# 软件安全与漏洞分析

## 3.3 返回导向编程的发展（2）

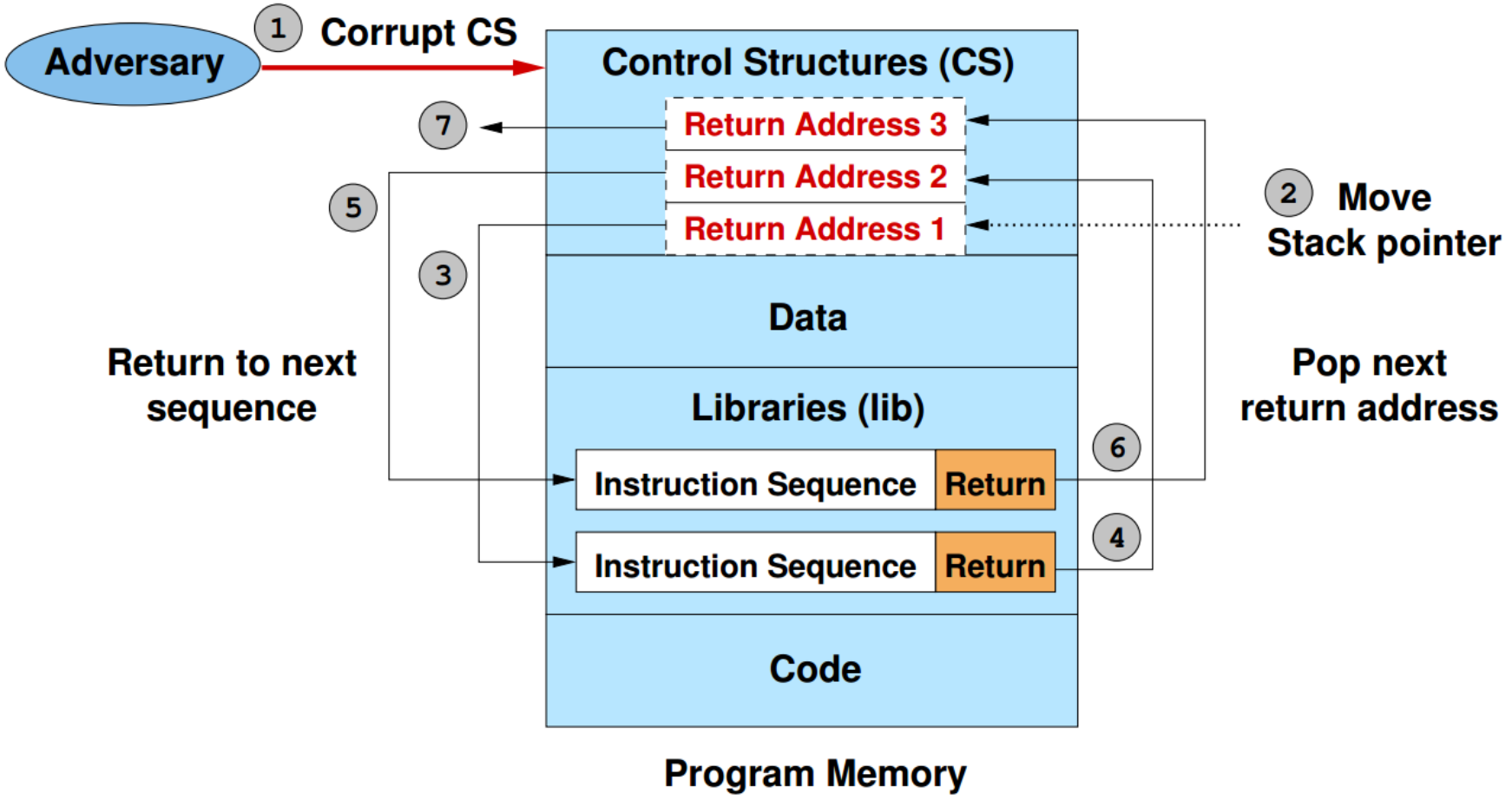
# Previously in Software Security

- RISC环境下的返回导向编程
- 返回导向编程的防御思路之一
  - 防御：破坏返回导向编程的原子指令组件（gadget）
  - 应对性的新型返回导向编程：不使用ret的ROP

# 返回导向编程的发展（2）

- 本节主题 – 能否从结构上阻止程序控制流被劫持（进而防范返回导向编程）？
  - 程序控制流完整性保护（control flow integrity, CFI）的基本思想
  - 实用（粗粒度）的CFI
  - 基于动态优化的CFI
  - 利用硬件特性和虚拟化技术的CFI

# 回顾：返回导向编程的基本流程



# 回顾：返回导向编程的基本流程

- 始于对程序控制流的篡改（**控制流异常**）
- 各gadget由ret指令（或者pop-jump组合）代替eip加以链接（**大量控制流异常**）
- 原始栈结构遭到破坏，ROP过程中栈指针单向移动（**栈的行为异常**）
- 有时，栈指针可能被篡改并指向不属于栈区的内存（**栈的行为异常**）

# 控制流完整性保护

- 上述异常可否作为返回导向编程的特征？
  - 要求程序按照程序猿所规定的逻辑去执行
  - 当出现不应出现的控制转移行为时，阻止程序执行
  - 当程序的栈结构发生异常变化时，阻止程序执行
- 由此产生的返回导向编程防御思路：控制流完整性保护（CFI）技术

# 控制流完整性保护

- **理想的CFI**的提出：Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity[C]//Proceedings of the 12th ACM conference on Computer and communications security. ACM, 2005: 340-353.
- 基本思想：
  - 通过预设的运行时校验，确保程序执行与预先定义的控制流图严格吻合
  - 通过对二进制码的静态分析来获取CFI所需保证的控制流图
  - 藉由静态的二进制代码改写为程序添加运行时的自我校验

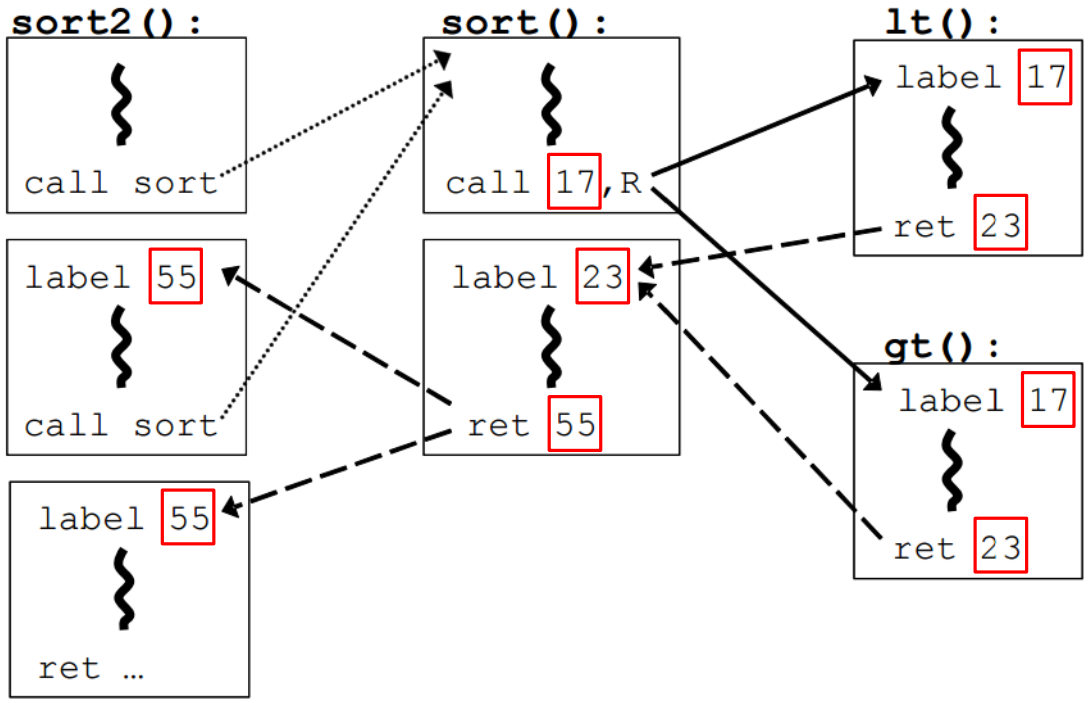
# 控制流完整性保护

- What to check?
  - 多数情况下，程序中的控制转移**指向某个常量目标**（正确性可静态验证）
  - 但程序中同样存在**控制转移目标在执行时才被计算出来**的情况（需要CFI动态校验）
- Where to check?
  - 间接跳转（jmp *x）/函数指针（call *x）/函数返回（ret）
  - 时机：在这些指令执行之前
- Check what?
  - 被校验控制转移的目的位置的合法性

