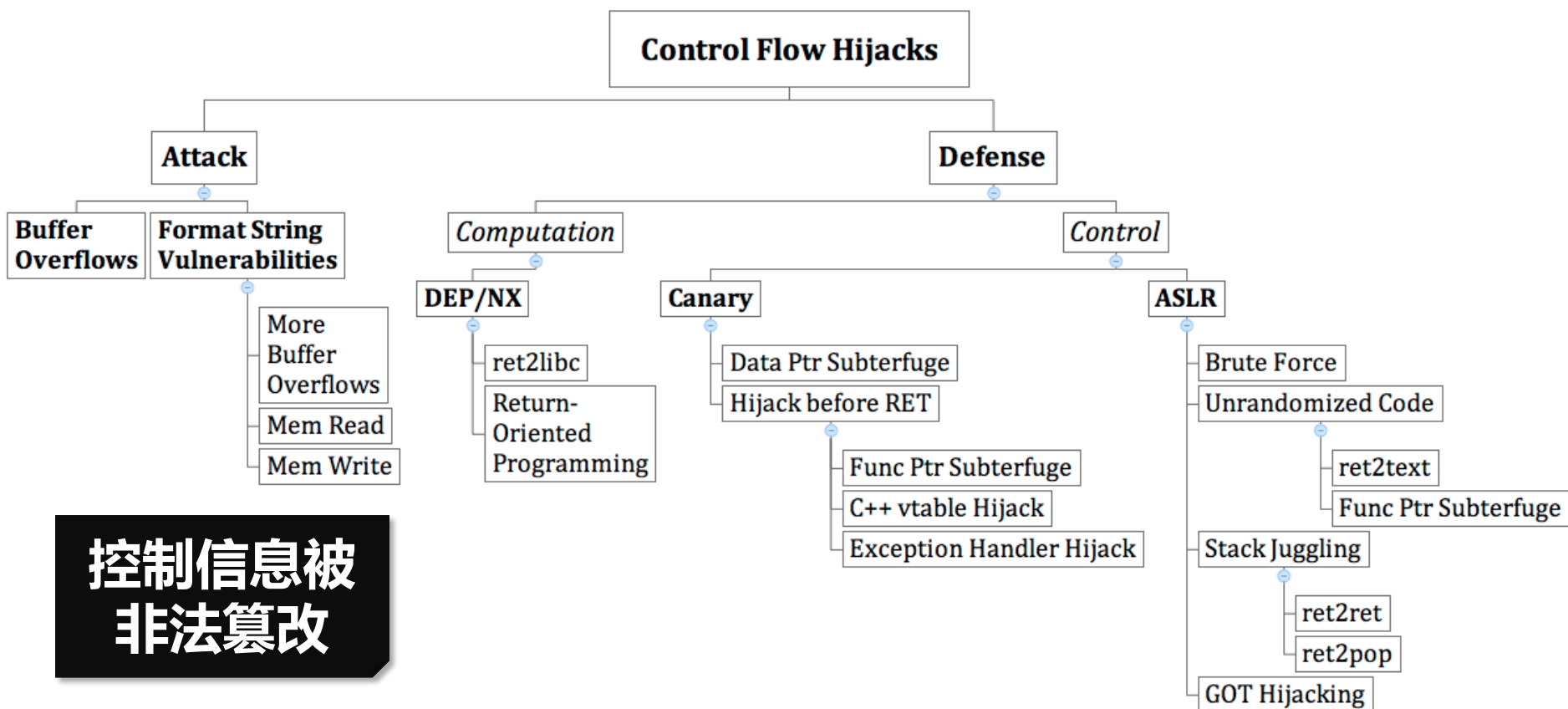


第2讲

软件安全 (3)

控制流完整性

控制流劫持



控制信息被
非法篡改

Behavior-based Detection

- Stack canaries, non-executable data, ASLR的目标在于使得攻击步骤复杂化
 - 但是这些方法仍然可能失败
- 基于行为的检测
 - 观测程序的行为，即其是否按照我们的期望运行
 - If not, 程序可能被compromised

Control-flow Integrity (CFI)

■ Challenges

- Define “Expected behavior”
- Detect deviations from expectation efficiently
- Avoid compromise of the detector

■ Solutions

- Control flow graph (CFG)
- In-line reference monitor (IRM)
- Sufficient randomness, immutability

CFI的主要思想

在程序执行期间，每当一条机器指令**转移控制时**，通过预先构建的**控制流图**(CFG, Control Flow Graph)，来确定转移目标的有效性。

基本块(Basic Block)

基本块：一个连续的指令/代码序列

- 任何位置的指令比其后面的所有指令先执行
- 没有任何外部指令可以在序列中的两个指令之间执行

```
1. x = y + z  
2. z = t + i
```

```
3. x = y + z  
4. z = t + i  
5. jmp 1
```

```
6. jmp 3
```

```
1. x = y + z  
2. z = t + i  
3. x = y + z  
4. z = t + i  
5. jmp 1
```

基本块(Basic Block)

只有基本块**开始的位置**有跳转目标
只有基本块**结束的位置**有跳转指令

1. $x = y + z$
2. $z = t + i$

3. $x = y + z$
4. $z = t + i$
5. `jmp 1`

6. `jmp 3`

1. $x = y + z$
2. $z = t + i$
3. $x = y + z$
4. $z = t + i$
5. `jmp 1`

CFG 的定义

静态控制流图:

- 每一个顶点 v_i 是一个基本块
- 边 (v_i, v_j) 表示从基本块 v_i 到基本块 v_j 的控制转移

**函数/过程内部的基本块构成intra-procedural
范围的CFG**

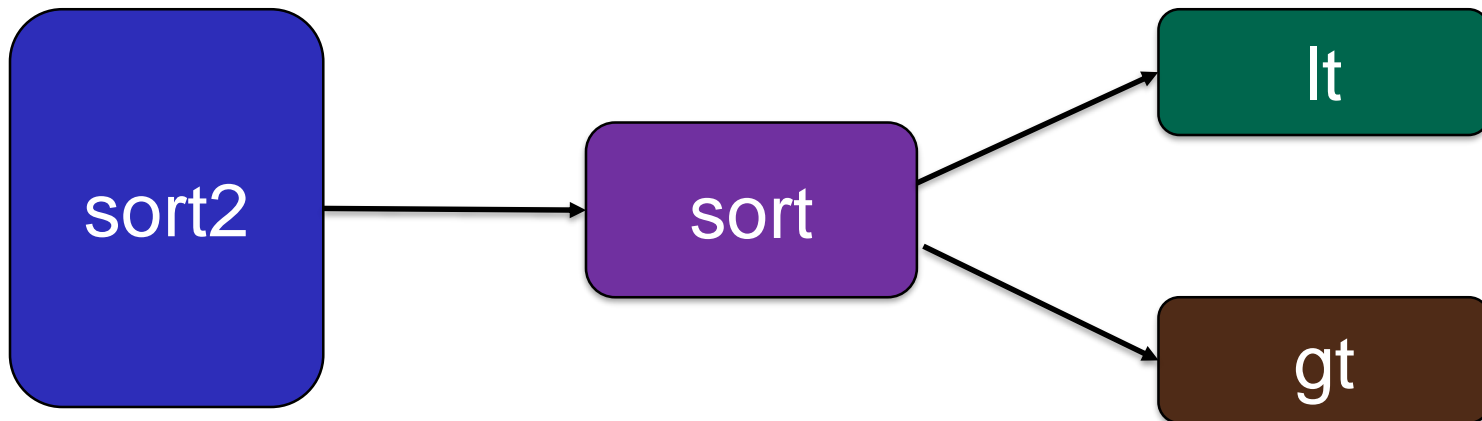
Call Graph

每一个函数是一个节点，边 (v_i, v_j) 表示函数 v_i 调用函数 v_j

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

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

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

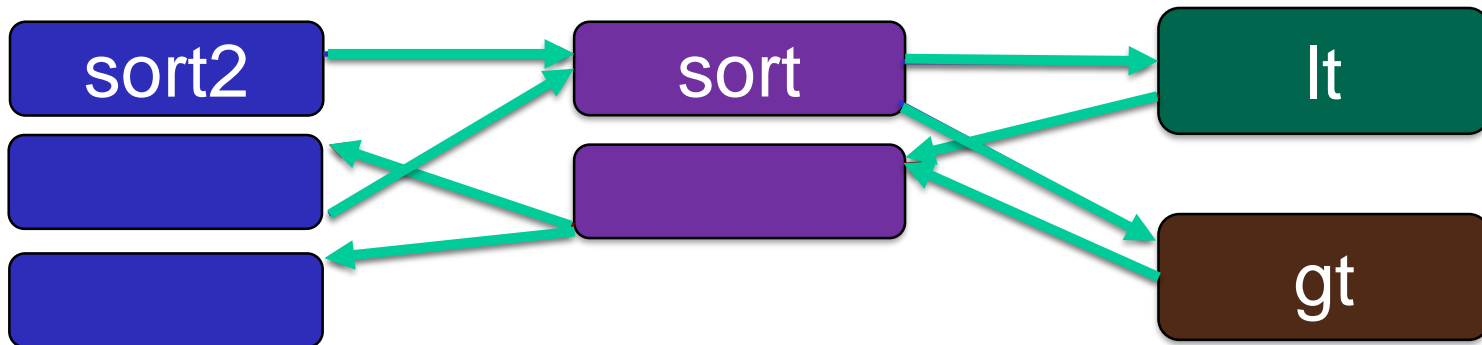


Control Flow Graph

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

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

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



Break into **basic blocks**
Distinguish *calls* from *returns*

CFI: Compliance with CFG

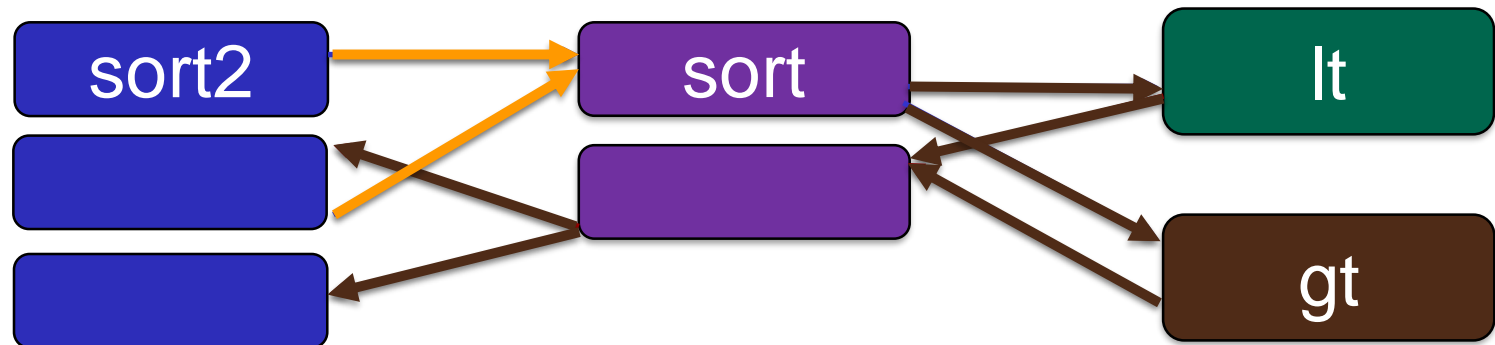
- **Compute the call/return CFG** in advance
 - During compilation, or from the binary
- **Monitor the control flow** of the program, and ensure that it only follows paths allowed by the CFG
- 直接调用(Direct Calls)、间接跳转(Indirect Calls)
 - 只有间接调用需要被监控
 - jmp, call, ret (via **registers**)
 - 直接调用不需要被监控
 - Code is immutable
 - Target address cannot be changed

Control Flow Graph

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

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

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



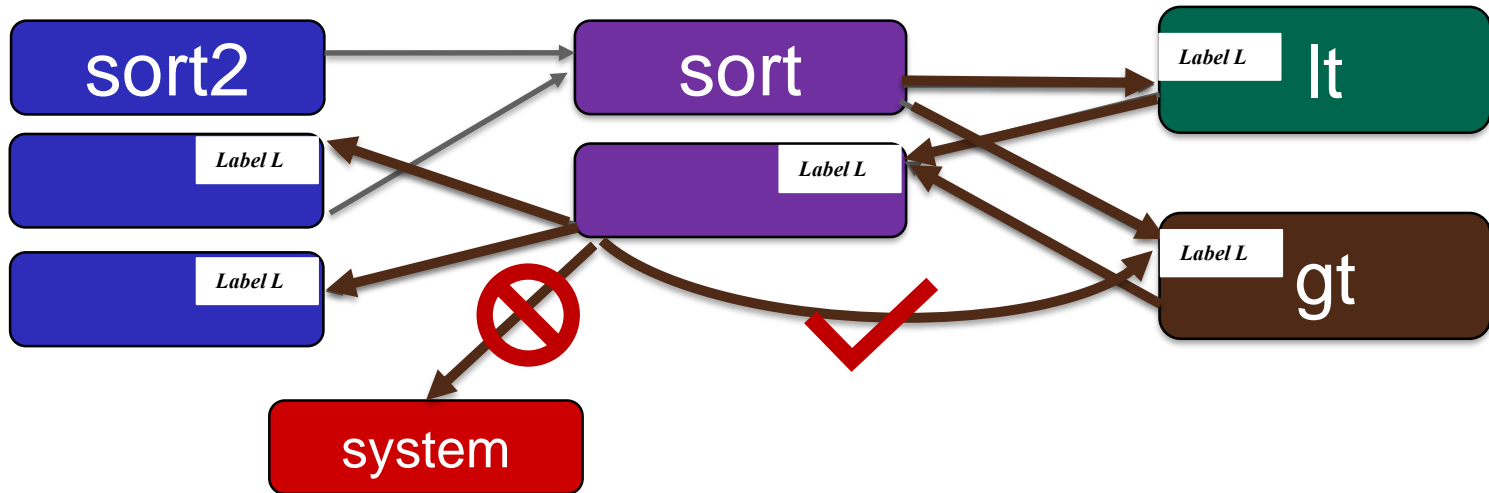
Direct calls

Indirect transfers (call via *register*, or ret)

