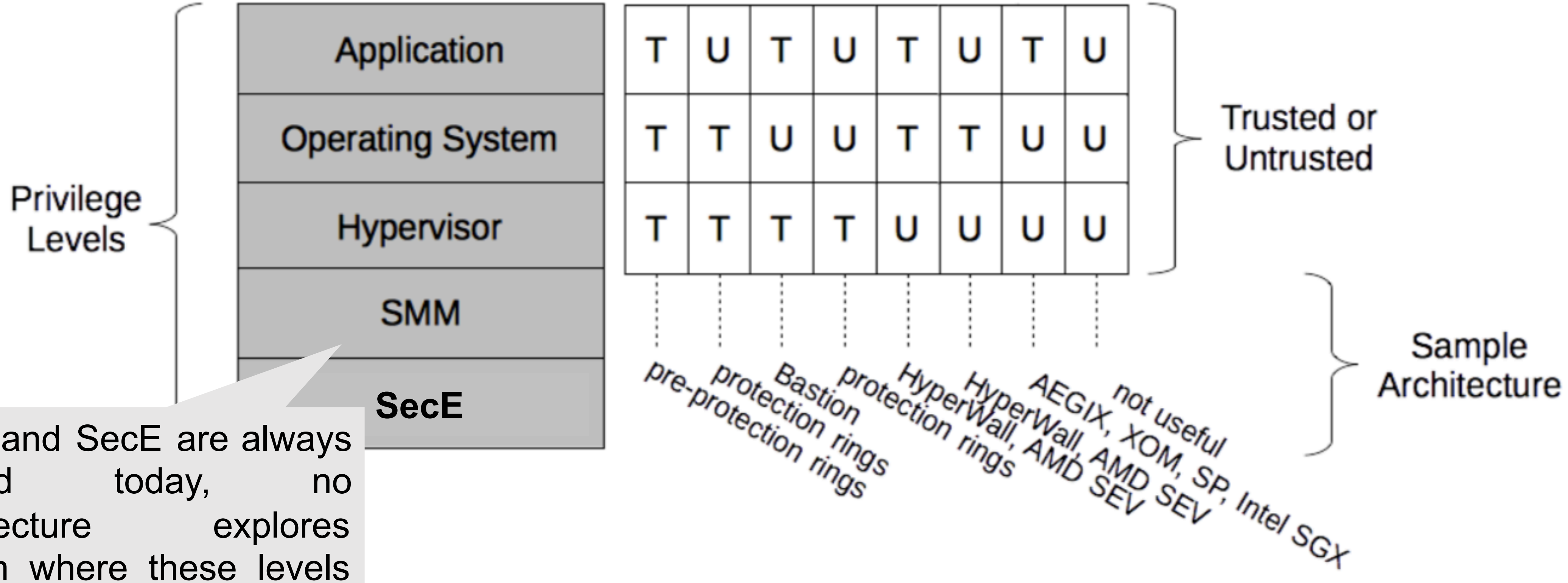
Integrity is the prevention of unauthorized modification of protected information without detection.

- Availability is usually not provided usually

Confidentiality and integrity protections are from attacks by other components (and hardware) not in the TCB. **There is typically no protection from malicious TCB.**

Protections Categorized by Architecture

Secure processor architectures break the linear relationship (where lower level protection ring is more trusted):



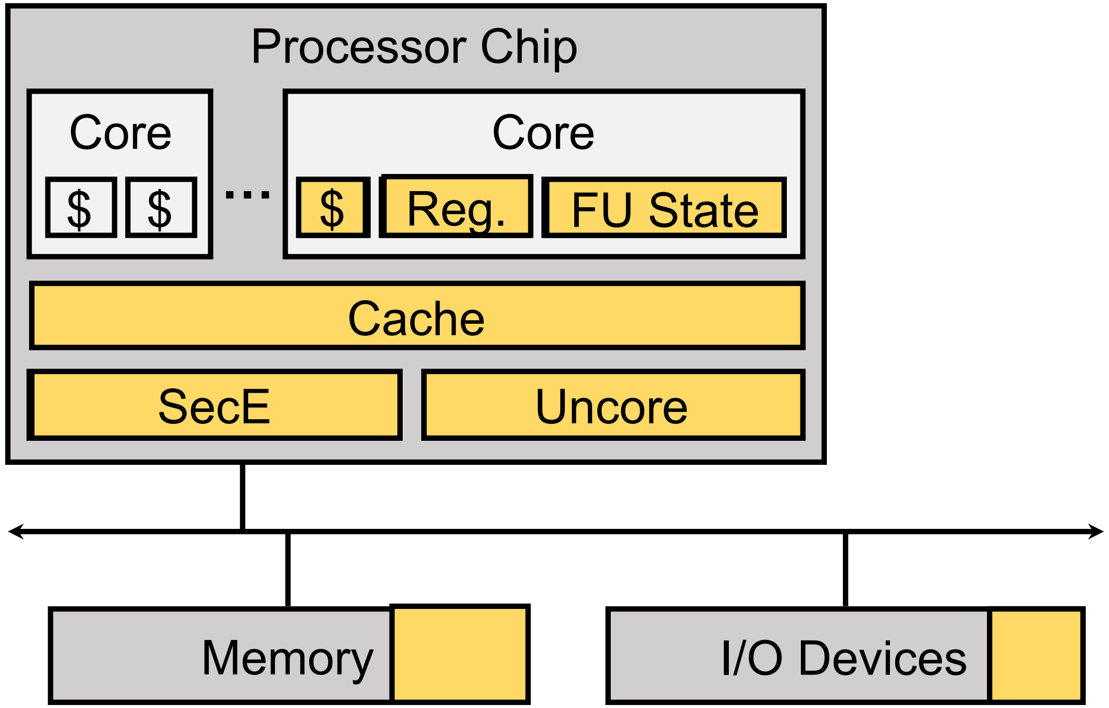
Privilege Levels	pre-protection rings	protection rings	Bastion	protection rings	HyperWall, AMD SEV	HyperWall, AMD SEV	AEGIX, XOM, SP, Intel SGX	not useful
Application	T	U	T	U	T	U	T	U
Operating System	T	T	U	U	T	T	U	U
Hypervisor	T	T	T	T	U	U	U	U
SMM								
SecE								

SMM and SecE are always trusted today, no architecture explores design where these levels are untrusted.

Protecting State of the Protected Software

Protected software's **state** is distributed throughout the processor. All of it needs to be protected from the untrusted components and other (untrusted) protected software.

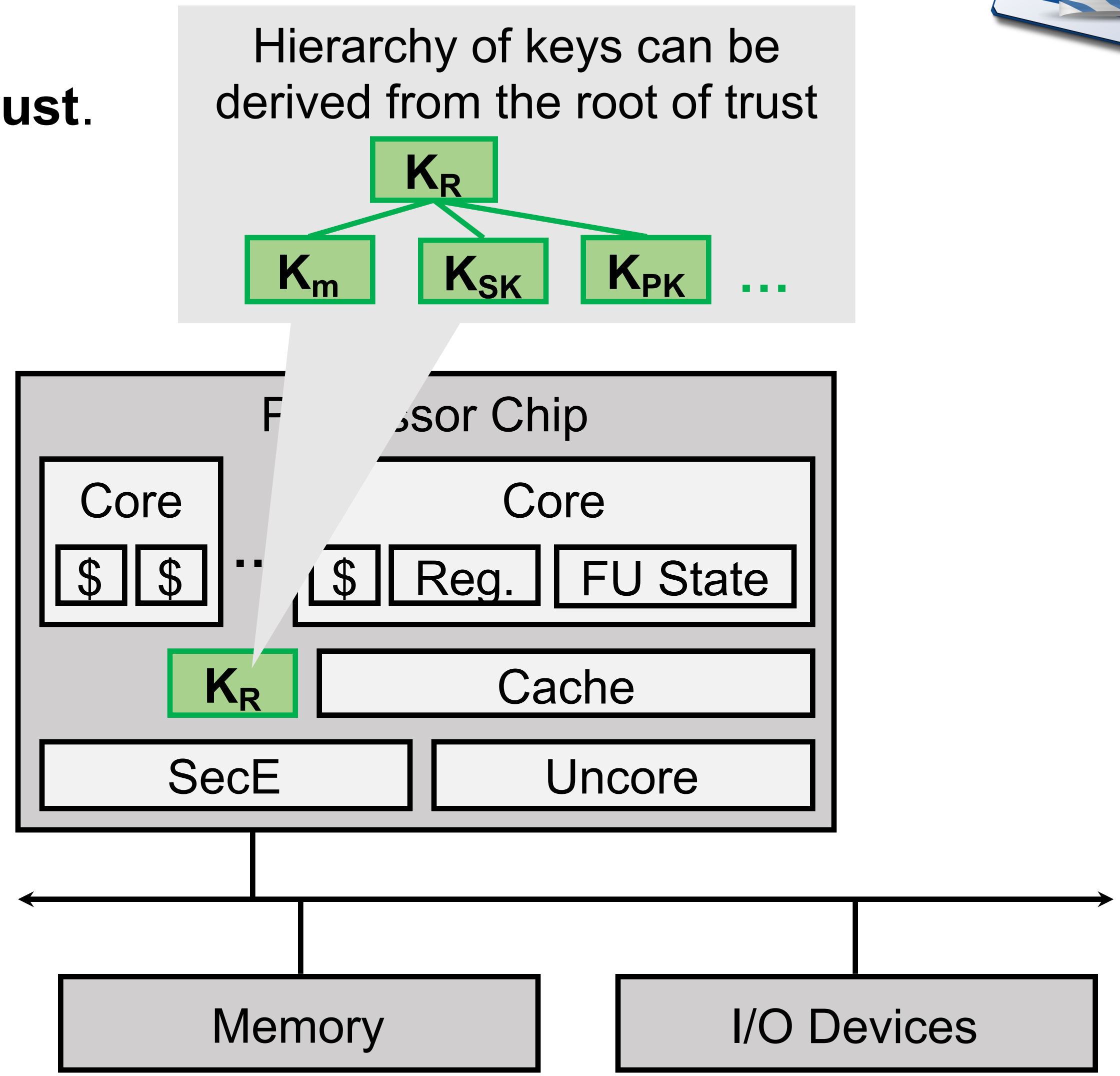
- Protect memory through encryption and hashing with integrity trees
- Flush state, or isolate state, of functional units in side processor cores
- Isolate state in uncore and any security modules
- Isolate state in I/O and other subsystems



Root of Trust for TCB

Security of the system is derived from a **root of trust**.

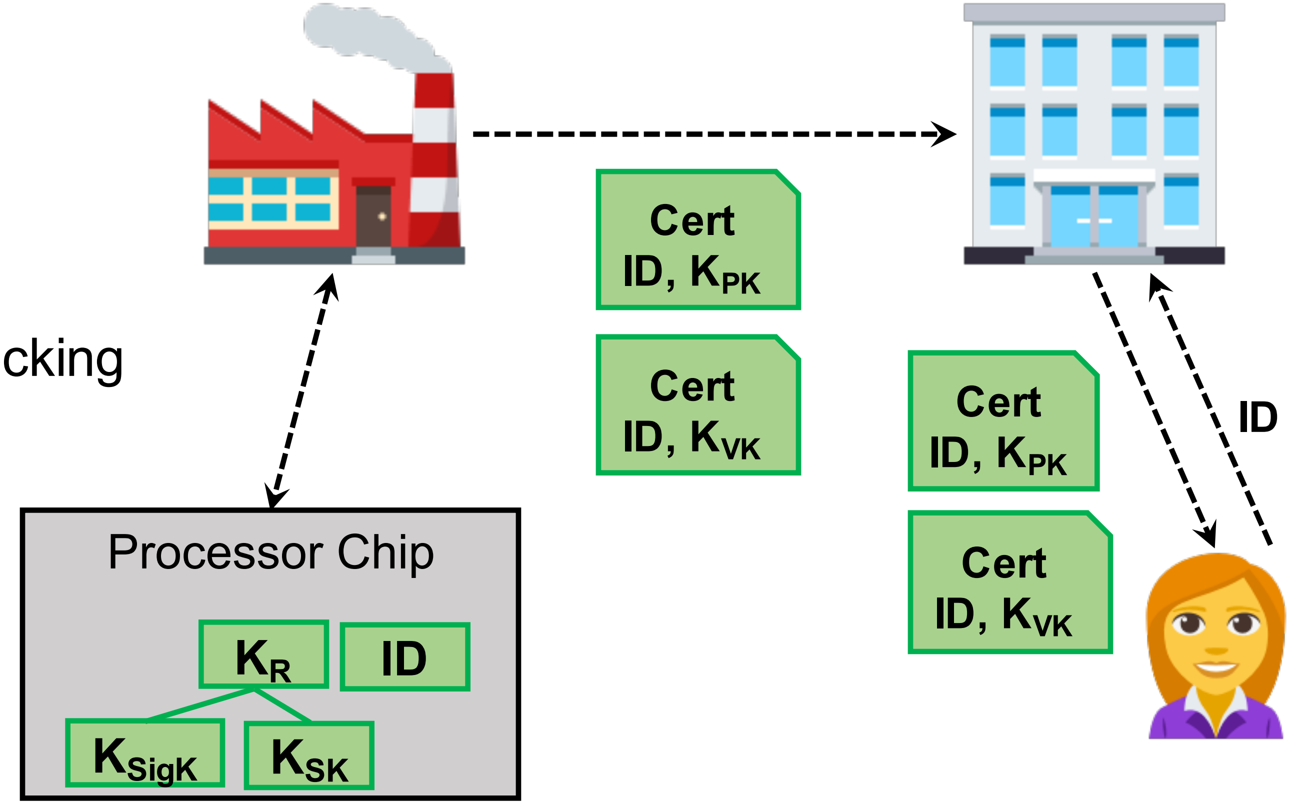
- A secret (cryptographic key) only accessible to TCB components
- Derive encryption and signing keys from the root of trust
- **Burn in at the factory** by the manufacturer (but implies trust issues with manufacturer and the supply chain)
  - E.g. One-Time Programmable (OTP) fuses
- Use **Physically Uncloneable Functions** (but requires reliability)
  - Extra hardware to derive keys from PUF
  - Mechanisms to generate and distribute certificates for the key



Derived Keys and Key Distribution

Derived form the root of trust are signing and verification keys.

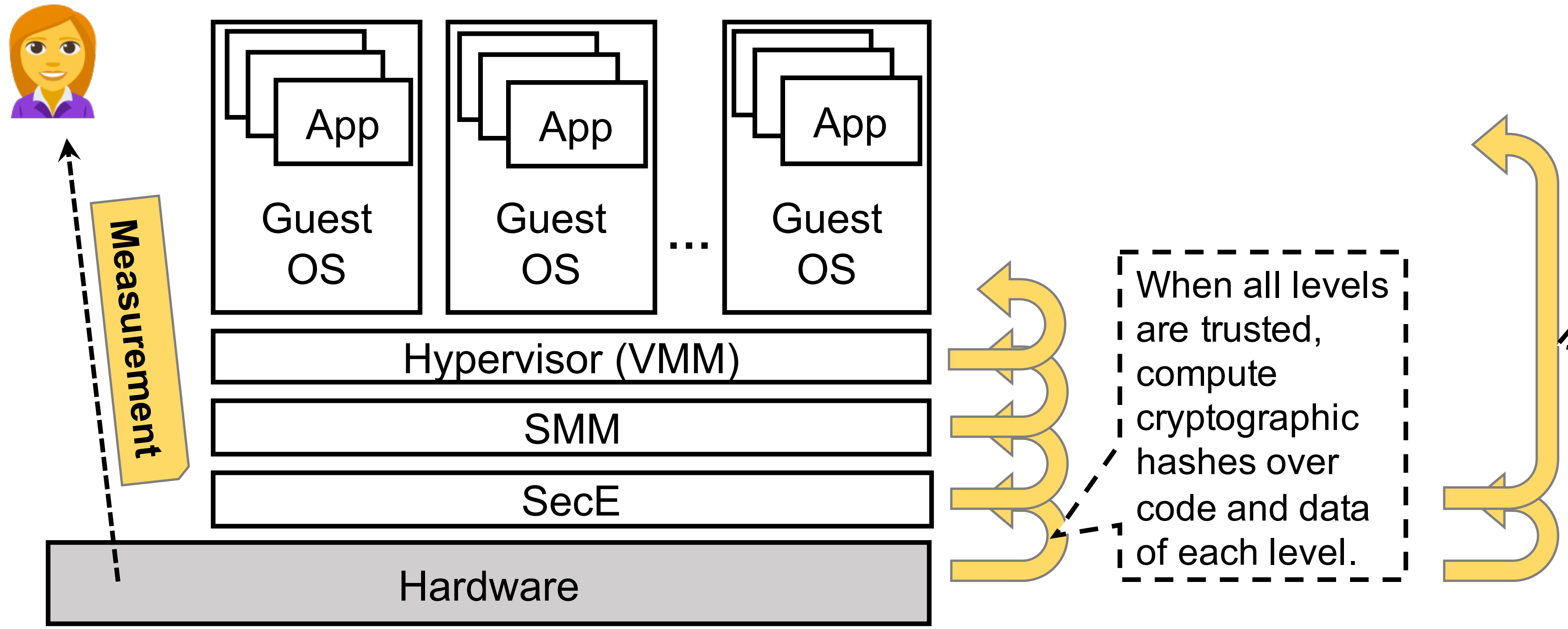
- Public key, K_{PK}, for encrypting data to be sent to the processor
  - Data handled by the TCB
- Signature verification key, K_{VK}, for checking data signed by the processor
  - TCB can sign user keys
- Key distribution for PUF based designs will be different
- Need infrastructure!



Emoji Image:
https://www.emojione.com/emoji/1f3ed
https://www.emojione.com/emoji/1f469-1f4bc
https://www.emojione.com/emoji/1f3e2

Software Measurement

With an embedded signing key, the software running in the TEE can be "measured" to attest to external users what code is running on the system.



When all levels are trusted, compute cryptographic hashes over code and data of each level.

Some architectures, e.g. SGX or SEV, "skip" untrusted layers when computing hashes

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

Using Software Measurement

Trusted / Secure / Authenticated Boot:

- Abort boot when wrong measurement is obtained
- Or, continue booting but do not decrypt secrets
- Legitimate software updates will change measurements, may prevent correct boot up

Remote attestation:

- Measure and digitally sign measurements that are sent to remove user

Data sealing (local or remote):

- Only unseal data if correct measurements are obtained

TOC-TOU attacks and measurements:

- Time-of-Check to Time-of-Use (TOC-TOU) attacks leverage the delay between when a measurement is taken, and when the component is used
- Cannot easily use hashes to prevent TOC-TOU attacks

Memory Protections in Secure Processors

Sources of Attacks on Memory

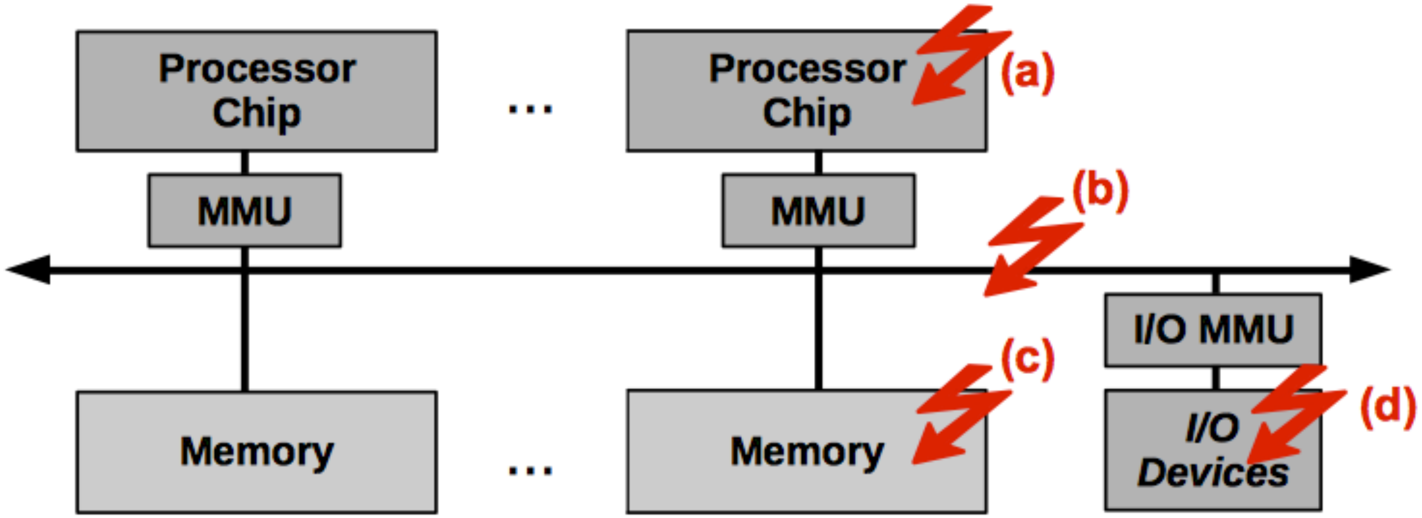
Memory is vulnerable to different types of attacks:

a) Untrusted software running no the processor

b) Physical attacks on the memory bus, other devices snooping on the bus, man-in-the-middle attacks with malicious device

c) Physical attacks on the memory (Coldboot, …)

d) Malicious devices using DMA or other attacks



Common attack types:
- Snooping
- Spoofing
- Splicing
- Replay
- Disturbance

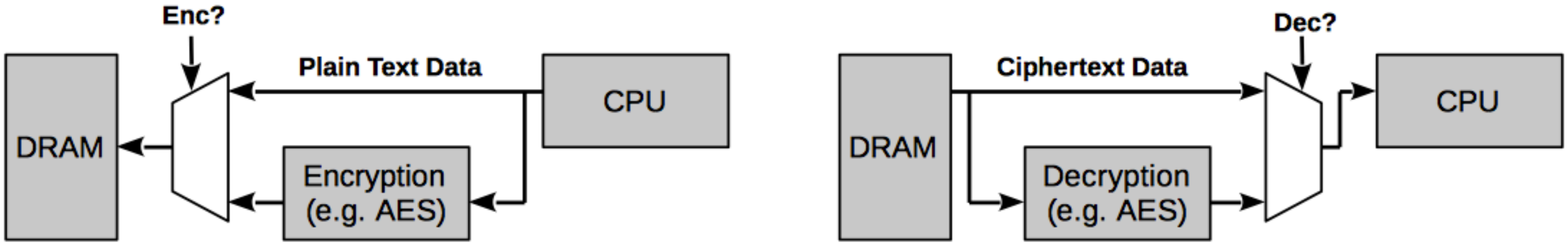
Confidentiality Protection with Encryption

Contents of the memory can be protected with encryption. Data going out of the CPU is encrypted, data coming from memory is decrypted before being used by CPU.

a) Encryption engine (usually AES in CTR mode) encrypts data going out of processor chip

b) Decryption engine decrypts incoming data

Pre-compute encryption pads, then only need to do XOR; speed depends on how well counters are fetched / predicted.