# 控制流完整性保护

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



# 控制流完整性保护

| Bytes (opcodes) | x86 assembly code | Comment |
|---|---|---|
| FF E1 | jmp ecx | ; a computed jump instruction |
| 81 39 78 56 34 12 | cmp [ecx], 12345678h | ; compare data at destination |
| 75 13 | jne error_label | ; if not ID value, then fail |
| 8D 49 04 | lea ecx, [ecx+4] | ; skip ID data at destination |
| FF E1 | jmp ecx | ; jump to destination code |

| Bytes (opcodes) | x86 assembly code | Comment |
|---|---|---|
| 8B 44 24 04 | mov eax, [esp+4] | ; first instruction |
| ... | | ; of destination code |
| 78 56 34 12 | DD 12345678h | ; label ID, as data |
| 8B 44 24 04 | mov eax, [esp+4] | ; destination instruction |
| ... | | |

# 控制流完整性保护

| Bytes (opcodes) | x86 assembly code | Comment |
|---|---|---|
| FF E1 | jmp ecx | ; a computed jump instruction |
| B8 77 56 34 12 | mov eax, 12345677h | ; load ID value minus one |
| 40 | inc eax | ; increment to get ID value |
| 39 41 04 | cmp [ecx+4], eax | ; compare to destination opcodes |
| 75 13 | jne error_label | ; if not ID value, then fail |
| FF E1 | jmp ecx | ; jump to destination code |

| Bytes (opcodes) | x86 assembly code | Comment |
|---|---|---|
| 8B 44 24 04 | mov eax, [esp+4] | ; first instruction |
| ... | | ; of destination code |
| 3E 0F 18 05 78 56 34 12 | prefetchnta [12345678h] | ; label ID, as code |
| 8B 44 24 04 | mov eax, [esp+4] | ; destination instruction |
| ... | | |

# 控制流完整性保护

| Bytes (opcodes) | x86 assembly code | Comment |
|---|---|---|
| FF 53 08 | call [ebx+8] | ; call a function pointer |

is instrumented using `prefetchnta` destination IDs, to become:

| | | |
|---|---|---|
| 8B 43 08 | mov eax, [ebx+8] | ; load pointer into register |
| 3E 81 78 04 78 56 34 12 | cmp [eax+4], 12345678h | ; compare opcodes at destination |
| 75 13 | jne error_label | ; if not ID value, then fail |
| FF D0 | call eax | ; call function pointer |
| 3E 0F 18 05 DD CC BB AA | prefetchnta [AABBCCDDh] | ; label ID, used upon the return |

| | | |
|---|---|---|
| C2 10 00 | ret 10h | ; return, and pop 16 extra bytes |

is instrumented using `prefetchnta` destination IDs, to become:

| | | |
|---|---|---|
| 8B 0C 24 | mov ecx, [esp] | ; load address into register |
| 83 C4 14 | add esp, 14h | ; pop 20 bytes off the stack |
| 3E 81 79 04 DD CC BB AA | cmp [ecx+4], AABBCCDDh | ; compare opcodes at destination |
| 75 13 | jne error_label | ; if not ID value, then fail |
| FF E1 | jmp ecx | ; jump to return address |

# 控制流完整性保护

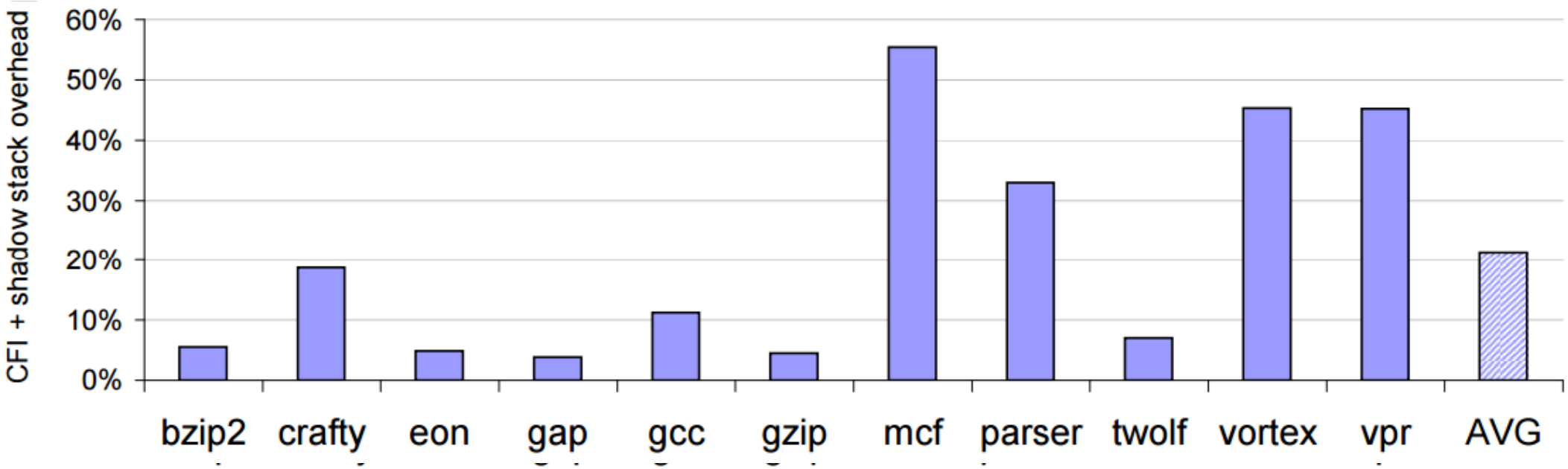
- CFI的基本安全性假设
  - 控制转移目标的标示符（ID）具有唯一性
  - 程序代码不可写
  - 程序数据不可执行

# 控制流完整性保护

- 进一步搭建：内存访问控制（software memory access control，SMAC）
  - 通过与运算，确保控制转移目标的最高位字节内容为40h
  - 使得间接控制转移总是指向合法的代码区段
  - 由此在一定程度上实现软件故障隔离（software fault isolation，SFI）
- 进一步搭建：受保护的影子栈（Protected Shadow Call Stack）
  - 回顾影子栈 – 专门划定的内存区域，用于检查ret指令是否响应了正确的call指令
  - 为影子栈预设地址前缀，结合SMAC，可确保只有CFI的校验指令可以修改该区域