In-line Monitor

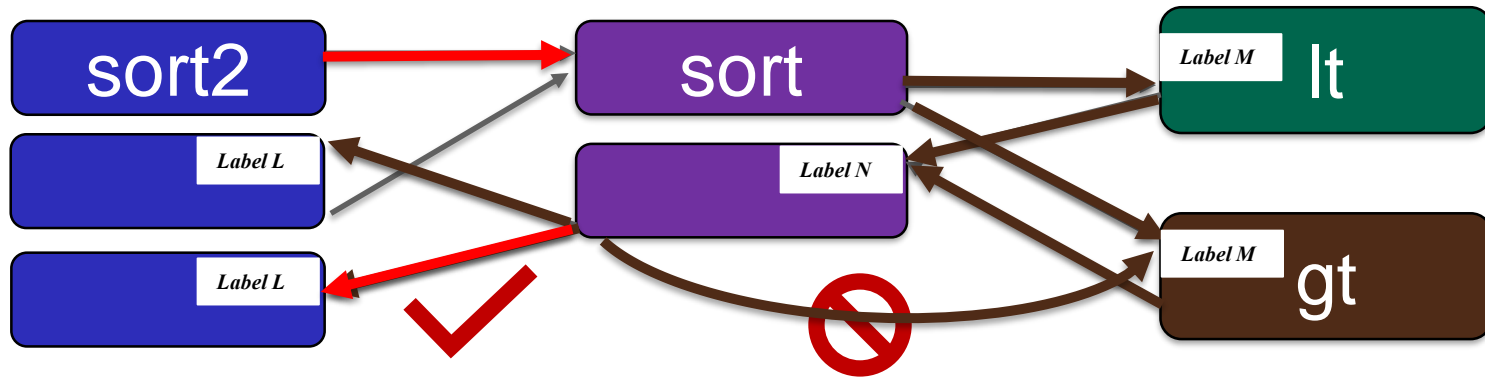
- Implement the monitor in-line, as a **program transformation**
- Insert a **label** just before the target address of a indirect transfer
- Insert **code** to check the label of the target at each indirect transfer
- The **labels** are determined by the CFG

Simplest labeling



使用相同的label

Detailed labeling



使用不同的label

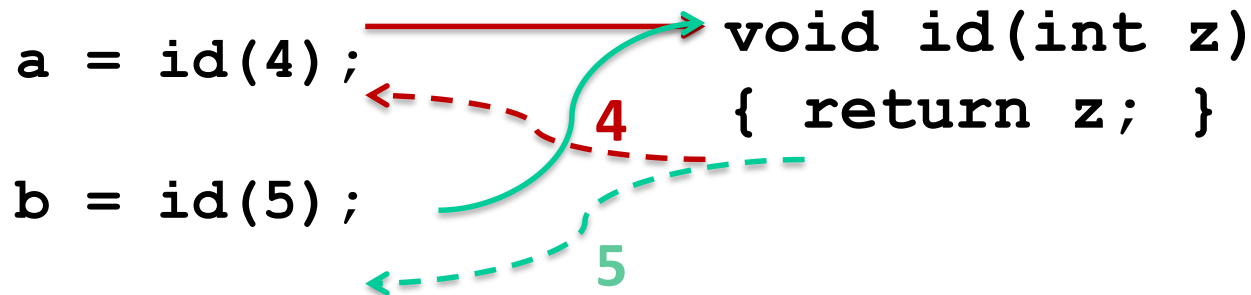
问题?

Context-insensitive vs. Context-sensitive

基于不同的调用上下文返回不同的地址

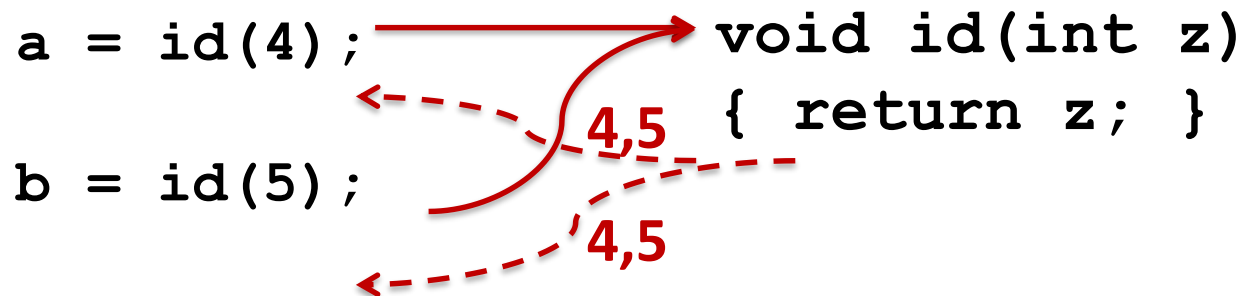
上下文敏感示例

```
a = id(4);  
b = id(5);  
  
void id(int z)  
{ return z; }
```



Context-Sensitive

```
a = id(4);  
b = id(5);  
  
void id(int z)  
{ return z; }
```



Context-Insensitive

CFI-步骤

目标：程序执行必须按照控制流图（CFG）的路径执行。

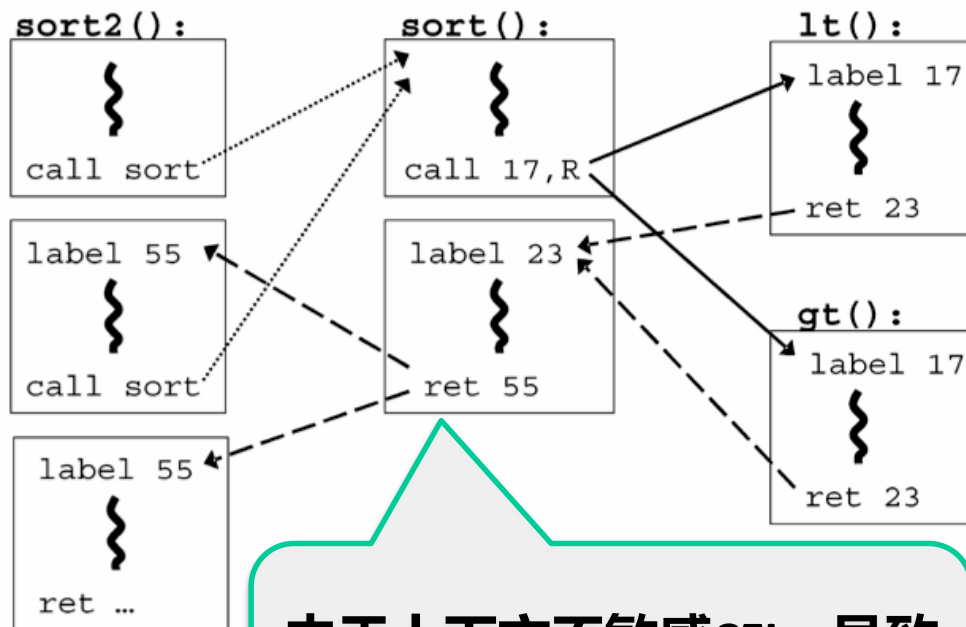
步骤：

- **静态构建CFG，比如在编译期**
- **二进制插桩, 比如在软件安装时**
 - 添加ID和ID checks，维护ID的唯一性
- **在装载时校验CFI的插桩**
 - 直接跳转目标，确保ID和ID checks存在，ID唯一性
- **程序运行时对ID进行检查**
 - 间接跳转具有匹配的ID

构建 CFG

.....> 直接调用
————> 间接调用

```
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 );  
}
```

