Control-Flow Integrity

principles, implementations, and Applications

Mart´ın Abadi, Mihai Budiu, Ulfar Erlingsson and Jay Ligatti

Daejin Lee

7th September, 2020 @ IS893

Introduction

- Control Flow Integrity(CFI) tries to prevent attacks from arbitrarily controlling program behavior
- Adopts binary instrumentation to enforce CFI on Windows x86
- Compatible with existing software and simple to enforce with low overhead.

Mitigations

- StackGuard(USENIX `98)
  - Buffer Overflow Detector by inserting random value

- CRED(NDSS `04)
  - Runtime Elimination of Buffer Overflows

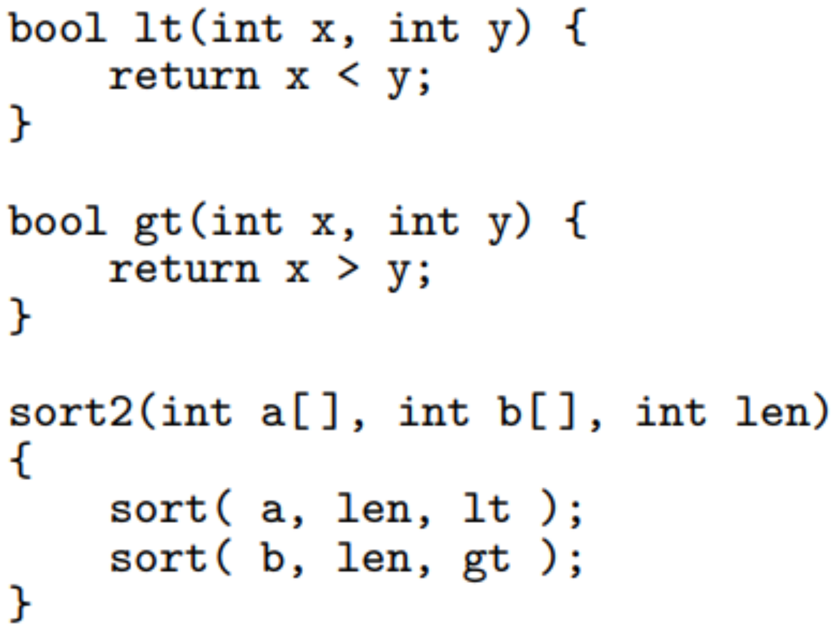
- Secure Program Execution via Dynamic Information Flow Tracking(ASPLOS `04)
  - Tainting of suspect Data

Mitigations

- StackGuard(USENIX `98)
  - Buffer Overflow Detector by inserting random value
- CRED(NDSS `04)
  - Runtime Elimination of Buffer Overflows
- Secure Program Execution via Dynamic Information Flow Tracking(ASPLOS `04)
  - Tainting of suspect Data

Hard to catch practicalness & performance

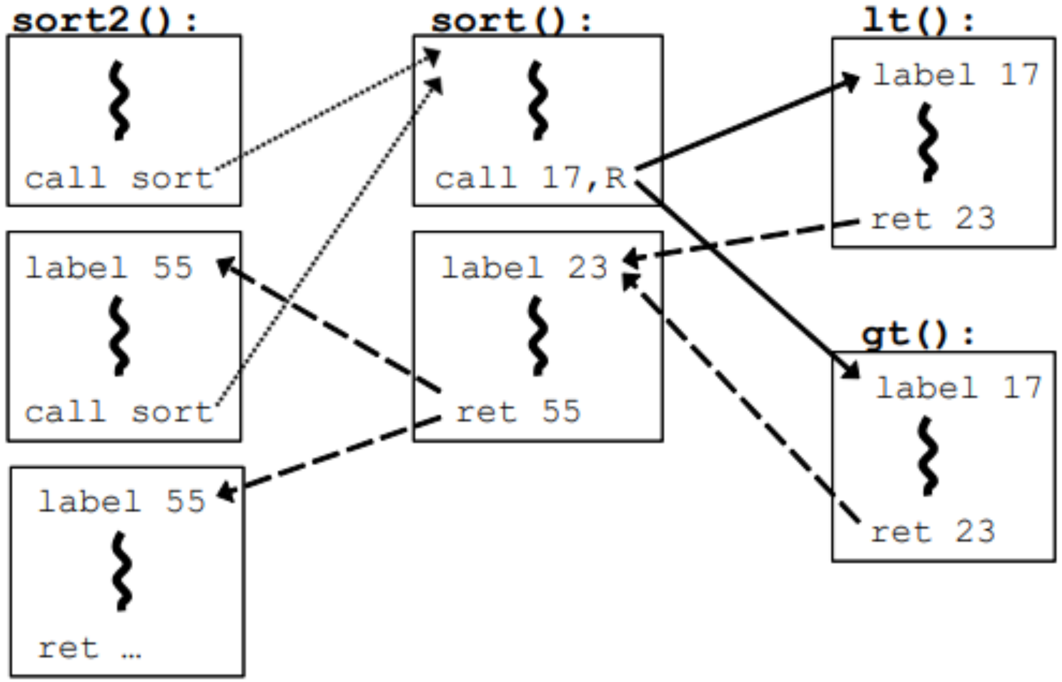
Is call/ret targets a valid destination?



