

2023-2024学年春季学期

计算机体系结构安全
Computer Architecture Security

授课团队：史岗，陈李维

计算机体系结构安全

Computer Architecture Security

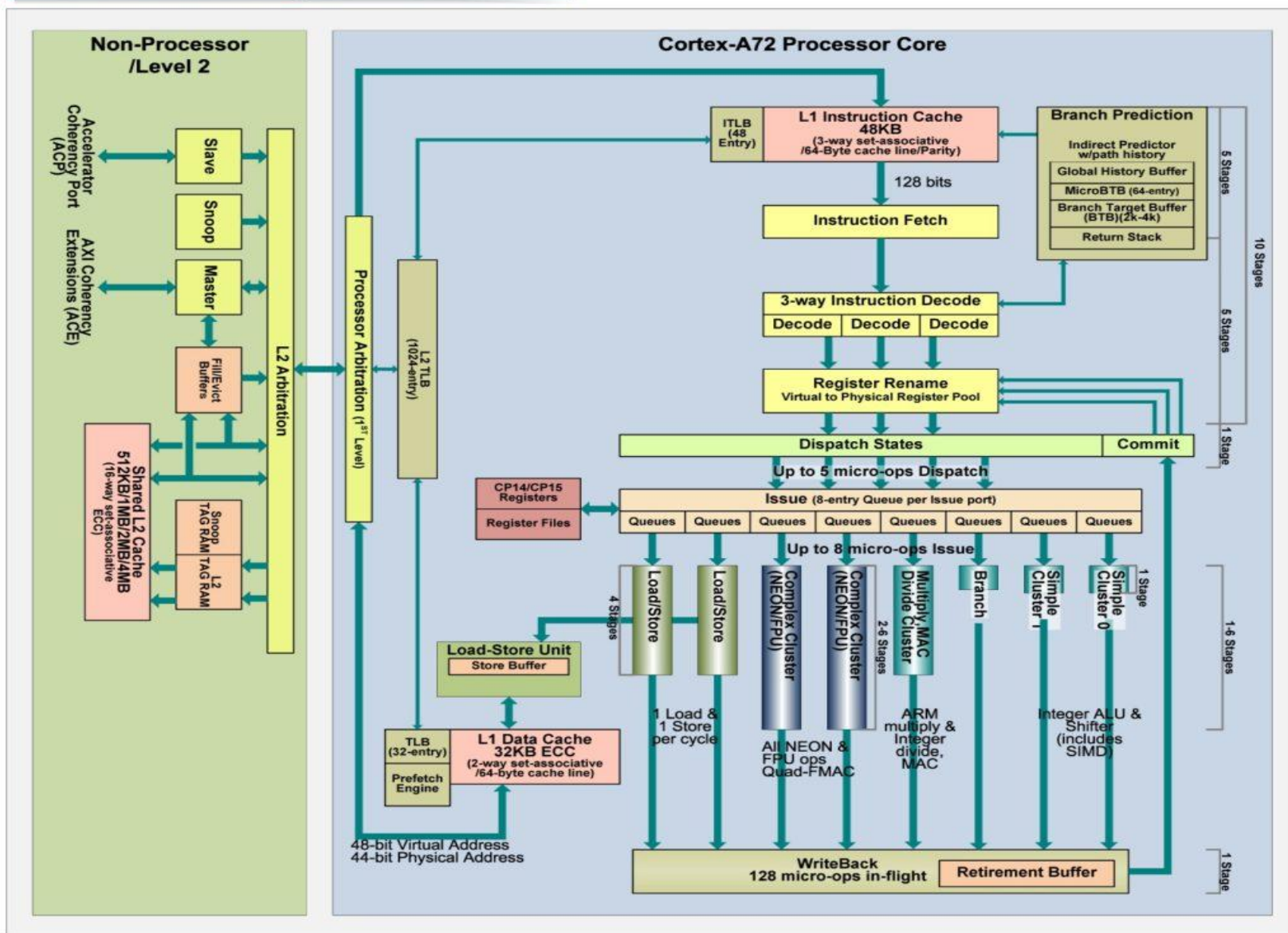
[第9次课] 安全微体系结构实践

授课教师：史岗

授课时间：2024. 04. 22

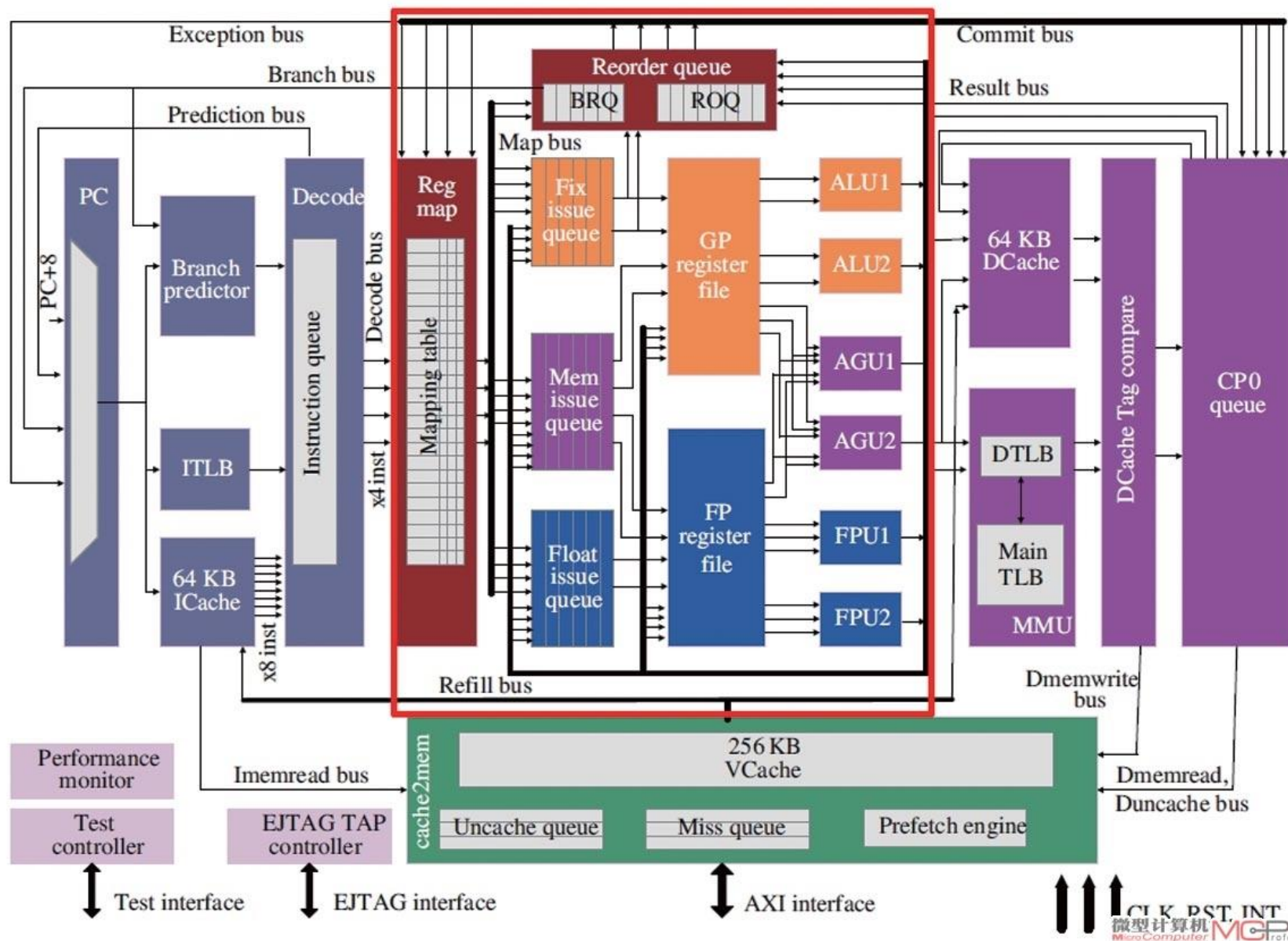
微体系结构概念

ARM Cortex-A72 Block Diagram



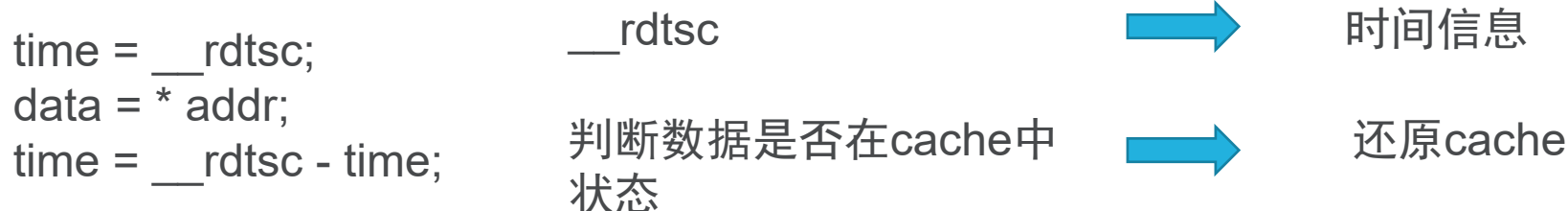
Copyright (c) 2015 Hiroshige Goto All rights reserved.

微体系结构概念



- 微体系结构指处理器负责完成指令集功能的具体部件的内部结构。
- 可以通过软件层是否可见来进行区分。
- 体系结构状态是指系统软件（OS/编译器）可见的状态，如内存中的数据，通用寄存器的值等。
- 微体系结构状态则是指系统软件不可直接感知的状态，如Cache中的数据，TLB内容，分支预测器的内容，处理器流水线状态，读写缓冲区内容等等。

- 针对处理器微体系结构部件，还原微体系结构部件的状态，从而获取敏感数据的攻击，**被称为微体系结构攻击 (Micro-architectural Attack)**。
- 微体系结构攻击通过时间、功耗、温度等信息，还原出微体系结构部件的状态，目前最主要的方法还是处理器时间信息。