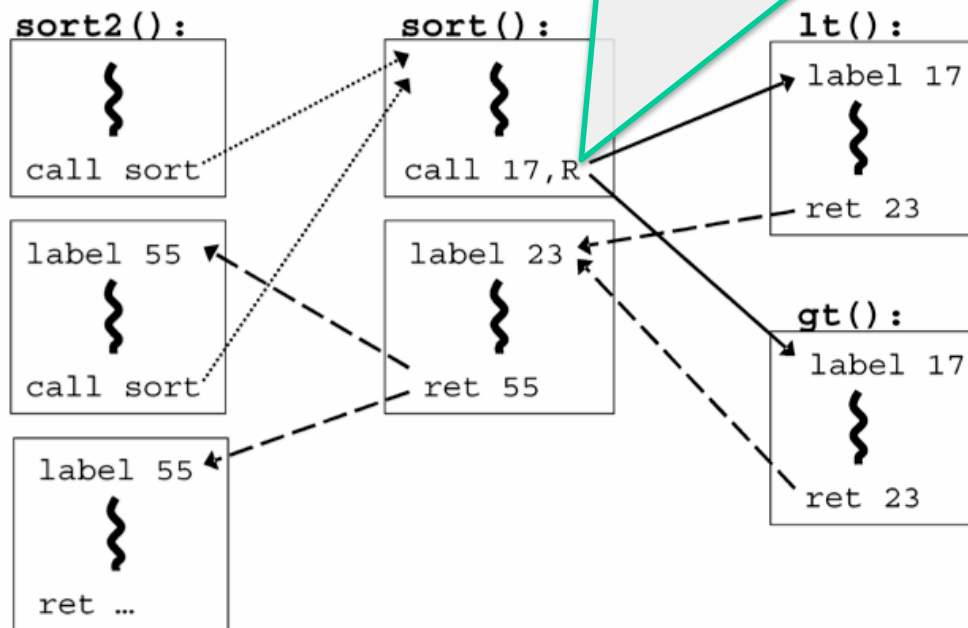


由于上下文不敏感CFI，导致
sort调用有两个返回目标

二进制插桩

call 17, R: 仅当R的标签为17时, 控制转移至R

```
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 );  
}
```



- 在每个目的位置插入一个唯一的ID

插桩示例

原始汇编代码

Opcode bytes	Source Instructions
FF E1	jmp ecx ; computed jump

Opcode bytes	Destination Instructions
8B 44 24 04	mov eax, [esp+4] ; dst

插装后汇编代码

B8 77 56 34 12	mov eax, 12345677h	; load ID-1
40	inc eax	; add 1 for ID
39 41 04	cmp [ecx+4], eax	; compare w/dst
75 13	jne error_label	; if != fail
FF E1	jmp ecx	; jump to label

3E 0F 18 05	prefetchnta	; label
78 56 34 12	[12345678h]	; ID
8B 44 24 04	mov eax, [esp+4]	; dst
...		

当目标位置的标签是“12345678”时，
控制跳转到目标位置

在二进制的目标位置插入标签“12345678”

插桩示例

Bytes (opcodes)	x86 assembly code	Comment
FF 53 08	call [ebx+8]	; call a function pointer
is instrumented using prefetchnta destination IDs, to become: 检查目标label		
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

Bytes (opcodes)	x86 assembly code	Comment
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

检查目标label

校验CFI插桩

- **直接跳转目标(e.g., call 0x12345678)**
 - 根据CFG判断所有目标的有效性
- **IDs**
 - 验证每一个进入点(entry point)后是否有一个ID
 - 验证是否有非法ID
- **ID checks**
 - 验证控制转移的位置前是否有ID check
 - 验证check代码是不遵循 CFG?

安全假设

- **UNQ: Unique IDs**
 - 用来维护CFG的语义
- **NWC: Non-Writable Code (Immutable)**
 - 保证攻击者无法修改label
 - 动态加载和动态生成的代码不满足NWC(例如JIT)
- **NXD: Non-Executable Data**
 - 保证攻击者无法插入的具有合法label的code

安全保证

- CFI可以防护基于非法控制流转移的攻击
 - 基于栈的缓冲区溢出攻击， ROP/return-to-libc攻击
- 不能防护**不违反程序的原始CFG**的攻击
 - Single-labeled CFG
 - 基于数据的攻击（data leaks or corruptions）
 - heartbleed
 - 对系统调用参数的篡改
 - 替换文件名
 - 错误的逻辑实现

```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, str);
    if(authenticated) { ...
}
```


CFI实现

- **基于软件 (Software-based)**
- **硬件辅助 (Hardware-assisted)**

Software-based vs Hardware-assisted CFI Enforcement

- **Software-based Control-flow Integrity Enforcement**
 - 方法: implementing CFI enforcement in software only
 - 举例: Microsoft **CFG**, **RFG**; Google **VTV**, **IFCC**
 - 优点: 更快地实现/产品化, 更灵活, 适应各种应用场景
- **Hardware-assisted Control-flow Integrity Enforcement**
 - 方法: enforcing CFI with the support of dedicated hardware (new ISA feature etc.)
 - 举例: Intel **CET**, ARM **PAC**
 - 优点: 更小的性能下降, 更有效地抵御攻击/绕过

控制流劫持

- **Forward-edge** control-flow hijacking
- **Backward-edge** control-flow hijacking

```
typedef int (*func_ptr)(struct foo *);  
  
func_ptr saved_actions[] = {  
    do_simple,  
    do_fancy,  
    ...  
};  
  
int action_launch(int idx)  
{  
    func_ptr action;  
    int rc;  
    ...  
    action = saved_actions[idx];  
    ...  
    rc = action(info);  
    ...  
}
```

forward edge

```
int do_simple(struct foo *info)  
{  
    stuff;  
    and;  
    things;  
    ...  
    return 0;  
}
```

backward edge

<https://blog.csdn.net/pwl999>

CFG (Control Flow Guard)

CFG implements **coarse-grained control-flow integrity** for indirect calls

Compile time

```
void Foo(...) {  
    // SomeFunc is address-taken  
    // and may be called indirectly  
    Object->FuncPtr = SomeFunc;  
}
```

Metadata is automatically added to the image which identifies functions that may be called indirectly

```
void Bar(...) {  
    // Compiler-inserted check to  
    // verify call target is valid  
    _guard_check_icall(Object->FuncPtr);  
    Object->FuncPtr(xyz);  
}
```

A lightweight check is inserted prior to indirect calls which will verify that the call target is valid at runtime

Runtime

Process Start

- Map valid call target data

Image Load

- Update valid call target data with metadata from PE image

Indirect Call

- Perform O(1) validity check
- Terminate process if invalid target
- Jump if target is valid

CFG is a deterministic mitigation, its security is not dependent on keeping secrets.