# 控制流完整性保护

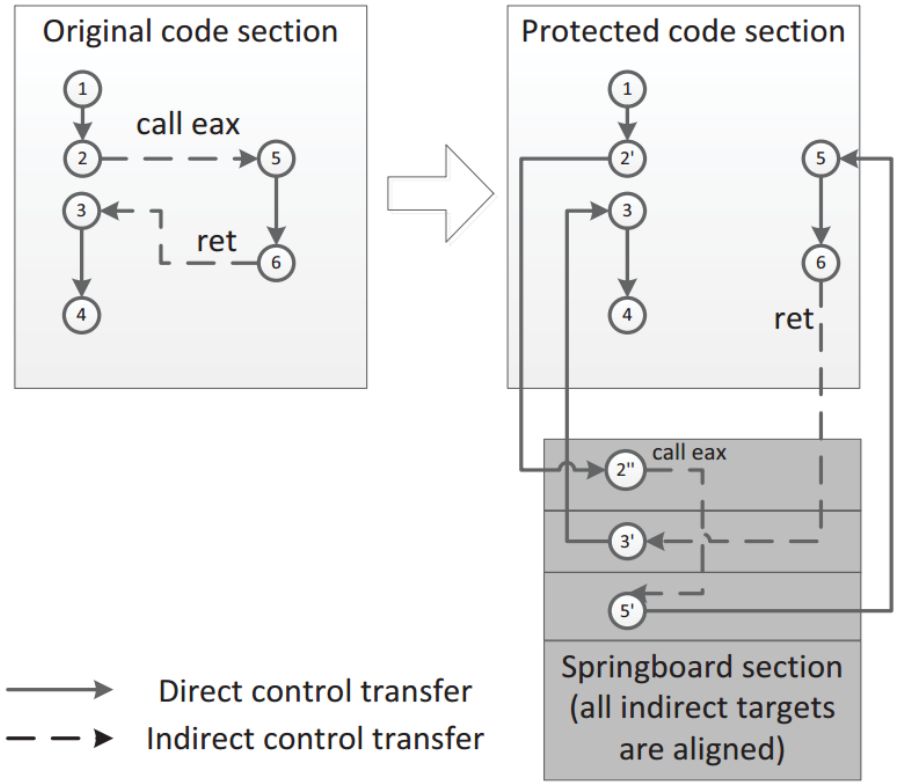
- 理想CFI的缺陷1 — 性能开销过大

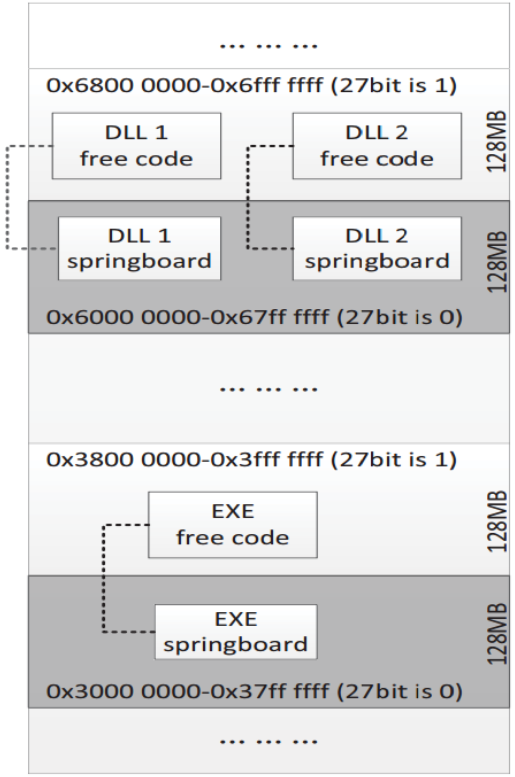



- 理想CFI的缺陷2 — 缺乏对兼容性问题的考虑（e.g. 共享库问题）

# 实用（粗粒度）的控制流完整性保护

- 改进思路：借鉴SFI的思想，优化CFI的校验机制



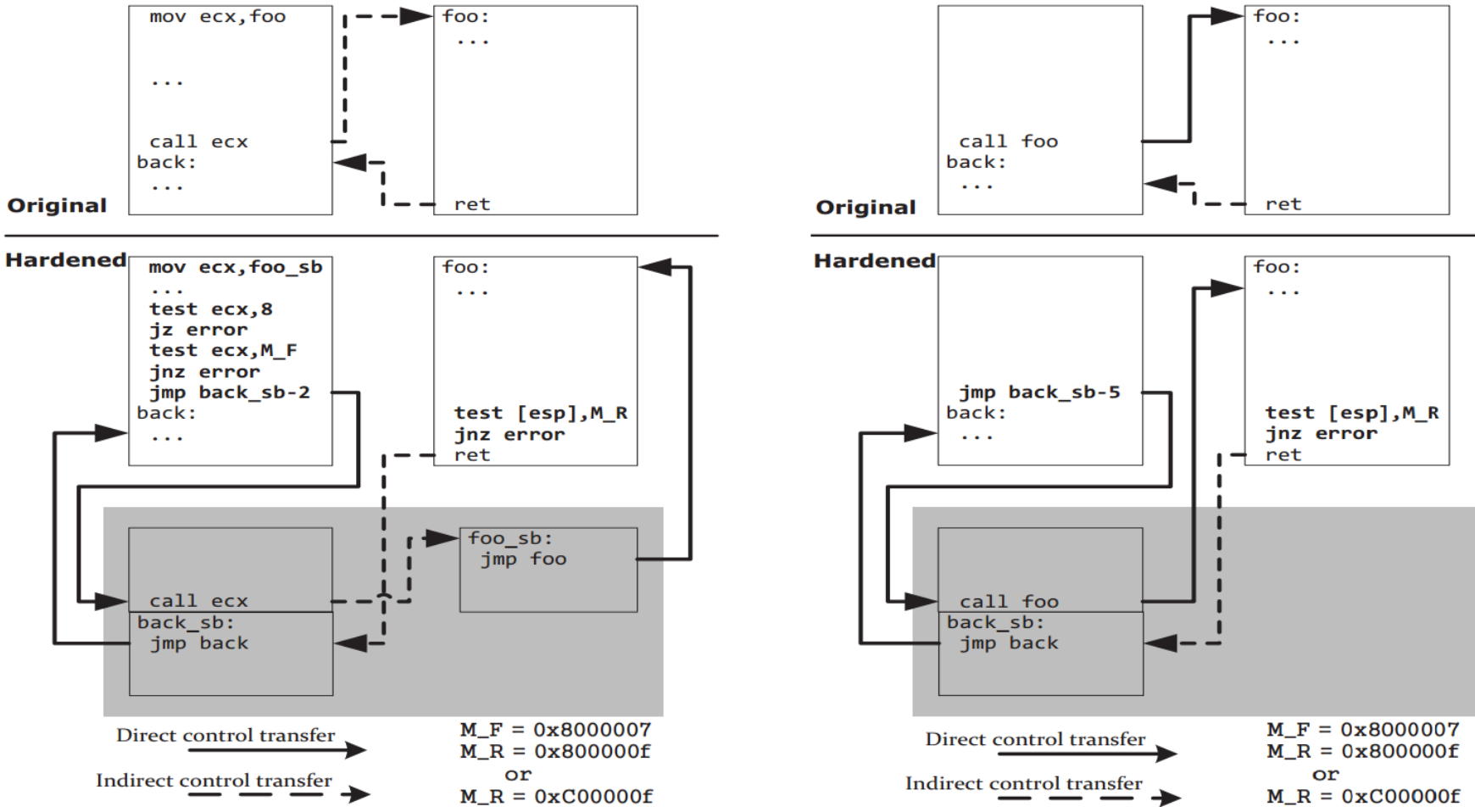
# 实用（粗粒度）的控制流完整性保护

- Springboard段的格式，使得**间接控制转移可以通过位校验进行检查**
  - Springboard段内地址的**第27位总是为0**，且仅有**Springboard段满足此条**
  - 规定**掩码**，使得Springboard段内的**各类不同存根总是按各自的规则对齐**