内容概要

- 微体系结构安全
 - 高速缓存 (Cache) 安全
 - 乱序执行安全
 - 推测执行安全
- 总结

内容概要

- 微体系结构安全
 - 高速缓存 (Cache) 安全
 - 乱序执行安全
 - 推测执行安全
- 总结

○安全防护的思路

○敏感指令限制

○硬件防御方法

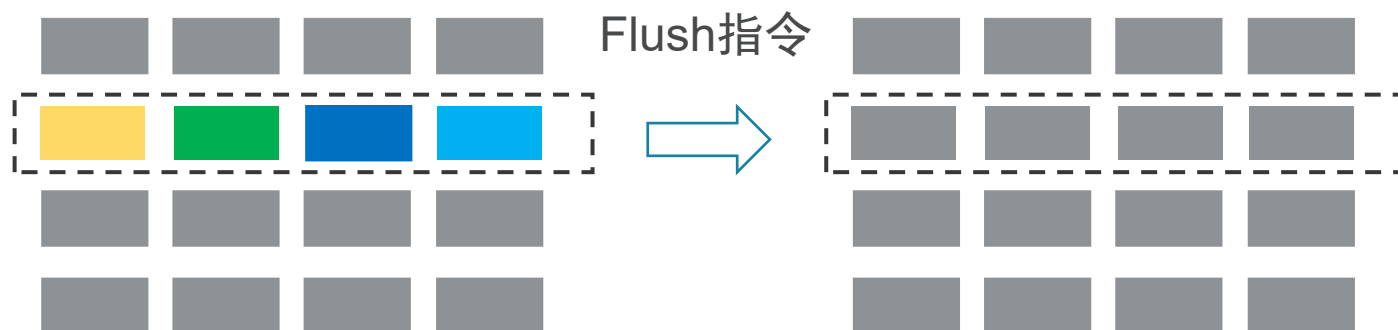
- 分区机制

- 随机化方法

- delay-on-miss

敏感指令限制

- 一部分Cache测信道攻击，如Flush+Reload, Flush+Flush等，通过cache刷新指令完成Flush操作，实现对cache状态的设置。



- x86指令集的clflush指令，Arm架构提供IC和DC指令，精确刷新指定cache line。

Apple M1等Arm架构处理器，不允许用户权限执行cache刷新指令。很多云服务平台也限制用户使用Cache刷新指令。

高速缓存 (Cache) 安全

- Cache攻击以及其他微体系结构攻击普遍依赖于精确的时间数据。

```
time = __rdtsc;           // 通过测量访存操作的时钟周期，推测cache状态
data = * addr;
time = __rdtsc - time;
```

- x86指令集的rdtsc指令，Arm指令集读取cntfrq_el0，cntvct寄存器，RISC-V指令集的rdcycle，rdtime指令等都可以获取精确的处理器时钟周期。
- 一些Linux发行版已经限制用户权限下获取处理器时钟周期的精度，或者只允许内核态获取处理器时钟周期。

这一方法只能提高攻击门槛，并不能消除攻击。攻击者仍然有可能获取较精确的时间数据，比如通过构造一个循环程序实现计时功能。

○安全防护的思路

○敏感指令限制

○硬件防御方法

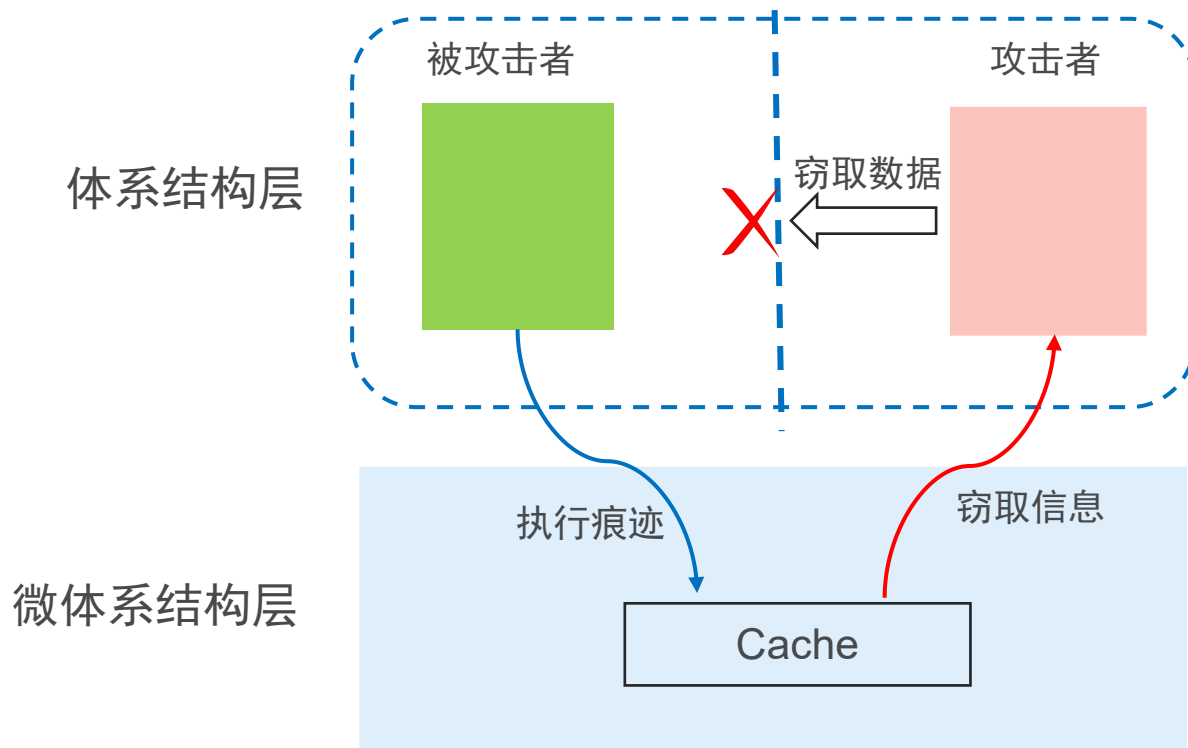
○分区机制

○随机化方法

○delay-on-miss

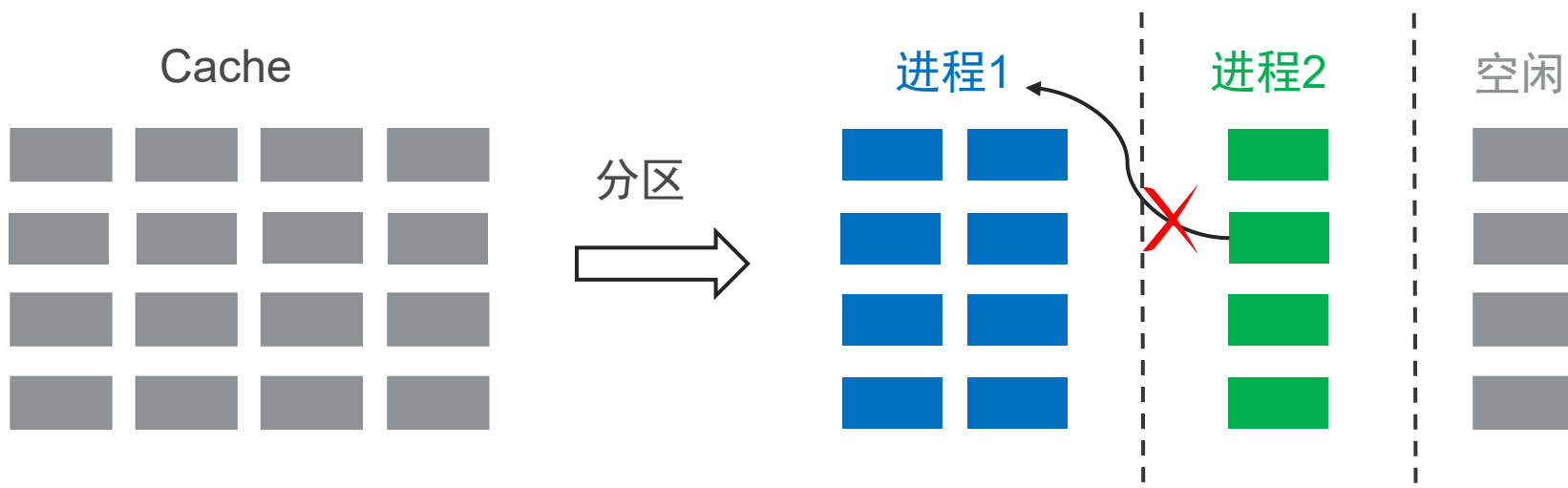
○硬件防御方法-分区机制

- Cache的安全问题是由于Cache的共享所导致的，攻击者和被攻击者虽然在体系结构层被分割，但是仍然共享了Cache。



○ 硬件防御方法-分区机制

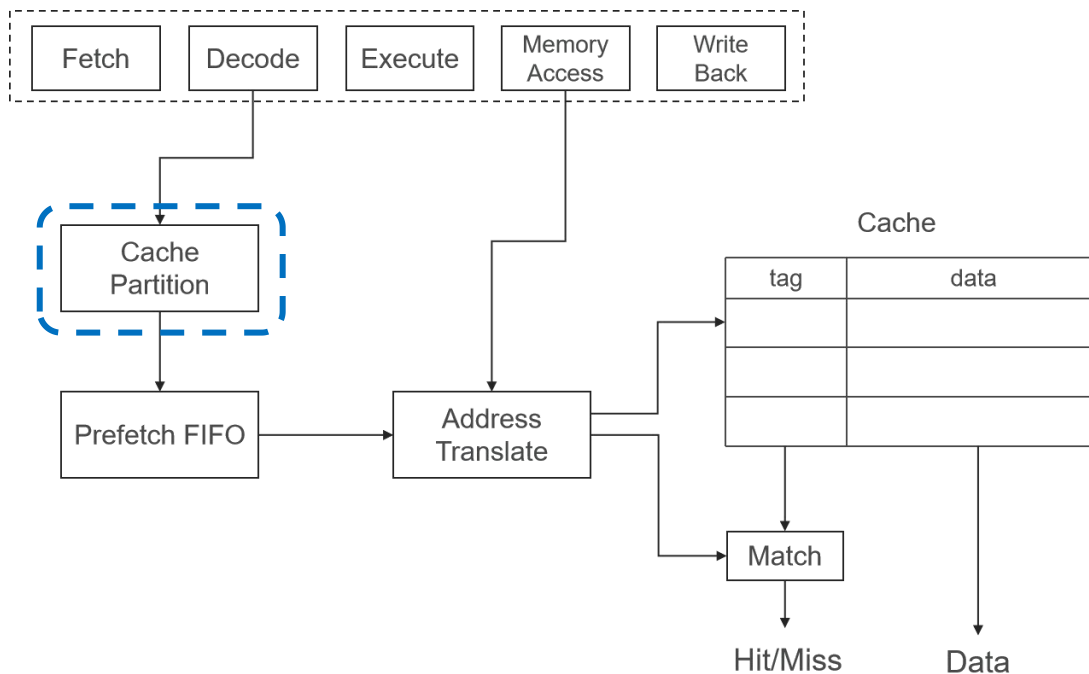
- Cache分区机制 (Cache partition) 尝试在微体系结构层对Cache进行划分, 阻止攻击者利用Cache获取被攻击者的相关信息。
- Partitioned Cache Architecture as a Side-Channel Defence Mechanism (2005) 第一次提出, 将Cache分区机制用于微体系结构安全防护。
- 核心思想: 对Cache进行划分, 每个权限或进程享有单独的cache区域, 禁止访问不属于该权限或进程的cache分区。



高速缓存 (Cache) 安全

Cache分区机制 (静态)

- 静态分区方法，即分区确定后不可增加或减小。
- 在解码阶段确定分区，增加效率。
- 需要额外的指令集ISA拓展：
 - ADDRPAR pid 增加分区
 - DELPAR pid 删除分区
 - INVPAR pid 刷新分区