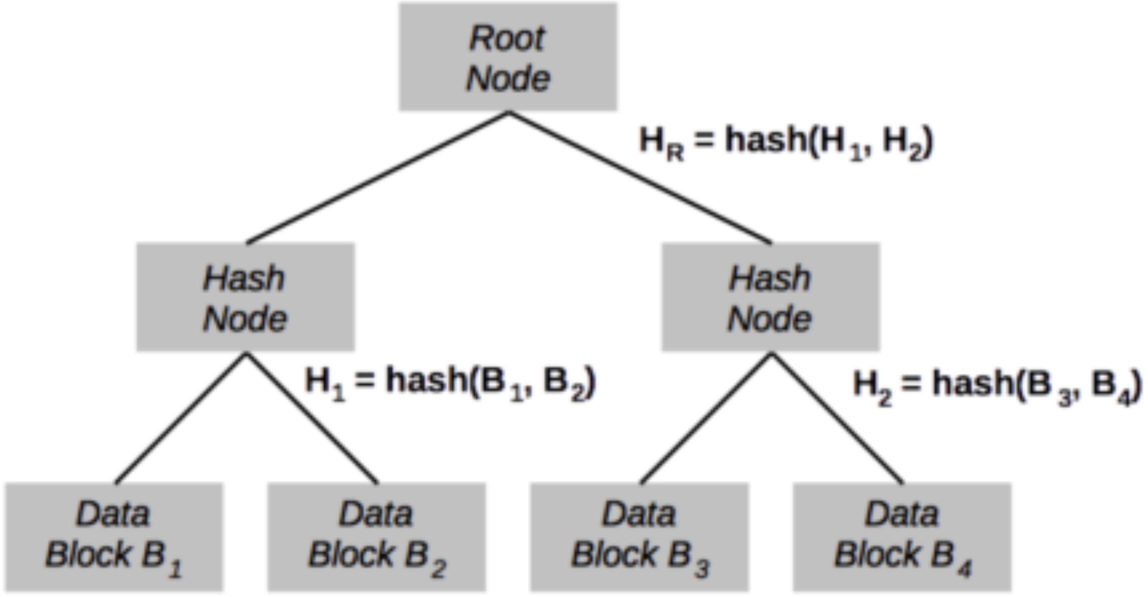


(a) Memory Write

(b) Memory Read

Integrity Protection with Hash Trees

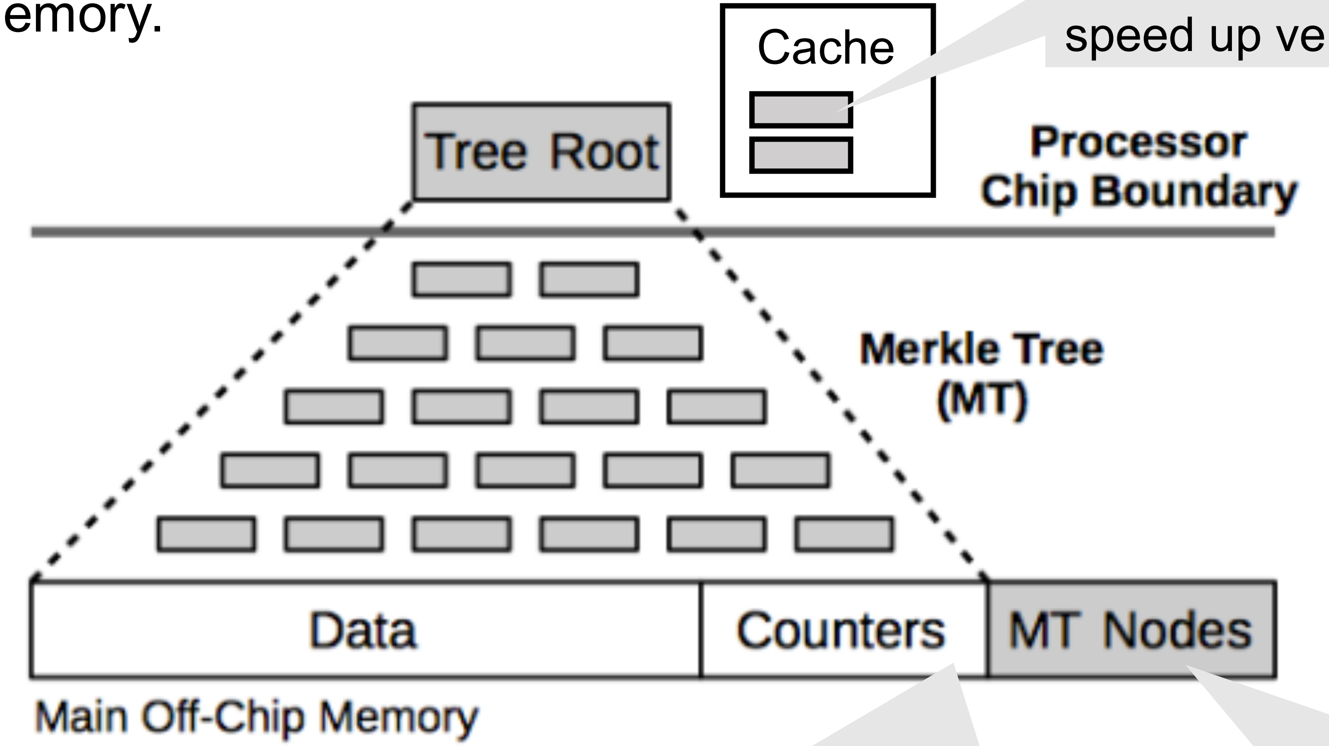
Hash tree (also called **Merkle Tree**) is a logical three structure, typically a binary tree, where two child nodes are hashed together to create parent node; the root node is a hash that depends on value of all the leaf nodes.



Integrity Protection with Hash Trees

Memory blocks can be the leaf nodes in a Merkle Tree, the tree root is a hash that depends on the contents of the memory.

On-chip (cached) nodes are assumed trusted, used to speed up verification.

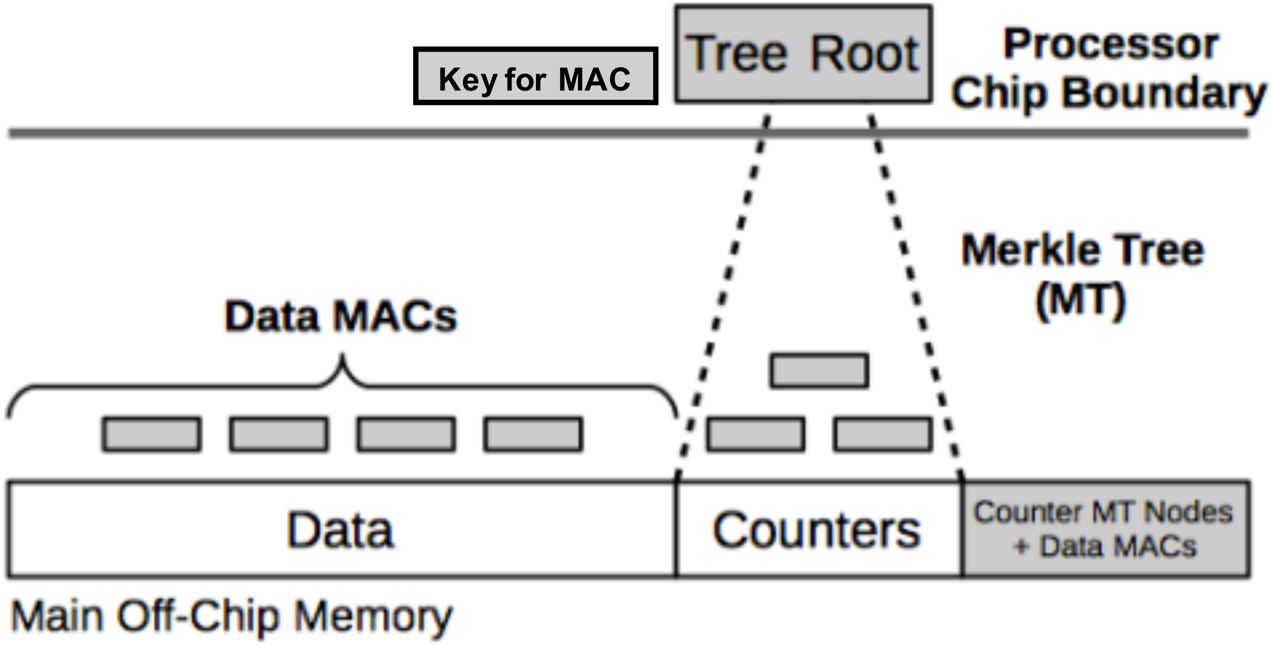


Counters are included in hashes for freshness.

Hash tree nodes are stored in (untrusted) main memory.

Integrity Protection with Bonsai Hash Trees

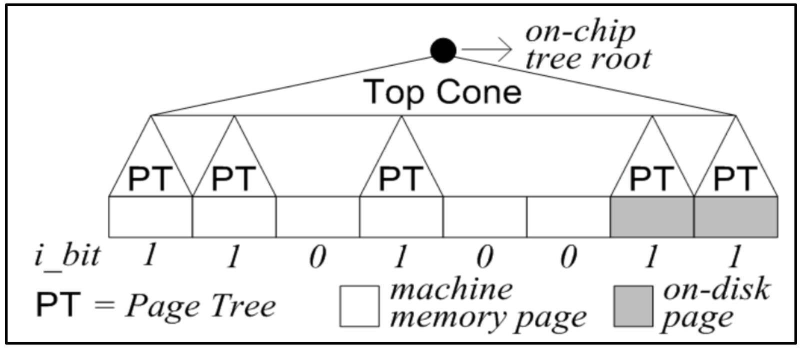
Message Authentication Codes (MACs) can be used instead of hashes, and a smaller "Bonsai" tree can be constructed.

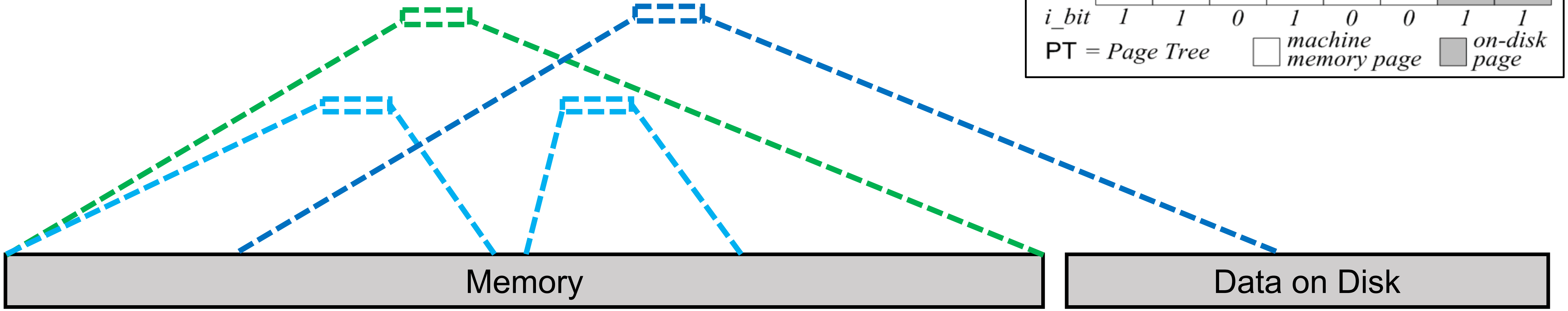


Integrity Protection of Selected Memory Regions

- For encryption, type of encryption does not typically depend on memory configuration
- For integrity, the integrity tree needs to consider:
  - Protect whole memory
  - Protect parts of memory (e.g. per application, per VM, etc.)
  - Protect external storage (e.g. data swapped to disk)

E.g., Bastion’s memory integrity tree (Champagne, et al., HPCA ‘10)



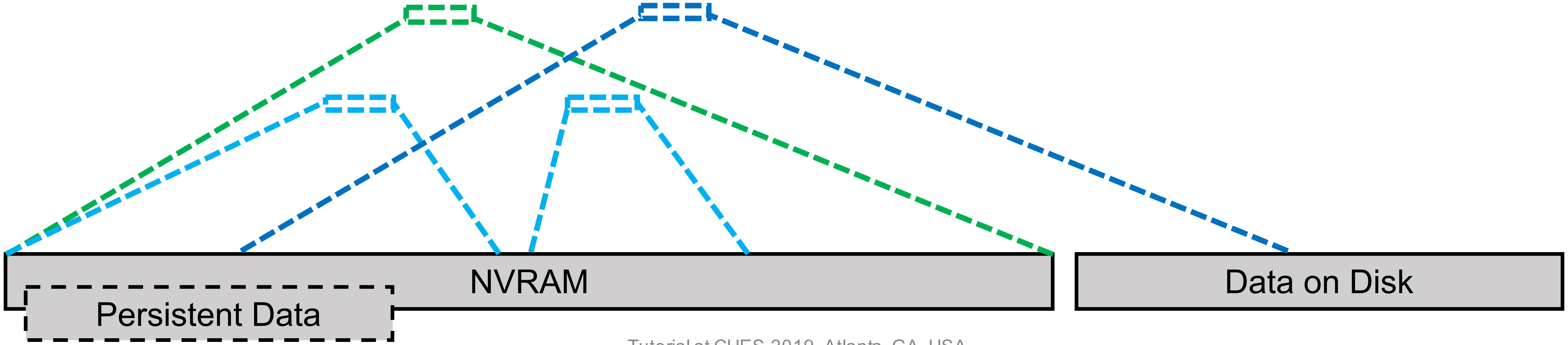


Integrity Protection of NVRAMs

- Non-volatile memories (NVMs) can store data even when there is no power, they are suitable to serve as a computer system's main memory, and replace or augment DRAM
  - Data remanence makes passive attacks easier (e.g. data extraction)
  - Data is maintained after reboot or crash (security state also needs to be correctly restored after reboot or crash)

Integrity considerations

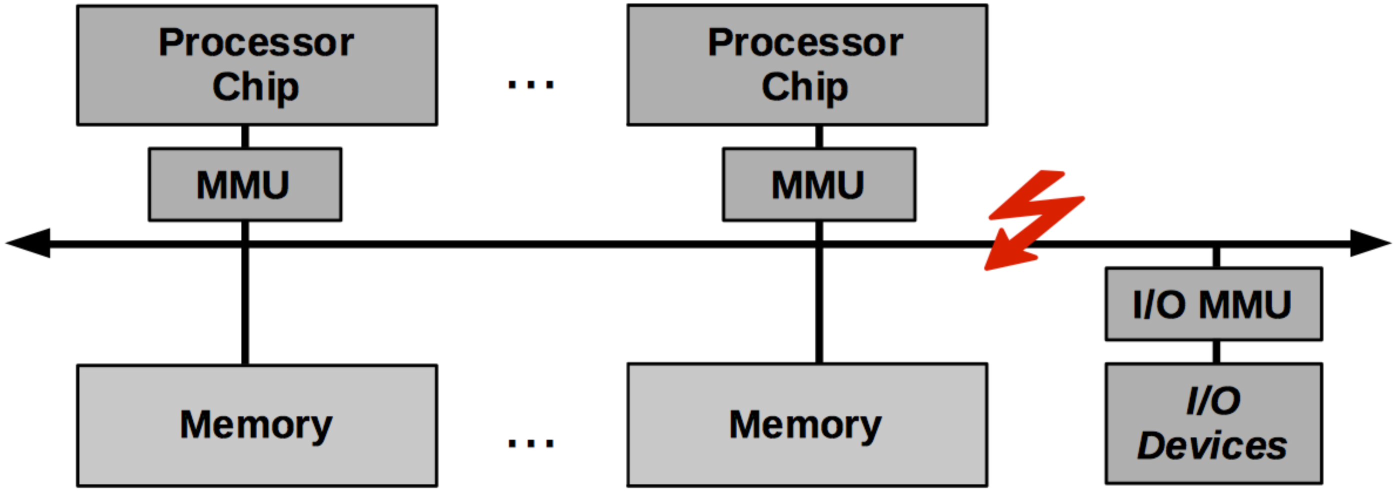
  - Atomicity of memory updates for data and related security state (so it is correct after reboot or a crash)
  - Which data in NVRAM is to be persisted (i.e. granularity)



Memory Access Pattern Protection

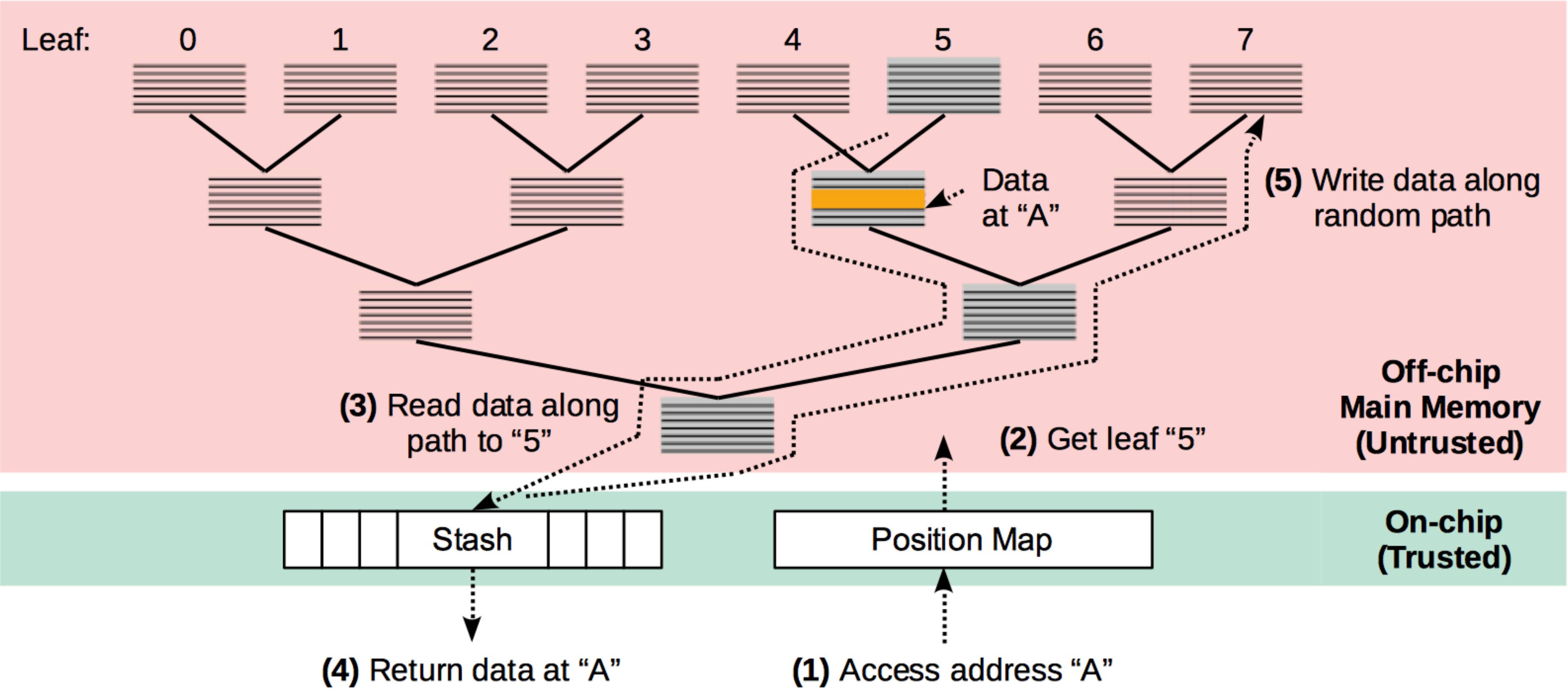
Snooping attacks can target extracting data (protected with encryption) or **extracting access patterns** to learn what a program is doing.

- Easier in Symmetric multiprocessing (SMP) due to shared bus
- Possible in other configuration if there are untrusted components



Memory Access Pattern Protection

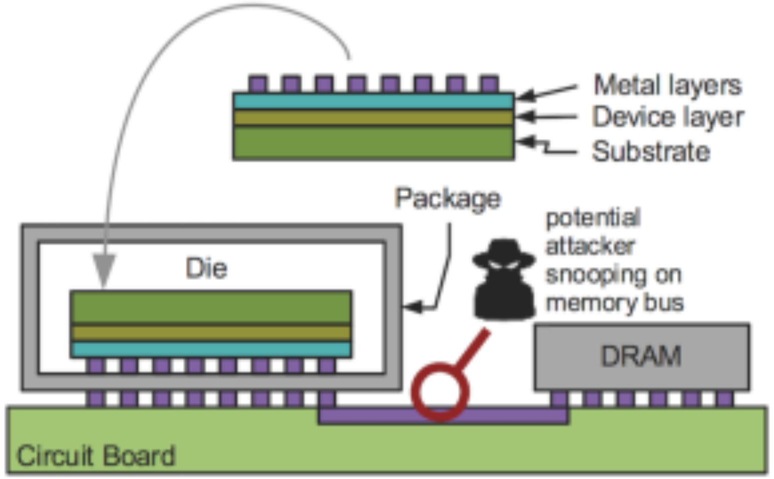
Access patterns (traffic analysis) attacks can be protected with use Oblivious RAM, such as Path ORAM. This is on top of encryption and integrity checking.

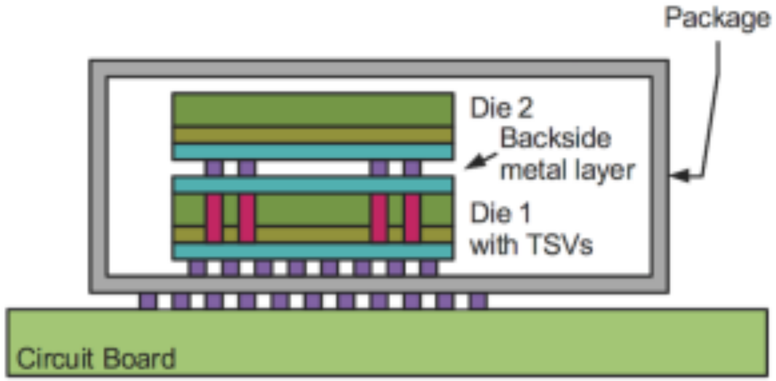


Leveraging 2.5D and 3D Integration

With 2.5D and 3D integration, the memory is brought into the same package as the main processor chip. Further, with embedded DRAM (eDRAM) the memory is on the same chip.

- Potentially probing attacks are more difficult
- Still limited memory (eDRAM around 128MB in 2017)





Principles of Design of Secure Processors

Principles of Secure Processor Architecture Design

Four principles for secure processor architecture design based on existing designs and also on ideas about what ideal design should look like are:

1. **Protect Off-chip Communication and Memory**
2. **Isolate Processor State among TEE Execution and other Software**
3. **Allow TCB Introspection**
4. **Authenticate and Continuously Monitor TEE and TCB**

- Architectural state
- Micro-architectural state
- Due to spatial or temporal sharing of hardware

Focus of other two parts of the tutorial

Additional design suggestions:

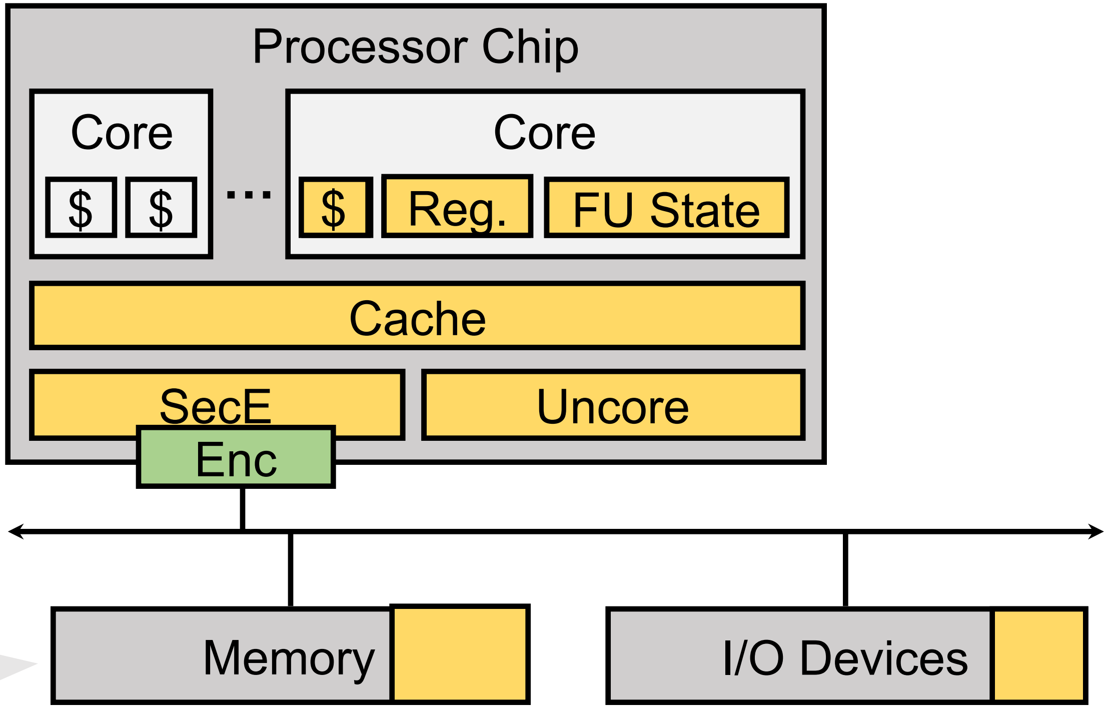
- Avoid code bloat
- Minimize TCB
- Ensure hardware security (Trojan prevention, supply chain issues, etc.)
- Use formal verification

Protect Off-chip Communication and Memory

Off-chip components and communication are untrusted, need protection with **encryption**, **hashing**, **access pattern protection**.