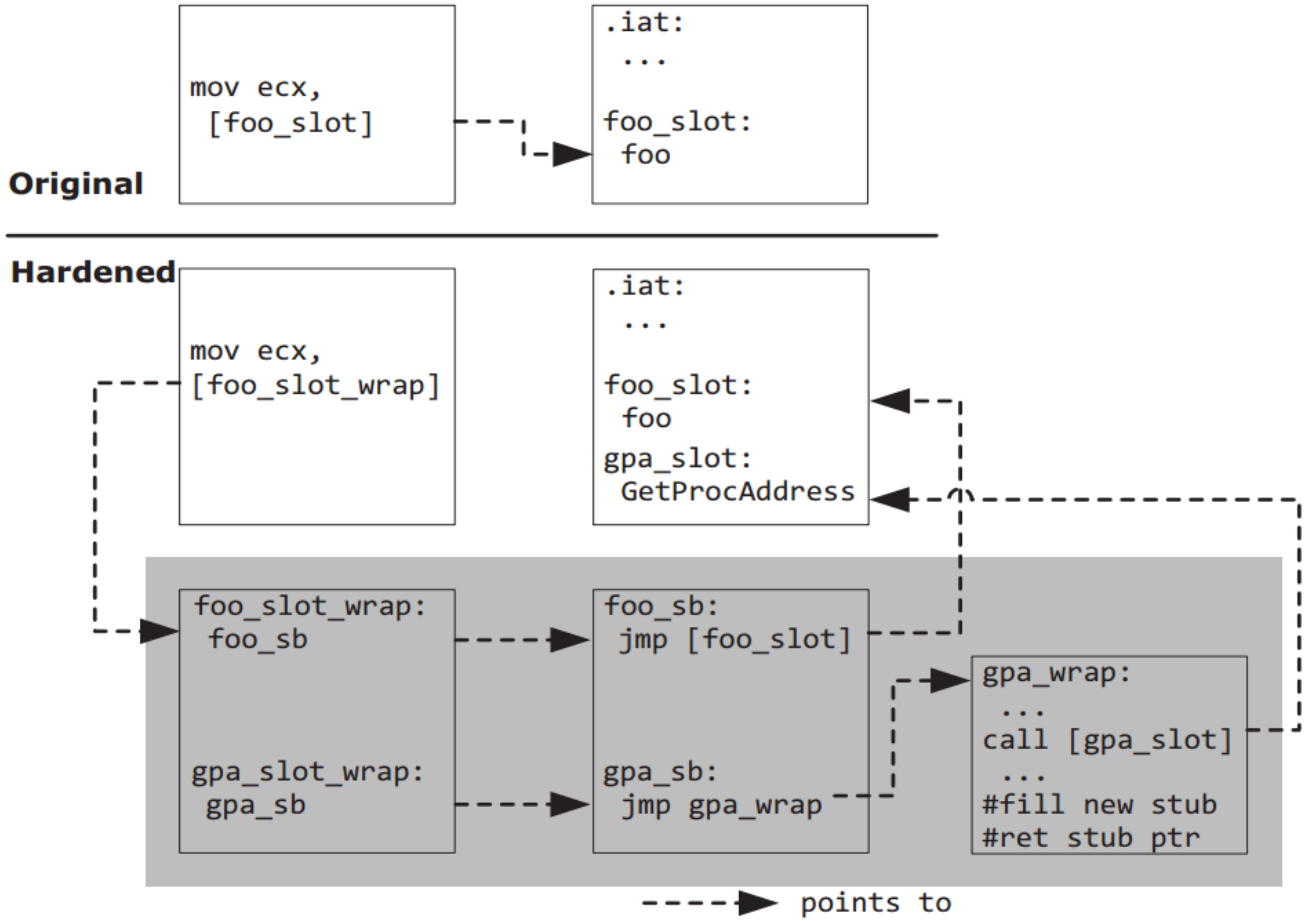
| Executable | Bits | | | | Meaning |
|---|---|---|---|---|---|
| | 27 | 26 | 3 | 2-0 | |
| no | * | * | * | *** | Non-executable section |
| yes | 1 | * | * | *** | Normal code section |
| yes | 0 | * | * | !000 | Springboard's invalid entry |
| yes | 0 | * | 1 | 000 | Springboard's function pointer stub |
| yes | 0 | 1 | 0 | 000 | Springboard's sensitive return stub |
| yes | 0 | 0 | 0 | 000 | Springboard's normal return stub |