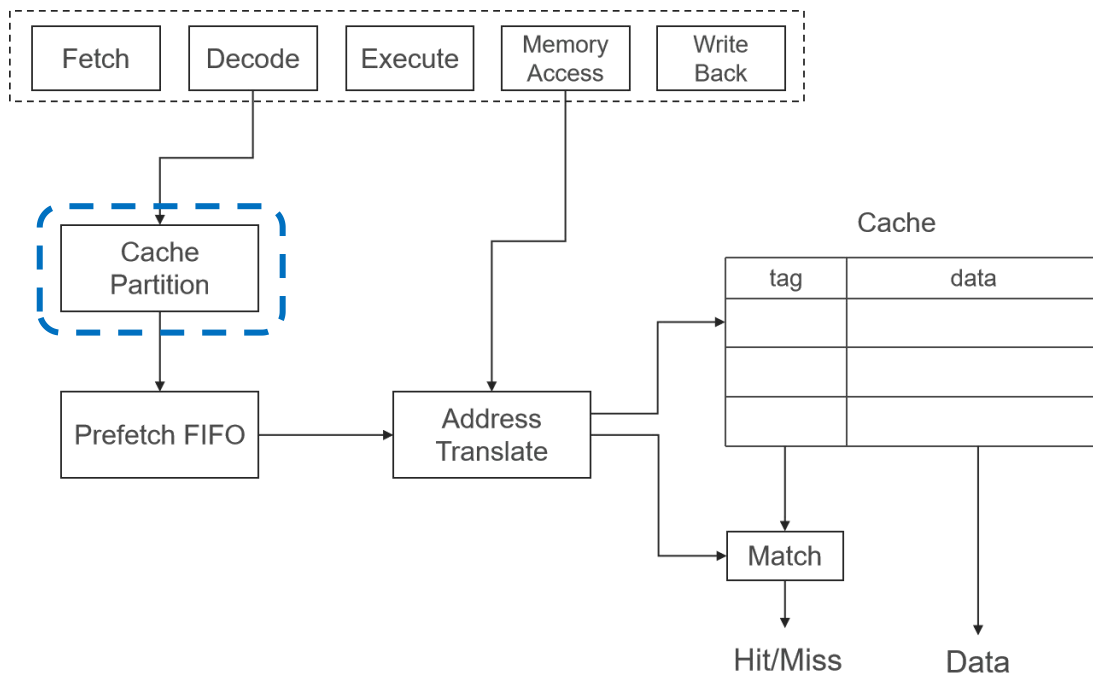
高速缓存 (Cache) 安全

Cache分区机制的评价 (静态)

- 方法意义在于提出了利用Cache分区机制防御微体系结构攻击的理论可行性

- 注意点:

- 不可能由用户对Cache进行任意管理, 必须由用户进程发起申请, 由管理员进行操作。
 - 需要用指令集拓展来申请和释放Cache, 但这些指令的权限不能交由用户进程进行执行。
- Cache分区确定后不能增加或减小, 导致Cache的使用率很低, 使计算机系统的性能明显下降。



○Cache分区机制 (动态)

- 动态方法即可以根据进程的Cache使用需求，动态的增加或者减小专属Cache区域。
- 各进程的Cache请求需要高权限管理软件（如操作系统，hypervisor）来负责维护和管理。
- 优缺点评价：
 - 优点：性能好，Cache利用效率高；
 - 缺点：一个敏感进程对Cache的动态使用也可以反映该进程的运行时信息，构成潜在的侧信道。攻击者可以通过在攻击进程内申请Cache使用来评估处理器Cache的占用情况，从而推测出敏感进程的运行时信息。设计者必须要考虑消除这类信息泄漏。

○安全防护的思路

○敏感指令限制

○硬件防御方法

○分区机制

○随机化方法

○delay-on-miss

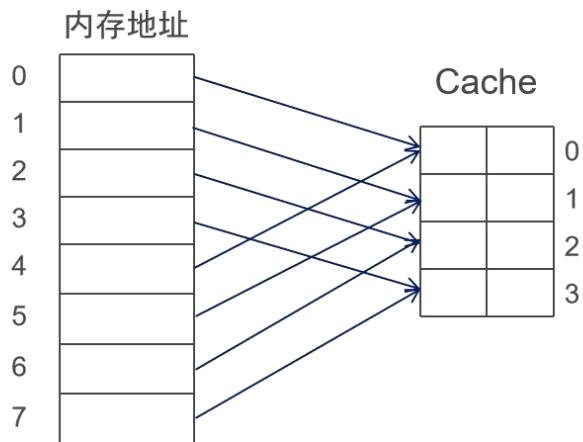
○Cache随机化

- Cache微体系结构攻击的核心在于攻击者构造与被攻击者的Cache条目碰撞，并根据碰撞信息还原出相关数据。Cache随机化就是在替换时不再遵循确定性的替换策略，而是引入一定的随机量。
- 随机化方法不会牺牲Cache的使用效率，只是改变了地址到Cache的映射关系。
- 随机化方法使得攻击者无法推测出Cache中的数据布局，从而难以构造碰撞，即使是偶尔发生了碰撞，随机化方法也能保证攻击者难以从中获取有效信息

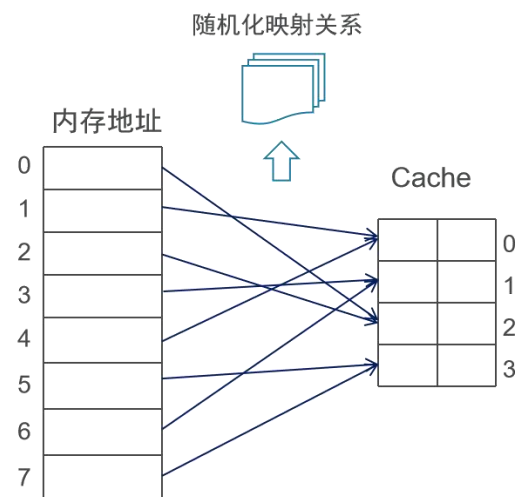
Cache随机化两个核心思想

随机化映射 \Rightarrow 空间上的随机化

重随机化 \Rightarrow 时间上的随机化



空间上的随机化



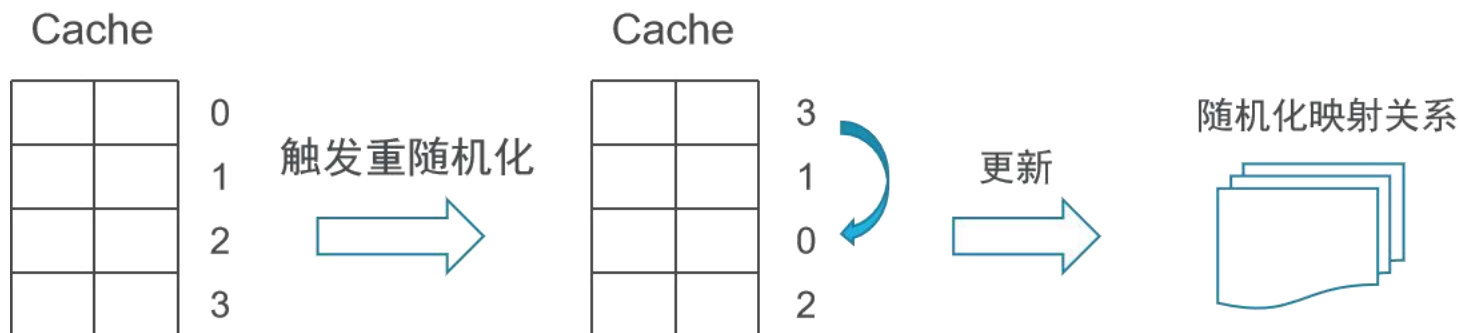
- 空间随机化：打乱原地址到Cache的映射关系。
- 随机化映射关系需要保存，一般是比如随机化算法的密钥，或者映射表等信息。
- 每个进程有独立的随机化映射关系。

New Cache Designs for Thwarting Software Cache-based Side Channel Attacks (2007 ISCA)

○ Cache随机化两个核心思想

○ 随机化映射 \Rightarrow 空间上的随机化

○ 重随机化 \Rightarrow 时间上的随机化



- **时间随机化**：需要对Cache运行状态进行统计分析，一些事件后（比如一定数量的Cache miss），要对随机化映射关系进行随机化调整。
- 考虑性能开销问题，一般不会重随机化整个映射关系，而是部分随机化
- 目的：防止攻击者在时间上反复尝试（replay attack），破解随机化映射关系。

New Cache Designs for Thwarting Software Cache-based Side Channel Attacks (2007 ISCA)

○安全防护的思路

○敏感指令限制

○硬件防御方法

○分区机制

○随机化方法

○delay-on-miss

该机制用于Spectre攻击的防御，会在后面介绍

内容概要

- 微体系结构安全
 - 高速缓存 (Cache) 安全
 - 乱序执行安全
 - 推测执行安全
- 总结

乱序执行安全

乱序执行回顾：熔断（Meltdown）漏洞

体系结构层（软件层）视角

微体系结构层视角

1 rcx = kernel address

1 rcx = kernel address

2 rbx = array_addr in userspace

2 rbx = array_addr in userspace

3 mov al, byte [rcx];

Exception

越权访问触发例外

4 shl rax, 0xc;

5 mov rbx, qword [rbx + rax];

3 mov al, byte [rcx]; 越权访问触发例外

4 shl rax, 0xc;

5 mov rbx, qword [rbx + rax];