Open research challenges:

- Performance



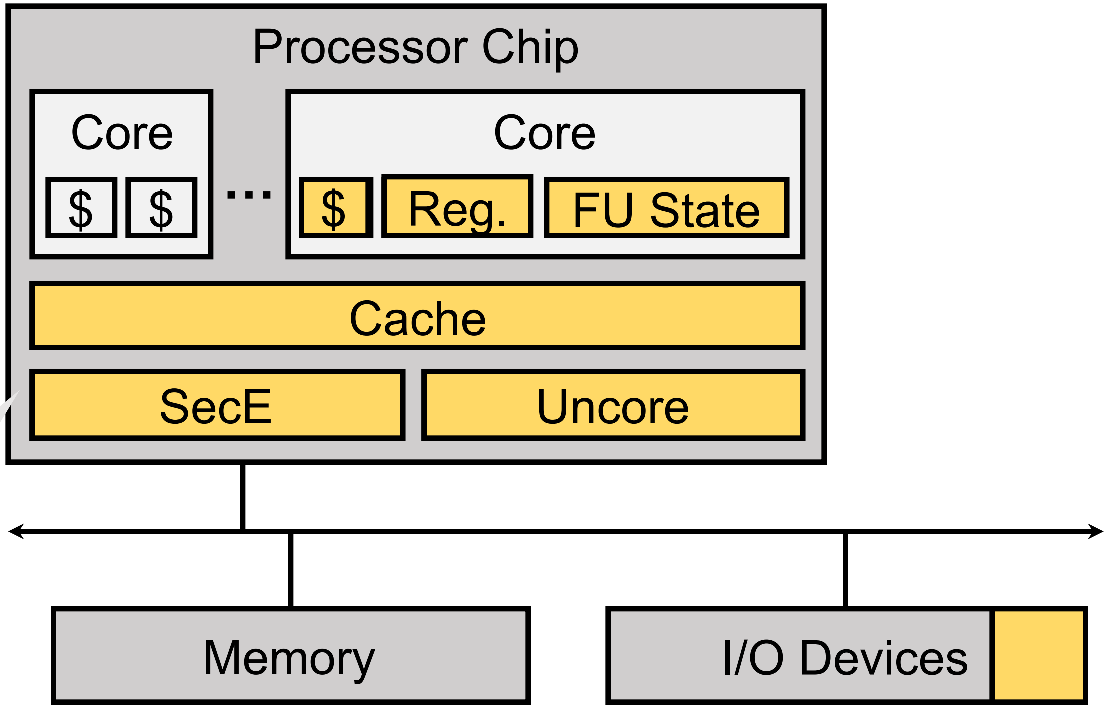
E.g. encryption defends Cold boot style attacks on main memory.

Isolate Processor State among TEE Execution

When switching among protected software and other software or other protected software, need to flush the state, or save and restore it, to prevent one software influencing another.

Open research challenges:

- Performance
- Finding all the state to flush or clean
- Isolate state during concurrent execution
- ISA interface to allow state flushing



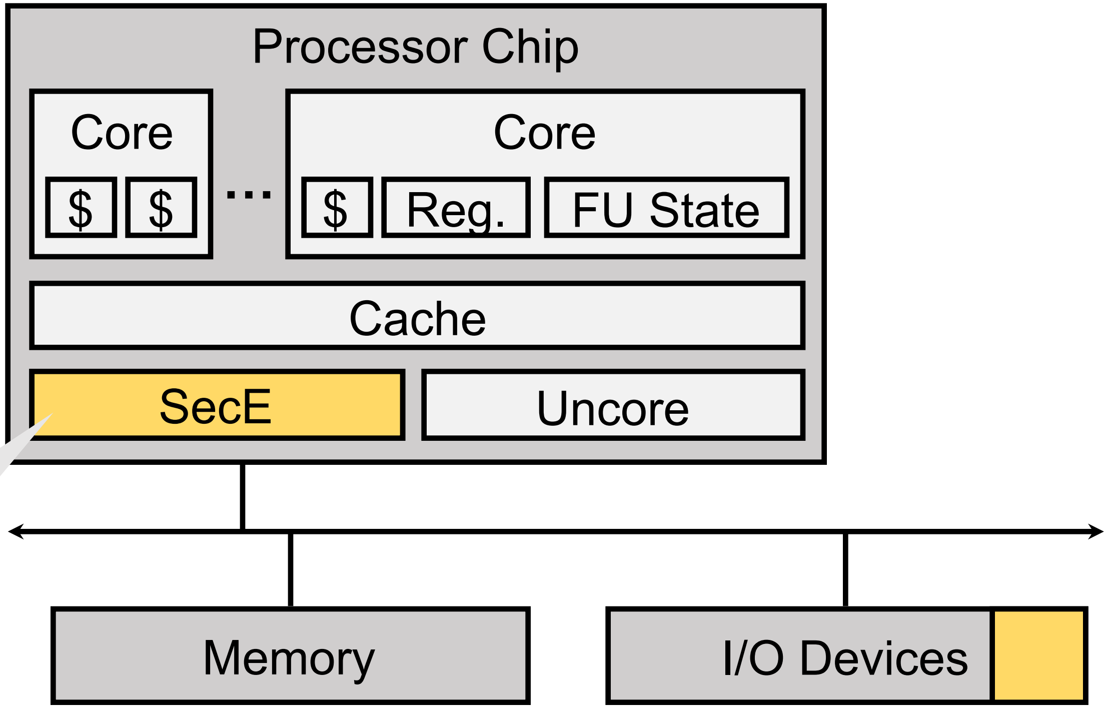
E.g. flushing state helps defend Spectre and Meltdown type attacks.

Allow TCB Introspection

Need to ensure correct execution of TCB, through **open access to TCB design**, **monitoring**, **fingerprinting**, and **authentication**.

Open research challenges:

- ISA interface to introspect TCB
- Area, energy, performance costs due extra features for introspection
- Leaking information about TCB or TEE



E.g. open TCB design can minimize attacks on ME or PSP security engines

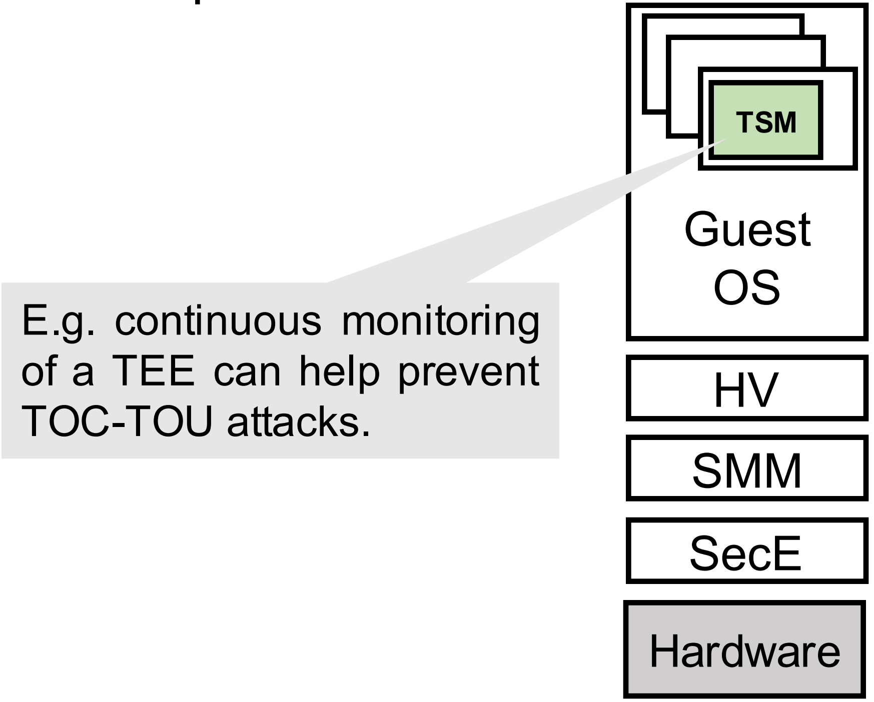
Authenticate and Continuously Monitor TEE and TCB

Monitoring of software running inside TEE, e.g. TSMs or Enclaves, gives assurances about the state of the protected software.

Likewise monitoring TCB ensures protections are still in place.

Open research challenges:

- Interface design for monitoring
- Leaking information about TEE



E.g. continuous monitoring of a TEE can help prevent TOC-TOU attacks.

Design of Secure Processor Architectures

Part 2

Timing Channels: Attacks and Hardware Defenses

Jakub Szefer
Assistant Professor
Dept. of Electrical Engineering
Yale University

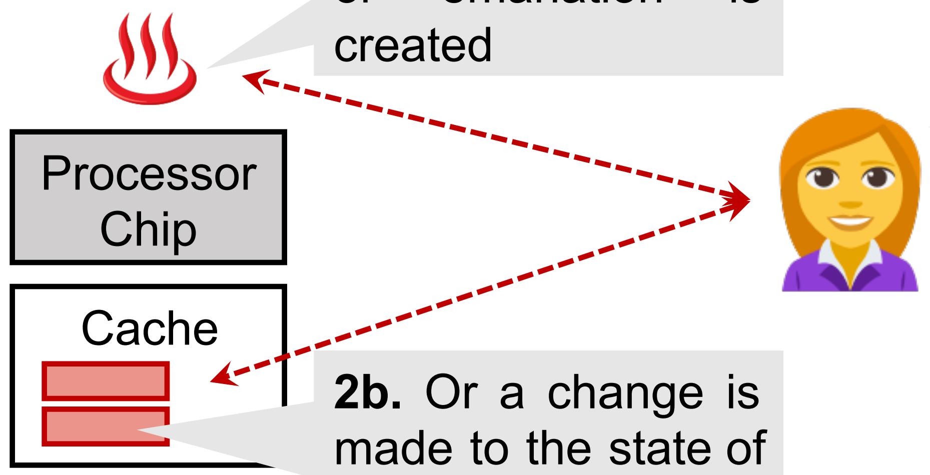
CHES 2019 – August 25, 2019

Slides and information available at: https://caslab.csl.yale.edu/tutorials/ches2019/

Side and Covert Channels

Side and Covert Channels in Processors

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

3. “Receiver” observes the emanation or state change

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

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

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

Goals of Side and Covert Channels

Goal of side or covert channels is to break the logical protections of the computer system and leak confidential or sensitive information.

- Typically attack confidentiality (leak data from secure to insecure)
  - All attacks fall in this category, they establish a channel to exfiltrate information
- Could be used in “reverse” to attack integrity (insecure data leaks to, and affects secure data)
  - Power, thermal, or EM fault attacks can also fall in this category
- Beyond leaking data:
  - Leak control flow or execution patterns
  - Leak memory access patterns

Channels: Victim-to-Attacker and Attacker-to-Victim

Typically a channel is **from an unsuspecting victim to an attacker**:

- Goal is to extract some information from victim
- Victim does not observe any execution behavior change

Victim's operation sends information to attacker

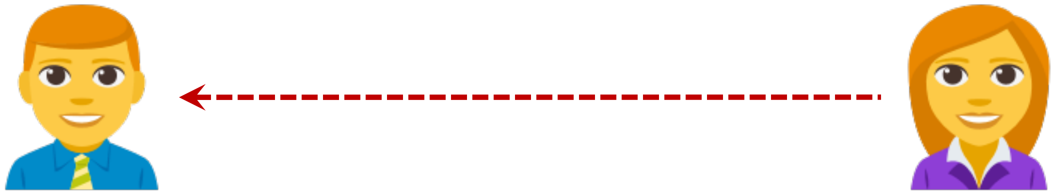
Attacker obtains information via the side channel

A channel can also exist **from attacker to victim**:

- Attacker's behavior can "send" some information to the victim
- The information, in form of processor state for example, affects how the victim behaves unbeknownst to them

E.g. modulate branch predictor state to affect execution of the victim

Victim's operation depends on the information sent from attacker



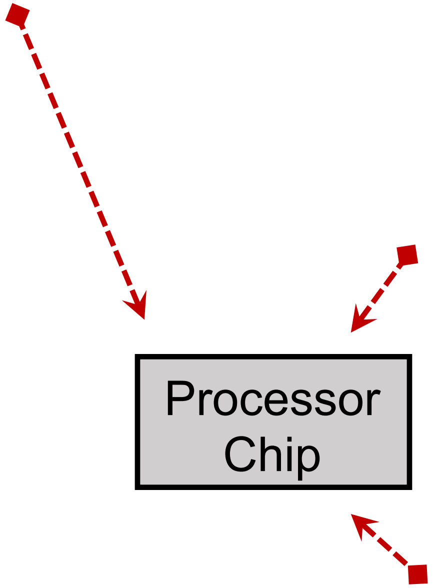
Attacker modulates some information that is sent to victim

Emoji Image:
https://www.emojione.com/emoji/1f468-1f4bc
https://www.emojione.com/emoji/1f469-1f4bc

Side and Covert Channels and Physical Proximity

Distance: infinity
(assuming network connection)

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.



Processor Chip

Distance: small
(physical connection)

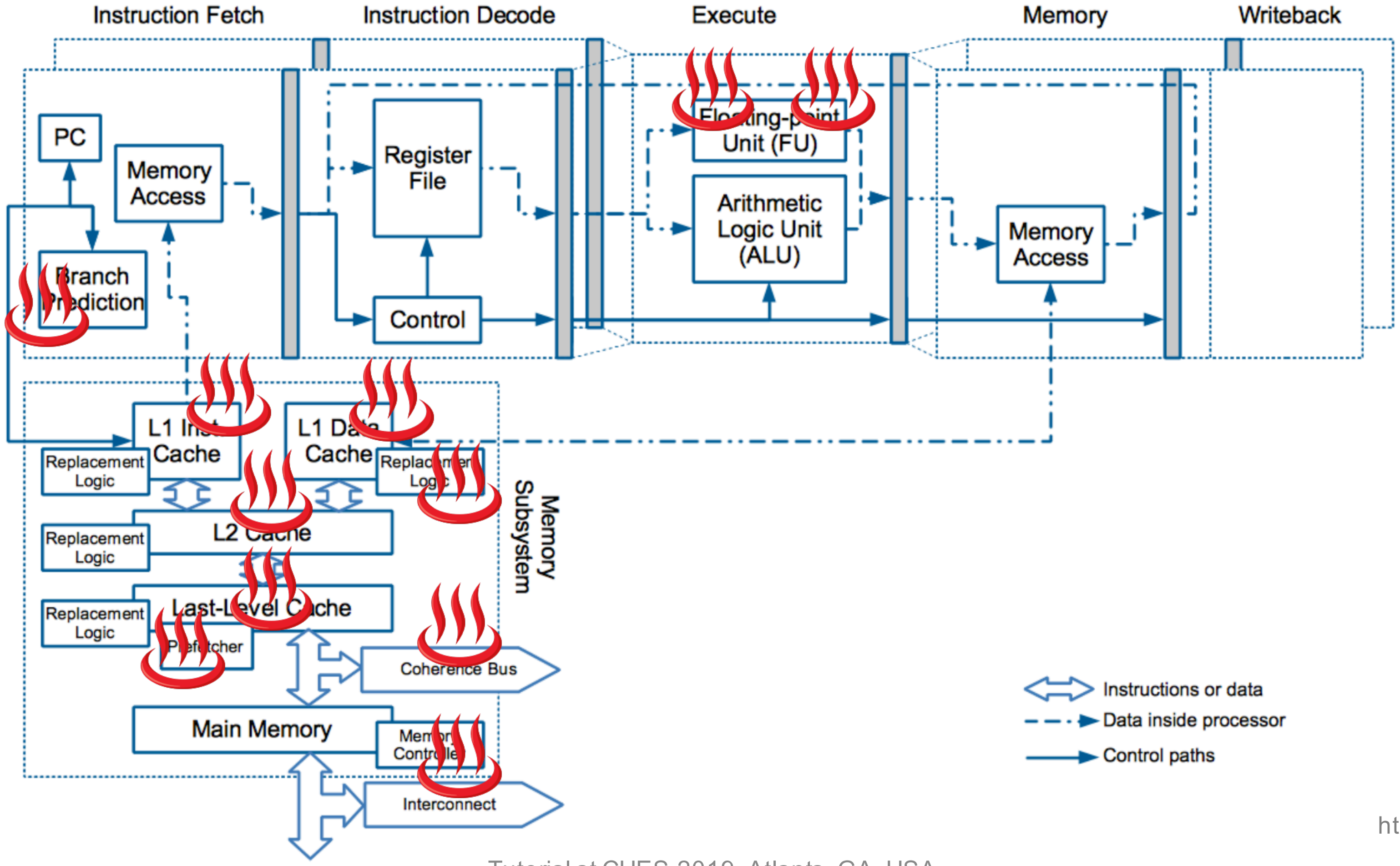
Power channels require physical connection to measure the power consumed by the CPU (assuming there is no on-chip sensor that can be abused to get power measurements).

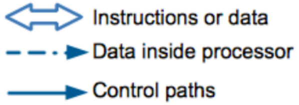
Distance: medium
(emanations signal range)

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

Timing Channels Inside a Processor

Many components of a modern processor pipeline can contribute to timing channels.





Emoji Image:
https://www.emojione.com/emoji/2668

Sources of Timing Side Channels

Six source of timing channels that can lead to attacks:

1. **Instruction with Different Execution Timing** – Execution of different instructions takes different amount of time
2. **Variable Instruction Timing** – Execution of a specific instruction takes different time, e.g. depending on the state of the unit
3. **Functional Unit Contention** – Sharing of hardware leads to contention, whether a program can use some hardware leaks information about other programs
4. **Stateful Functional Units** – Program's behavior can affect state of the functional units, and other programs can observe the output (which depends on the state)
5. **Prediction Units** – Prediction units can be used to build timing channels, this is different from prediction units being used as part of transient attacks
6. **Memory Hierarchy** – Data caching creates fast and slow execution paths, leading to timing differences depending on whether data is in the cache or not

Instruction with Different Execution Timing

Computer architecture principles of **pipelining** and **making common case fast** drive processor designs where certain operations take more time than others – program execution timing may reveal which instruction was used.

- Multi-cycle floating point operations vs. single cycle addition
- Execution time of a piece of code depends on the types of instructions it uses, especially, between different runs of software can distinguish from timing if different instructions were executed

Constant time software implementations strive to choose instructions to try to make software run in constant time independent of any secret values

- Instructions with different execution timing are easiest to deal with
- Other sources of timing differences make it more difficult or even not possible to make software run in constant time
    - Note, ”constant time” is not always same time, just that time is independent of secret values

Variable Instruction Timing

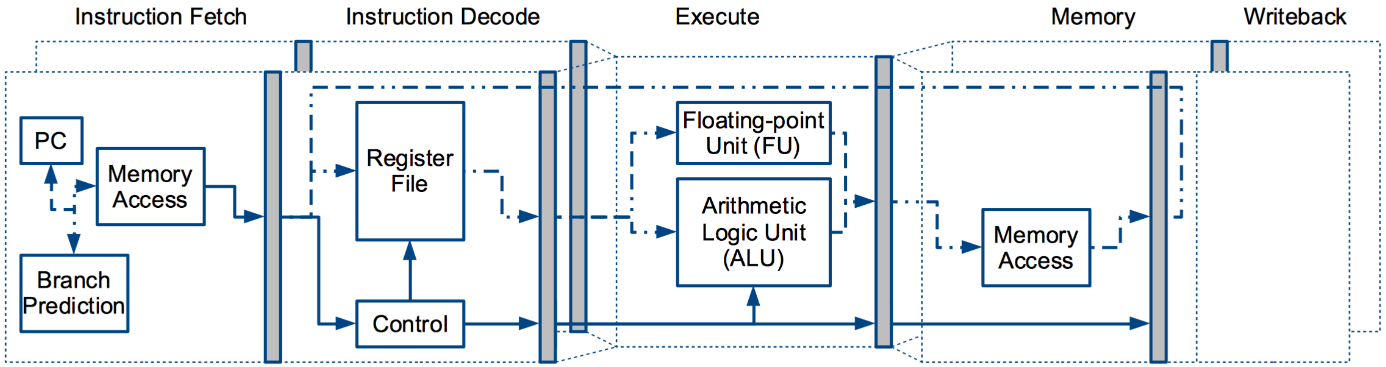
For a specific instruction, its timing depends on the state of the processor. Different state, or different execution history of instructions, affect timing of certain instructions:

- **Memory loads and stores**: memory access hitting in the cache vs. memory access going to DRAM
- **Multimedia instructions**: whether AVX unit is powered on or not affects timing
- **Reading from special registers**: e.g., random number generator slows down if it is used a lot and entropy drops
- **Instructions that trigger some state cleanup**, e.g. interrupt latency for SGX enclaves depends on amount of data processor has to clean up and secure before handling the interrupt

Functional Unit Contention

Functional units within processor are re-used or shared to save on area and cost of the processor resulting in varying program execution.

- Contention for functional units causes execution time differences



Spatial or Temporal Multiplexing allows to dedicate part of the processor for exclusive use by an application

- Negative performance impact or need to duplicate hardware

Stateful Functional Units

Many functional units inside the processor keep some history of past execution and use the information for prediction purposes.

- Execution time or other output may depend on the state of the functional unit
- If functional unit is shared, other programs can guess the state (and thus the history)
- E.g. caches, branch predator, prefetcher, etc.

Flushing state can erase the history.

- Not really supported today
- Will have negative performance impact

Prediction Units

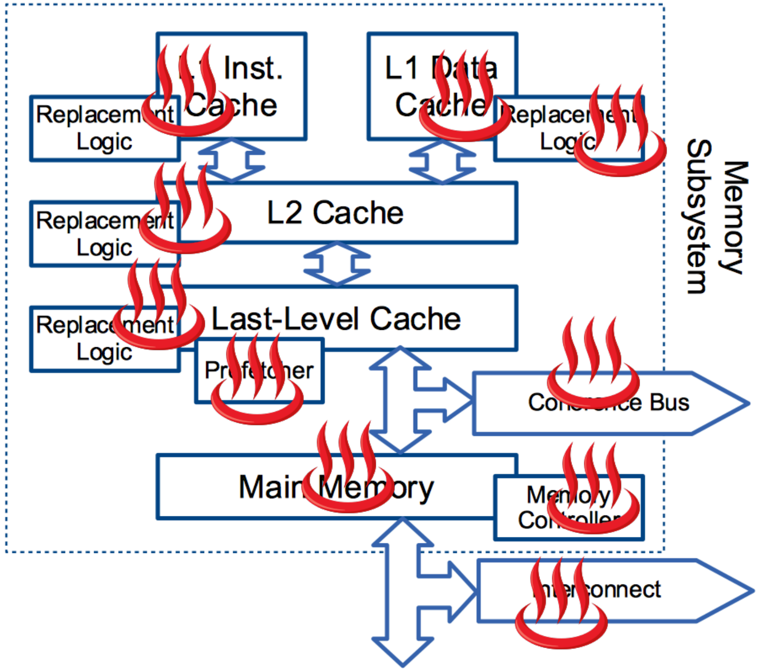
Prediction units can be used to build timing channels, this is different from prediction units being used as part of transient attacks.

- The **prediction units make prediction based on history** of executed instructions and the processor’s state
- The **prediction units are often shared** between threads running on the same core
- Victim’s or **sender’s execution history can affect the prediction observed by the attacker** thread, and the attacker observe the timing difference

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 area 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**



Emoji Image: https://www.emojione.com/emoji/2668

Timing Channels in Caches

Cache Timing Attacks Continue to Raise Concerns

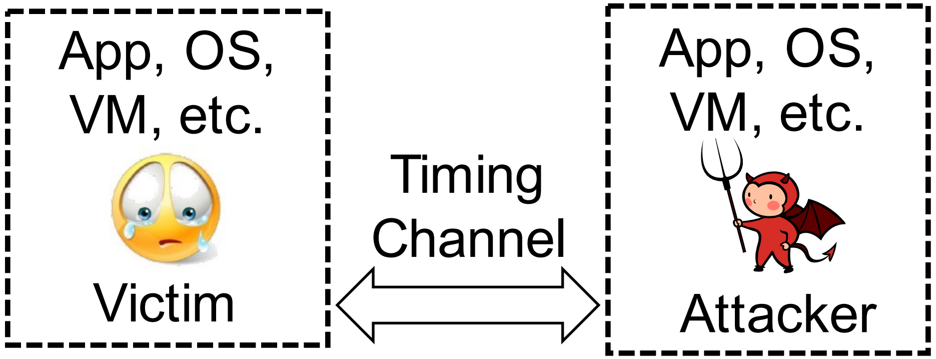
- There is renewed interest in timing attacks due to Transient Execution Attacks
- Most of them use **transient executions <u>and</u>** leverage **cache timing attacks**
- Variants using cache timing attacks (side or covert channels):

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



App, OS, VM, etc.

