Control Flow Integrity

Behavior-based detection

- Stack canaries, non-executable data, and ASLR aim to complicate various steps in a standard attack
  - But they still may not stop it
- Idea: **observe** the program's **behavior — is it doing what we expect it to?**
  - If not, might be compromised
- Challenges
  - Define "expected behavior"
  - Detect deviations from expectation efficiently
  - Avoid compromise of the detector

Control-flow Integrity (CFI)

- *Define "expected behavior":*

  Control flow graph (CFG)

- *Detect deviations from expectation efficiently*

  In-line reference monitor (IRM)

- *Avoid compromise of the detector*

  Sufficient randomness, immutability

Efficient?

- **Classic CFI** (2005) imposes **16% overhead** on average, **45%** in the **worst case**
  - Works on arbitrary executables
  - Not modular (no dynamically linked libraries)
- **Modular CFI** (2014) imposes **5% overhead** on average, **12%** in the **worst case**
  - C only (part of LLVM)
  - Modular, with separate compilation
  - http://www.cse.lehigh.edu/~gtan/projects/upro/

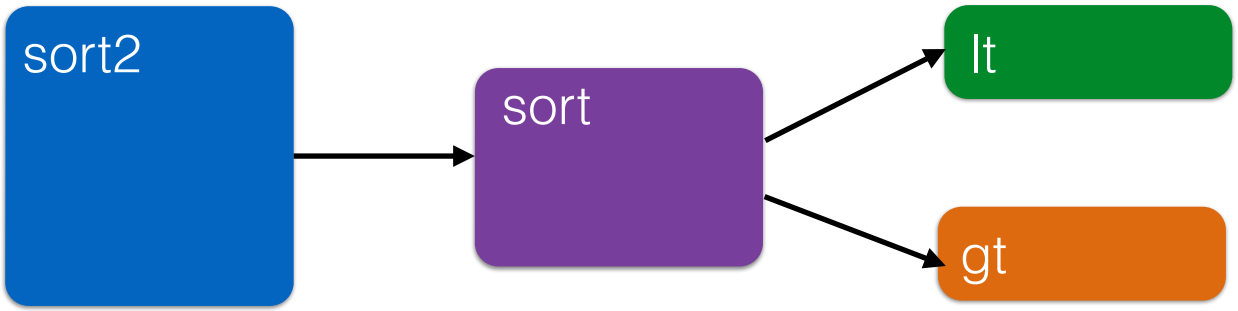
Secure?

- MCFI can **eliminate 95.75% of ROP gadgets** on x86-64 versions of SPEC2006 benchmark suite
  - By ruling their use non-compliant with the CFG
- Average Indirect-target Reduction (AIR) **> 99%**
  - AIR is, in essence, the percentage of **possible targets of indirect jumps that CFI rules out**
    - **For CFI: nearly all of them**