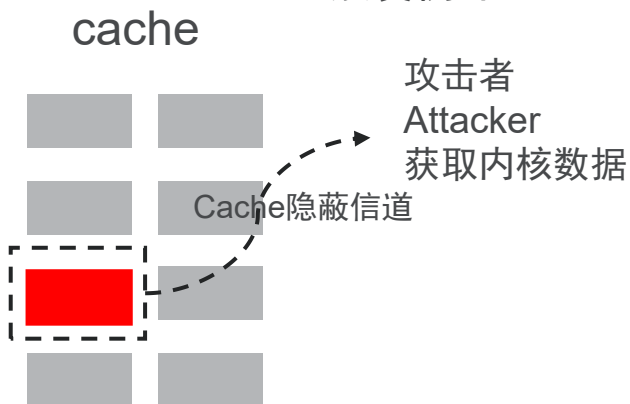
Exception

执行后，保留在ROB中，等待提交

第3行代码提交，触发例外

第3行代码如果处于乱序执行状态下，则会越权访问rcx中的内核地址kernel address。

第4、5行代码构成Cache隐蔽信道，将加载入Cache的内核数据传递出来。由于乱序执行，第4、5条指令在第3条指令触发异常前就会完成。



乱序执行安全

○ 熔断（Meltdown）漏洞防御思路

1 `rcx = kernel address`

2 `rbx = array_addr in userspace`

3 `mov al, byte [rcx];`

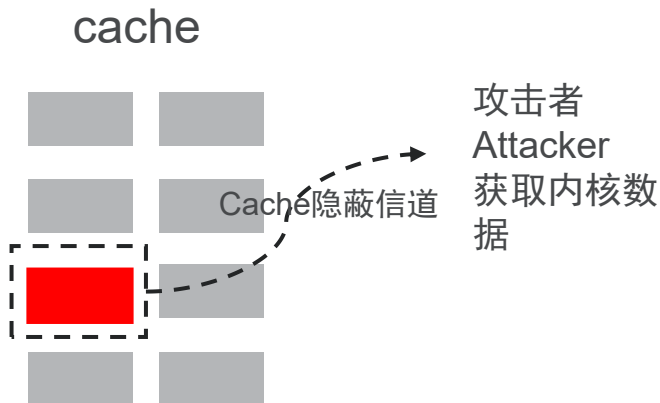
Exception

越权访问触发例外

4 `shl rax, 0xc ;`

5 `mov rbx, qword [rbx + rax];`

Cache隐藏信道



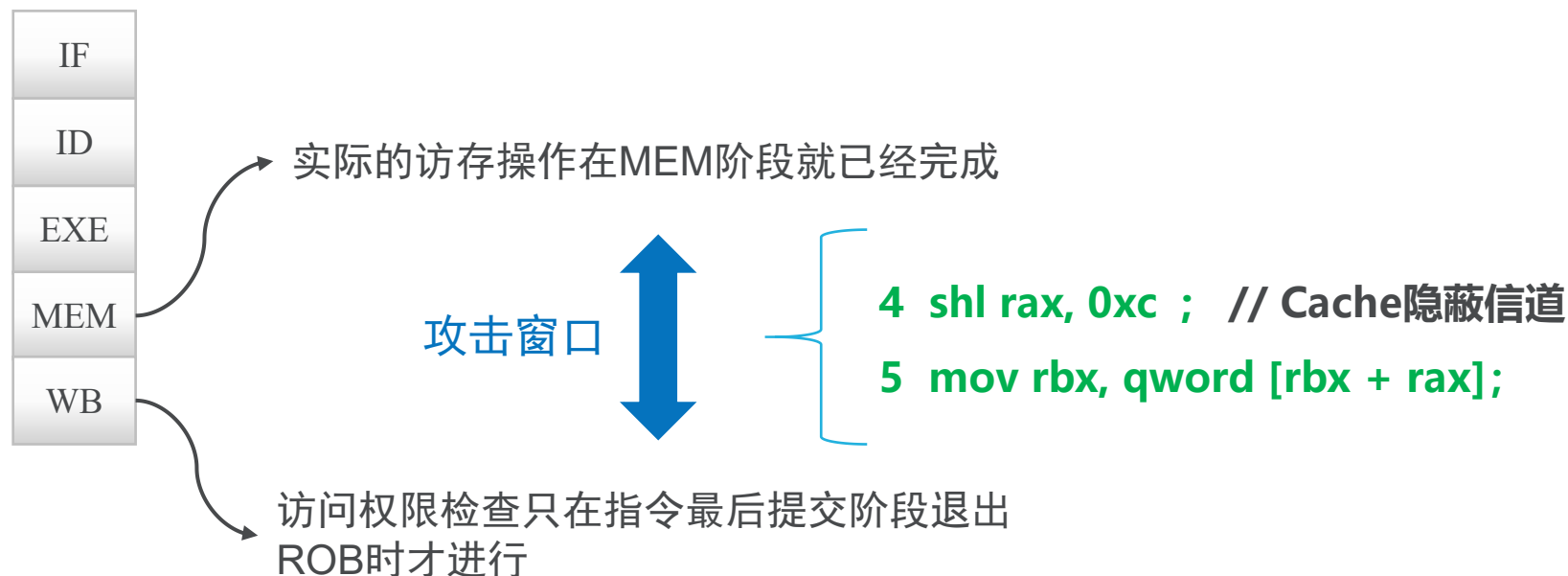
○ 熔断漏洞需要结合隐蔽信道将数据传输给攻击者，最早的攻击原型中采用了Cache实现。所以前面介绍的针对Cache的防御方法也能起到防御效果。

○ 但能采用的隐蔽信道很多，如TLB，BTB等微体系结构部件都可以用作隐蔽信道，因而针对Cache进行防御并不能完全阻止此类攻击。

乱序执行安全

思考熔断（Meldown）漏洞产生的原因：

3 `mov al, byte [rcx] ; //访问内核数据`

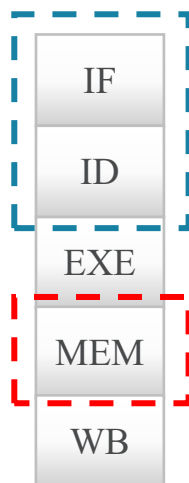


处理器只关注了指令在**体系结构层**的权限检查（最后WB阶段），而忽视了**微体系结构层**的权限问题。

- 现在的高性能处理器中，乱序执行深度很深，ROB深度往往有数百条，因而很容易存在足够的攻击窗口。

- 最直接的解决方法：提前进行权限检查操作

3 `mov al, byte [rcx];`



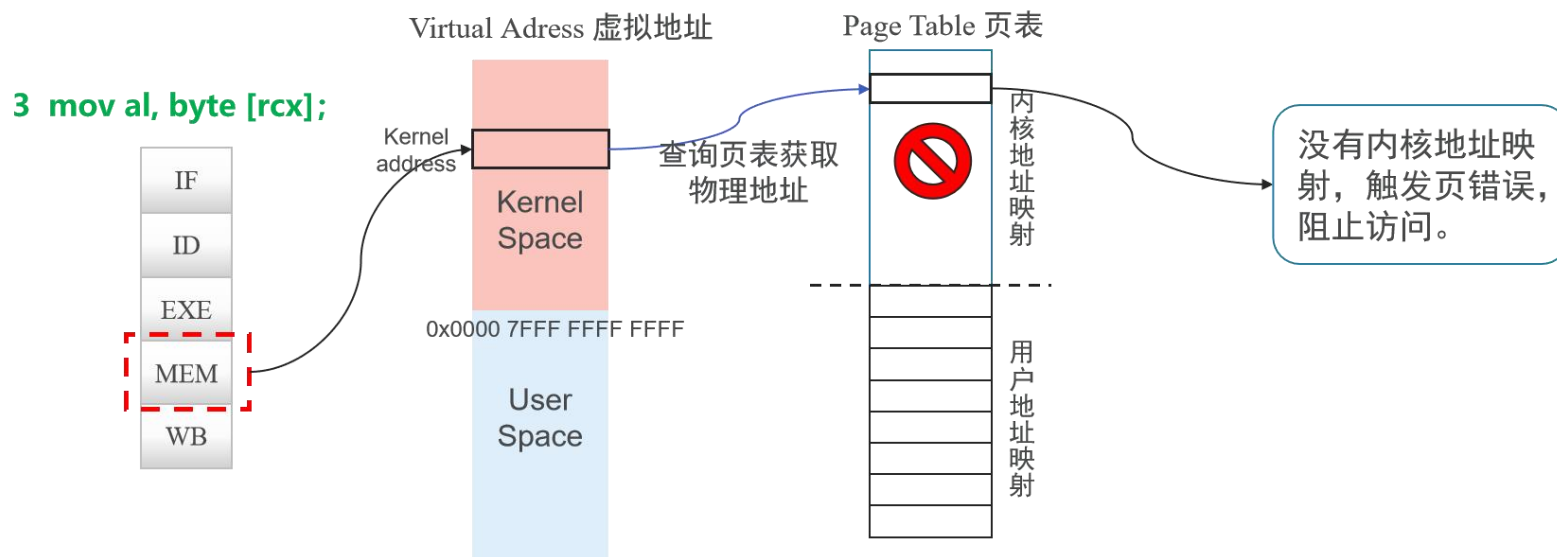
在IF（取指）和ID（解码）阶段就完成权限检查。发现违反权限直接触发异常。

越权访问已经被提前发现

- 目前较新的商用高性能处理器普遍在微体系结构层加强了指令的各类权限检查。
- 但这一方法需要对处理器微架构进行修改，对于已经部署的老处理器平台无法生效。

乱序执行安全

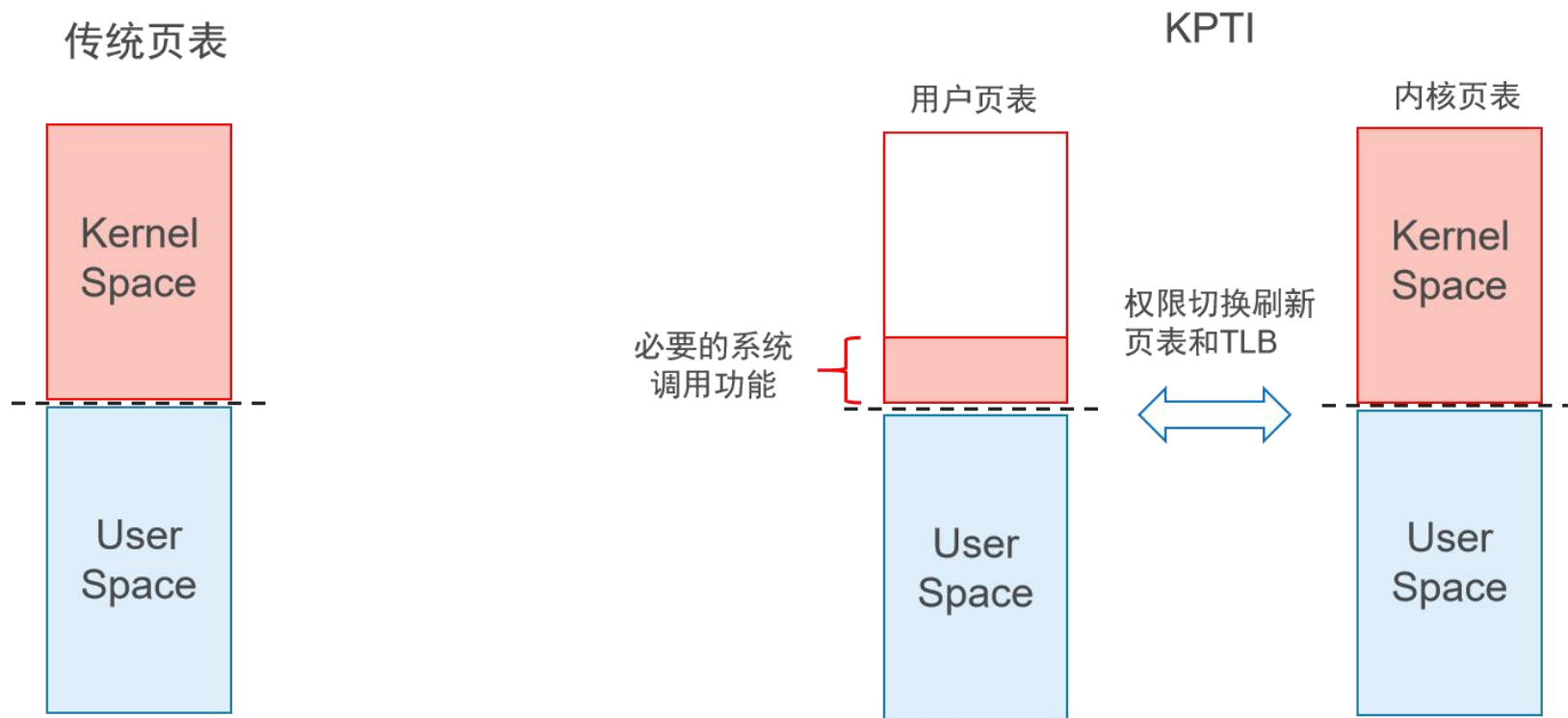
- 软件的解决思路：KPTI (Kernel Page Table Isolation) 内核页表隔离。
- KPTI的提出早于熔断 (Meltdown) 攻击，当时提出的目的是防止微体系结构攻击对内核地址随机化 (KASLR) 的破解，以及其他针对内核的微体系结构攻击。