Victim

Timing Channel

App, OS, VM, etc.

Attacker

- **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

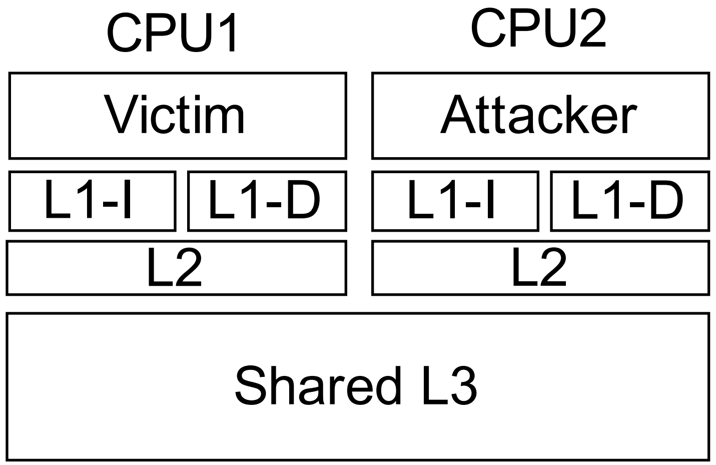
- **Side-channel attack vs. covert-channel attack**
  - Side channel: victim is not cooperating
  - Covert channel: victim (sender) works with attacker – easier to realize and higher bandwidth

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

Prime-Probe Attacks

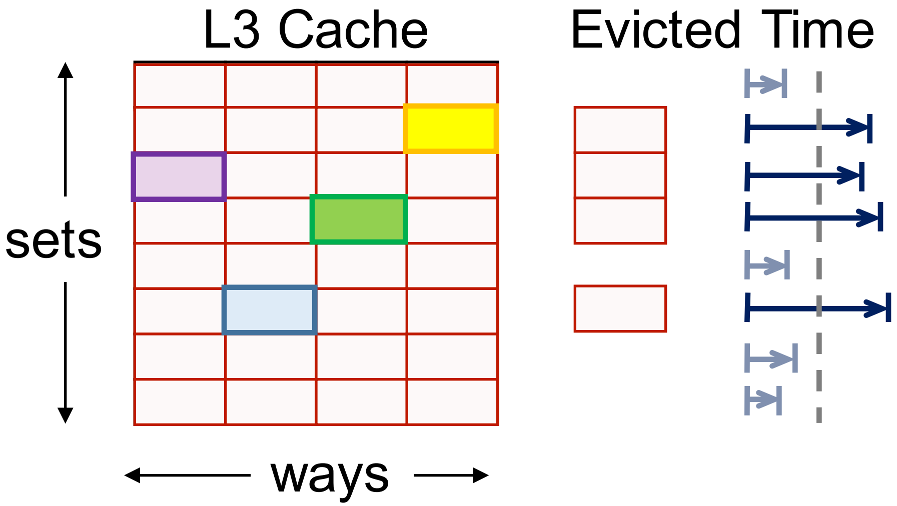
Osvik, D. A., Shamir, A., & Tromer, E, “Cache attacks and countermeasures: the case of AES”. 2006.

2- Victim accesses critical data



1- Attacker primes each cache set

3- Attacker probes each cache set (measure time)

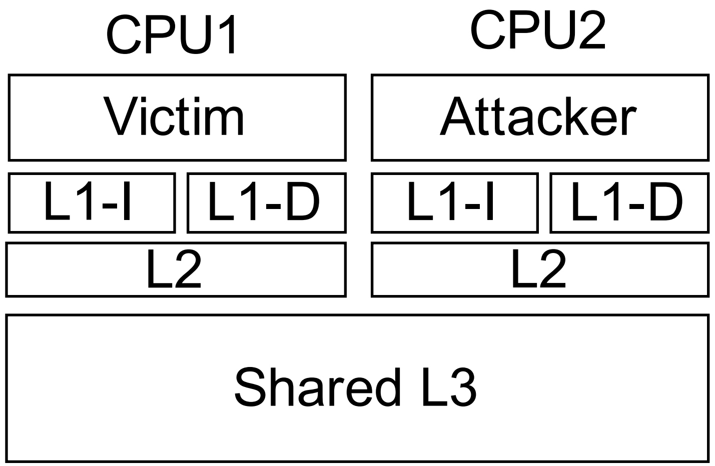


Data sharing is not needed

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



CPU1

CPU2

Victim

Attacker

L1-I

L1-D

L1-I

L1-D

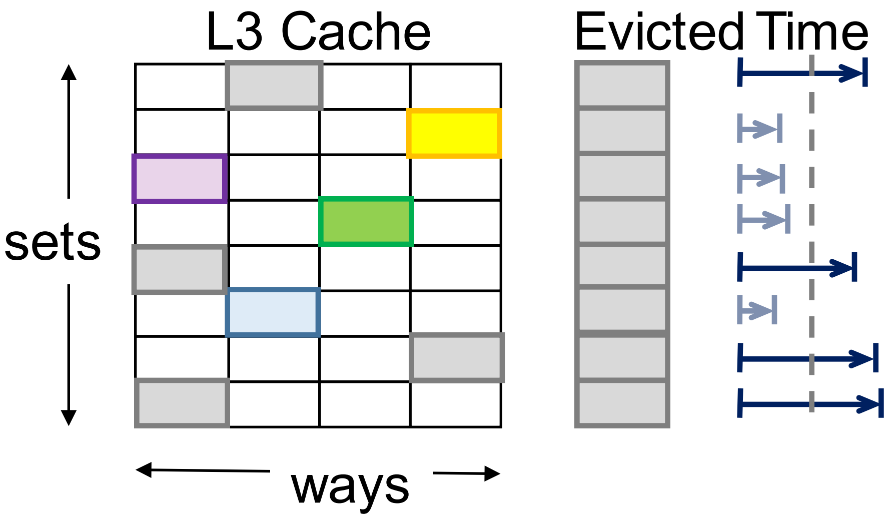
L2

L2

Shared L3

1- Attacker flushes each line in the cache

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



Data sharing is needed

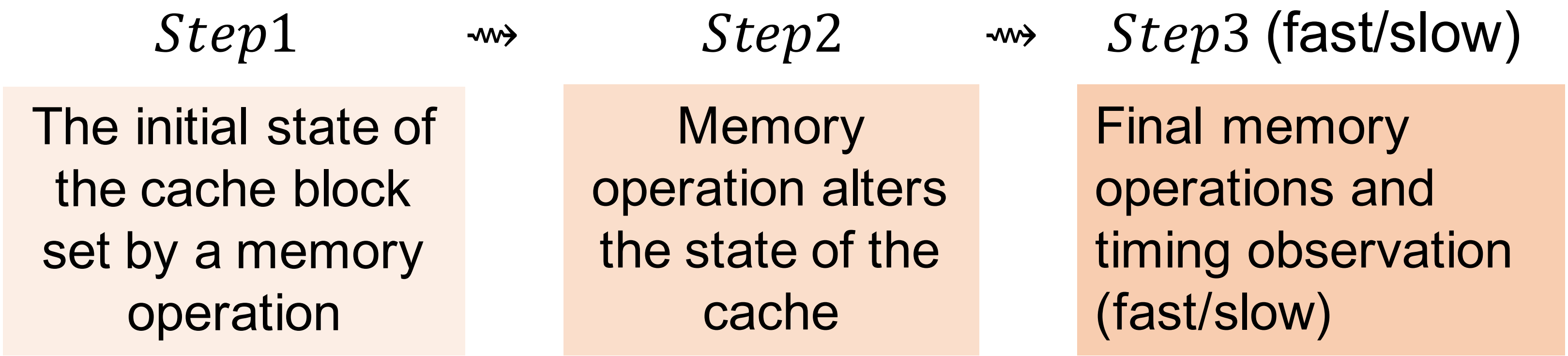
A Three-Step Model for Cache Timing Attack Modeling

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



$Step1 \rightsquigarrow Step2 \rightsquigarrow Step3$ (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

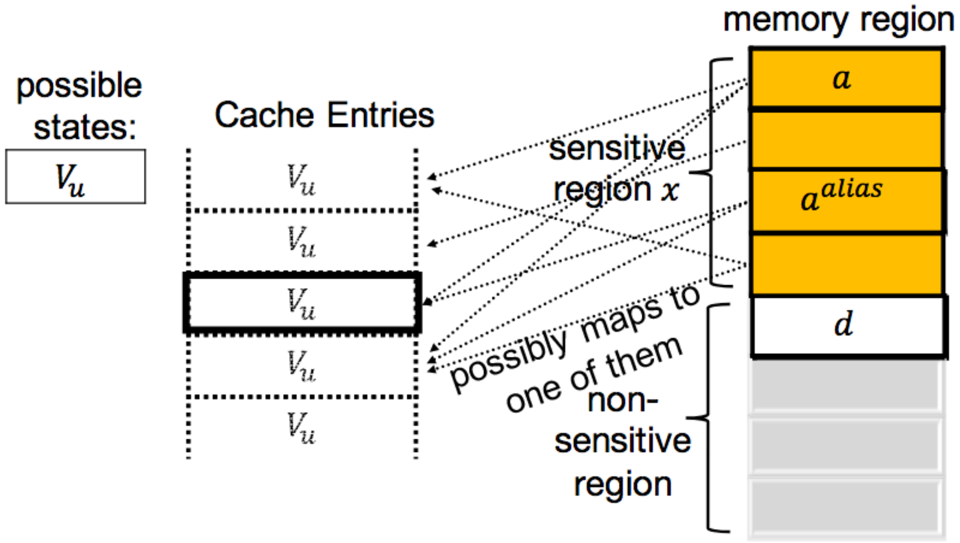
- **There are 72 possible cache timing attack types**

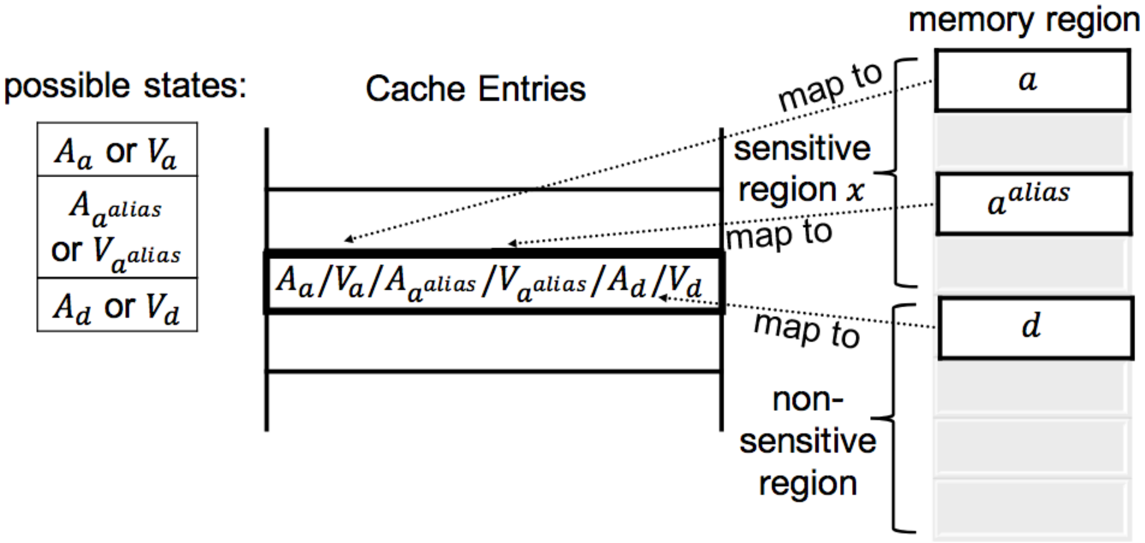
A Three-Step Model for Cache Timing Attack Modeling

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

There are 17 possible states for each step in the model:

V_u, A_a, V_a, A_a^{alisa}, V_a^{alias}, A_d, V_d, …



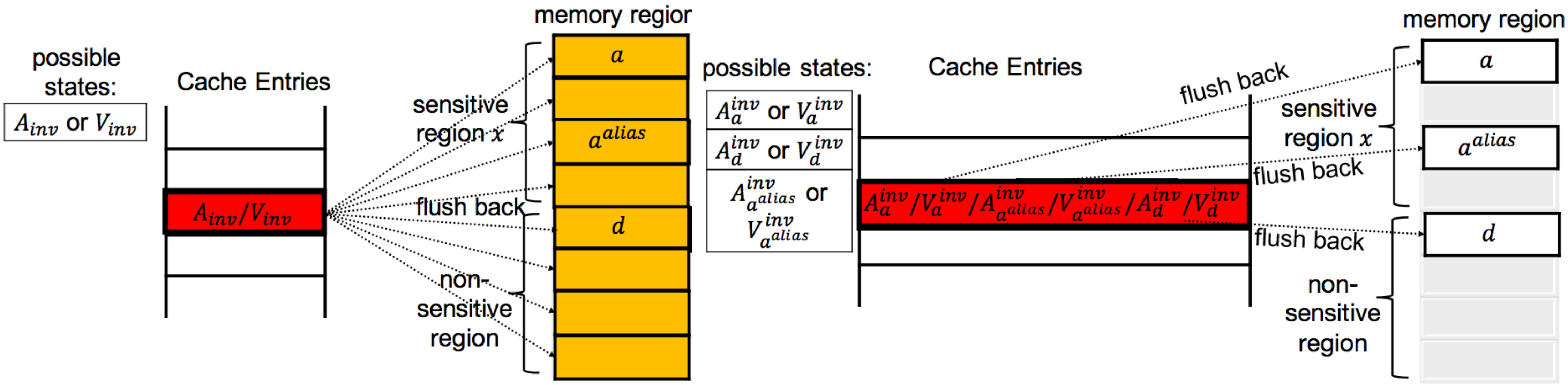


A Three-Step Model for Cache Timing Attack Modeling

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

There are 17 possible states for each step in the model:

V_u, A_a, V_a, A_a^{alisa}, V_a^{alias}, A_d, V_d, A^{inv}, V^{inv}, A_a^{inv}, V_a^{inv}, $A_a^{aliasinv}$, $V_a^{aliasinv}$, A_d^{inv}, V_d^{inv}, …

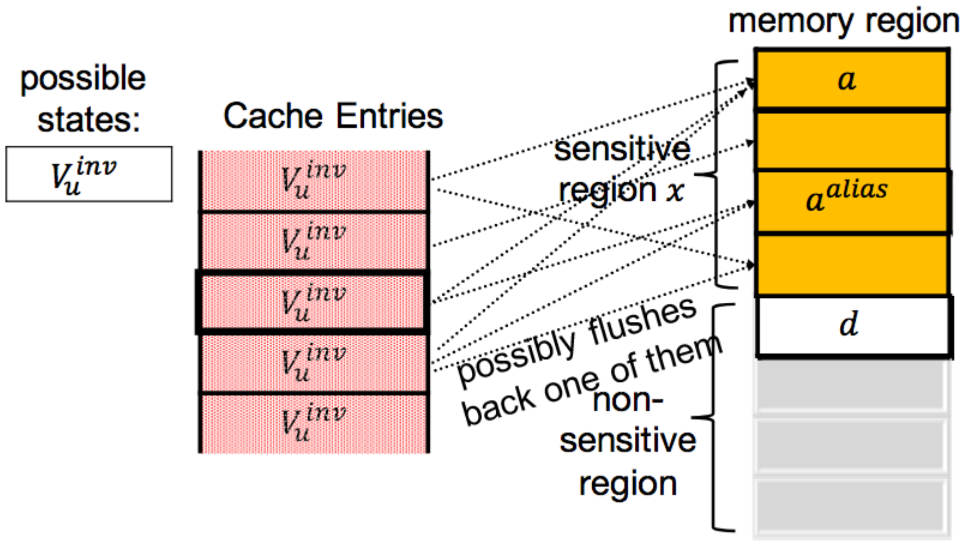


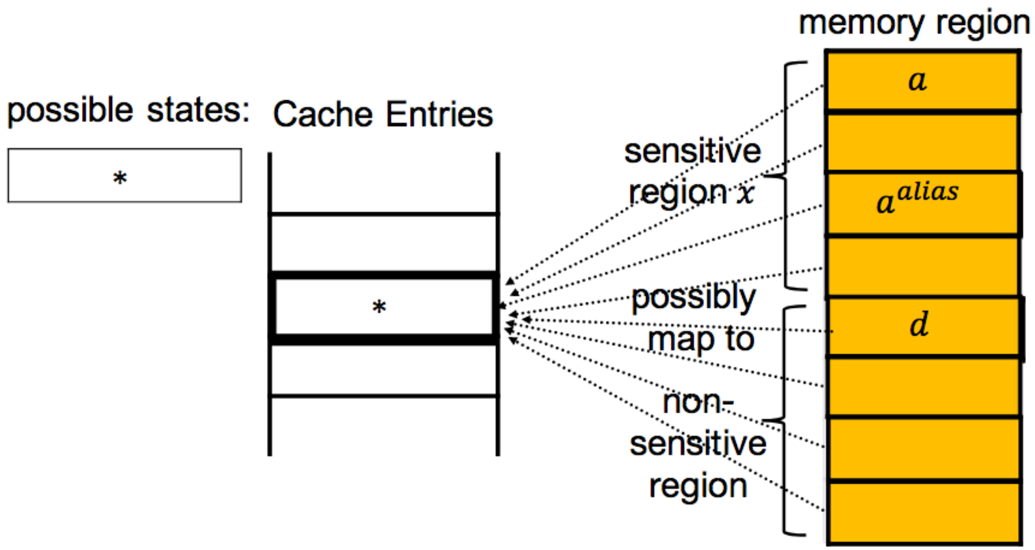
A Three-Step Model for Cache Timing Attack Modeling

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

There are 17 possible states for each step in the model:

V_u, A_a, V_a, A_a^{alisa}, V_a^{alias}, A_d, V_d, A^{inv}, V^{inv}, A_a^{inv}, V_a^{inv}, $A_a^{aliasinv}$, $V_a^{aliasinv}$, A_d^{inv}, V_d^{inv}, V_u^{inv}, *

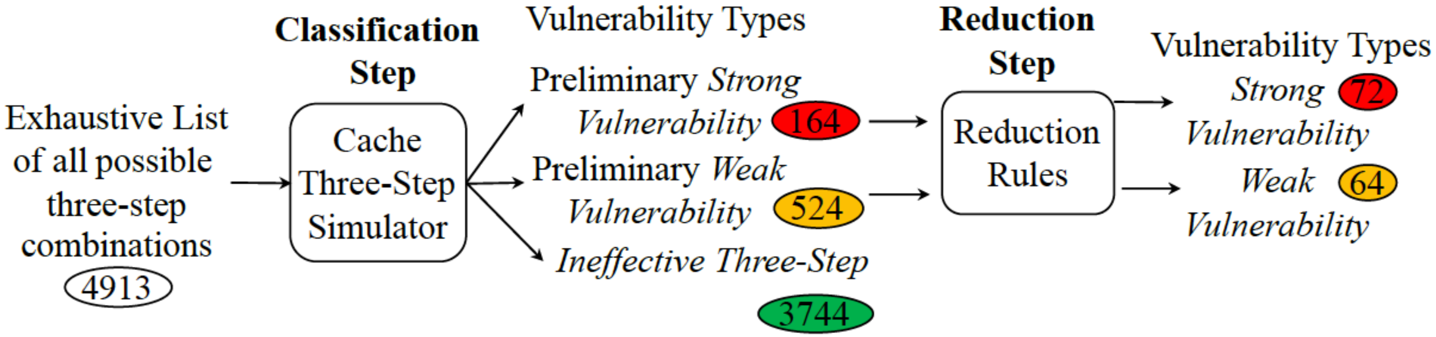


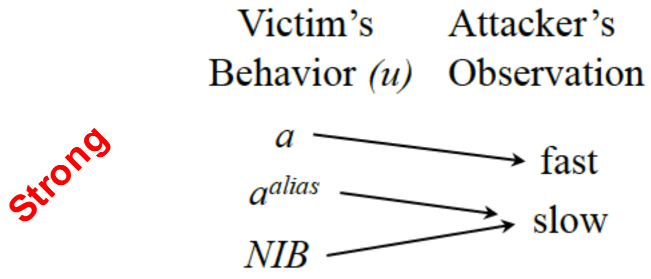


A Three-Step Model for Cache Timing Attack Modeling

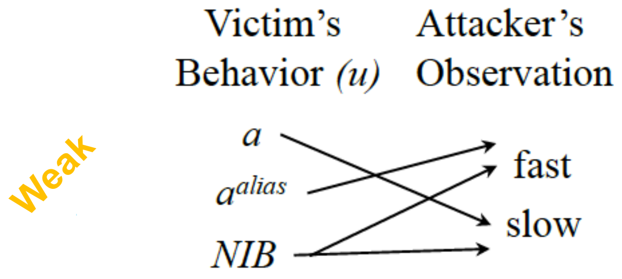
“Analysis of Secure Caches and Timing-Based Side-Channel Attacks”, S. Deng, et al., 2019
“Cache Timing Side-Channel Vulnerability Checking with Computation Tree Logic”, S. Deng, et al., 2018

- Exhaustively evaluate all 17 (step1) * 17 (step2) * 17 (step3) = 4913 three-step patterns
- Used cache three-step simulator and reduction rules to find all the strong effective vulnerabilities
- In total 72 strong effective vulnerabilities were derived and presented

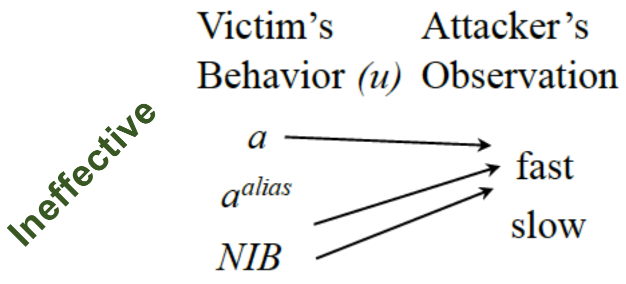




Example Vulnerability: $\boxed{V_d \rightsquigarrow V_u \rightsquigarrow A_a}$



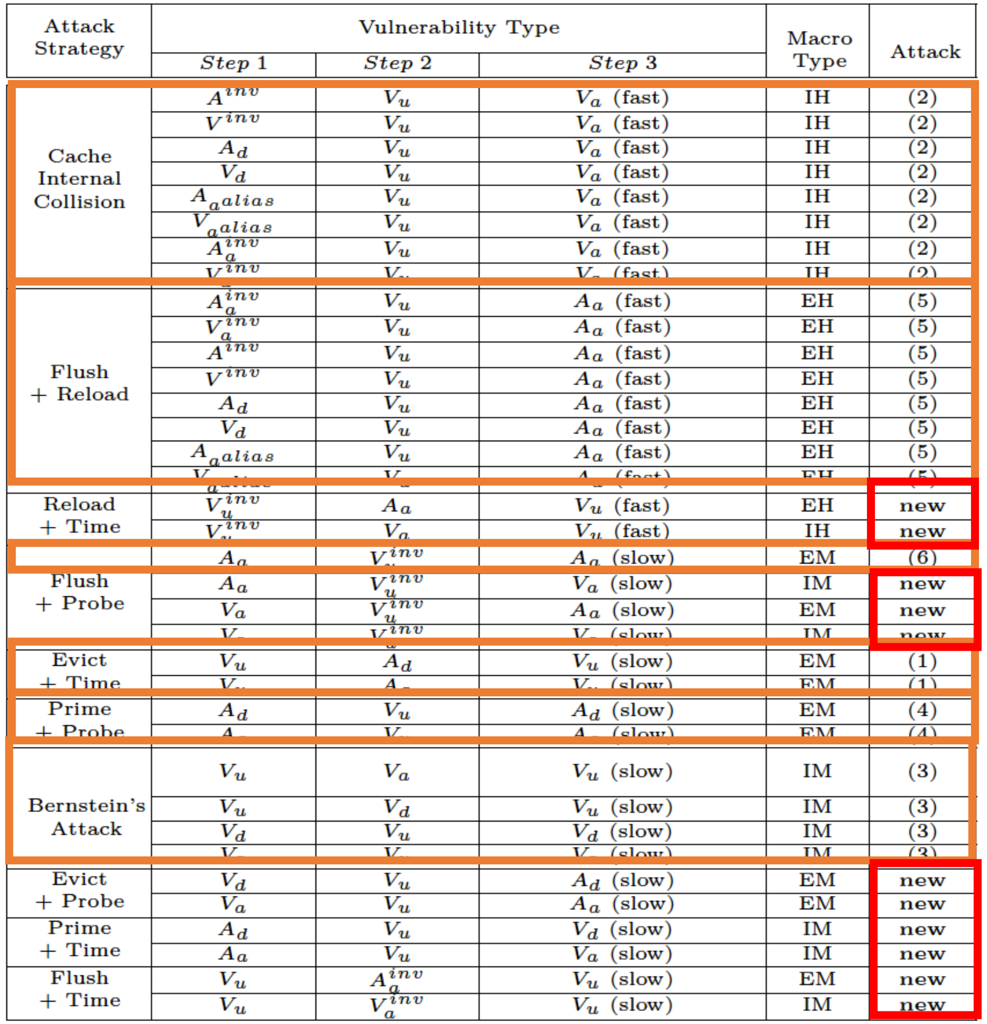
Example Vulnerability: $\boxed{* \rightsquigarrow V_u \rightsquigarrow A_a^{invalid}}$