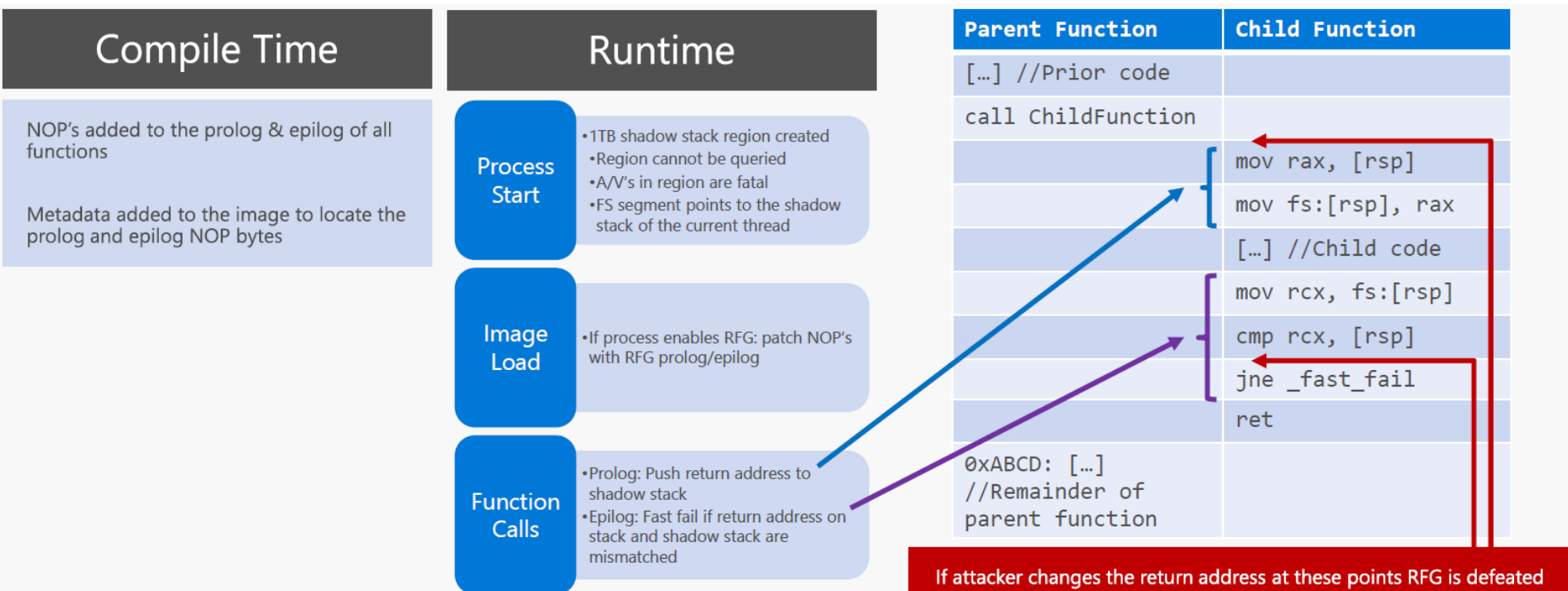
For C/C++ code, CFG requires no source code changes.

```
ntdll!LdrpDispatchUserCallTarget:  
00007ffb`4e100e10 4c8b1d59e50d00 mov     r11,qword ptr  
[ntdll!LdrSystemDllInitBlock+0xb0]  
00007ffb`4e100e17 4c8bd0      mov     r10,rax  
00007ffb`4e100e1a 49c1ea09    shr     r10,9  
00007ffb`4e100e1e 4f8b1cd3    mov     r11,qword ptr [r11+r10*8]  
00007ffb`4e100e22 4c8bd0      mov     r10,rax  
00007ffb`4e100e25 49c1ea03    shr     r10,3  
00007ffb`4e100e29 a80f      test    al,0Fh  
00007ffb`4e100e2b 7509      jne     ntdll!LdrpDispatchUserCallTarget+0x26  
ntdll!LdrpDispatchUserCallTarget+0x1d:  
00007ffb`4e100e2d 4d0fa3d3    bt      r11,r10  
00007ffb`4e100e31 7303      jae     ntdll!LdrpDispatchUserCallTarget+0x26  
ntdll!LdrpDispatchUserCallTarget+0x23:  
00007ffb`4e100e33 48ffe0      jmp     rax
```

CFG不依赖于secret

RFG (Return Flow Guard)

RFG: MS的Software Shadow Stack



RFG依赖于secret, 即shadow stack 的虚拟地址

RFG (Return Flow Guard)

- **问题:**

- 依赖于secret, shadow stack的映射可以被成功的泄露
 - AnC Attack: <https://www.vusec.net/projects/anc/>
- Race Condition

"When we have to do it in **software**, we have to introduce 'no ops'; when you're entering and exiting the function, we pad them with blanks and so people are able to massage the memory, people are able to massage the **race condition**s of the system and skip the checks completely," Hari Pulapaka, principal group program manager of the Windows kernel team, explained. There's no **race condition** when the shadow stack is stored in **hardware**, so the checks don't get skipped.

<https://www.techrepublic.com/article/windows-10-security-how-the-shadow-stack-will-help-to-keep-the-hackers-at-bay/>

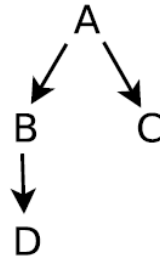
Google's CFI

- Forward-Edge CFI for Virtual Calls (**VTV**, Virtual-Table Verification)
- Forward-Edge CFI for Indirect Function Calls (**IFCC**, Indirect Function-Call Checks)

VTV

```
class A {  
public:  
    int mA;  
    virtual void foo();  
}  
class B : public A {  
public:  
    int mB;  
    virtual void foo();  
    virtual void bar();  
}  
class C : public A {  
public:  
    int mC;  
    virtual void baz();  
}  
class D : public B {  
public:  
    int mD;  
    virtual void foo();  
    virtual void boo();  
}
```

(a) C++ Code



(b) Class Hierarchy

```
C1: A* a = ...  
    a->foo();  
...  
C2: B* b = ...  
    b->bar();
```

(c) Sample Callsites

```
C1: $a = ...  
    (1) $avptr = load $a  
    (2) $foo_fn = load $avptr  
    (3) call $foo_fn  
  
C2: $b = ...  
    (1) $bvptra = load $b  
    (2) $bar_fn = load ($bvptra+0x8)  
    (3) call $bar_fn
```

(d) Callsite Instructions

Fig. 1: C++ Example

VTV

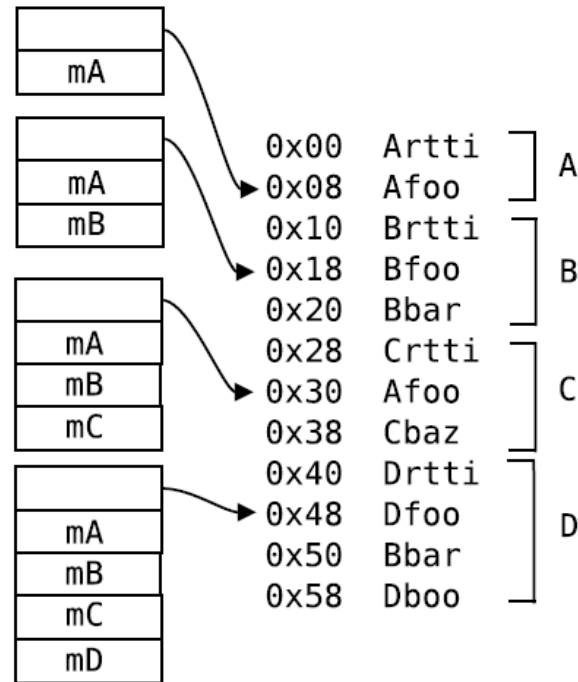


Fig. 2: Normal Vtable Layout in Memory

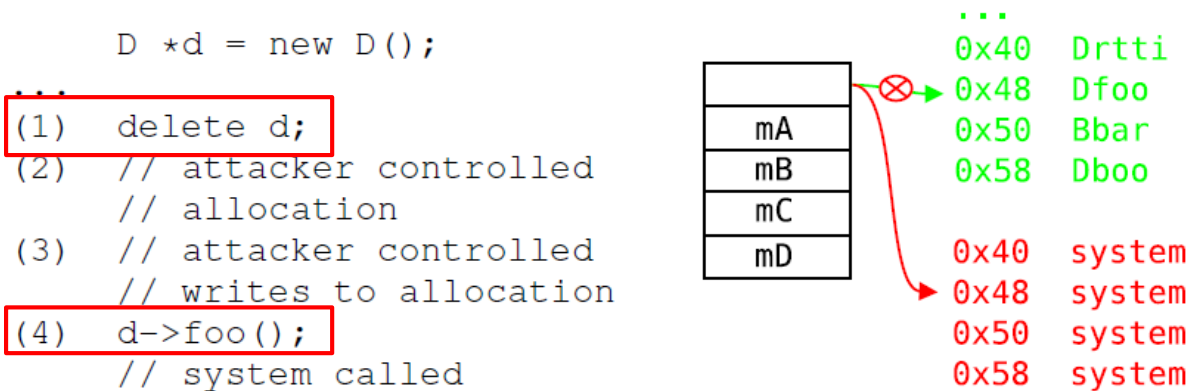


Fig. 3: VTable hijacking example

VTV

<pre>C1: \$a = ... \$avptr = load \$a assert invalid \$avptr, A \$foo_fn = load \$avptr call \$foo_fn</pre>	<pre>C1: \$a = ... \$avptr = load \$a assert \$avptr ∈ {0x8, 0x18, 0x30, 0x48} \$foo_fn = load \$avptr call \$foo_fn</pre>
<pre>C2: \$b = ... \$bvptr = load \$b assert invalid \$bvptr, B \$bar_fn = load (\$bvptr+0x8) call \$bar_fn</pre>	<pre>C2: \$b = ... \$bvptr = load \$b assert \$bvptr ∈ {0x18, 0x48} \$bar_fn = load (\$bvptr+0x8) call \$bar_fn</pre>
(a) Abstract Check	(b) Vptr check semantics