- 核心思想：访存操作需要访问页表完成虚拟地址到物理地址的转换，这一操作即使在微体系结构层也不可能被绕过。如果在用户空间运行时，不对内核地址空间进行物理地址映射，就可以避免越权访问。

乱序执行安全

- 具体实现：为用户地址空间和内核地址空间维护两个单独的页表，用户地址空间中内核部分只保留状态切换必要的部分系统调用，不映射完整的内核地址空间。
- 用户态和内核态切换时，需要对页表和TLB进行刷新。



小结:

- KPTI通过修改操作系统即可实现，不需要做硬件修改。
- 目前KPTI已经被Windows, Linux, MacOS等操作系统支持。
- KPTI的主要问题是在用户态和内核态切换时有很大的性能开销，实验表明在部分系统调用频繁的程序中开销能达到30%。
- 更严格、且提前的硬件权限检查是解决乱序执行安全的根本之道。

内容概要

- 微体系结构安全
 - 高速缓存 (Cache) 安全
 - 乱序执行安全
 - 推测执行安全
- 总结

推测执行安全

推测执行回顾：幽灵（Spectre）漏洞

// 幽灵漏洞代码

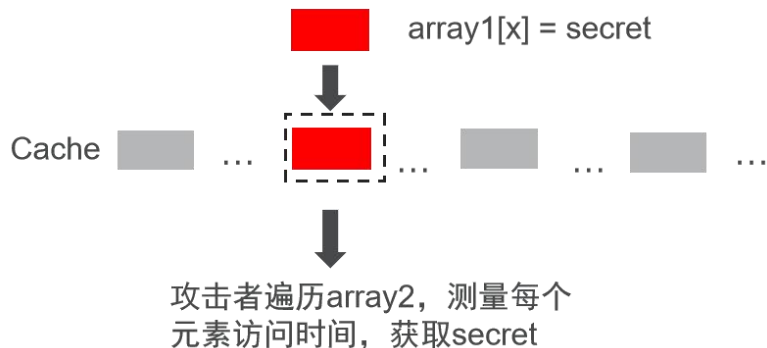
```
1  x = secret_index
   // x > array1_size 且 array1[x] = secret
2  if ( x < array1_size)
3      y = array2 [ array1[x] * 4096 ];
```

// 体系结构视角

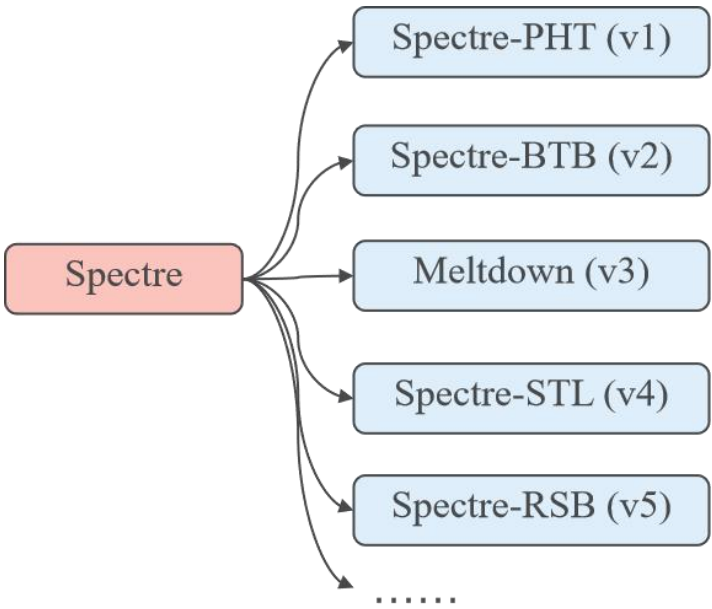
```
1  x = secret_index
2  x < array1_size = False //判定不执行分支
3  ..... // 后续指令
```

// 微体系结构视角

```
1  x = secret_index
2  预测 x < array1_size = True
3  y = array2 [ array1[x] * 4096 ];
处理器发现x < array1_size = True预测错误,
撤销y = array2 [ array1[x] * 4096 ];
```



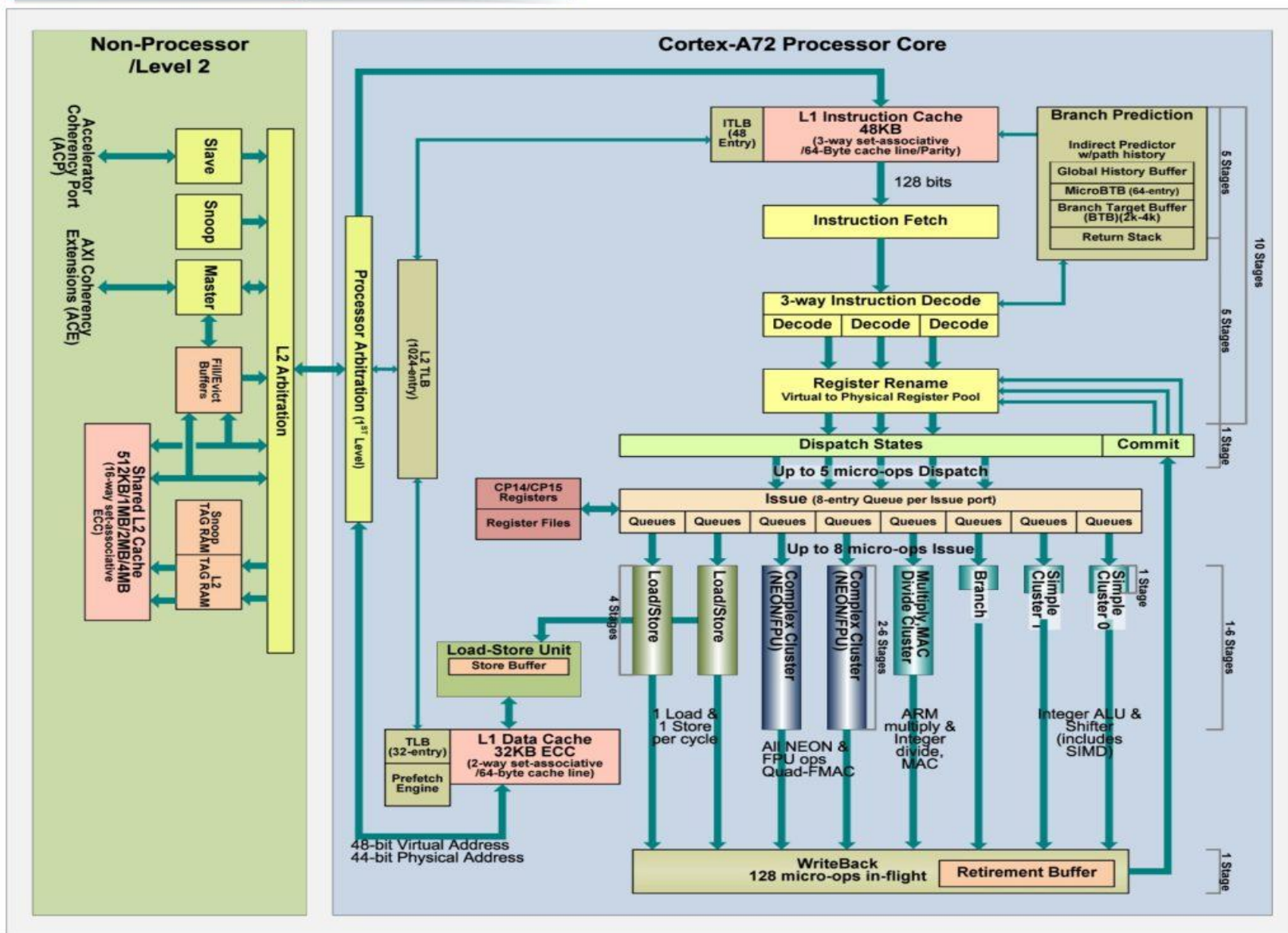
- 预测机制在处理器中广泛存在，利用各类预测机制，幽灵（Spectre）也衍生出很多变种



漏洞	Spectre变种编号	CVE漏洞号	原理
Spectre-PHT	Variant 1	CVE-2017-5753	PHT预测使访存绕过边界检查。
Spectre-PHT	Variant 1.1	CVE-2018-3693	PHT预测使访存绕过边界检查并访存。
Spectre-BTB	Variant 2	CVE-2017-5715	BTB预测，分支目标注入。
Meltdown	Variant 3	CVE-2017-5754	利用乱序执行缺少权限检查，实现越权信息泄露。
Spectre-STL	Variant 4	CVE-2018-3639	利用处理器对读写数据的预测前递操作。
Spectre-RSB	Variant 5		利用处理器对返回地址的预测。
.....			

微体系结构概念

ARM Cortex-A72 Block Diagram



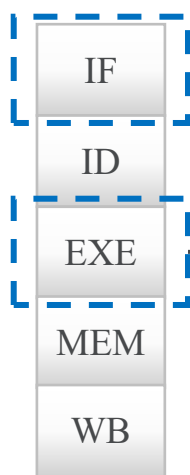
Copyright (c) 2015 Hiroshige Goto All rights reserved.

推测执行安全

思考幽灵漏洞根源分析

- 处理器的预测功能会被攻击者误导。
- 预测执行的指令会对Cache等微体系结构产生改变。
- 攻击者通过Cache等隐蔽信道还原出敏感数据。

if (x < array1_size)



在取指IF后，分支预测器判定指令类型并作出跳转预测

攻击窗口

EXE执行阶段，处理器会解出分支指令的实际跳转目标。

处理器按错误分支执行

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

改变Cache状态

Cache



○安全防护的思路

- 分支预测器隔离

- Cache delay-on-miss

- 控制流与数据流分析

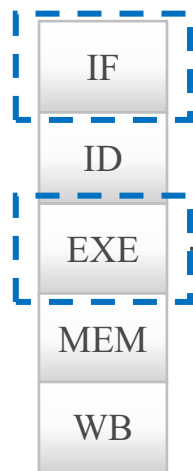
挑战：既要考虑安全，又要兼顾性能

推测执行安全

思考幽灵漏洞产生的原因

- 处理器的预测功能会被攻击者误导。 \Rightarrow 分支预测器隔离
- 预测执行的指令会对Cache等微体系结构产生改变。
- 攻击者通过Cache等隐蔽信道还原出敏感数据。

if (x < array1_size)



在取指IF后，分支预测器判定指令类型并作出跳转预测

攻击窗口

处理器按错误分支执行

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

EXE执行阶段，处理器会解出分支指令的实际跳转目标。

改变Cache状态

Cache



...