Example Vulnerability: $\boxed{V_d \rightsquigarrow V_u^{invalid} \rightsquigarrow V_d}$

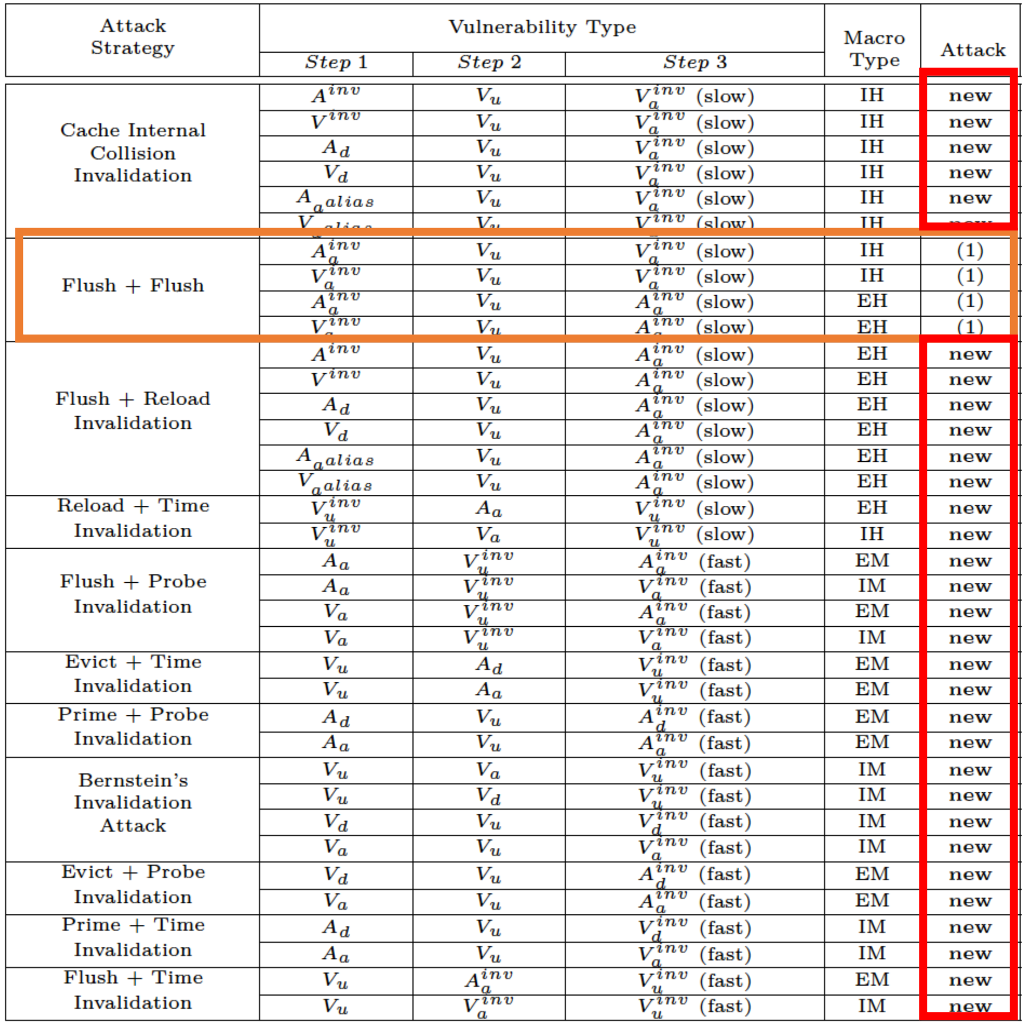
Exhaustive List of Cache Timing Side- Channel Attacks

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].

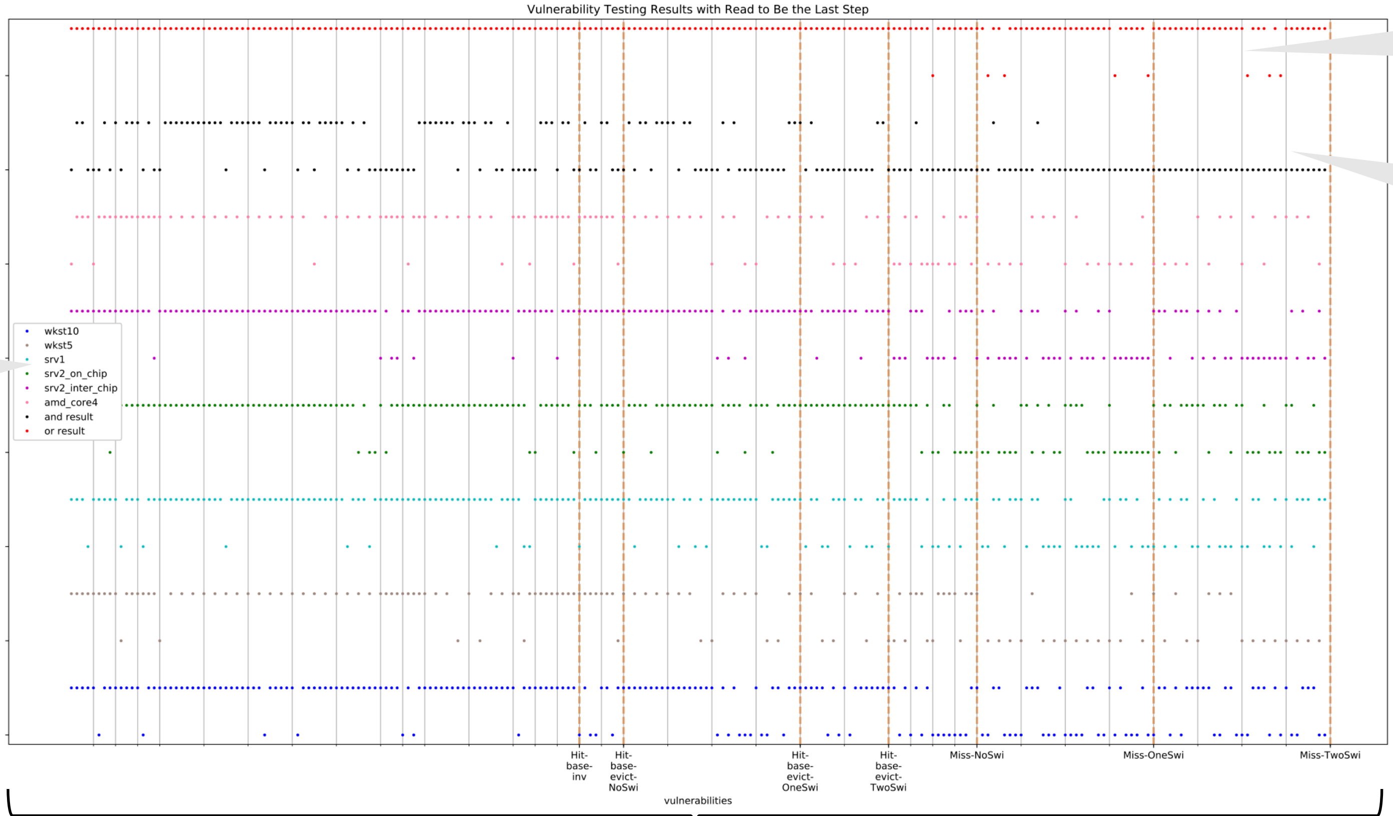


Attack Strategy	Vulnerability Type			Macro Type	Attack
	Step 1	*Step* 2	*Step* 3		
Cache Internal Collision Invalidation	A^{inv}	V_u	V_a^{inv} (slow)	IH	new
	V^{inv}	V_u	V_a^{inv} (slow)	IH	new
	A_d	V_u	V_a^{inv} (slow)	IH	new
	V_d	V_u	V_a^{inv} (slow)	IH	new
	$A_{a\,alias}$	V_u	V_a^{inv} (slow)	IH	new
	$V_{a\,alias}$	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].

Security Micro-Benchmarks for Cache Timing Attacks

- On-going research in our group looks into development of open-source benchmarks for quantifying cache timing attacks



Different CPUs tested

Tests that pass for one or more CPUs

Tests that pass for all CPUs

Tests for individual CPUs

Three-step tests: 72 * 8 or 16 variants

Timing Channels in Other Parts of Mem. Hierarchy

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

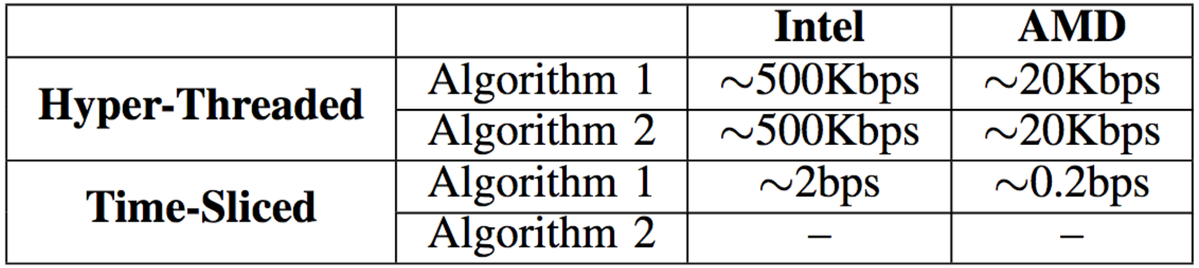
LRU Timing Attacks

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	–	–

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

```
void _gcry_mpi_powm (gcry_mpi_y_ res,
                gcry_mpi_t base, gcry_mpi_t expom gcry_mpi_t_mod)
{
        mpi_ptr_t rp, xp; /* pointers to MPI data */
        mpi_ptr_t tp;
        ...
        for(;;) {
                /* For every exponent bit in expo*/
                _gcry_mpih_sqr_n_basecase(xp, rp);
                if(secret exponent || e bit is1) {
                      /* unconditional multiply if exponent is
                       * secret to mitigate FLUSH+RELOAD
                       */
                      _gcry_mpih_mul(xp, rp);
                }
                if(e_bit_is1) {
                       /*e bit is 1, use the result*/
                       tp = rp; rp = xp; xp = tp;
                       rsize = xsize;
                }
        }
}
```

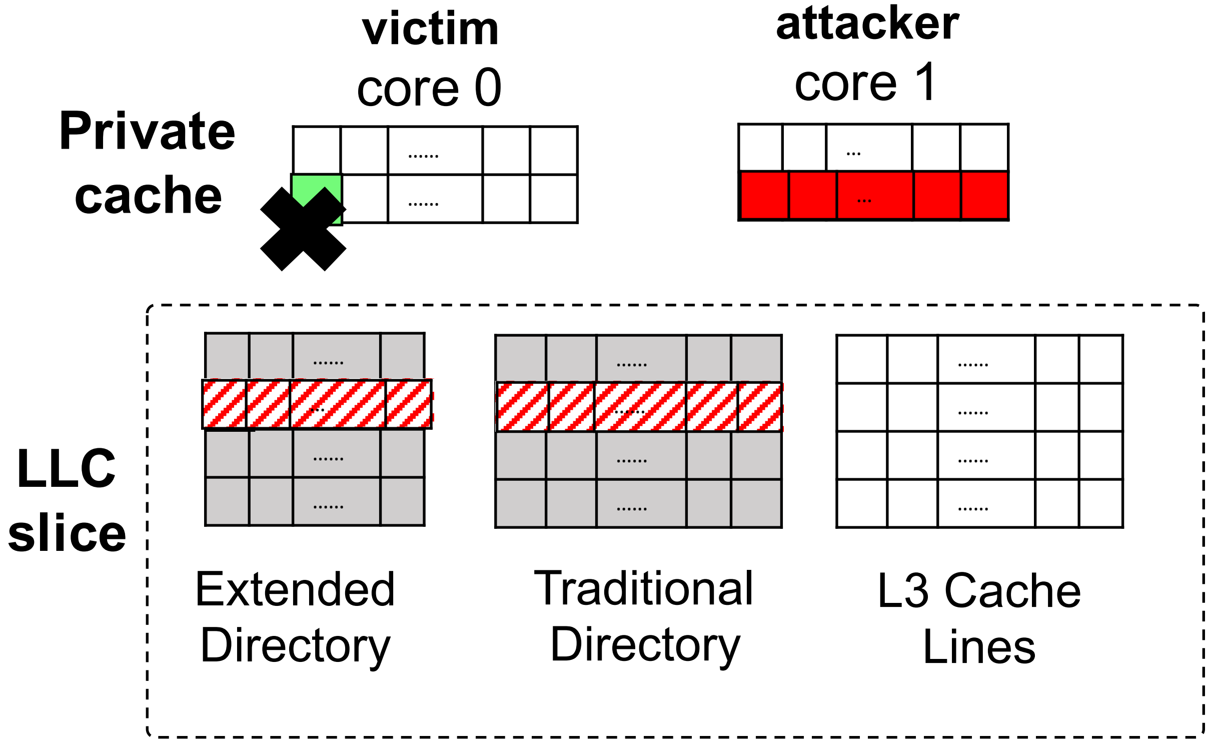
Existing TLBleed work can extract the cryptographic key from the RSA public-key algorithm* with a 92% success rate.

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

Cache Directory Timing Attacks

Mengjia Yan et al. “SecDir: A Secure Directory to Defeat Directory Side-Channel Attacks”. 2019

- Directories have been shown to be vulnerable to side-channel attacks
- Every cache line in the cache hierarchy has an associated directory entry
- Directory attack outline:
  1. Directory conflict
  2. Evicts victim’s directory entry
  3. Evicts victim’s cache line



Animation adapted from slides by Mengjia Yan

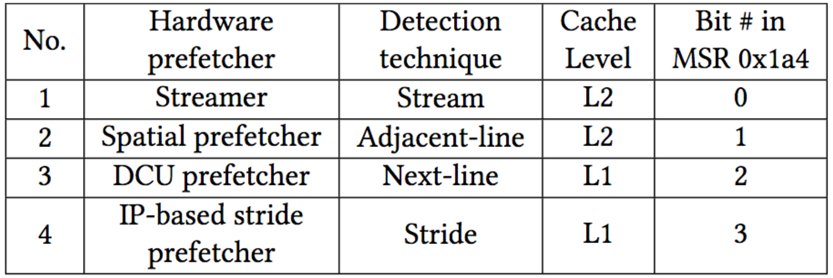
Timing Attacks in Prefetchers

"Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage", Y. Shin, et al., CCS 2018

Prefetchers have been abused for timing attacks

- E.g. IP-based stride prefetcher, has been used to break cryptographic algorithm implementations
- Any cryptographic algorithm implementation that utilizes a lookup table is subject to the attack
  - Pattern of accesses in the table will be revealed by the data that is prefetched
- Prefetching is a type of prediction or speculation

Prefetchers in Intel processors



No.	Hardware prefetcher	Detection technique	Cache Level	Bit # in MSR 0x1a4
1	Streamer	Stream	L2	0
2	Spatial prefetcher	Adjacent-line	L2	1
3	DCU prefetcher	Next-line	L1	2
4	IP-based stride prefetcher	Stride	L1	3

Classical vs. Speculative Side-Channels

Side channels can now be classified into two categories:

- **Classical** – which do not require speculative execution
- **Speculative** – which are based on speculative execution

Difference is victim is not fully in control of instructions they execute (i.e. some instructions are executed speculatively)

Root cause of the attacks remains the same

State of functional unit is modified by victim and it can be observed by the attacker via timing changes

Defending classical attacks defends speculative attacks as well, but not the other way around

Focusing only on speculative attacks does not mean classical attacks are prevented, e.g. defenses for cache-based attacks

Timing Side Channels which Use Speculation

- Modern computer architectures gain performance by using prediction mechanisms:
  - Successful prediction = fast execution and performance gain
  - Mis-prediction = slow execution and performance loss

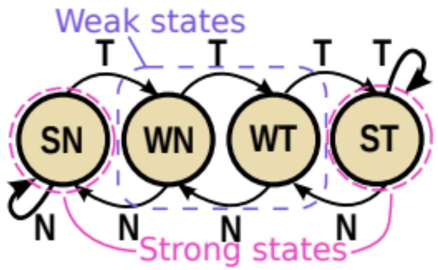
- The prediction units (e.g., branch predictor, prefetcher, memory disambiguation prediction, etc.) make prediction based on prior history of executed instructions and data
- The prediction units are often shared between threads in SMT cores
- Victim's execution history can affect the prediction observed by the attacker thread, and the attacker can observe the timing difference

- **These types of side channels are different from the transient executions attacks**
  - In transient execution attacks, secrets are accessed during mis-prediction
  - In timing side channels using speculation victim's behavior is leaked to the attacker, through the mis-prediction (or lack there of) by the attacker

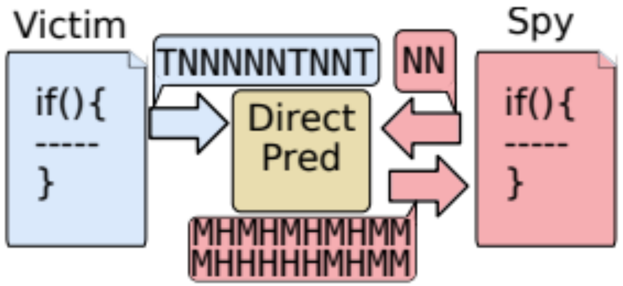
Timing Channels Through Pattern History Table

D. Evtyushkin, et al., “BranchScope: A New Side-Channel Attack on Directional Branch Predictor”, 2018
D. Evtyushkin, et al., “Covert Channels Through Branch Predictors: A Feasibility Study”, 2015

- Branch predictors Pattern History Table (PHT) is shared among all processes on a core, and is not flushed on context switches
  - The branch predictor stores its history in the form of a 2-bit saturating counter which is the PHT



- Attack on PHT using Prime+Probe strategy
  1. Prime the branch predictor by executing branches at specific address
  2. Let victim or sender run
  3. Observe the branch outcomes



Original	0	1	1	1	1	1	0	1	1	0
Spy Measurements	M M	H H	M H	H H	M H	H H	M M	H H	M M	M M
Decoded Results	0	1	1	1	1	1	0	1	0	0

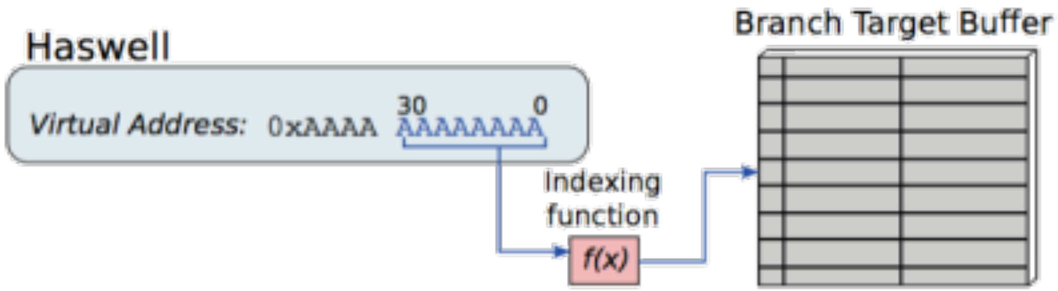
Spy Dict: MM, HM – 0
MH, HH – 1

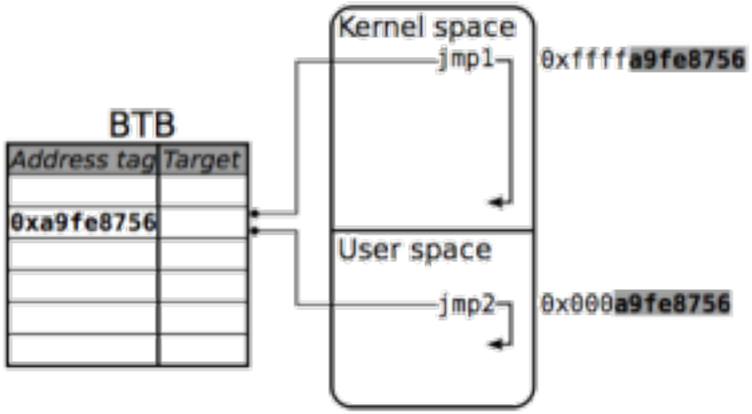
- Existing attacks:
  - Covert channels
  - Attacks on SGX enclave code

Timing Channels Through Branch Target Buffer

D. Evtyushkin, et al., “Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR”, 2016

- The Branch Target Buffer (BTB) stores target addresses of recently executed branch instructions, so that those addresses can be obtained directly from a BTB lookup
  - BTB can be indexed using some bits of the virtual address
  - Conflicts will exist when branches have same low-order bits





- Attack strategy
  1. Prime the BTB by executing branches or jumps at specific address
  2. Let victim or sender run
  3. Observe the branch outcomes

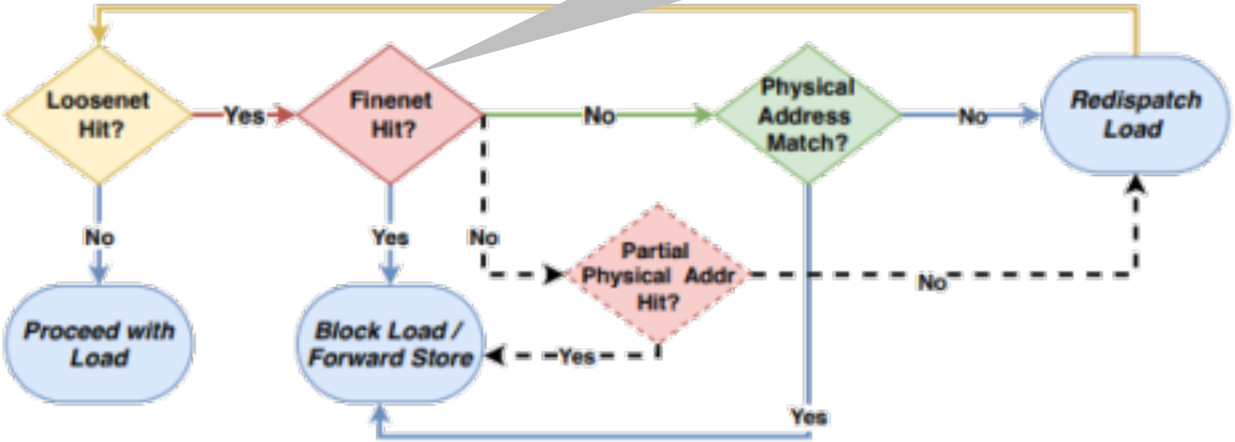
- Existing attacks:
  - Attack KASLR (Kernel address space layout randomization)

Timing Channels Through Memory Disambiguation

S. Islam, et al., “SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks”, 2019

- The processor can execute loads speculatively before the stores finish
  - Forward the data of a preceding store to the load if there is a potential dependency
  - Later check if the dependency was true
  - May not use all address bits or check permissions for fast execution

Loosenet = lower address comparision logic
Finenet = upper address comparison logic



- Existing attacks:
  - Leakage of the physical address mapping
  - Efficient eviction set finding for Prime+Probe attacks in L3 caches
  - Helps to construct DRAM row conflicts for Rowhammer type attacks

Secure Hardware Caches

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 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