Fig. 4: Instrumented Callsites


IFCC

- 通过为间接调用目标生成跳转表 (jump table) ;
- 并在间接调用点添加代码来转换函数指针, 以保护间接调用, 从而确保它们指向跳转表条目;
- 任何未指向相应表的函数指针都被视为CFI违规;

IFCC

- Addresses of all functions, which have their addresses taken, are replaced with the **addresses of jump table entries**.
 - Jump table entry is created with **jump** to the original function address.
- Indirect calls are replaced with **instructions** that **transform pointers** to force function calls into right table.
- Pointer transformation uses **jump table base address** and a **special mask** to force every call into the right table.
- Mask depends on sizes of jump table and its entries: for example, if the table size is 32 entries, and each entry is 4 bytes, the mask will be 1111100.
- Simple transformation: $\text{Pointer} = ((\text{Pointer} - \text{BaseAddress}) \& \text{Mask}) + \text{BaseAddress}$;
- If the pointer already is in the right table, it is unchanged. If not, it is **forced into the table**.

IFCC

Function	Address		Function	Address
Func_a	1101110		Func_a	1001111
Func_b	1110010		Func_b	1010011

FuncPtr fp;

...

fp = 1101110;

...

call fp;

...

// fp is hacked

fp = 1111110;

...

call fp;

FuncPtr fp;

...

//address replaced

fp = 1001111;

...

// transformation

fp = fp - 1000011; // 0001100

fp = fp & 1111100; // 0001100

fp = fp + 1000011; // 1001111

call fp;

fp = fp - 1000011; // 0111011

fp = fp & 1111100; // 0111000

fp = fp + 1000011; // 1111011

call fp; // forced to jump table



Entry Address

1000011 + 00000

1000011 + 00100

1000011 + 01000

1000011 + 01100

1000011 + 10000

1000011 + 10100

1000011 + 11000

1000011 + 11100

...

Jump Table

Base Address = 1000011

Entry size = 4 bytes

Mask = 1111100

jmp to crash

jmp to crash

jmp to crash

jmp to 1101110 (Func_a)

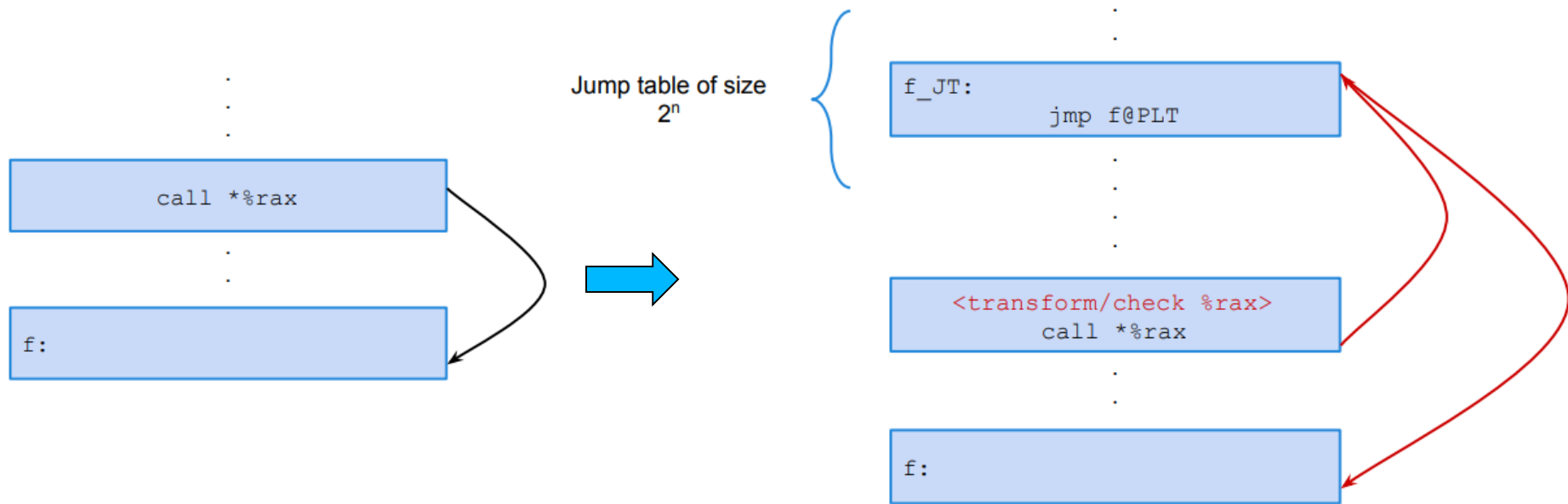
jmp to 1110010 (Func_b)

jmp to crash

jmp to crash

jmp to crash

IFCC



IFCC

```
<do_simple>:
201870: xor    %eax,%eax
201872: retq
...
<do_fancy>:
201880: mov    0x4(%rdi),%eax
201883: add    (%rdi),%eax
201885: retq
```

```
<action_launch>:
201890: push   %rbx
201891: movslq %edi,%rax
201894: mov    0x200550(,%rax,8),%rax

20189c: mov    $0x203b44,%edi
2018a1: callq  *%rax
...
```

<https://blog.csdn.net/pw1999>

①根据函数原型一致创建了间接调用表，能间接跳转到do_simple()和do_fancy()

```
<__typeid__ZTSFiP3fooE_global_addr>:
201860: jmpq   201870 <do_simple>
201865: int3
201866: int3
201867: int3
201868: jmpq   201880 <do_fancy>
20186d: int3
20186e: int3
20186f: int3
```

```
<do_simple>:
201870: xor    %eax,%eax
201872: retq
...
<do_fancy>:
201880: mov    0x4(%rdi),%eax
201883: add    (%rdi),%eax
201885: retq
```

```
<action_launch>:
201890: push   %rbx
201891: movslq %edi,%rax
201894: mov    0x200550(,%rax,8),%rax
20189c: mov    $0x201860,%ecx
2018a1: mov    %rax,%rdx
2018a4: sub    %rcx,%rdx
2018a7: ror    $0x3,%rdx
2018ab: cmp    $0x1,%rdx
2018b2: ja     2018dc <action_launch+0x4c>
2018b4: mov    $0x203b44,%edi
2018b9: callq  *%rax
...
2018dc: ud2
```