...



...

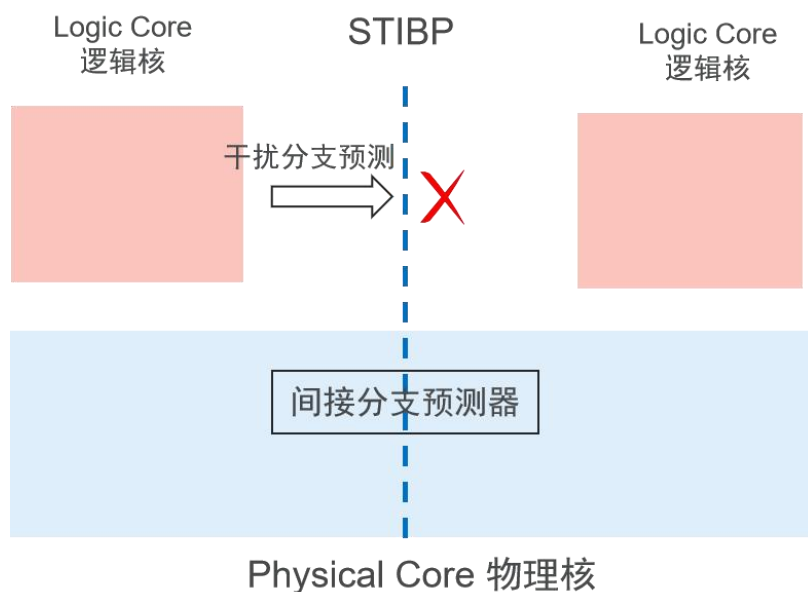


推测执行安全

○ 分支预测器隔离（产业界方案）

○ Intel/AMD – STIBP

- Single Thread Indirect Branch Predictors (独立线程间接分支预测器)
- 在超线程（Hyper-Threading）处理器上隔离同一物理核上不同逻辑核的间接分支预测，防止互相影响。

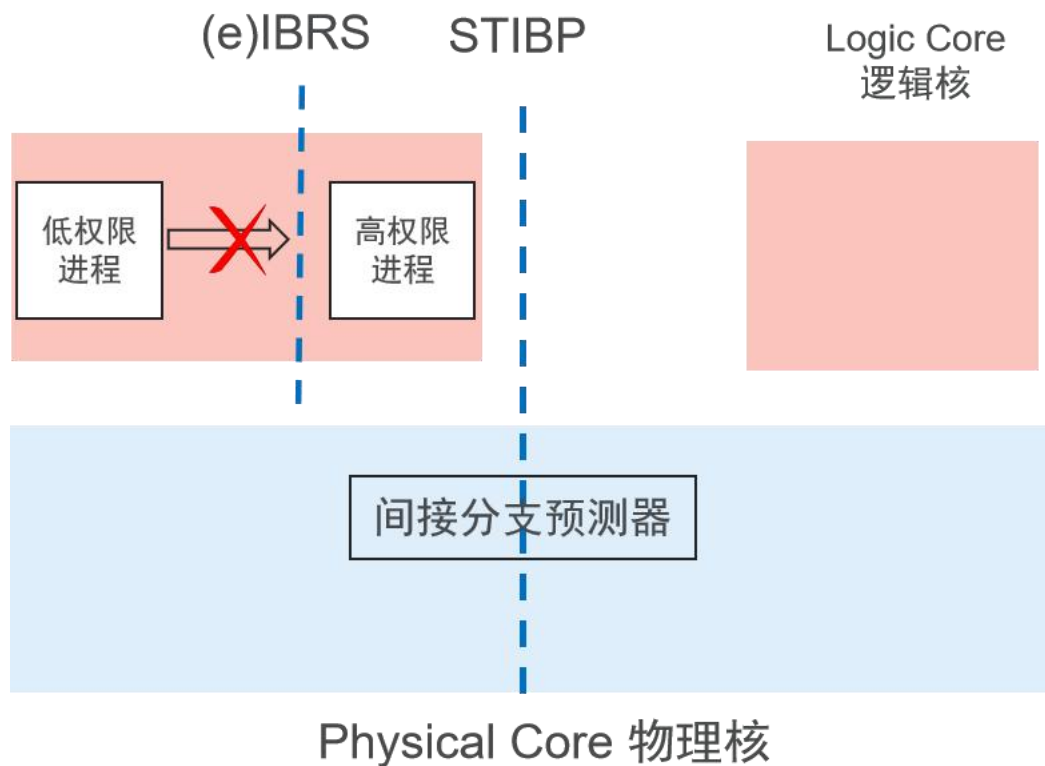


推测执行安全

○ 分支预测器隔离（产业界方案）

○ Intel/AMD – (e)IBRS

- (enhanced) Indirect branch restricted speculation (间接分支约束预测)
- 阻止低权限程序的间接分支预测干扰高权限程序。



○分支预测器隔离（产业界方案）尚有不足

- Linux 4.20更新对STIBP和IBRS的支持后，出现明显的性能下滑，在系统调用频繁以及多线程测试中的性能开销超过30%。Linux的发明者Linus Torvalds直接批评Intel的STIBP和IBRS就是垃圾。

*Is Intel really planning on making this shit architectural? Has anybody talked to them and told them they are f*cking insane?*

They do literally insane things. They do things that do not make sense. That makes all your [i.e. Woodhouse's] arguments questionable and suspicious. The patches do things that are not sane.

...So somebody isn't telling the truth here. Somebody is pushing complete garbage for unclear reasons. Sorry for having to point that out.

- enhanced IBRS优化了性能开销和开启步骤，Intel手册建议开启(e)IBRS不需要额外开启STIBP，IBRS也会对硬件线程进行检查。
- 最新研究发现(e)IBRS可以被绕过 (Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks (2023 Usenix Security))。

推测执行安全

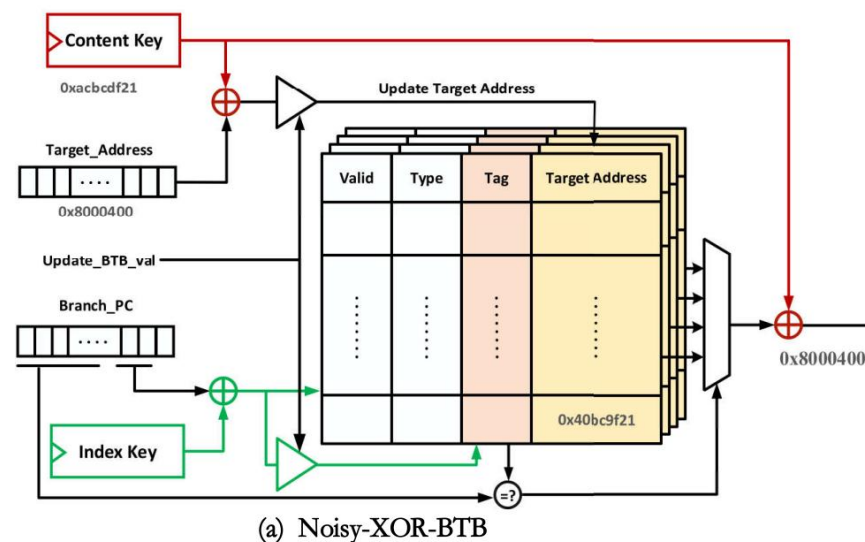
分支预测器隔离（学术界方案）

A Lightweight Isolation Mechanism for Secure Branch Predictors (Lutan Zhao, Peinan Li, Rui Hou, 2021 DAC)

对硬件线程通过加密提供隔离保护。

硬件生成随机数Content Key，不同硬件线程不同权限有独立的随机数。该随机数需要作为上下文信息维护。

用Content Key通过XOR异或操作实现对源地址和目标地址的加密。

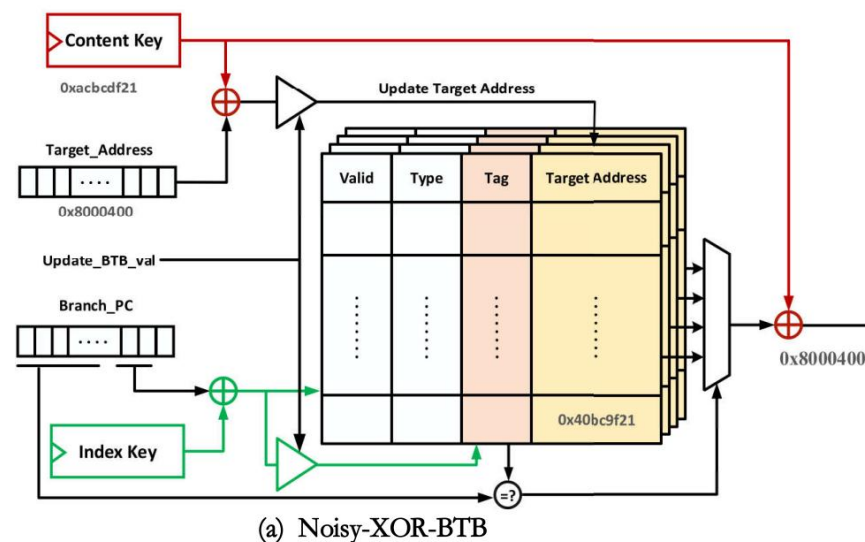


推测执行安全

分支预测器隔离（学术界方案）

A Lightweight Isolation Mechanism for Secure Branch Predictors (Lutan Zhao, Peinan Li, Rui Hou, 2021 DAC)

- 对BTB条目进行随机化。
- 硬件生成随机数Index Key，不同硬件线程不同权限有独立的随机数。该随机数需要作为上下文信息维护。
- 源地址与Index Key XOR异或后再检索BTB。
- Index Key需要定期更新（时间随机化）
- 平均性能开销1.3%



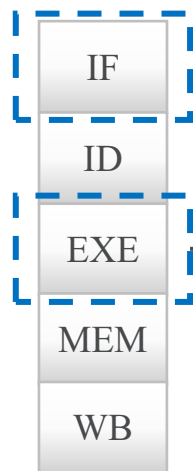
推测执行安全

思考幽灵漏洞产生的原因

- 处理器的预测功能会被攻击者误导。
- 预测执行的指令会对Cache等微体系结构产生改变。
- 攻击者通过Cache等隐蔽信道还原出敏感数据。

Cache隐蔽信道防御

if (x < array1_size)