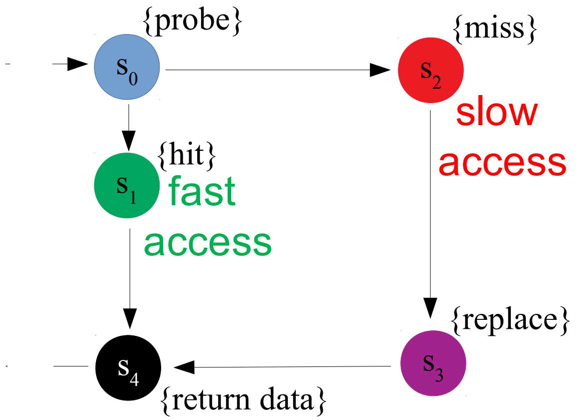
Deng, S., Xiong, W., Szefer, J., "Analysis of Secure Caches and Timing-Based Side-Channel Attacks", 2019

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

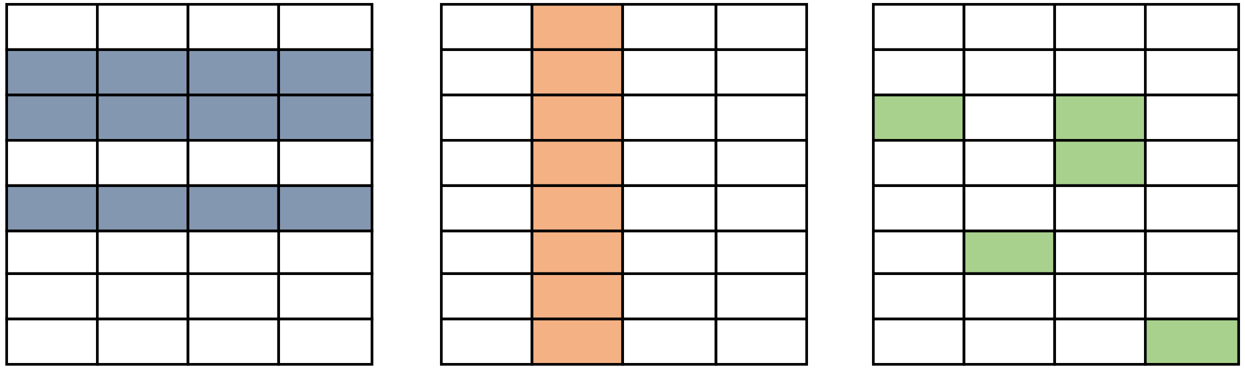
Memory reuse conditions

- 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)



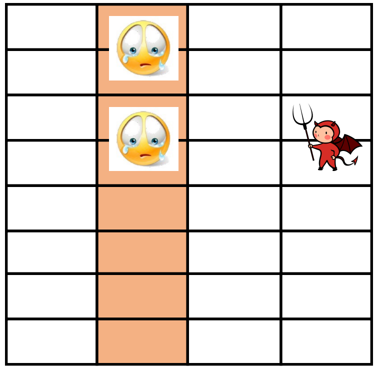
Partitioning

- **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



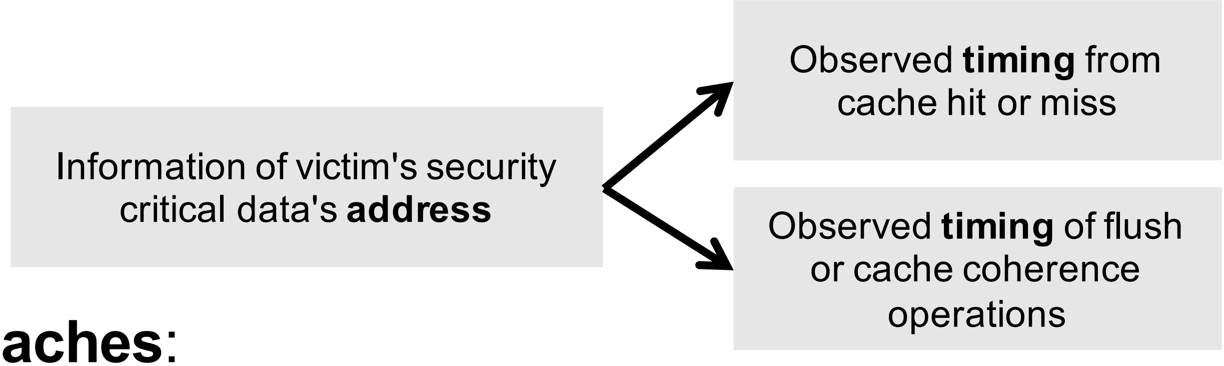
Partitioning (cont.)

- **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

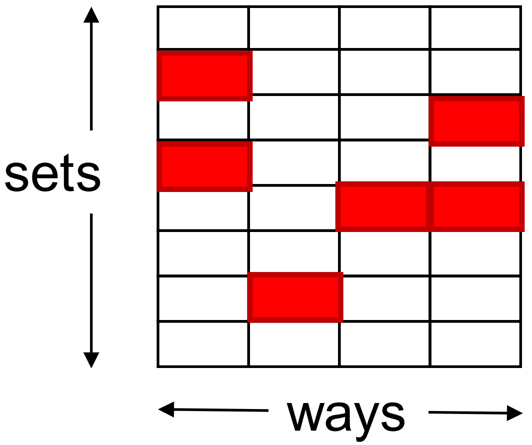

Information of victim's security critical data's **address**
Observed **timing** from cache hit or miss
Observed **timing** of flush or cache coherence operations

- **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

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

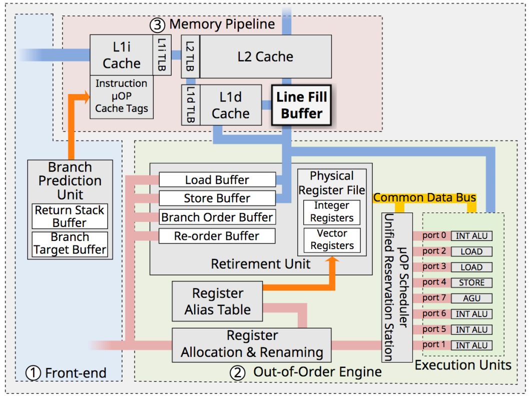
	SP*	SecVerilog	SecDCP	NoMo	SHARP	Sanctum	MI6	InvisiSpec	CATalyst	DAWG	RIC	PL	RP	Newcache	Random Fill	CEASER	SCATTER	Non Det.
Perf.	1%	-	12.5% better over SP cache	1.2% avr., 5% worst	3%-4%	-	-	reduce slowdown of Spectre from 74% to 21%	average slowdown of 0.7% for SPEC and 0.5% for PARSEC	L1 and L2 most 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 performance optimization	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%	-	-	-	-	-	-	-

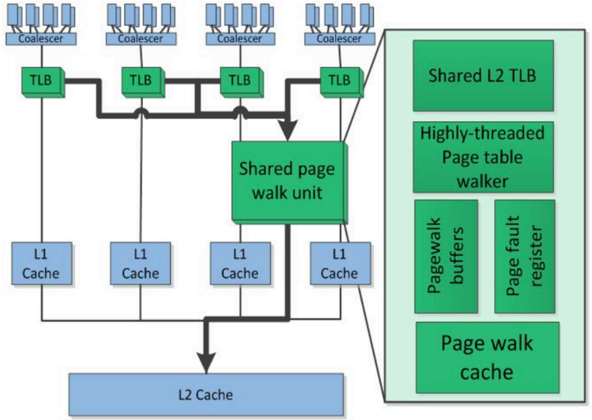
Secure Buffers, TLBs, and Directories

Buffers

Figures from Rogue In-Flight Data Load paper and UW-Madison CS slides

- Various buffers exist in the processor which are used to improve performance of caches and TLBs
- **Main types of buffers in caches:**
  - Line Fill Buffer (L1 cache ↔ L2 cache)
  - Load Buffer (core ↔ cache)
  - Store Buffer (core ↔ cache)
  - Write Combining Buffers (for dirty cache lines before store completes)
  - … (more could be undesclosed)
- **Main types of buffers in TLBs:**
  - Page Walk Cache





Secure Buffers

- Various buffers store data or memory translation based on the history of the code executed on the processor
- **Hits and misses in the buffers can potentially be measured and result in timing attacks**
  - This is different from recent MDS attacks, which abuse the buffers in another way: MDS attacks leverage the fact that data from the buffers is sometimes forwarded without proper address checking during transient execution
- **Towards secure buffers**
  - No specific academic proposal (yet)
  - **Partitioning** – can partition the buffers, already some are per hardware thread
  - **Randomization** – can randomly evict data from the buffers or randomly bring in data, may not be possible
  - Add **new instructions to conditionally disable some of the buffers**

TLBs

Deng, S., et al., “Secure TLBs”, ISCA 2019.

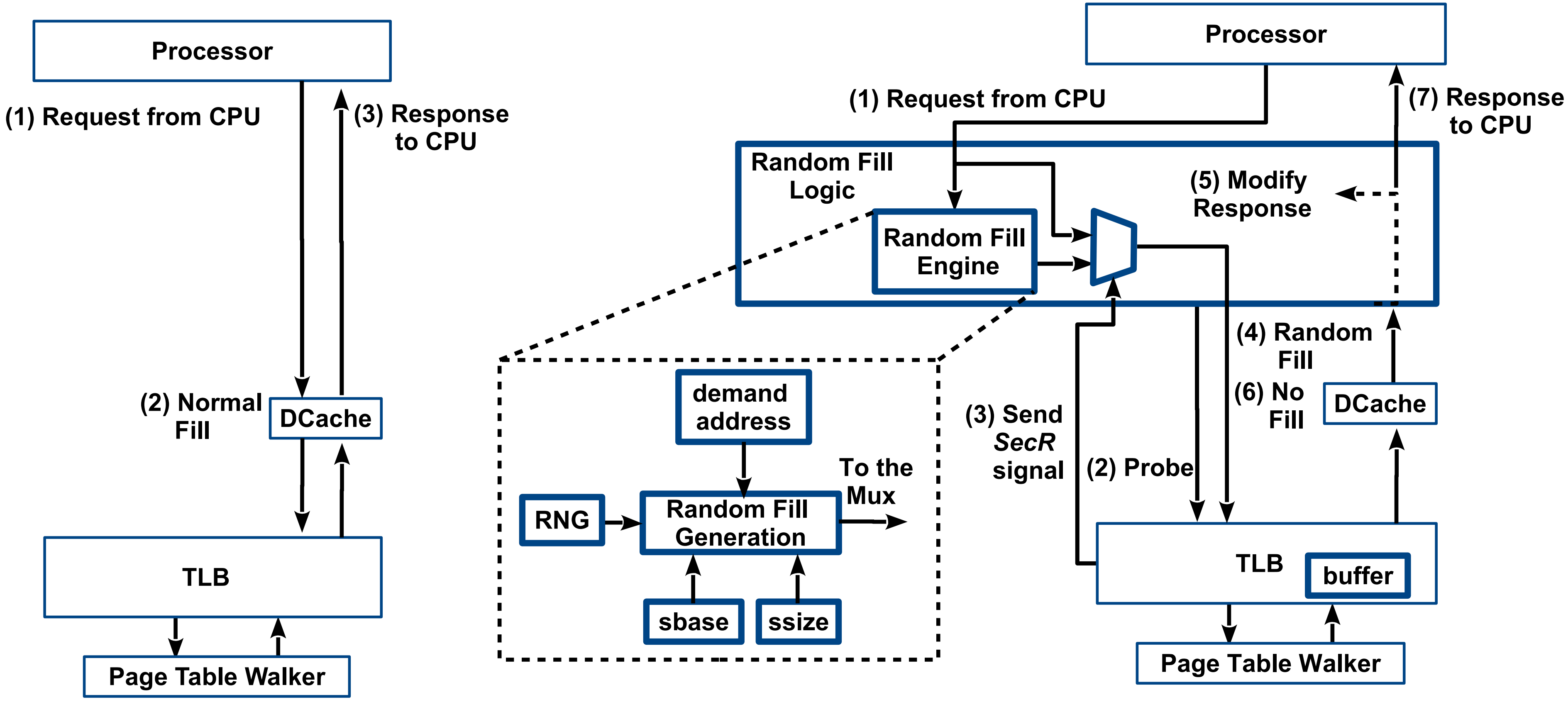
- All timing-based channels in microarchitecture pose threats to system security, and all should be mitigated
- TLBs are cache-like structures, which exhibit fast and slow timing based on the request type and the current contents of the TLB
  - Contents of the TLB is affected by past history of executions
  - Can leak information about other processes

- Timing variations due to hits and misses exist in **TLBs** and can be leveraged to build **practical timing-based attacks**:
  - TLB timing attacks are triggered by memory translation requests, not by direct accesses to data
  - TLBs have more complicated logic, compared to caches, for supporting various memory page sizes
  - Further, defending cache attacks does not protect against TLB attacks

Secure TLBs

Deng, S., et al., "Secure TLBs", ISCA 2019.

- Random Fill Engine and RF TLB microarchitecture.

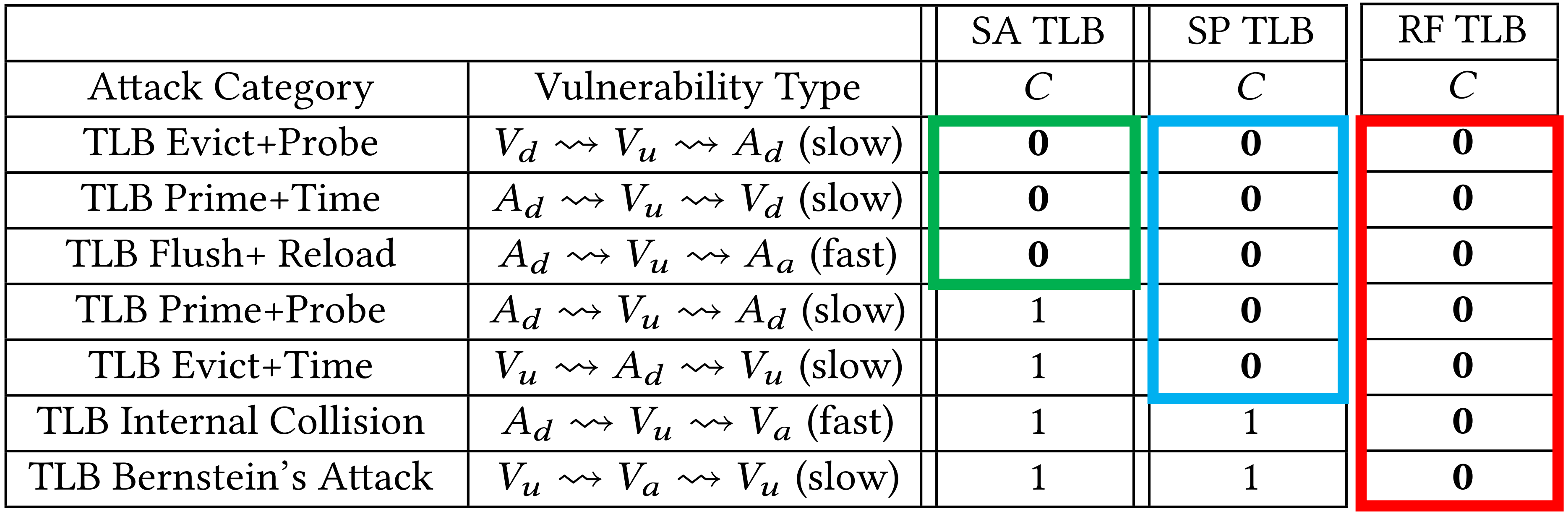


Probe ➡ Random Fill ➡ No fill

Secure TLBs

Deng, S., et al., “Secure TLBs”, ISCA 2019.

- Regular **Set-Associative TLBs** can prevent external hit-based vulnerabilities and vulnerabilities requiring getting hit for different processes
- **Static-Partitioned TLB** can prevent more external miss-based vulnerabilities than SA TLB
- **Random-Fill TLB** can prevent all types of vulnerabilities



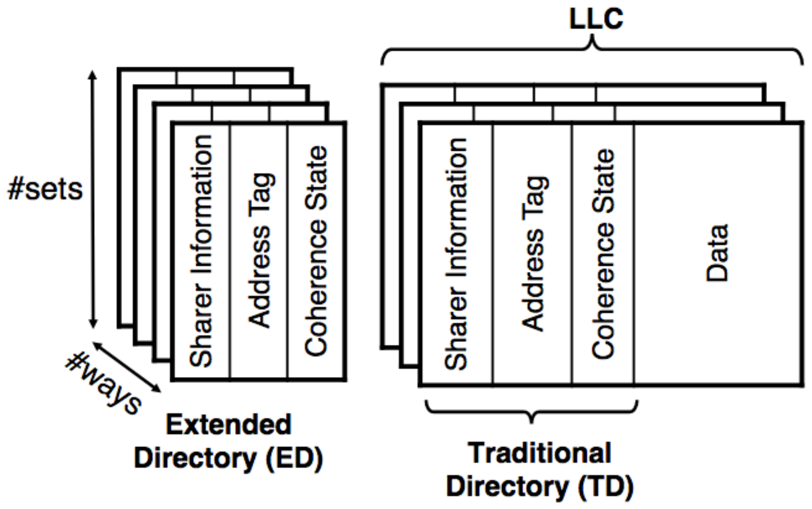
		SA TLB	SP TLB	RF TLB
Attack Category	Vulnerability Type	C	C	C
TLB Evict+Probe	$V_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow)	**0**	**0**	**0**
TLB Prime+Time	$A_d \rightsquigarrow V_u \rightsquigarrow V_d$ (slow)	**0**	**0**	**0**
TLB Flush+ Reload	$A_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	**0**	**0**	**0**
TLB Prime+Probe	$A_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow)	1	**0**	**0**
TLB Evict+Time	$V_u \rightsquigarrow A_d \rightsquigarrow V_u$ (slow)	1	**0**	**0**
TLB Internal Collision	$A_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	1	1	**0**
TLB Bernstein’s Attack	$V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow)	1	1	**0**

- *Evaluated on a 3-step model for TLBs; model and list of all attack types are in the cited paper.*

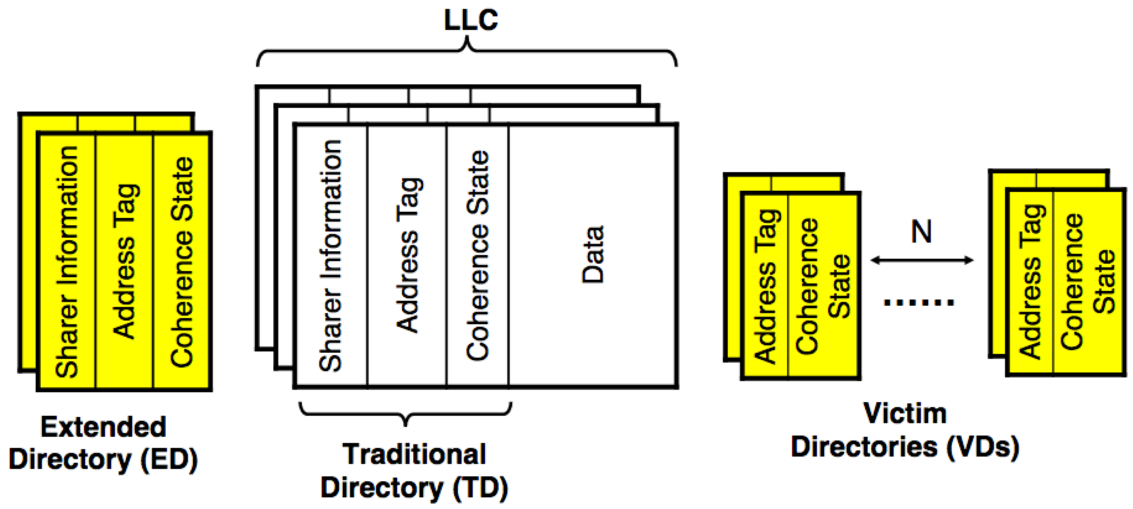
Secure Directories

Deng, S., et al., “Secure TLBs”, ISCA 2019.

- Directories are used for cache coherence to keep track of the state of the data in the caches
- By forcing directory conflicts, an attacker can evict victim directory entries, which in turn triggers the eviction of victim cache lines from private caches
- **SecDir** re-allocates directory structure to create per-core private directory areas used in a victim-cache manner called Victim Directories; the partitioned nature of Victim Directories prevents directory interference across cores, defeating directory side-channel attack.



Intel Directory in Skylake CPUs



Secure Directory (SecDir)

Mitigation Overheads

- Performance overhead of the different secure components and the benchmarks used for the evaluation

	Performance Overhead	Benchmark
Secure Buffers	n/a	n/a
Secure TLBs [S. Deng, et al., 2019]	For SR TLB: IPC 1.4%, MPKI 9%	SPEC2006
SecDir [M. Yan, et al., 2019]	few % (some benchmarks faster some slower)	SPEC2006

Summary of Timing Attacks and Defenses

- In response to timing attacks on caches, and other parts of the processor’s memory hierarchy, many secure designs have been proposed
- Caches are most-researched, from which we learned about two main defense techniques:
  - **Partitioning**
  - **Randomization**
- The techniques can be applied to other parts of the processor: Buffers, TLBs, and Directories

- Most claim modest overheads of few % on SPEC2006 workloads
  - Unclear of overhead on real-life applications

- Other parts of memory hierarchy are still vulnerable: memory bus contention, for example

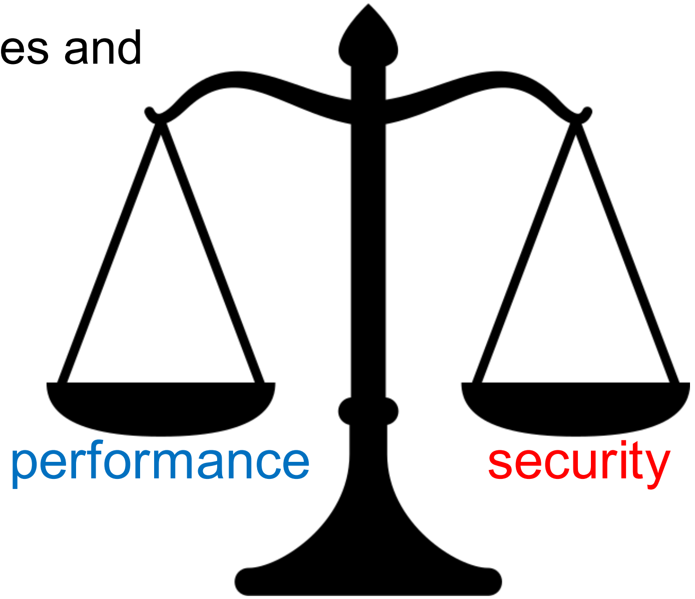
Research Challenges

- **Balance tradeoff between performance and security**
  - Curse of quantitative computer architecture: focus on performance, area, power numbers, but no easy metric for security – designers focus on performance, area, power numbers since they are easy to show ”better” design, there is no clear metric to say deign is “more secure” than another design

- **Evaluation on simulation vs. real machines**
  - Simulation workloads may not represent real systems, performance impact of security features is unclear
  - Real systems (hardware) can’t be easily modified to add new features and test security

- **Formal verification of the secure feature implementations**
  - Still limited work on truly showing design is secure
  - Also, need more work on modelling all possible attacks, e.g. the three-step model

- **Side channels can be used to detect or observe system’s operation**



Design of Secure Processor Architectures

Part 3

Transient Exec. Attacks and Hardware Defenses

Jakub Szefer
Assistant Professor
Dept. of Electrical Engineering
Yale University

CHES 2019 – August 25, 2019

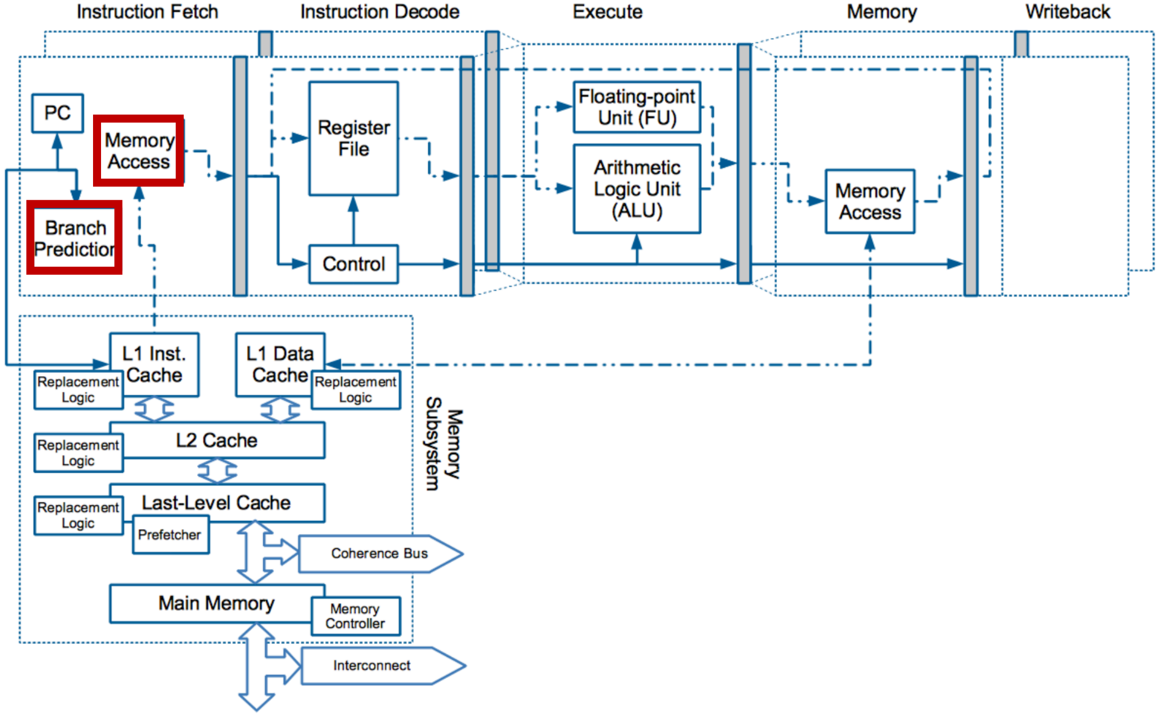
Slides and information available at: https://caslab.csl.yale.edu/tutorials/ches2019/