- 设计效果：**间接控制转移只能以Springboard段内的适当存根作为目标**

# 实用（粗粒度）的控制流完整性保护

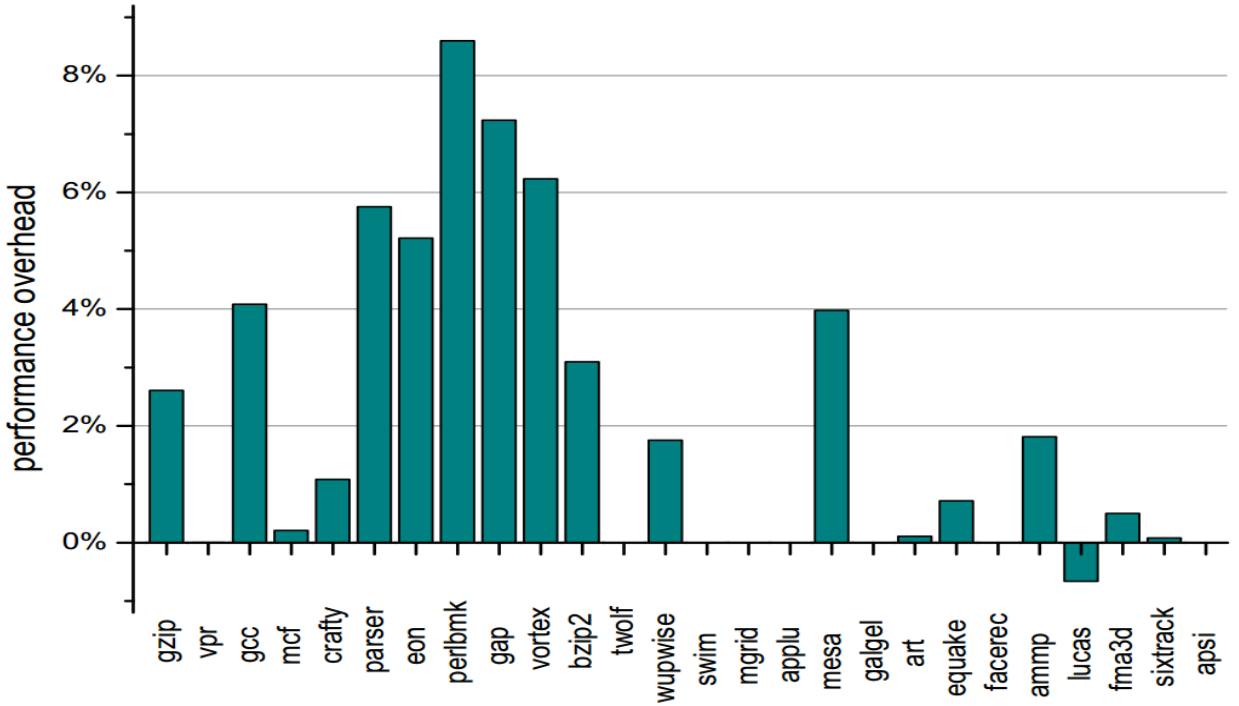


# 实用（粗粒度）的控制流完整性保护



# 实用（粗粒度）的控制流完整性保护

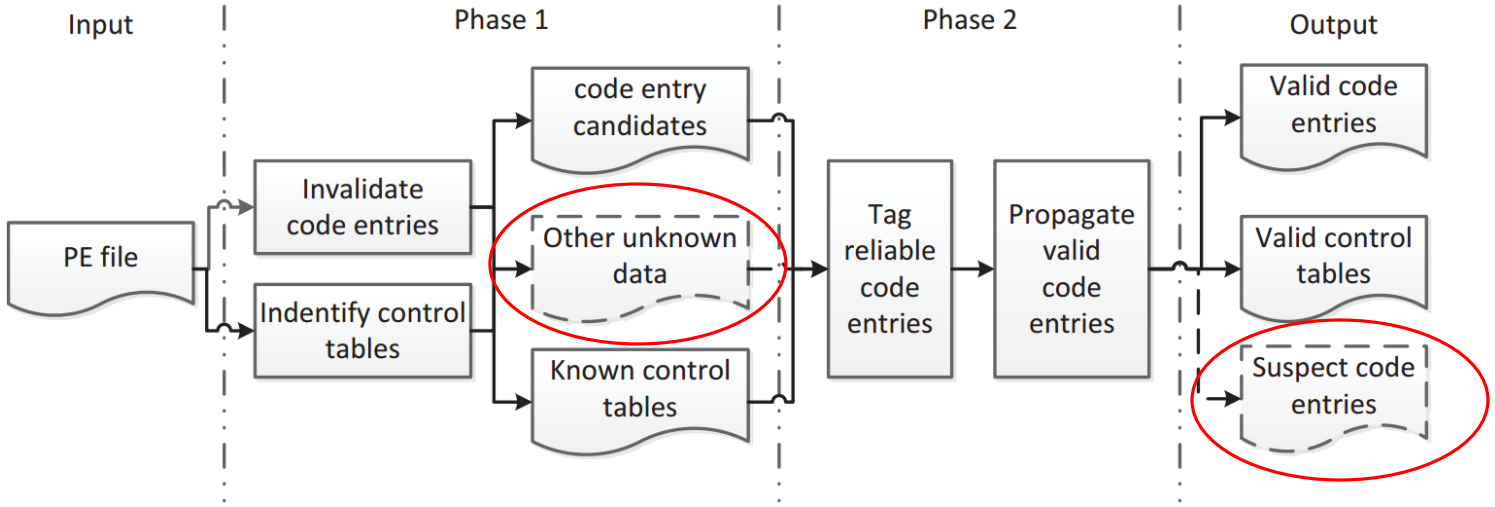
- 改进后的CFI性能开销




- 参考文献：Zhang C, Wei T, Chen Z, et al. Practical control flow integrity and randomization for binary executables[C]//Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013: 559-573.

# 基于动态优化的CFI

- 前述CFI方案仍然存在不尽人意之处：需要改写程序的二进制代码
  - 即使是最强大的二进制分析工具，也很难识别出程序中所有的合法控制转移目标



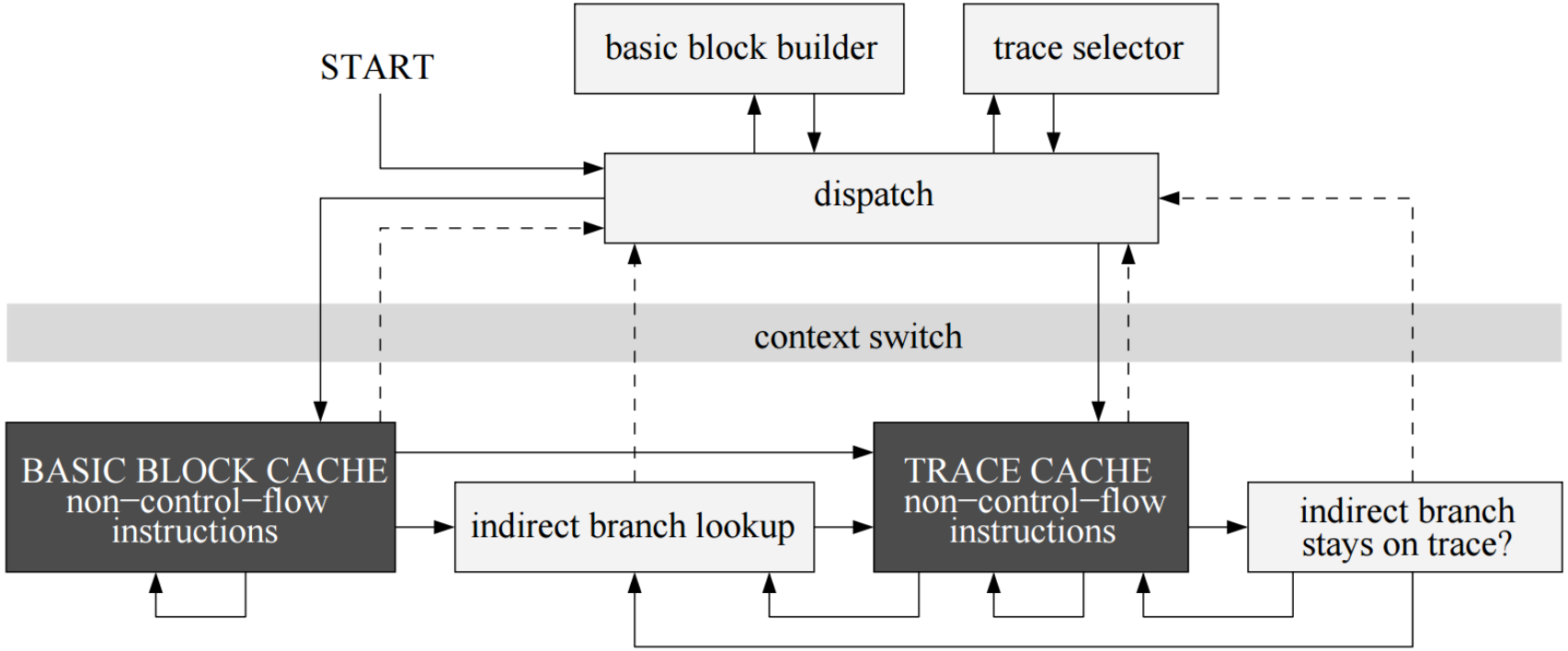

  - 对二进制代码的修改同样困难且容易出错，且兼容性问题在一定程度上仍然存在

# 基于动态优化的CFI

- 另一种思路：何不在程序执行过程中加以监视和约束？
  - 本类型CFI方案1 -- Kiriansky V, Bruening D, Amarasinghe S P. Secure Execution via Program Shepherding[C]//USENIX Security Symposium. 2002, 92: 84.
  - 本类型CFI方案2 -- Davi L, Sadeghi A R, Winandy M. ROPdefender: A detection tool to defend against return-oriented programming attacks[C]// Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM, 2011: 40-51.