②rax中获取到的函数指针不再是do_simple()/do_fancy()的地址，而是xxx_global_addr间接调用表中的地址

③在间接调用之前，判断rax中的地址是否合法，方法是：判断rax中的目的地址，是否超过xxx_global_addr间接调用表的最大范围

<https://blog.csdn.net/pw1999>

IFCC

■ 举例

```
struct foo {
    int_arg_fn int_funcs[1];
    int_arg_fn bad_int_funcs[1];
    float_arg_fn float_funcs[1];
    int_arg_fn not_entries[1];
};
static struct foo f = {
    .int_funcs = {int_arg},
    .bad_int_funcs = {bad_int_arg},
    .float_funcs = {float_arg},
    .not_entries = {(int_arg_fn)((uintptr_t)(not_entry_point)+0x20)}
};

int idx = argv[1][0] - '0';
f.int_funcs[idx](idx);
```

```
clang -fvisibility=hidden -flto -fno-sanitize-trap=all -fsanitize=cfi -o cfi_icall cfi_icall.c
```

https://github.com/trailofbits/clang-cfi-showcase/blob/master/cfi_icall.c

<https://github.com/trailofbits/clang-cfi-showcase>

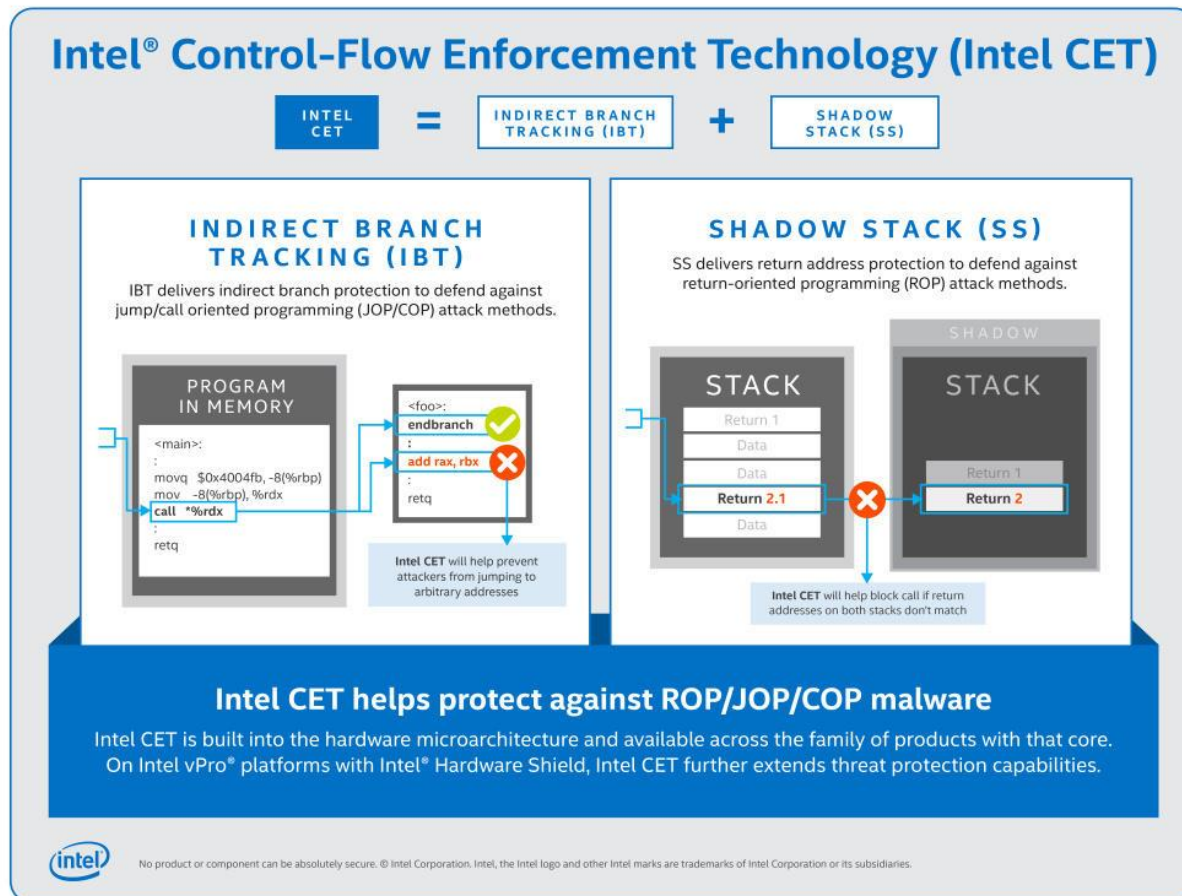
Intel CET

- **两个部分：**
 - Shadow Stack (SHSTK)
 - Indirect Branch Tracking (IBT)

Intel CET (Control-flow Enforcement Technology)

Control-flow Enforcement Technology (CET) provides the following capabilities to defend against ROP/JOP style control-flow subversion attacks:

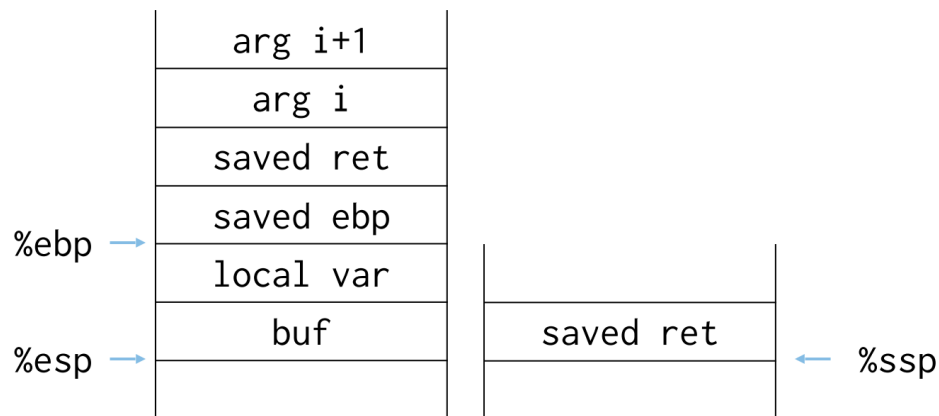
- Shadow Stack – return address protection to defend against Return Oriented Programming,
- Indirect branch tracking – free branch protection to defend against Jump/Call Oriented Programming.



Shadow Stack

■ 解决性能和安全问题

- ssp (shadow stack pointer)
- call 和 ret 自动更新 esp和ssp
- 无法手动更新shadow stack



Indirect Branch Tracking (IBT)

```
main() {  
    int (*f)();  
    f = test;  
    f();  
}
```

```
int test() {  
    return  
}
```

<main>:

ENDBR

:

movq \$0x4004fb, -8(%rbp)

mov -8(%rbp), %rdx

call *%rdx

:

retq

<test>:

ENDBR

:

add rax, rbx

:

retq

所有的Indirect Branch
必须以ENDBR开始

CET Instructions

- RDSSP – Read shadow stack pointer
- INCSSP – Shadow stack unwinding
- RSTORSSP, SAVEPREVSSP – Shadow stack context switching
- SETSSBSY, CLRSSBSY – Mark shadow stack in-use
- ENDBR, No-Track – Indirect branch tracking

Intel CET

- CET 支持

- Glibc: <https://man7.org/linux/man-pages/man1/gcc.1.html>
- Microsoft: Hardware-enforced Stack Protection, <https://www.zdnet.com/article/microsoft-announces-new-hardware-enforced-stack-protection-feature/>

Microsoft: Hardware-enforced Stack Protection

Our strategy for mitigating arbitrary code execution

Software vulnerabilities are typically exploited by hijacking control flow, injecting new code, and jumping to it