在取指IF后，分支预测器判定指令类型并作出跳转预测

攻击窗口

处理器按错误分支执行

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

EXE执行阶段，处理器会解出分支指令的实际跳转目标。

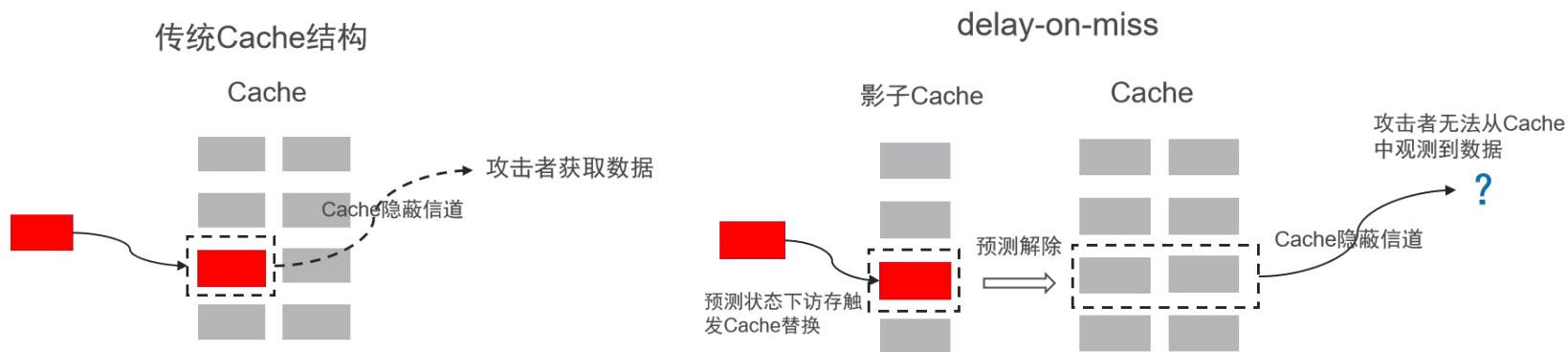
改变Cache状态

Cache



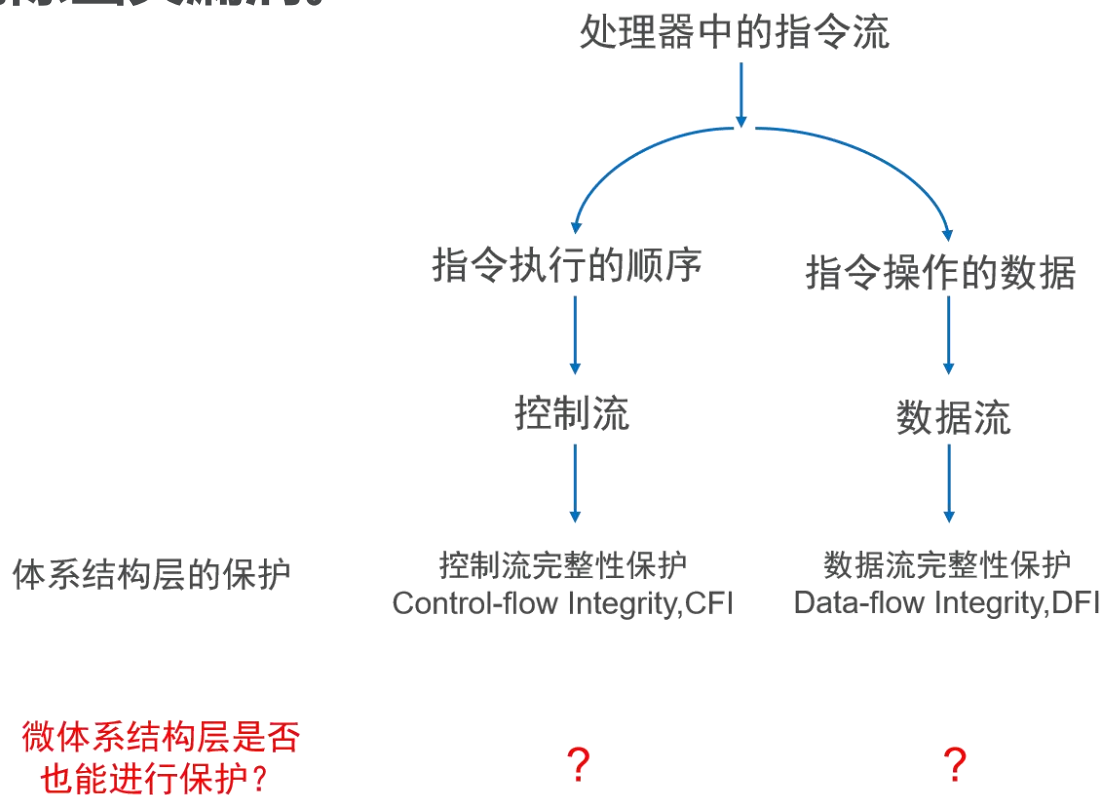
○ Delay on Miss

- 对Cache进行修改，添加额外的影子Cache。
- 关注预测状态下触发Cache miss的访存操作。
- 如果预测状态下执行的指令对内存的访问触发了Cache miss，需要更新Cache，Cache的更新内容存储在影子Cache中。
- 等待该指令预测状态解除，如果该指令有效，则影子Cache中的对应数据更新入Cache，如果无效，则丢弃影子Cache中的对应数据。



推测执行安全

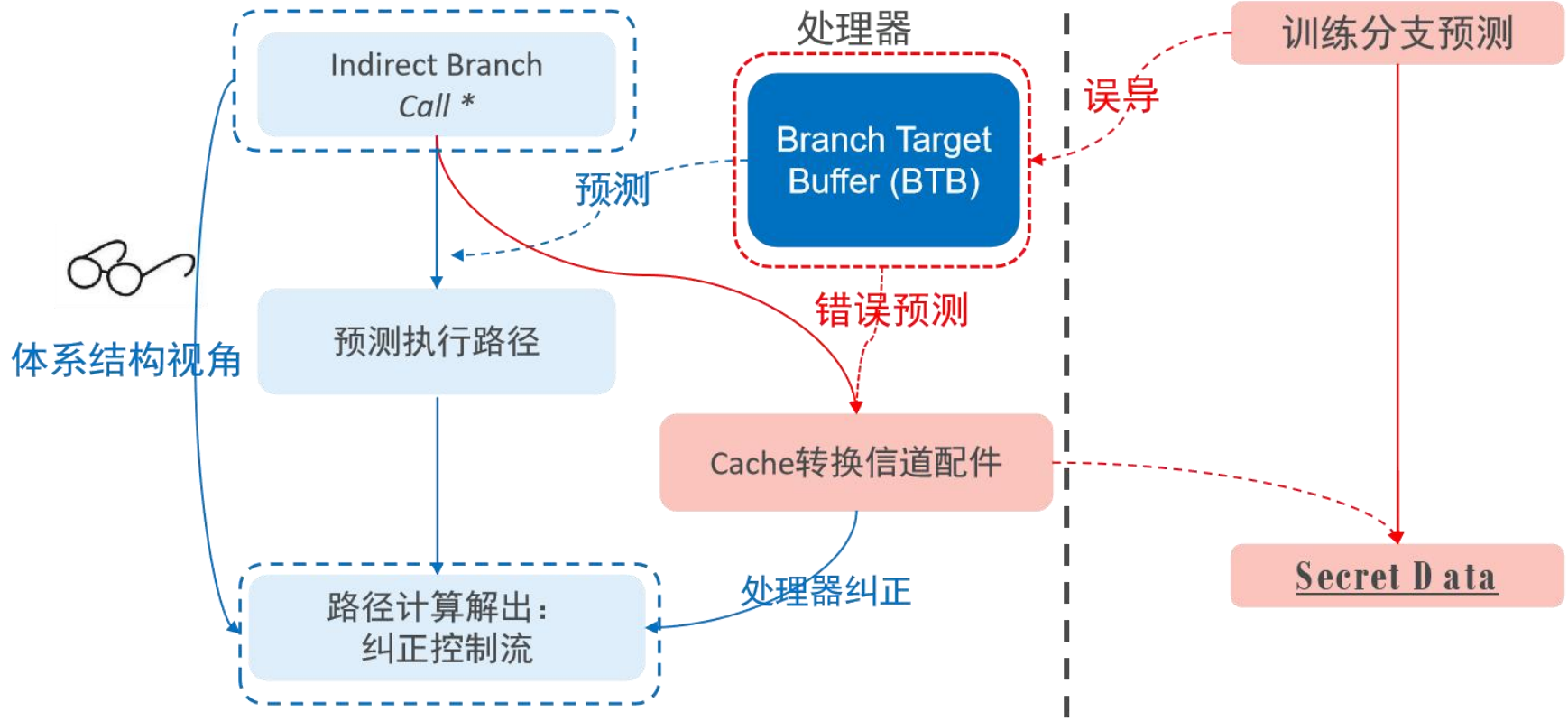
- 上述防御方法只针对幽灵漏洞攻击的部分关键点，并没有深入微体系结构层中执行的指令行为。
- 研究者尝试从微体系结构中执行的指令入手进行分析，从根源缓解甚至消除幽灵漏洞。