Transient Execution Attacks

Prediction and Speculation in Modern CPUs

Prediction is one of the six key features of modern processors

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



Transient Execution Attacks

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

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

Transient Execution Attacks = Transient Execution + Covert or Side Channel

Example: Spectre Bounds Check Bypass Attack

Example of Spectre variant 1 attack:

Victim code:

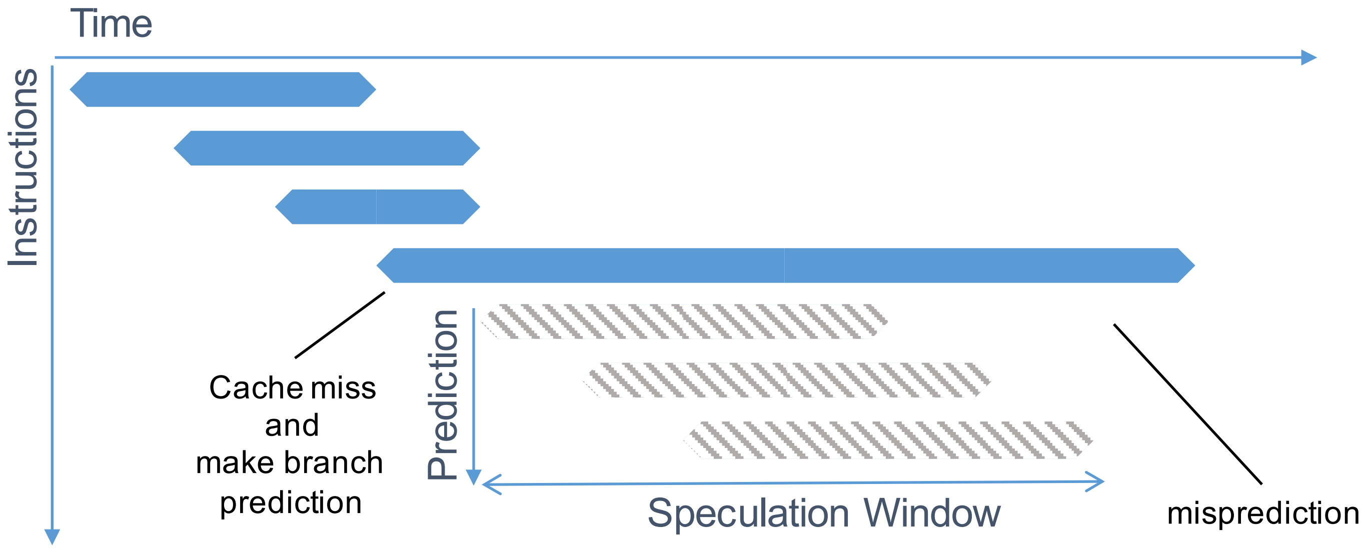
```
struct array *arr1 = ...;
struct array *arr2 = ...;
unsigned long offset = ...;
if (offset < arr1->len) {
  unsigned char value = arr1->data[offset];
  unsigned long index = value;
  unsigned char value2 = arr2->data[index * size];
...
```

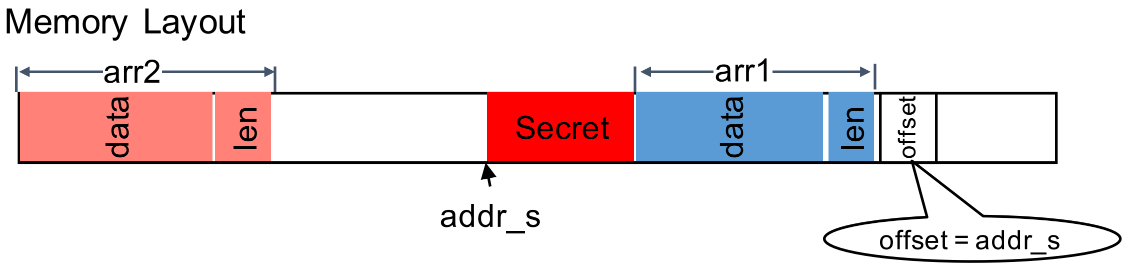
Probe array (side channel)

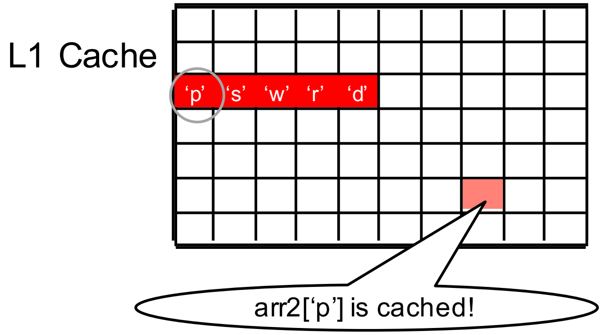
Controlled by the attacker

arr1->len is not in cache

change the cache state







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

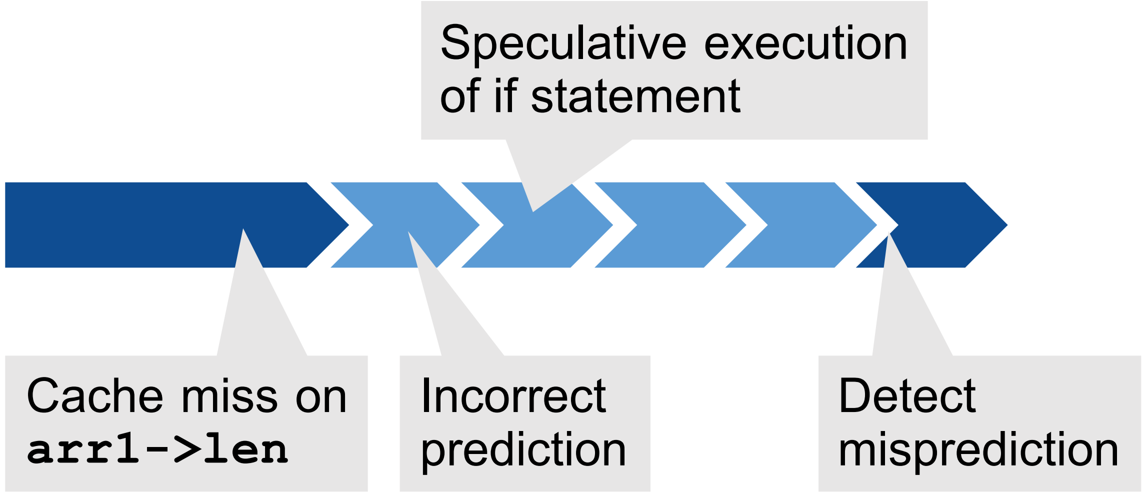
Transient Execution – due to Prediction

***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];
  ...
}
```



Speculative execution of if statement

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

Incorrect prediction

Detect misprediction

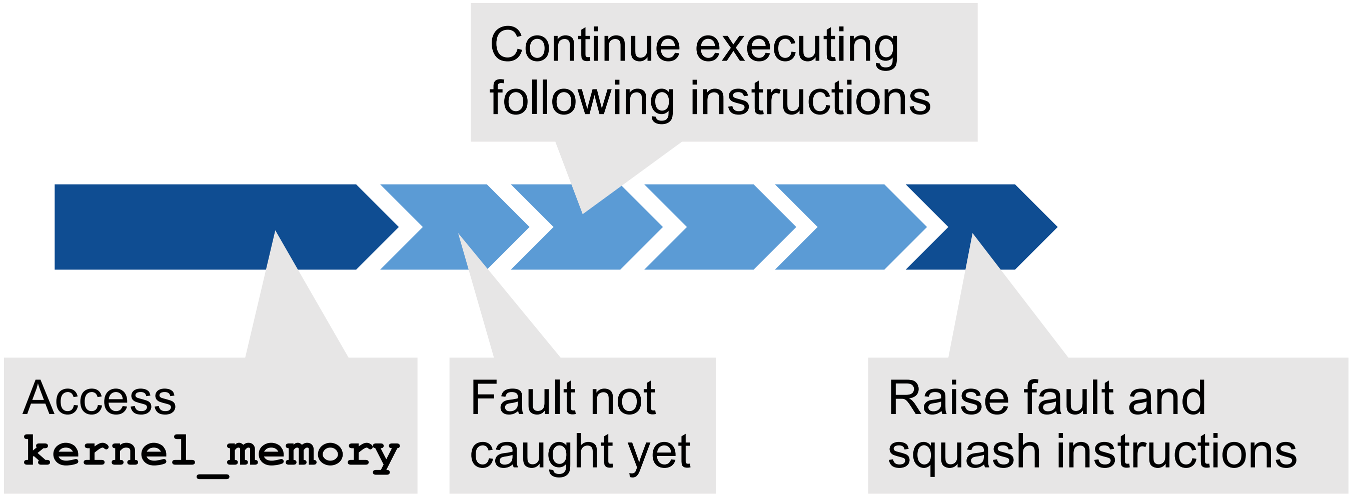
Transient Execution – due to Faults

***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 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];
...
```



Continue executing following instructions

Access **`kernel_memory`**

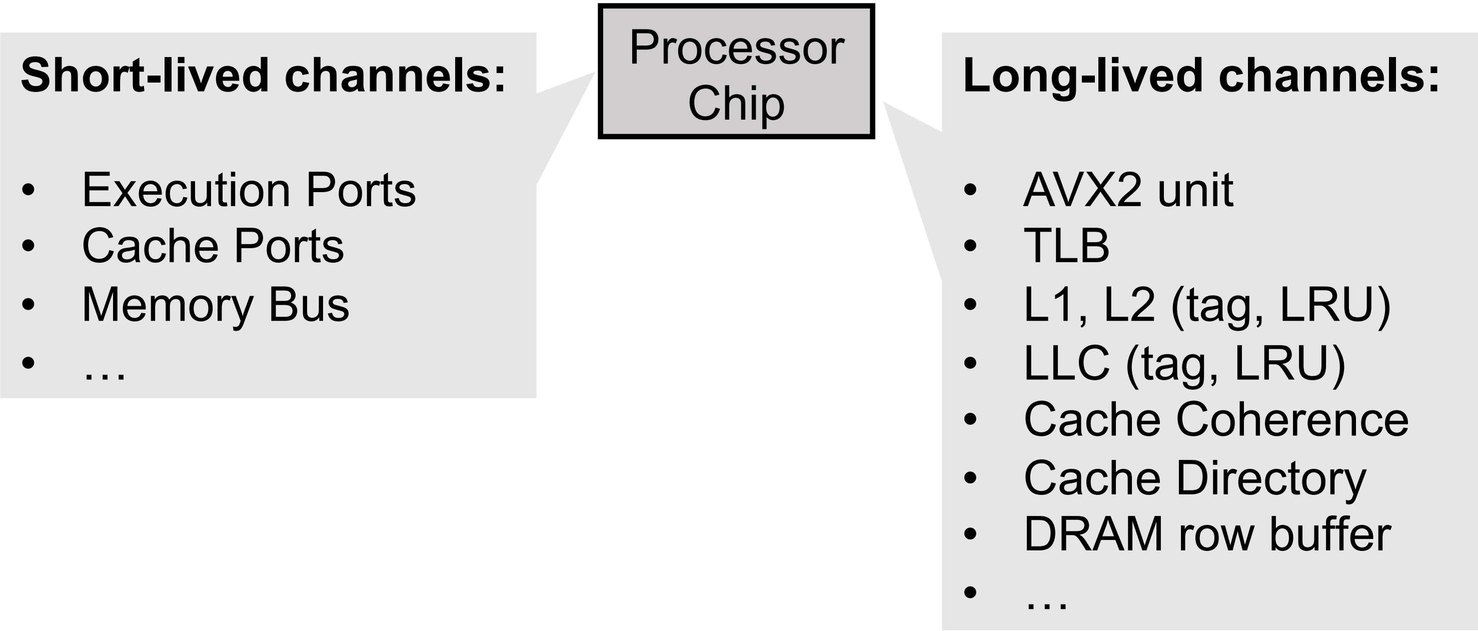
Fault not caught yet

Raise fault and squash instructions

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
- …

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
- 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

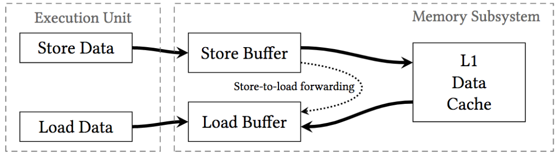
SGXPectre is also Spectre Variant 1 or 2 where code outside SGX Enclave can influence the branch behavior

More Spectre and Meltdown Variants

Micro-architectural Data Sampling (MDS) vulnerabilities:

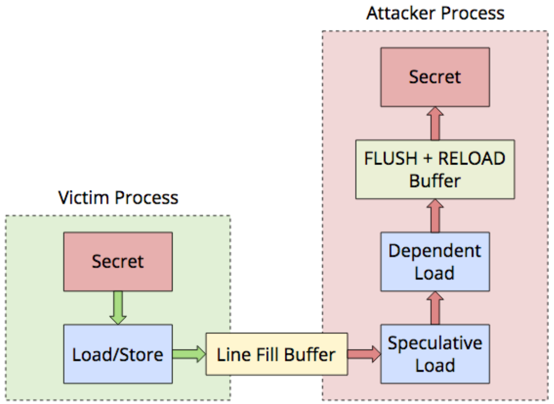
- **Fallout** – *Store Buffers*

> **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

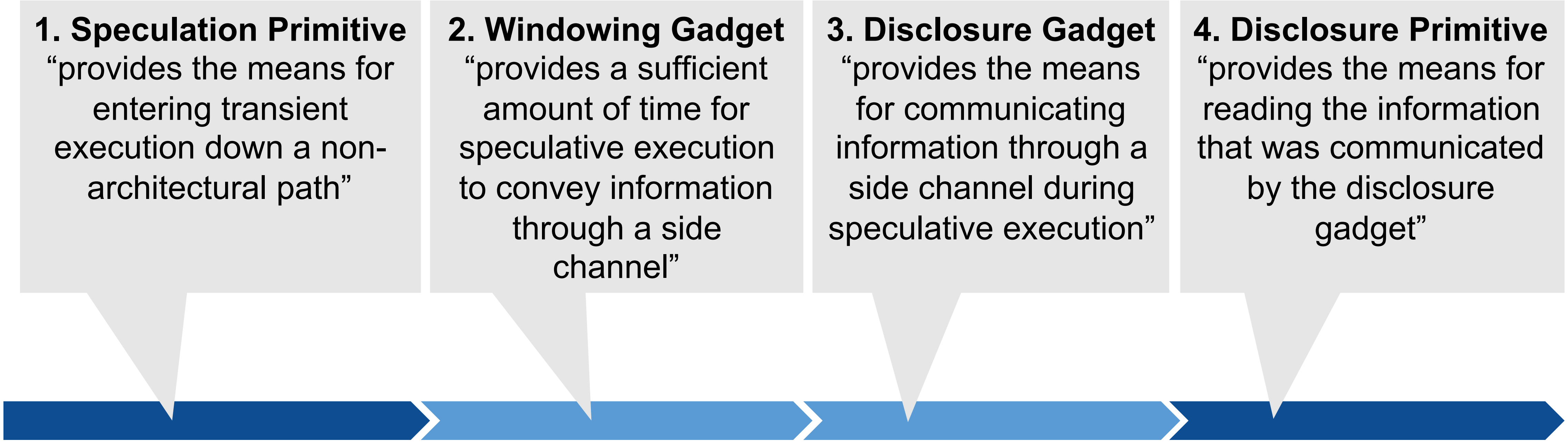
Microsoft, https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/

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"

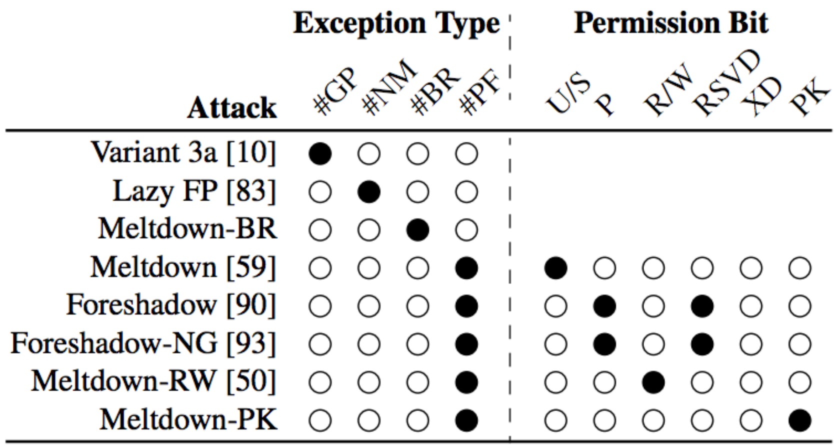
4. Disclosure Primitive
"provides the means for reading the information that was communicated by the disclosure gadget"

Speculation Primitives

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

1. Speculation Primitive

- **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”

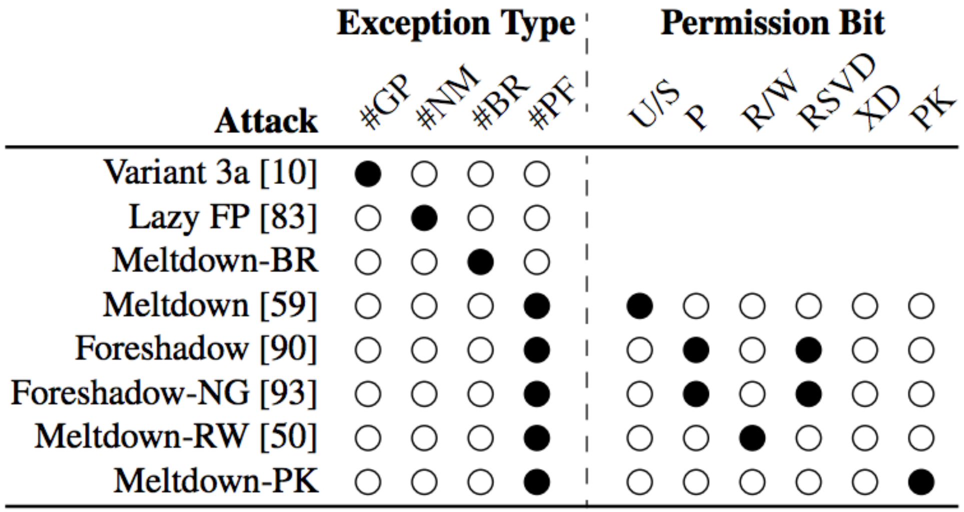
```
value2 = arr2[(*ptr)<<12];
```

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 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

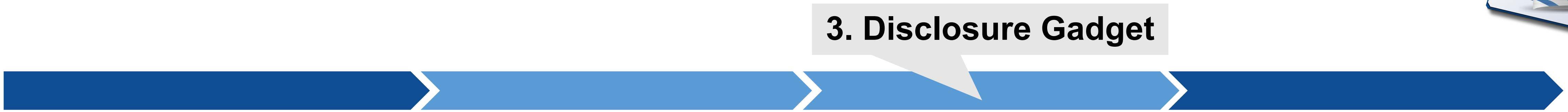
Memory access time determines how long it takes to resolve the branch

E.g.: Spectre v1 :

```
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

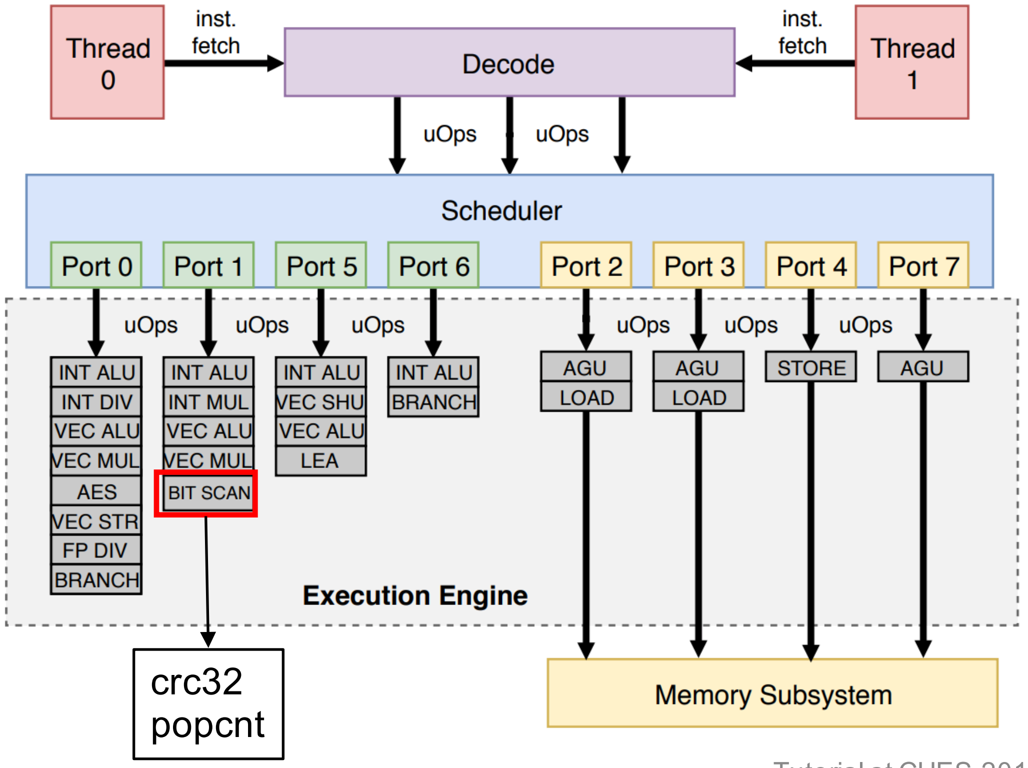
Two types of disclosure primitives:

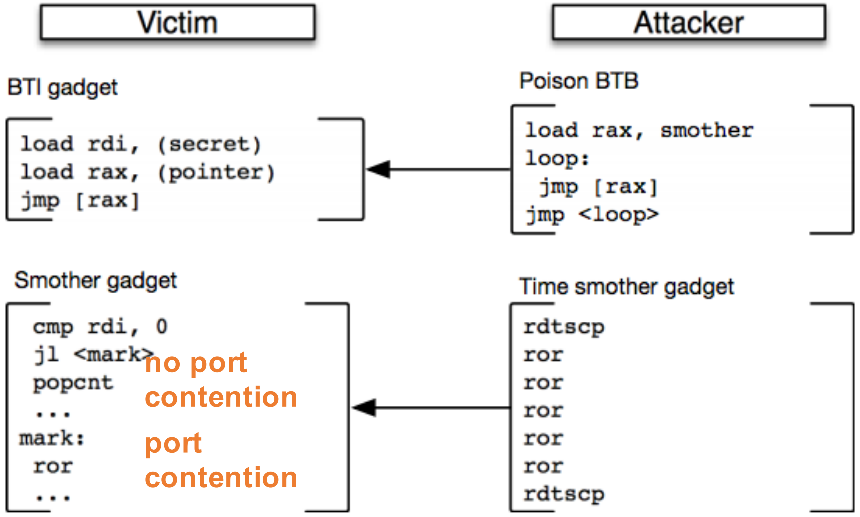
- **Short-lived** or **contention-based** (hyper-threading / multi-core scenario):
  1. Share resource on the fly (e.g., bus, port, cache bank)
  2. State change within speculative window (e.g., speculative buffer)
- **Long-lived channel**:
  - Change the state of micro-architecture
  - The change remains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





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
 ...
```

no port contention

port contention

Time smother gadget

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

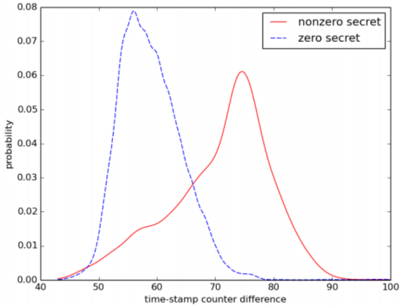


Fig. Probability density function for the timing of an attacker measuring ***crc32*** operations when running concurrently with a victim process that speculatively executes a branch which is conditional to the (secret) value of a register being zero.