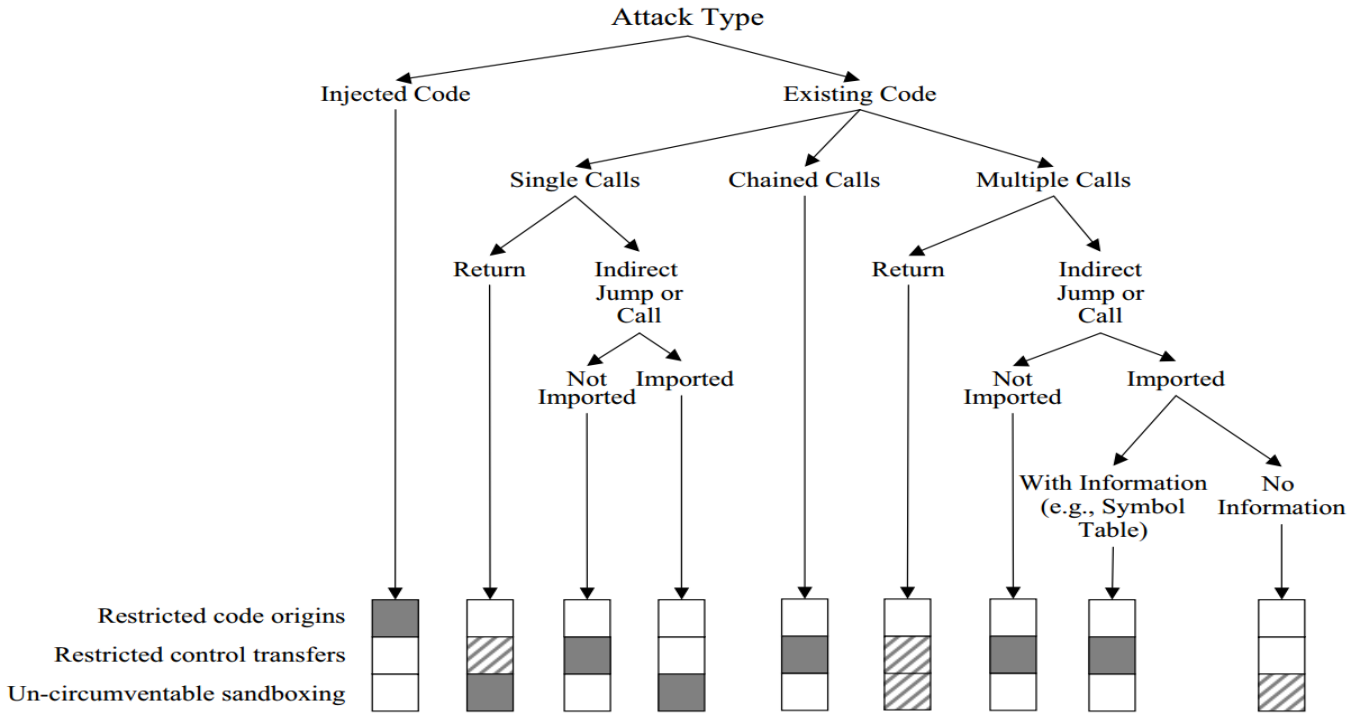
# 基于动态优化的CFI

- 设计基础：动态执行优化工具
  - 具体到Program Shepherding，基于DynamoRIO平台



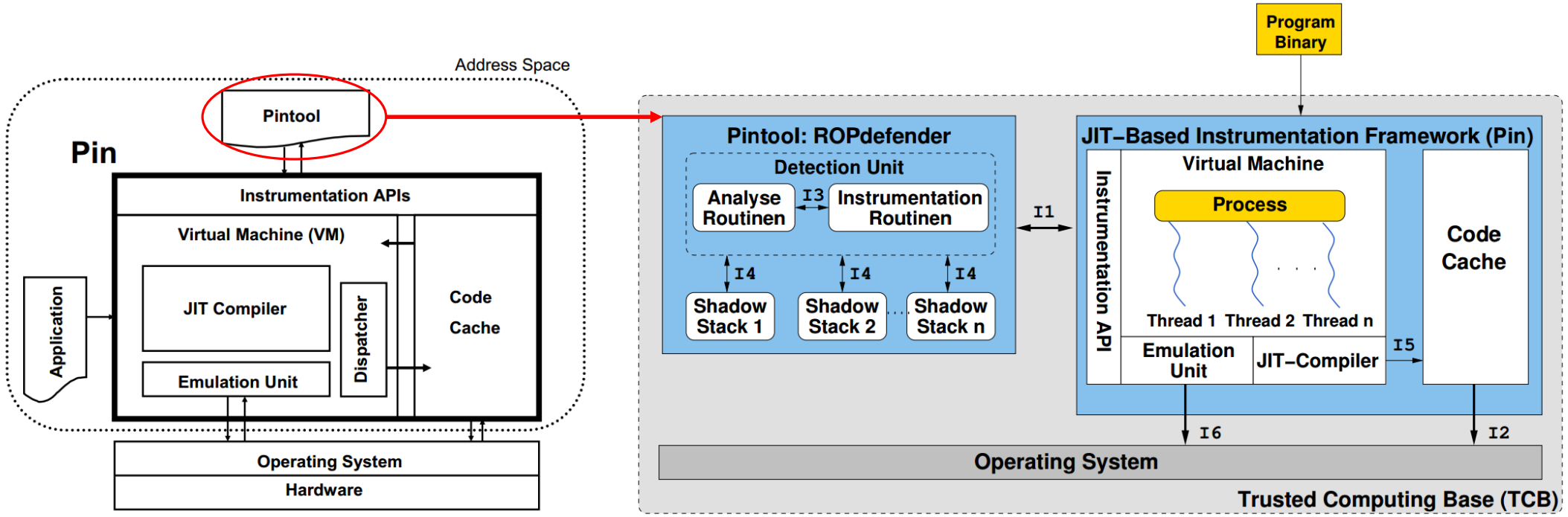
# 基于动态优化的CFI

- 设计基础：动态执行优化工具
  - 具体到Program Shepherding，基于DynamoRIO平台



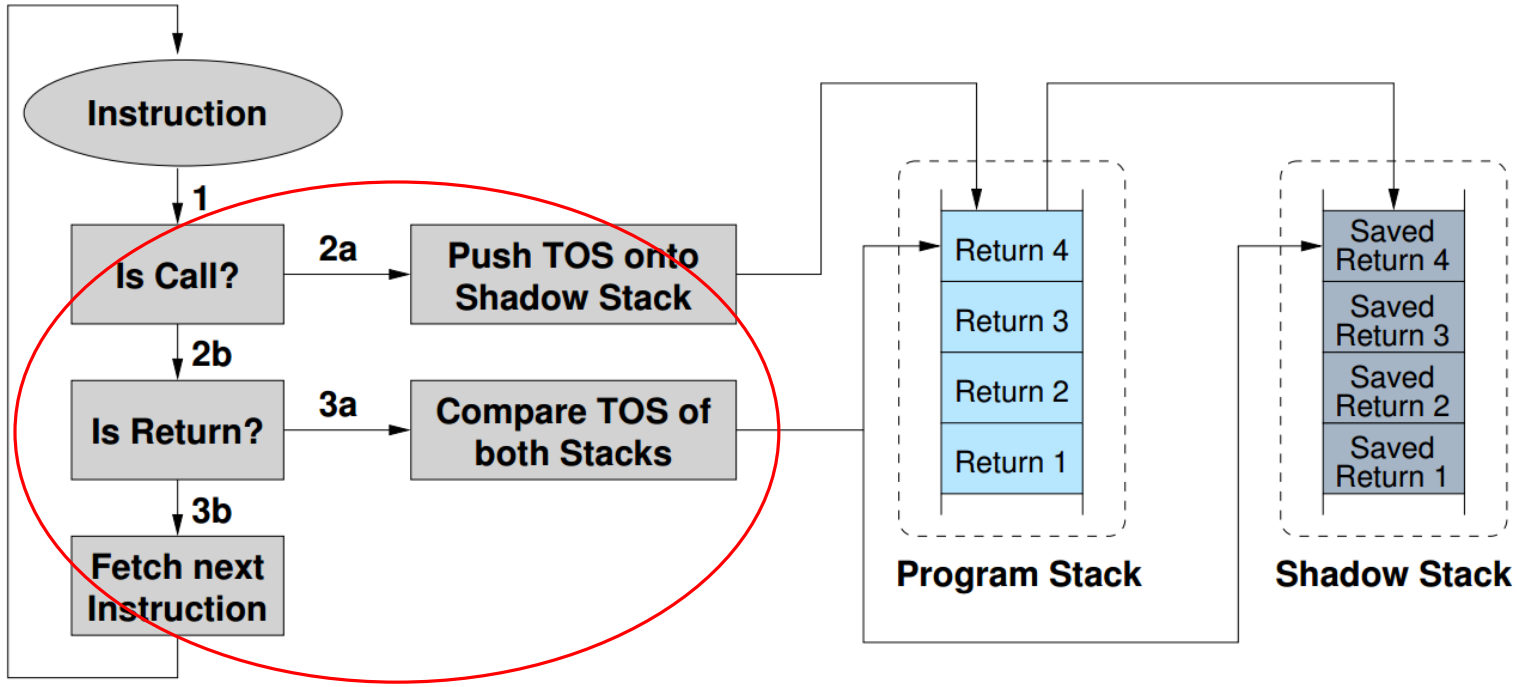
# 基于动态优化的CFI

- 设计基础：动态执行优化工具
  - 具体到ROPdefender，基于Pin平台