Prevent
arbitrary code
generation

Code Integrity Guard

Images must be signed and load
from valid places

Arbitrary Code Guard

Prevent dynamic code generation,
modification, and execution

Prevent
control-flow
hijacking

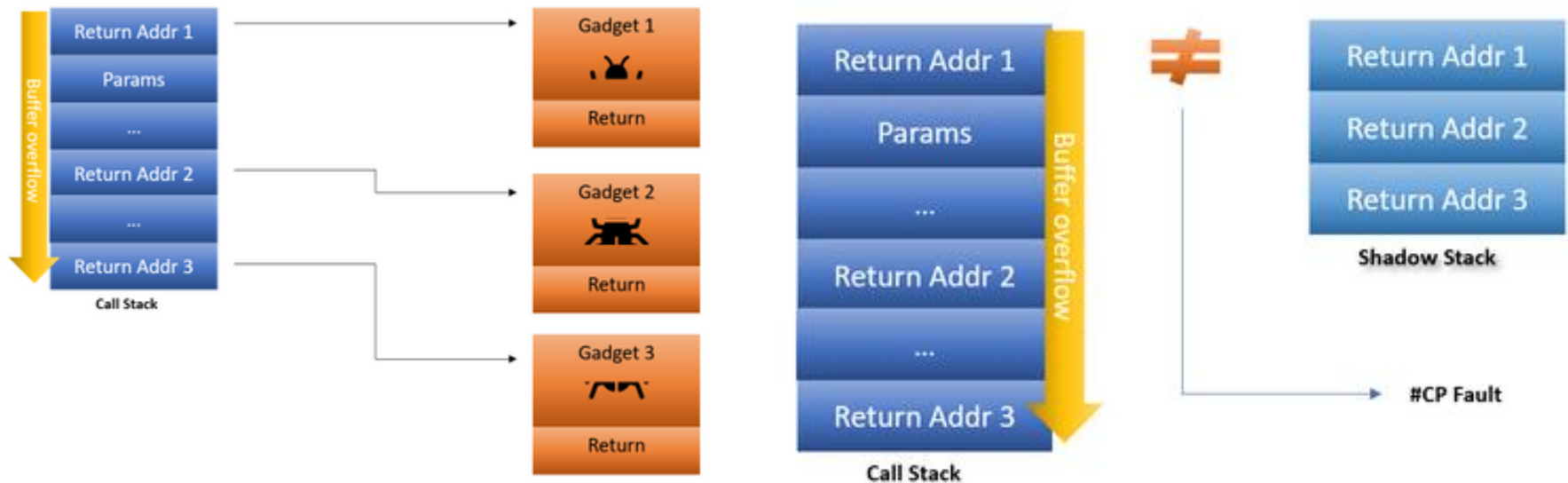
Control Flow Guard

Enforce control flow integrity
on indirect calls

Shadow Stack

Use a separate stack for return
addresses

Microsoft: Hardware-enforced Stack Protection



ROP问题

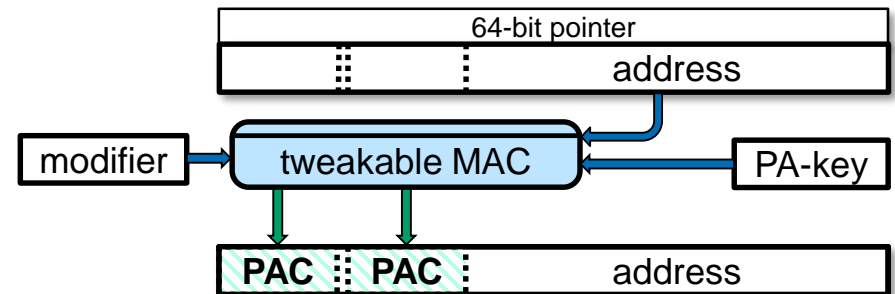
Shadow Stack

每条CALL指令上，返回地址都被压入栈和影子栈上；而在RET指令上，进行比较以确保完整性不受影响。

ARM 8.3-A Pointer Authentication

Pointer Authentication Codes (PAC)

- Set in unused bits of virtual address



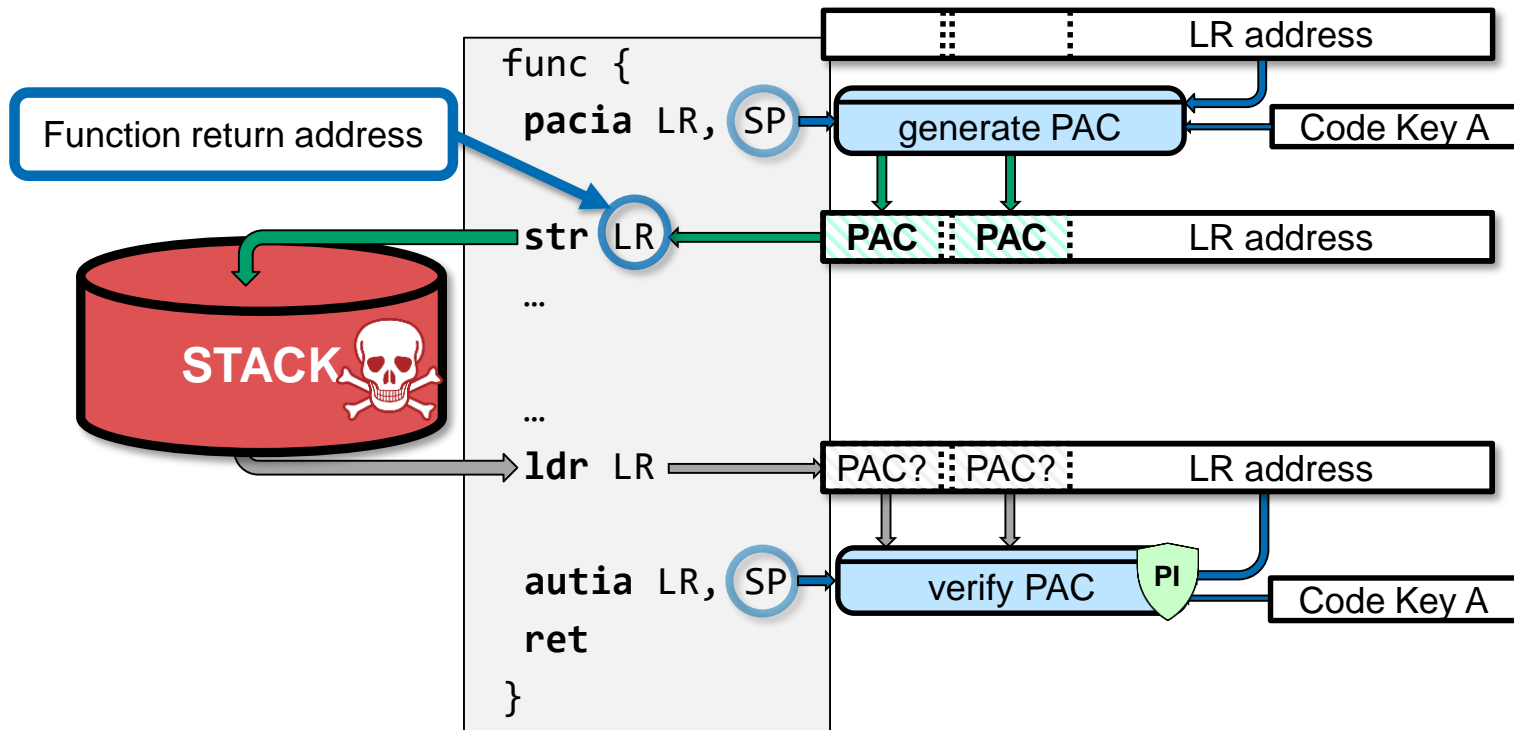
Key/configuration set at higher privilege level

Instrument with new PAC handling instructions

- Opcode determines used key
- Operands set **PA modifier** (tweak value)

instructions	Code-key		Data-key		Gen.-key
	A	B	A	B	
pacia	X				
pacib		X			
pacda			X		
pacdb				X	
pacga					X
autia	X				
autib		X			
autda			X		
autdb				X	

Example: PA-based return address signing



Qualcomm "[Pointer Authentication on ARMv8.3](#)", whitepaper 2017