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;
      }
}
```

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

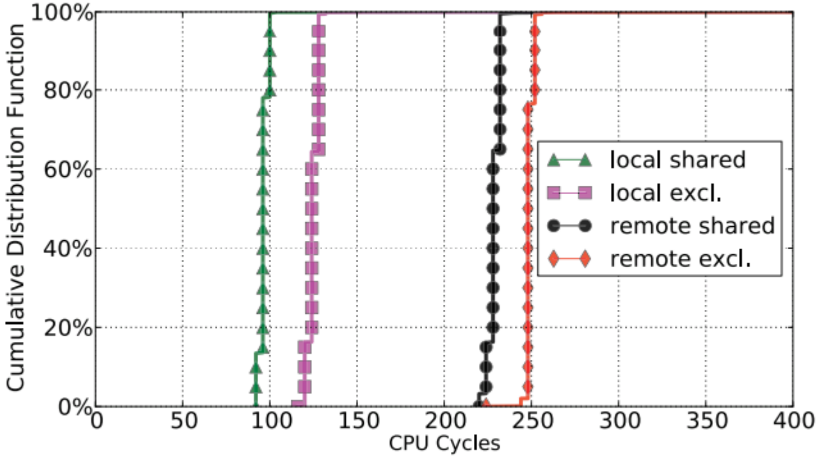


Fig. 2: Load operation latency in various (location, coherence state) combinations.

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

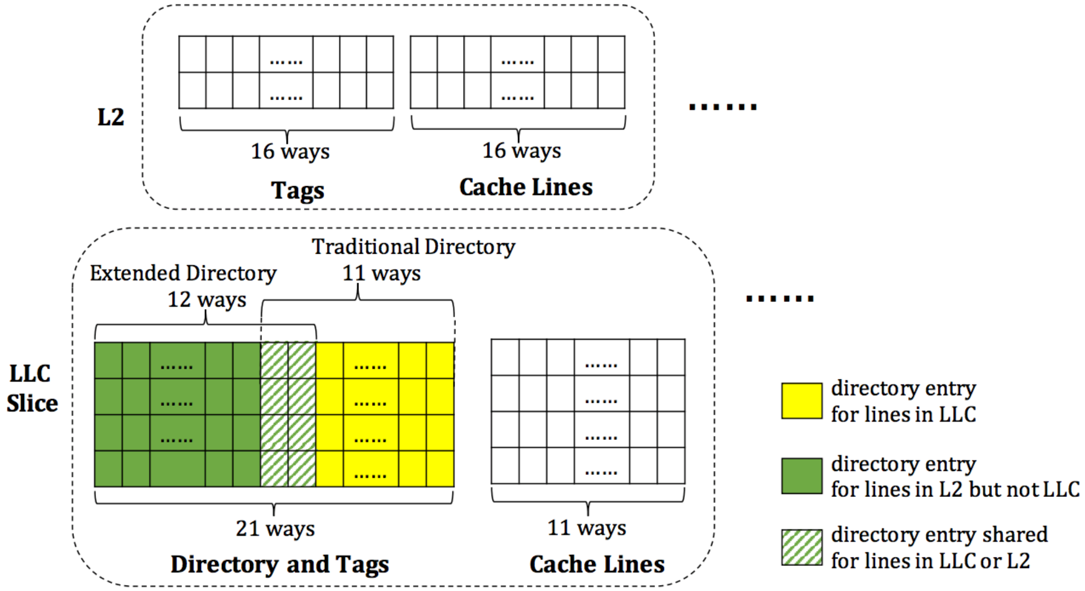


Fig. 9. Reverse engineered directory structure.

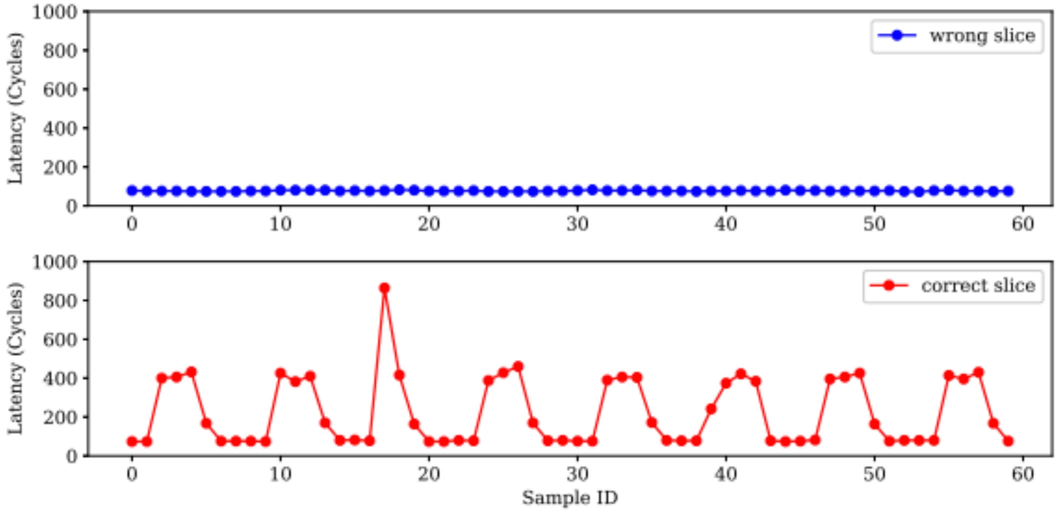


Fig. 13. The upper plot shows receiver’s access latencies on a slice not being used for the covert channel, while the lower one shows the one used in the covert channel. Sender transmits sequence “101010...”.

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

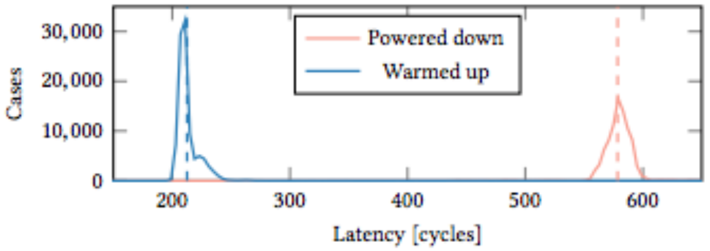


Figure 5: Differences in the execution time for AVX2 instructions (Intel i5-6200U). If the AVX2 unit is inactive (powered down), executing a 256-bit instruction takes on average 366 cycles longer than on an active AVX2 unit. The average values are shown as dashed vertical lines.

Gadget:

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

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

Transient Attack Hardware 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
   - Crate “secure” AVX, etc.

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

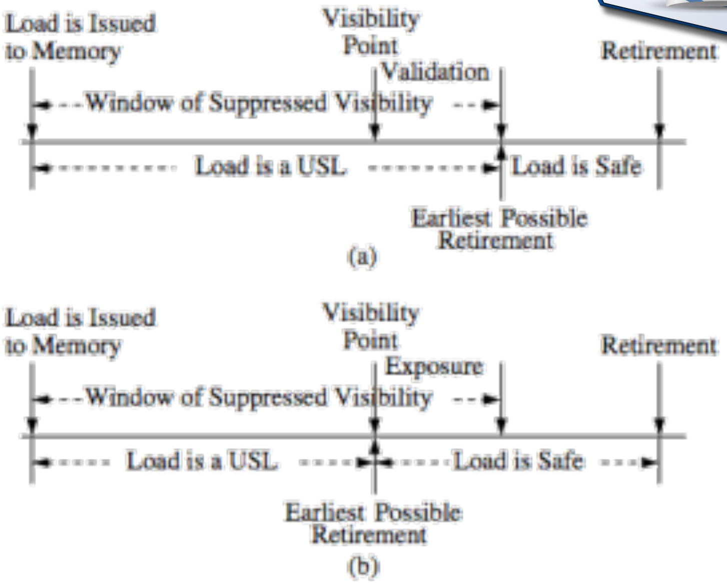
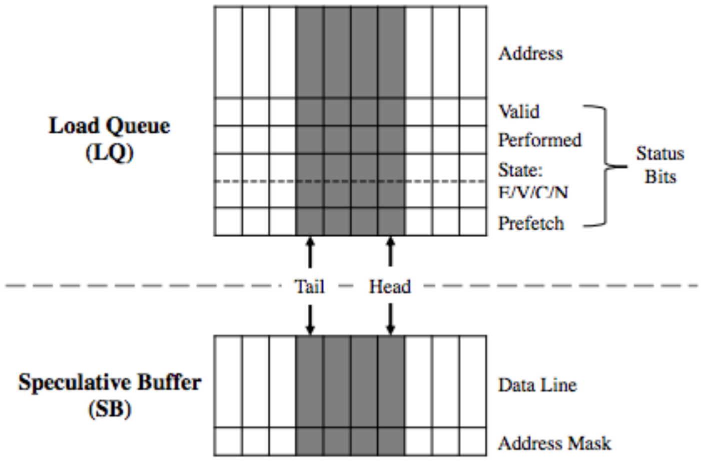


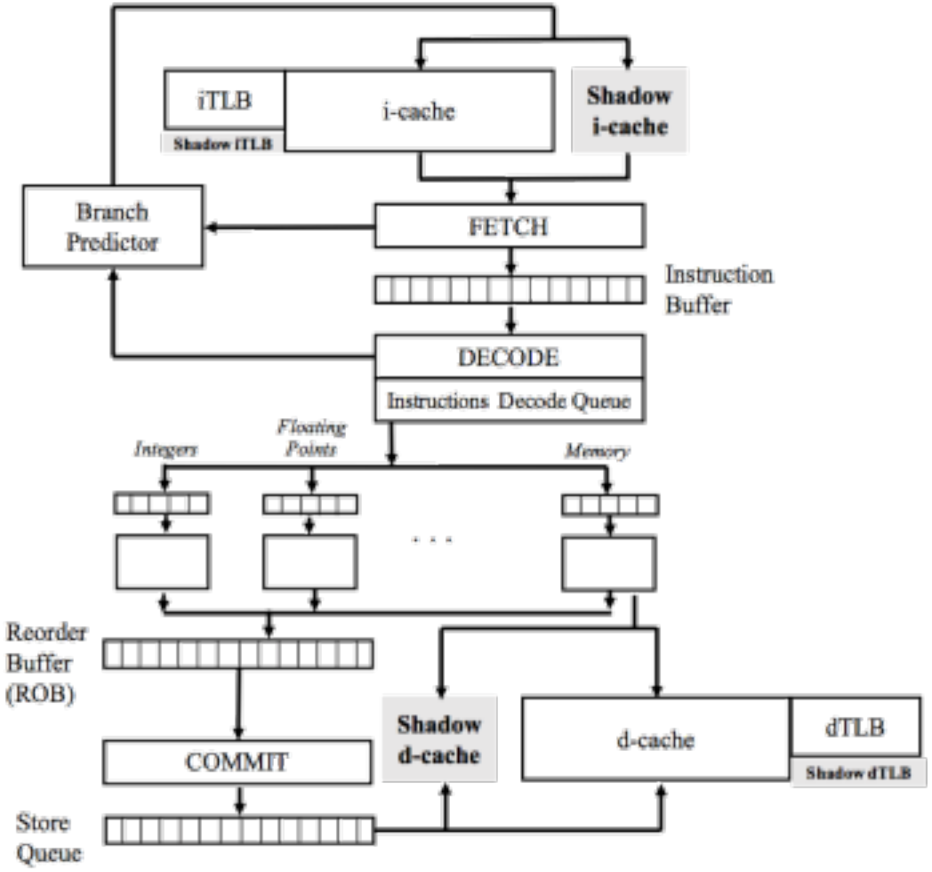
Fig. 2: Timeline of a USL with validation (a) and exposure (b).



Mitigations in Micro-architecture: SafeSpec

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

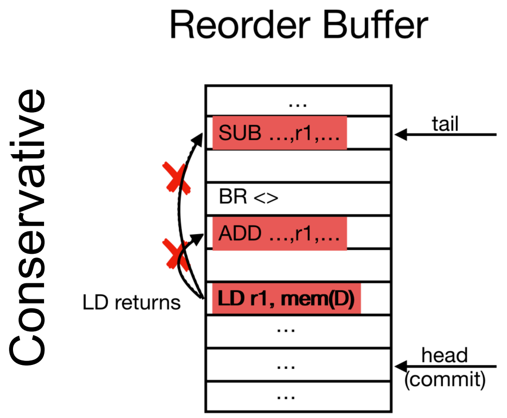
- Similar to InvisiSpec, shadow caches and TLBs are proposed to store the micro-architecture changes by speculative loads temporarily

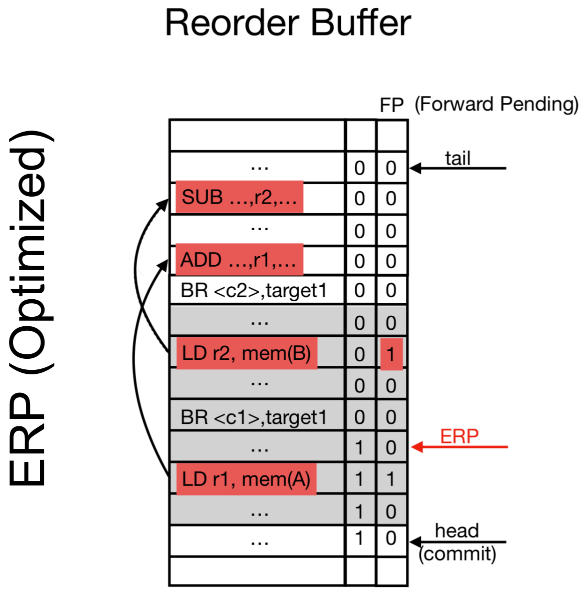


Mitigations in Micro-architecture: SpecShield

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





		FP
...	0	0
SUB ...,r2,...	0	0
...	0	0
ADD ...,r1,...	0	0
BR <c2>,target1	0	0
...	0	0
LD r2, mem(B)	0	1
...	0	0
BR <c1>,target1	0	0
...	1	0
LD r1, mem(A)	1	1
...	1	0
...	1	0

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

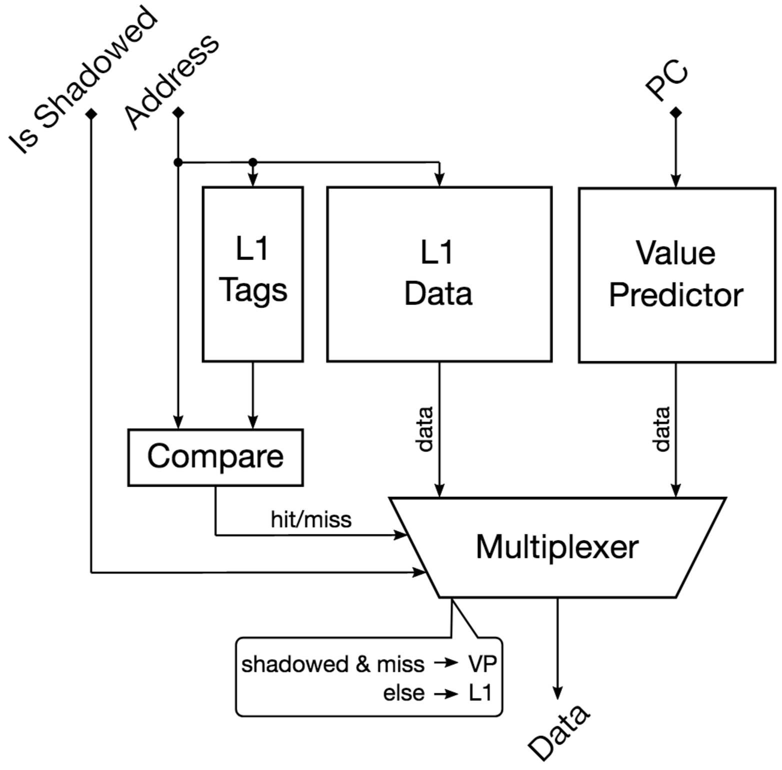
“Conditional Speculation: An Effective Approach to Safeguard Out-of-OrderExecution Against Spectre Attacks”, Peinan Li et al., HPCA 2019

- 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

Most hardware solutions have bigger overheads than reported overheads for secure caches – more motivation to look at secure caches

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 processor architectures

- E.g., Spectre v2 on SGX

Shared BTB allows the attacker to create speculative execution attack in enclave

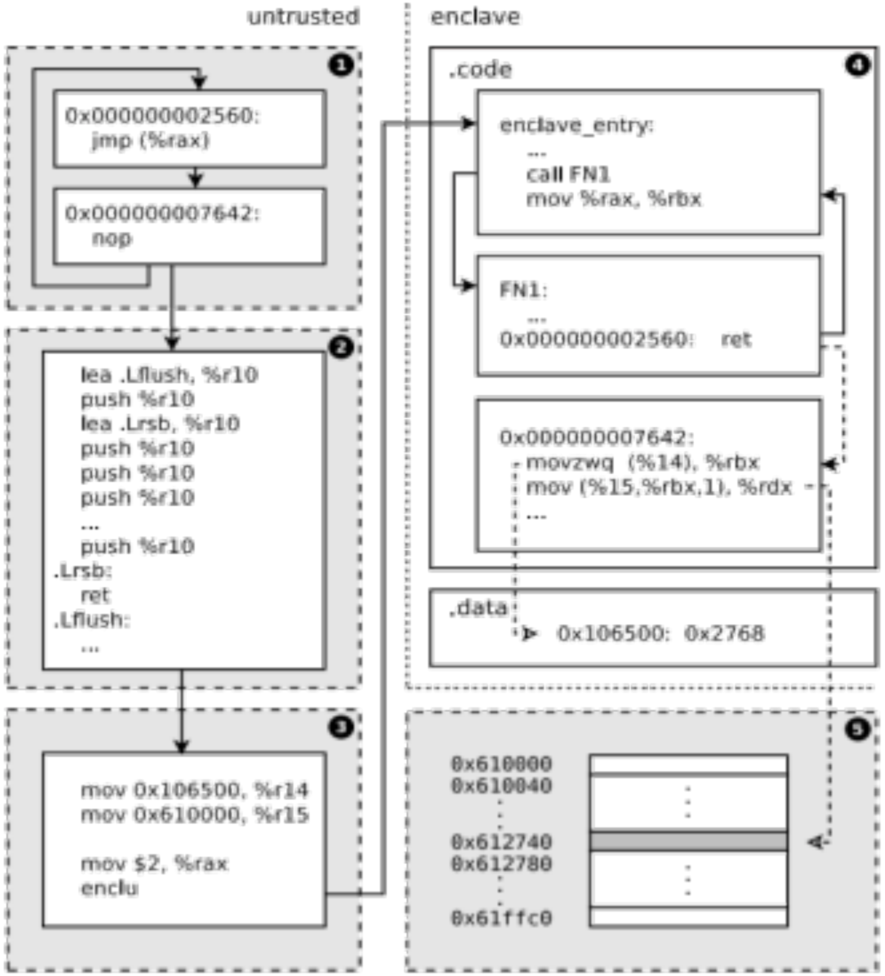
1. Poison BTB (Speculation Primitive)

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

3. Set secret address and probe array address

4. Execute victim code (Disclosure Gadget)

5. Obtain secret from covert channel (Disclosure Primitive)



```
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
```

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 processor architectures

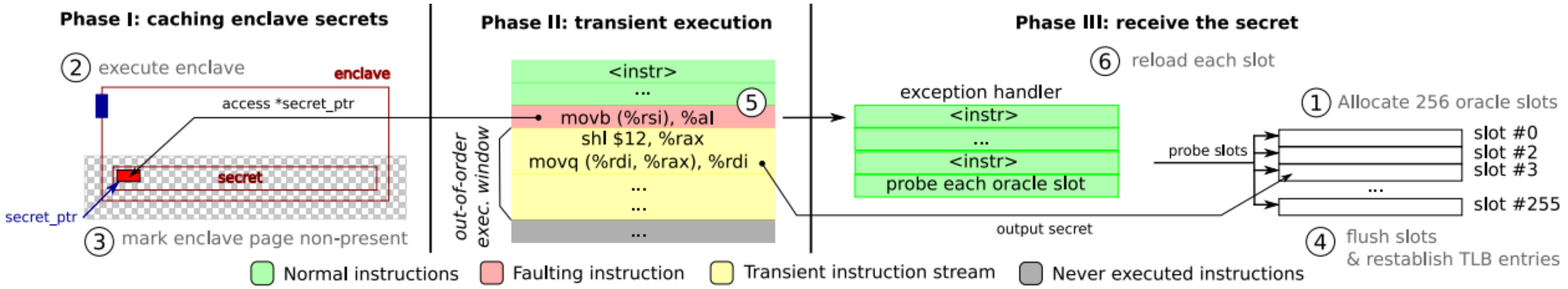


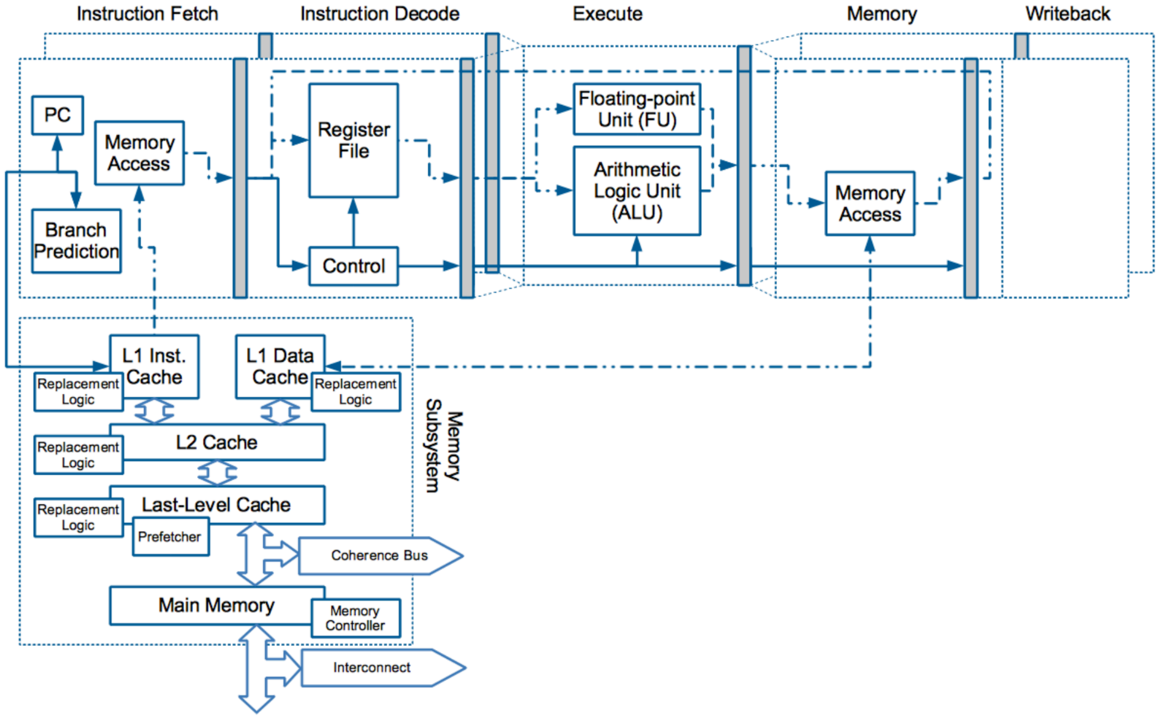
Figure 2: Basic overview of the Foreshadow attack to extract a single byte from an SGX enclave.

Delayed permission checks allows transient read of data by the attacker

Transient Execution Attacks 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