Spectre-v2

被攻击者

Spectre 攻击者

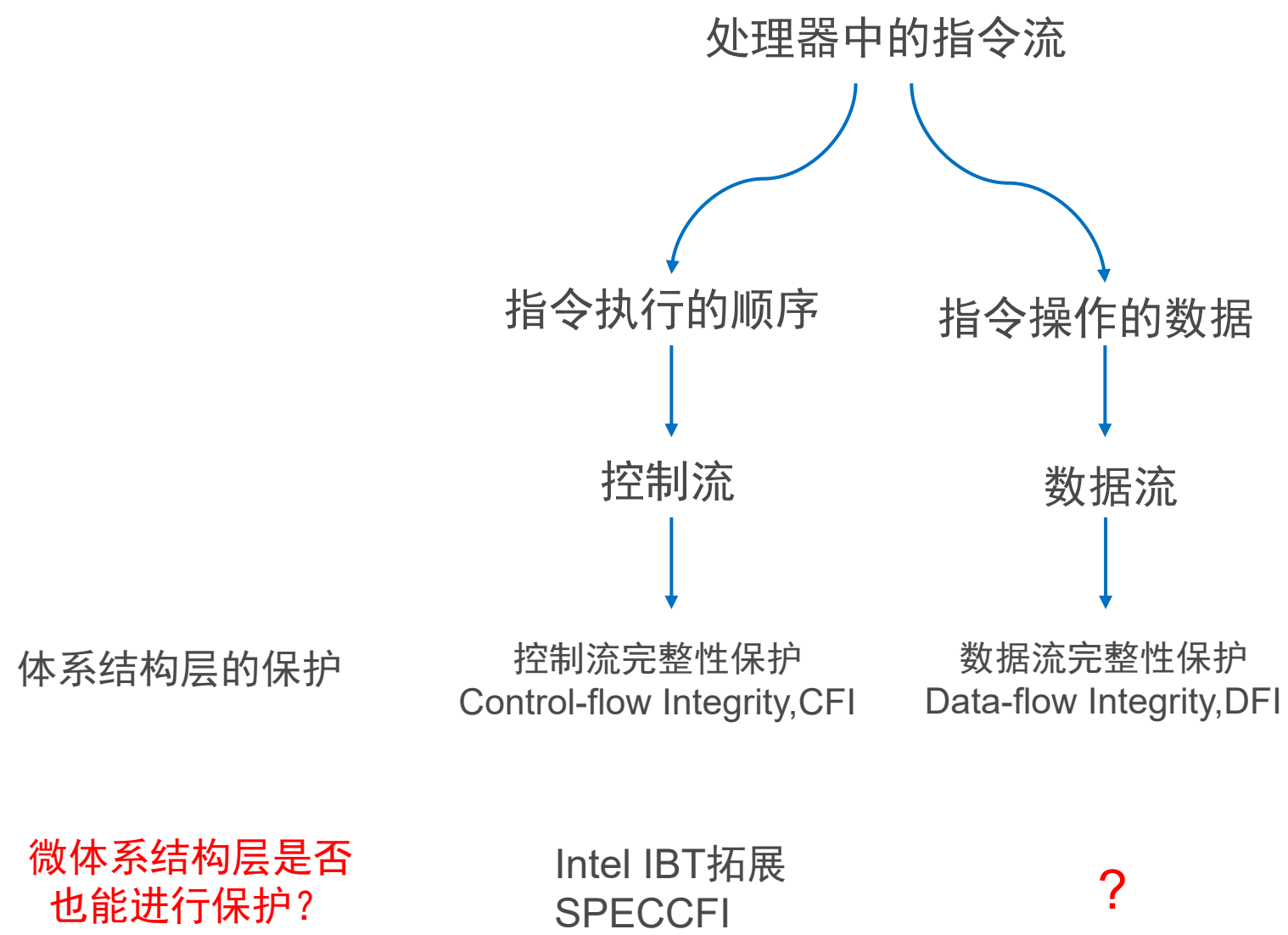



推测执行安全

- Spectre攻击中，攻击者误导处理器的预测部件，其实就是在微体系结构层篡改了程序的控制流。只是这种篡改处理器会进行纠正，使得体系结构层不可见。
- 可以用控制流完整性思想保护微体系结构层的程序控制流，从而阻止幽灵漏洞。
- 如果在微体系结构层中发现违反控制流完整性，应该进行什么处理？

控制流	关注点	处理方法
体系结构层	体系结构层状态是确定的，不可撤销的。要在体系结构层阻止攻击	报告错误，终止程序
微体系结构层	保证预测执行状态下没有危险指令执行	延迟指令执行，直到预测解除，但不需要报告错误。

- Intel CET中的前向控制流完整性保护机制IBT(Indirect Branch Tracking)拓展到处理器前端，用于防御Spectre-v2 (Spectre-BTB) 攻击。
- SpecCFI: Mitigating Spectre Attacks using CFI Informed Speculation (2020 IEEE Security & Privacy)则是进一步将细粒度的控制流完整性保护用于了微体系结构，并且将RSB对返回地址的预测用于防御Spectre-RSB攻击。
- 局限
 - 传统软件控制流完整性保护只涵盖间接分支跳转 (jmp *, call *) 和返回 (return)。在控制流视角中，if条件分支的两条路径都是合法控制流。控制流完整性思想防御幽灵漏洞只能用于**间接分支预测** (Spectre-v2, Spectre-BTB) 和**返回地址预测** (Spectre-RSB) 的保护。



- 数据流思想用于微体系结构安全  关注敏感数据在微体系结构中可能存在的泄漏。
- 污迹追踪 (Taint tracking) 是数据流保护中广泛使用的关键技术, 即将关注的敏感数据标记为被污染, 污染标记随着数据的传递一起传递, 当对被污染的数据进行关键操作时, 需要进行检查。
- 幽灵漏洞中哪些是敏感数据?
 - 预测执行状态下引入的可能改变微体系结构状态的数据是敏感数据。
- 哪些是敏感操作?
 - 会改变Cache等微体系结构状态的操作。

推测执行安全

```
1  x = secret_index
   // x > array1_size且array1[x] = secret
2  if ( x < array1_size)
3      y = array2 [ array1[x] * 4096 ];
```

预测
执行
窗口

1. load **a** <- array1[x];

2. **b** = **a** * 4096;

3. load y <- array2[**b**];

- 用污迹追踪来对预测执行的数据流进行分析。
- 第1条指令在预测状态下进行访存，第一次访存并不会造成信息泄露，认为是安全的。
- 但是第1条指令访存导致Cache中加载了**a**，应该标记为被污染。

推测执行安全

```
1  x = secret_index
   // x > array1_size且array1[x] = secret
2  if ( x < array1_size)
3      y = array2 [ array1[x] * 4096 ];
```

预测
执行
窗口

1. load **a** <- array1[x];

2. **b** = **a** * 4096;

3. load y <- array2[**b**];

- 第2条是算术操作，本身不构成信息泄漏的可能，所以预测状态下可以正常执行。
- 污迹追踪，需要维护污染标签的传递，**b**应该也标记被污染。

推测执行安全

```
1  x = secret_index
   // x > array1_size且array1[x] = secret
2  if ( x < array1_size)
3      y = array2 [ array1[x] * 4096 ];
```

预测
执行
窗口

1. load **a** <- array1[x];

2. **b** = **a** * 4096;

3. load y <- array2[**b**];

- 第3条指令是访存操作，属于危险的敏感操作。
- 第3条指令用了被污染的数据b。
- 敏感操作 + 使用被污染的数据 = 触发延迟执行
- 延迟执行第3条指令，直到if分支解出实际的路径。

推测执行安全

- Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data (2019 MICRO) 提出的预测污迹追踪STT方法。
- 这类方法关注预测窗口中数据的传递和泄漏可能，对潜在的可能泄露数据的指令执行延迟执行策略。

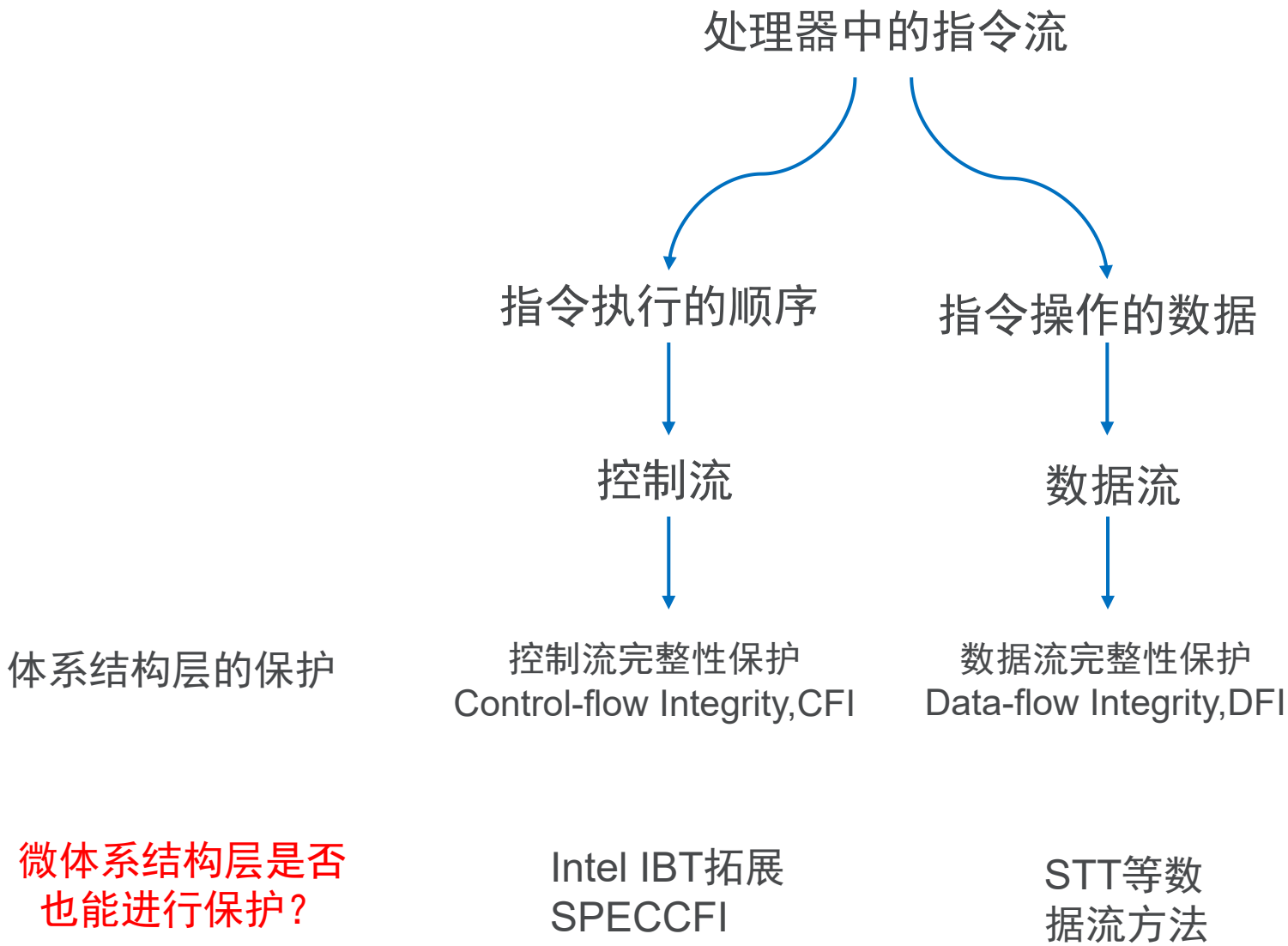
```
1  x = secret_index
    // x > array1_size且array1[x] =
    secret
2  if ( x < array1_size)
3      y = array2 [ array1[x] * 4096 ];
```

预测
执行
窗口

```
1. load  a <- array1[x];
2.  b = a * 4096;
3. load  y <- array2[b];
```

- 例子中选用了典型的Spectre-v1配合Cache信道，敏感操作选择了会改变Cache的访存指令。实际中根据不同的安全需求，可以对敏感指令进行不同的划分。
- 在条件分支解出时释放第3条指令的执行，根据不同的安全需求，可以进行更严格的策略，比如等待第3条指令到达ROB顶部后才可以执行。

- 数据流方法能实现对幽灵漏洞的综合性防御
 - 幽灵漏洞衍生出很多变种，但攻击者利用幽灵漏洞的核心目的始终是泄漏敏感数据。
 - 数据流方法始终围绕敏感数据进行追踪，所以能有效防御幽灵漏洞和各个变种，甚至可以对潜在的尚未发现的幽灵漏洞变种进行防御。
 - 根本原因：攻击目的与防御手段的统一。



内容概要

- 微体系结构安全
 - 高速缓存 (Cache) 安全
 - 乱序执行安全
 - 推测执行安全
- 总结

○ 介绍计算机微体系结构安全的保护

○ Cache保护

1. 敏感指令管理
2. 分区机制
3. 随机化方法

○ 熔断漏洞

1. 权限检查
2. 内核页表隔离 (KPTI)

○ 预测执行和幽灵漏洞

1. 隔离机制
2. Delay-on-miss
3. 控制流保护
4. 数据流保护

- 结合PoC代码研究幽灵漏洞及其变种（包括但不限于V1-V5）的攻击原理，并尝试至少一种进行实际运行。

Q&A