Call 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;
}
```

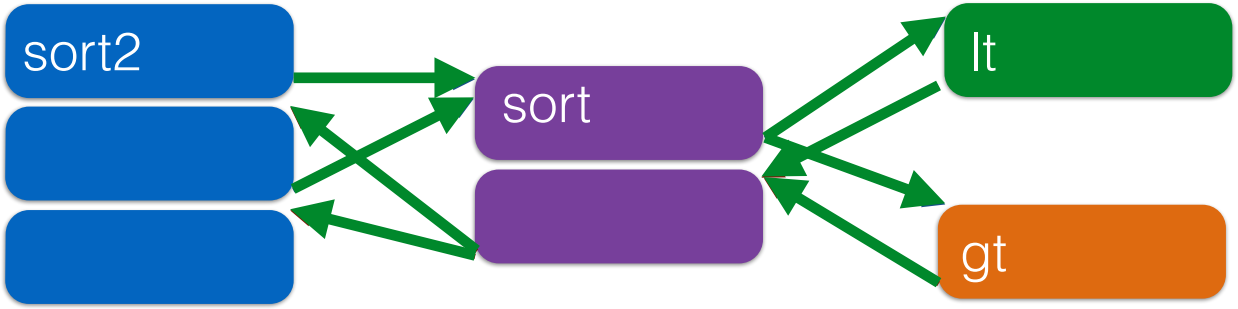


Which functions call other functions

Control Flow Graph

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

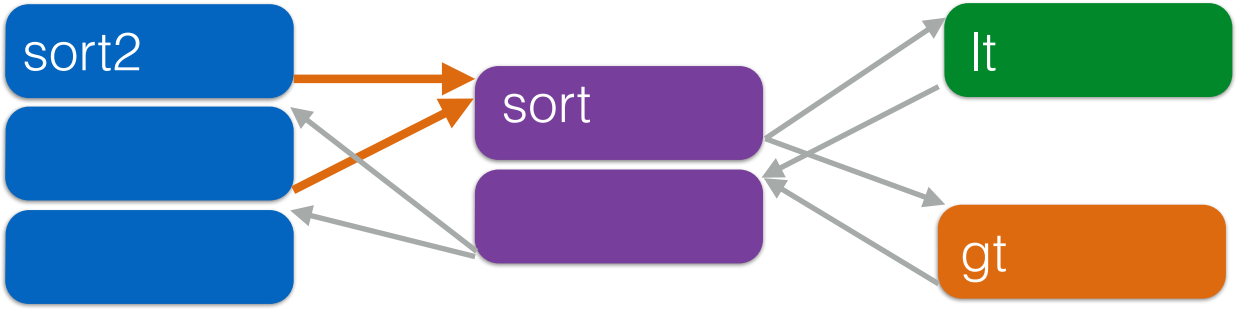
CFI: Compliance with CFG

- **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
- Observation: **Direct calls** need **not** be **monitored**
  - Assuming the code is immutable, the target address cannot be changed
- Therefore: **monitor only indirect calls**
  - `jmp`, `call`, `ret` via registers

Control Flow Graph

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

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

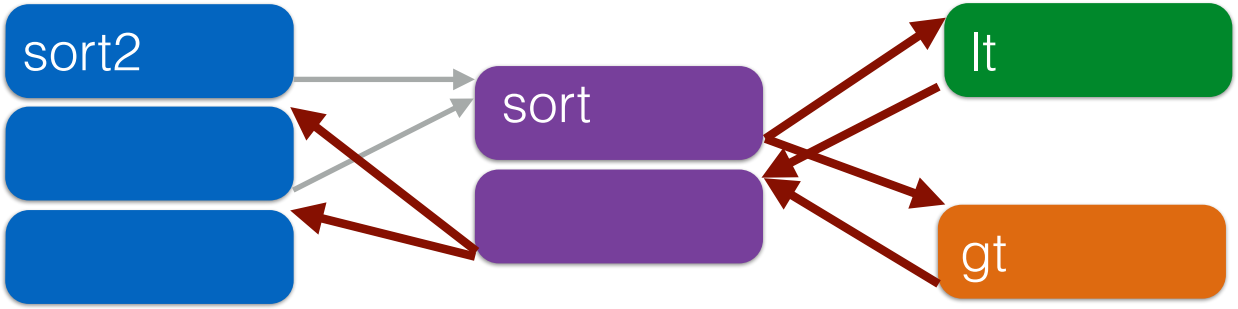


Direct calls *(always the same target)*

Control Flow Graph

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

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

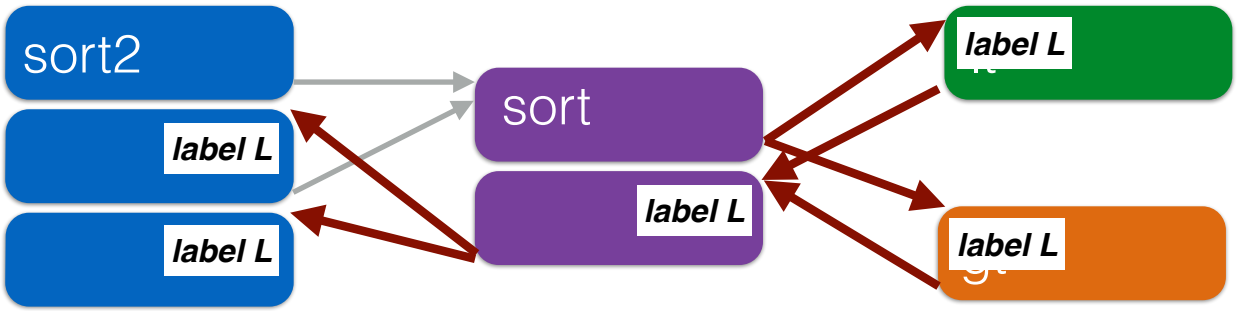


Indirect transfer (`call` *via register, or* `ret`)

In-line Monitor

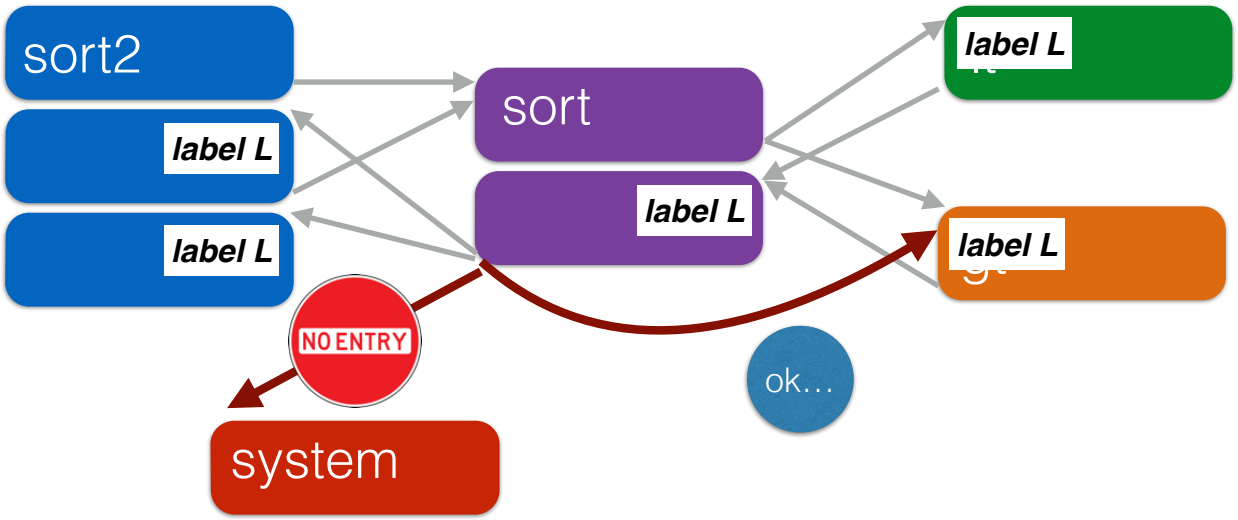
- Implement the monitor in-line, as a **program transformation**
- Insert a **label just before the target address** of an indirect transfer
- Insert **code to check the label of the target** at each indirect transfer
  - Abort if the label does not match
- The **labels are determined by the CFG**

Simplest labeling



Use the same label at all targets

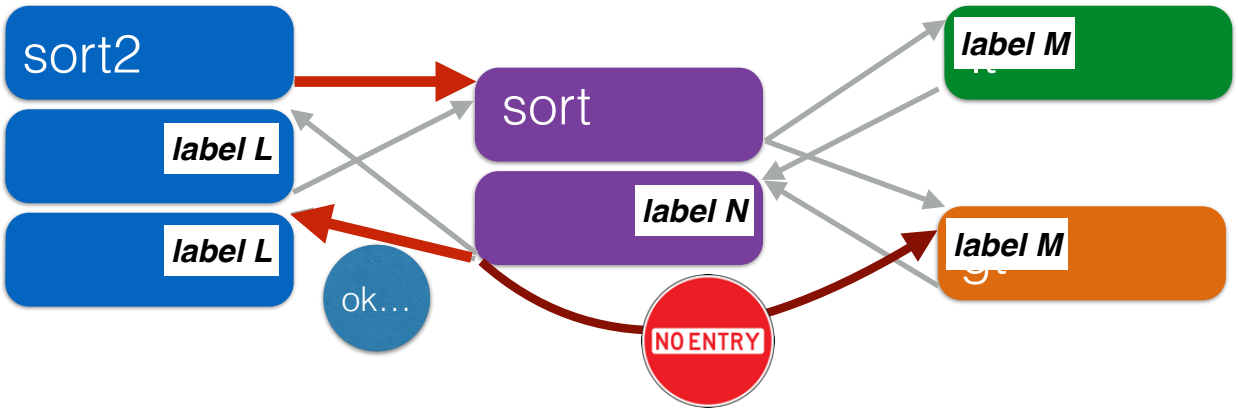
Simplest labeling



Use the same label at all targets

Blocks return to the start of direct-only call targets but not incorrect ones

Detailed labeling



Constraints:

- return sites from calls to `sort` must share a label (*L*)
- call targets `gt` and `lt` must share a label (*M*)
- remaining label unconstrained (*N*)

Still permits call from site A to return to site B

Classic CFI instrumentation