# 基于动态优化的CFI

- 设计基础：动态执行优化工具
  - 具体到ROPdefender，基于Pin平台

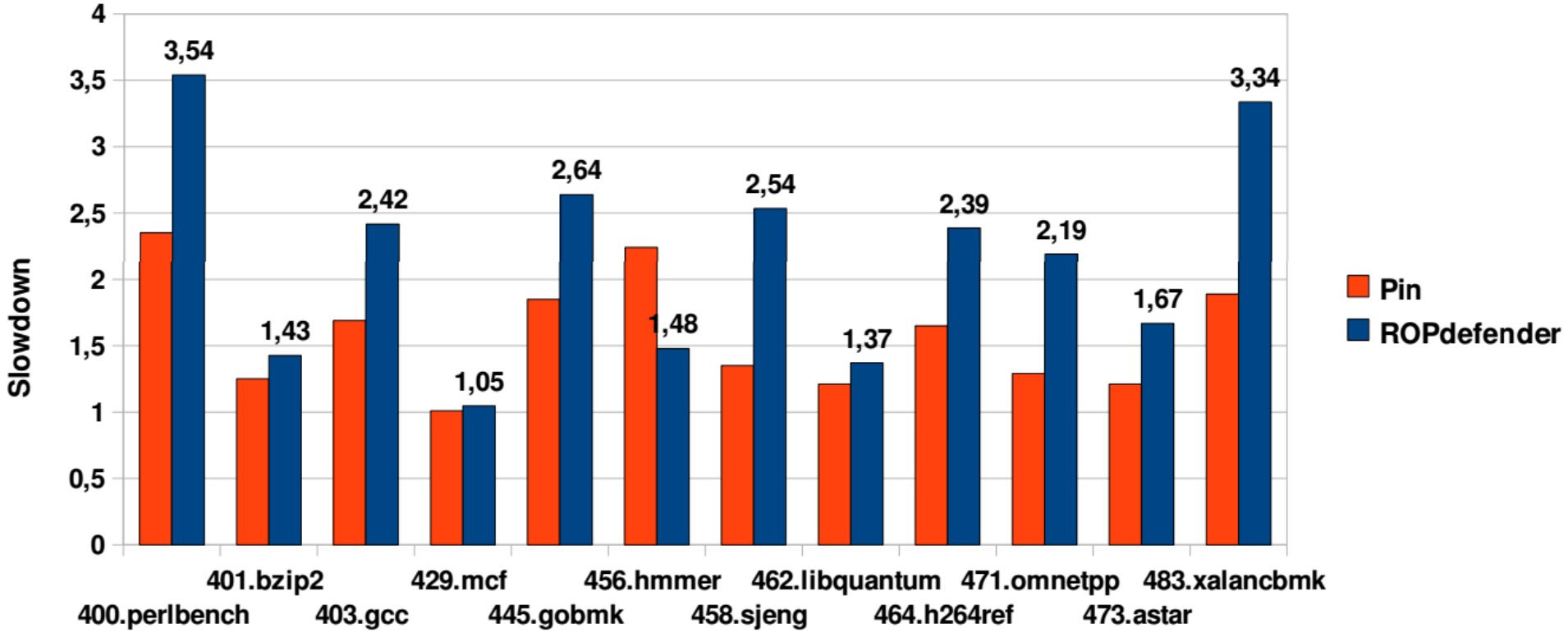


# 基于动态优化的CFI

- 设计基础：动态执行优化工具
  - 具体到ROPdefender，基于Pin平台
- 仍然是基于影子栈的防御思路，动态优化带来了什么变化？
  - 当遇到Setjmp/Longjmp：穷尽回溯直到寻获匹配，否则才认为异常
  - Unix signals和lazy binding：动态优化工具提供了相应的检测手段
  - C++异常：可以识别出异常处理流程，从而相应地寻获该流程计算出的返回地址