```
FF 53 08                  call  [ebx+8]            ; call a function pointer
```

is instrumented using `prefetchnta` destination ID

```
8B 43 08                  mov   eax, [ebx+8]       ; load
3E 81 78 04 78 56 34 12   cmp   [eax+4], 12345678h ;                ion
75 13                     jne   error_label        ; if nc
FF D0                     call  eax                ; call function pointer
3E 0F 18 05 DD CC BB AA   prefetchnta [AABBCCDDh]  ; label ID, used upon the return
```

Check target label

Fig. 4. Our CFI implementation of a call through a function pointer.

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 a
83 C4 14                  add   esp, 14h           ; pop 20
3E 81 79 04 DD CC BB AA   cmp   [ecx+4], AABBCCDDh ;                on
75 13                     jne   error_label        ; if not
FF E1                     jmp   ecx                ; jump to return address
```

Check target label

Can we defeat CFI?

- **Inject code** that has a **legal label**
  - *Won't work* because we assume **non-executable data**
- **Modify code labels** to allow the desired control flow
  - *Won't work* because the **code is immutable**
- **Modify stack during a check**, to make it seem to succeed
  - *Won't work* because **adversary cannot change registers** into which we load relevant data
    - No time-of-check, time-of-use bug (TOCTOU)

CFI Assurances

- **CFI defeats control flow-modifying attacks**
  - Remote code injection, ROP/return-to-libc, etc.
- But **not manipulation of control-flow** that is **allowed by the labels**/graph
  - Called **mimicry attacks**
  - The simple, single-label CFG is susceptible to these
- **Nor data leaks or corruptions**
  - Heartbleed would not be prevented
  - Nor the `authenticated` overflow
    - Control modification is allowed by graph

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