# 基于动态优化的CFI

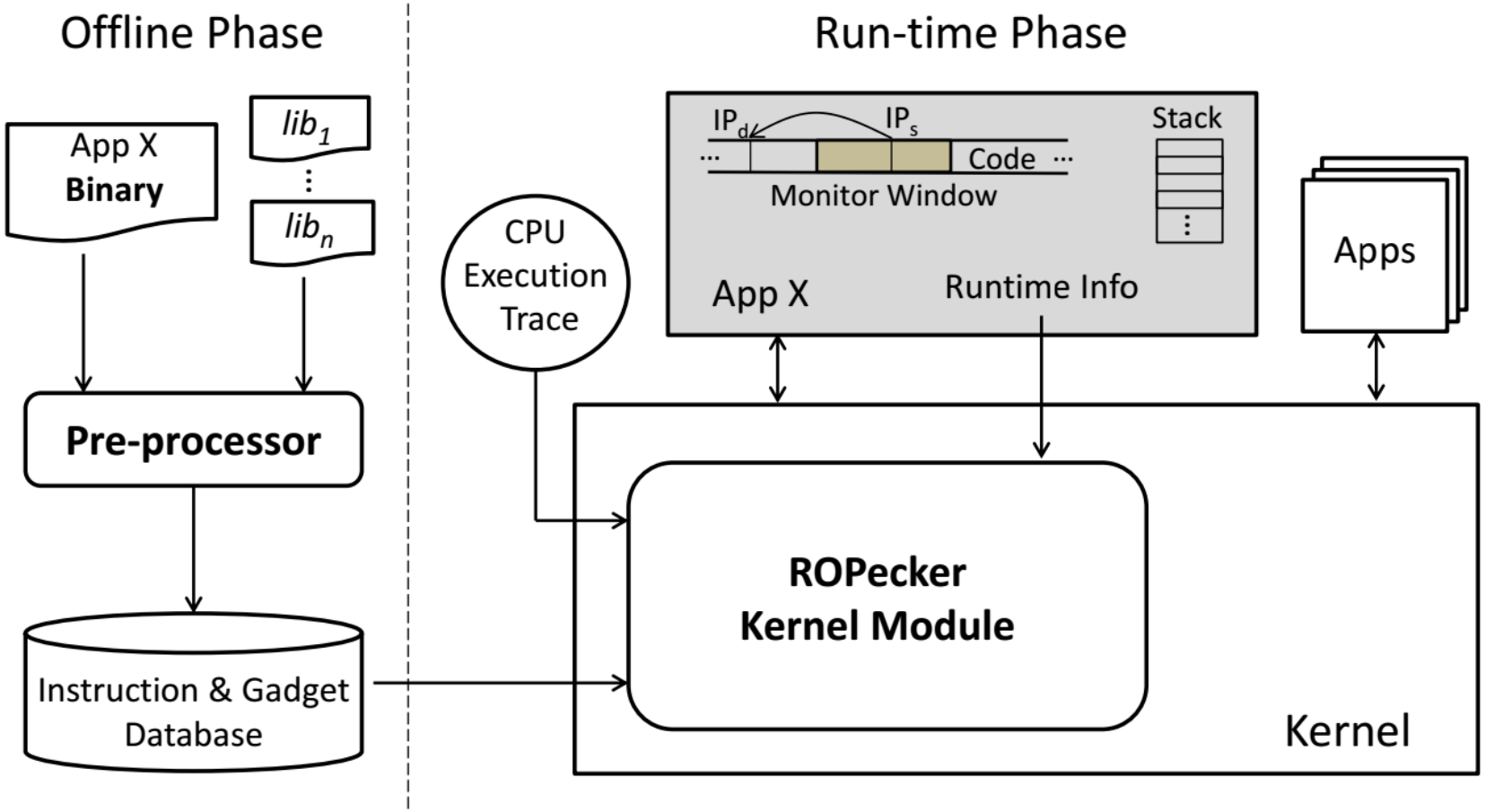
- 基于动态优化的CFI之不足：效率太低……



# 利用硬件特性和虚拟化技术的CFI

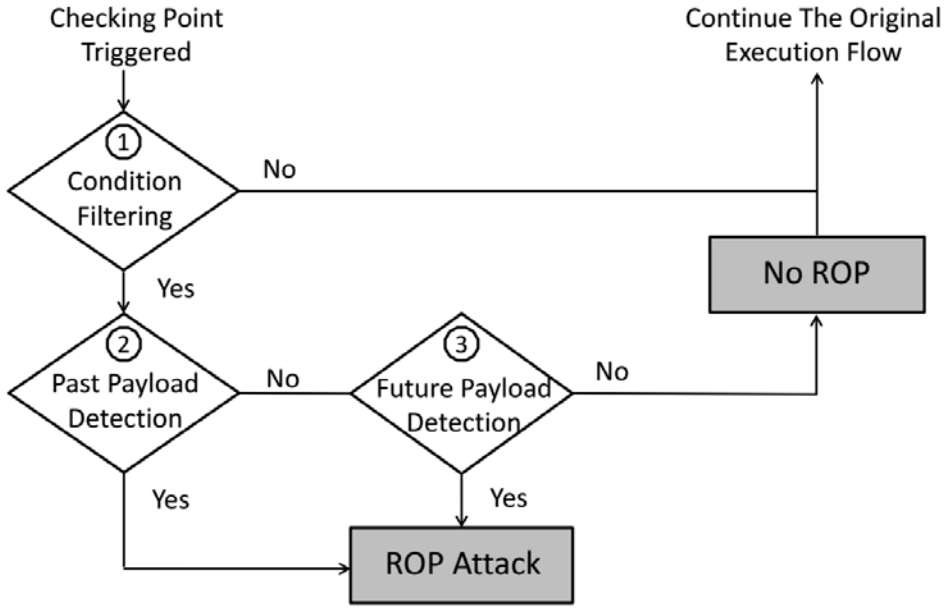
- 事实证明**软件虚拟机不靠谱**，但是**动态执行监视的思路仍然很诱人**
- 新思路：利用虚拟化技术
  - 内核模式下作为操作系统的一部分，性能开销低
  - 提供了动态优化工具可能提供的CFI优势
- 新机遇：Last Branch Record
  - 一组存在于CPU内部、记录其最后n次控制转移目标的寄存器（如Intel i5中设有16个）
  - 需要内核权限开启（恰好与虚拟化技术相适应）

# 利用硬件特性和虚拟化技术的CFI



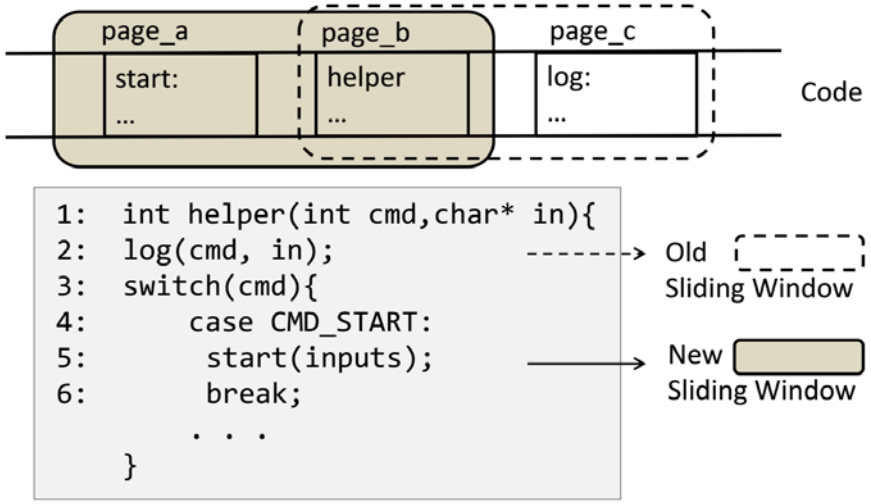
# 利用硬件特性和虚拟化技术的CFI

- 监视目标：**执行流由足够长的gadget序列组成**
  - 全部gadget列表在离线分析中可以容易地获取
  - 过往执行内容是否是gadget，可藉由LBR寄存器组准确地判断



# 利用硬件特性和虚拟化技术的CFI

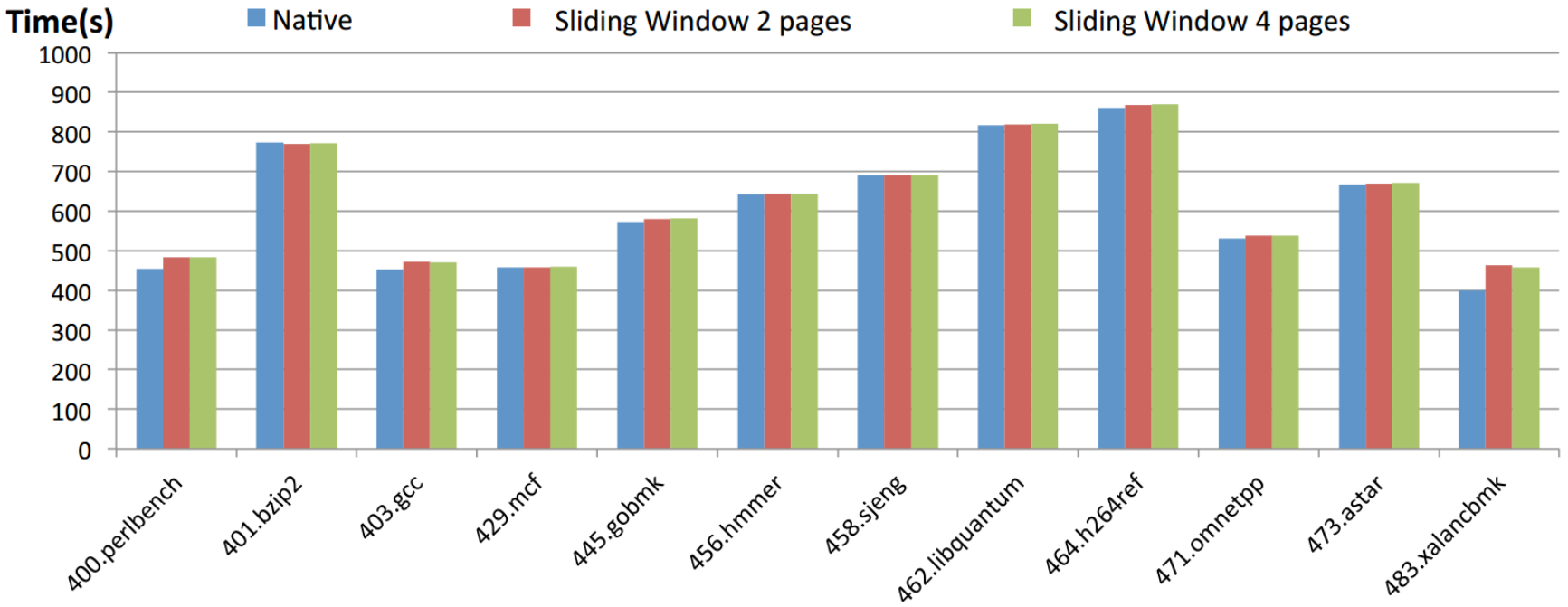
- 完整性检查如何被触发？
  - 以内存页为单位**设置滑动窗口**，所有窗口外内存均设为**不可执行**
  - 控制转移指向窗口外时，将触发异常并陷入内核，从而引起完整性检查




  - 额外地，敏感系统调用同样将引起完整性检查（系统调用必然陷入内核）

# 利用硬件特性和虚拟化技术的CFI

- 性能




- 参考文献： Cheng Y, Zhou Z, Miao Y, et al. Robert. ROPecker: A Generic and Practical Approach For Defending Against ROP Attack.(2014)[C]//Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14), February 23-26, 2014, San Diago, CA.

# What’s next?

- CFI的致命弱点
- 返回导向编程的新变种：一切为了绕过CFI