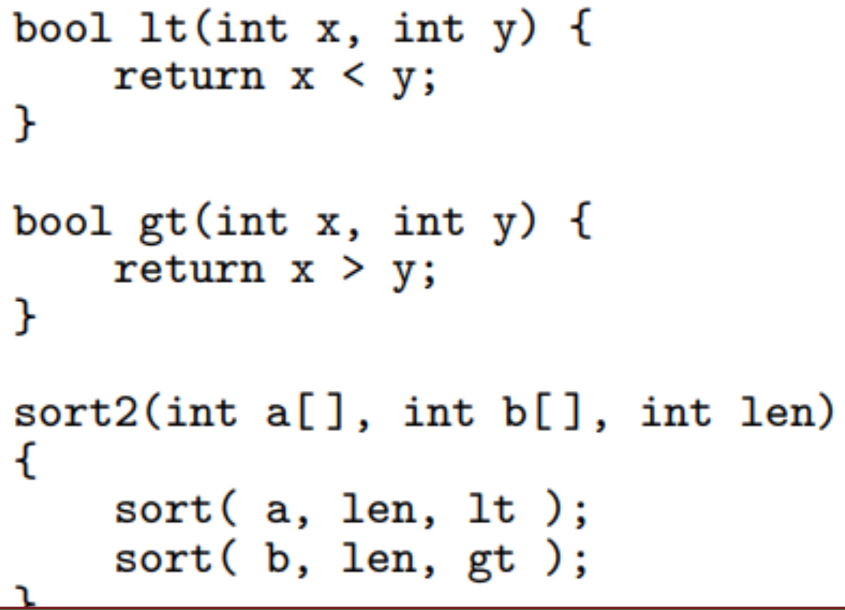
```
bool lt(int x, int y) {
    return x < y;
}

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

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



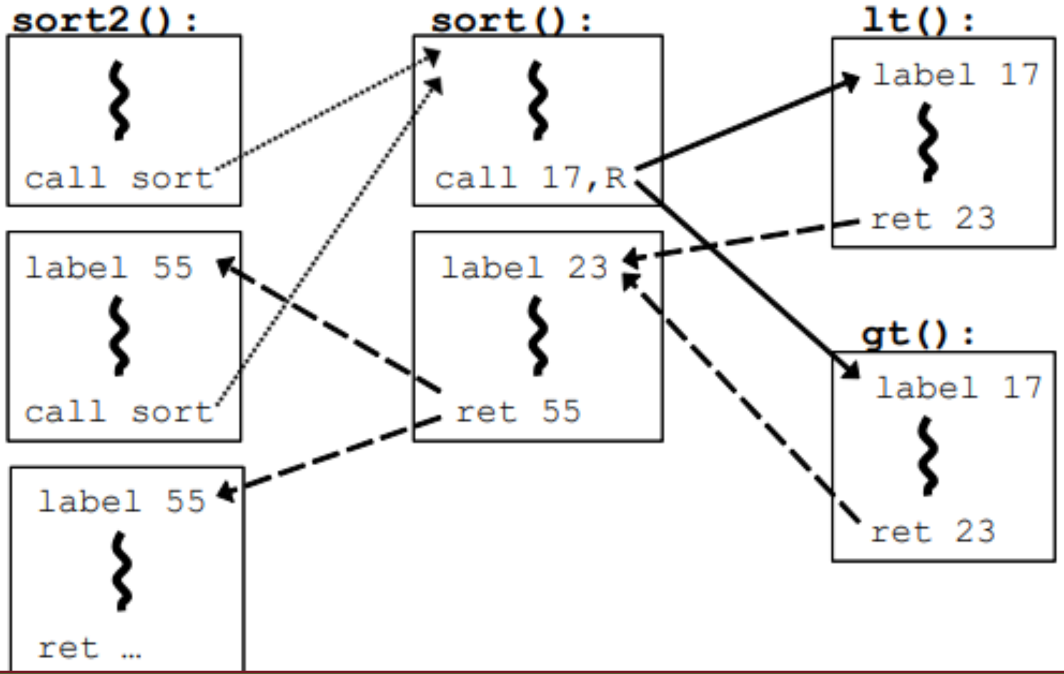
Is call/ret targets a valid destination?



```
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 );
}
```



Can be determined by a Control Flow Graph(CFG) ☺

Control Flow Integrity

- Software execution must follow a path of a CFG determined ahead of time.

- The CFG can be defined by static (source code, binary) analysis.

CFI Enforcement

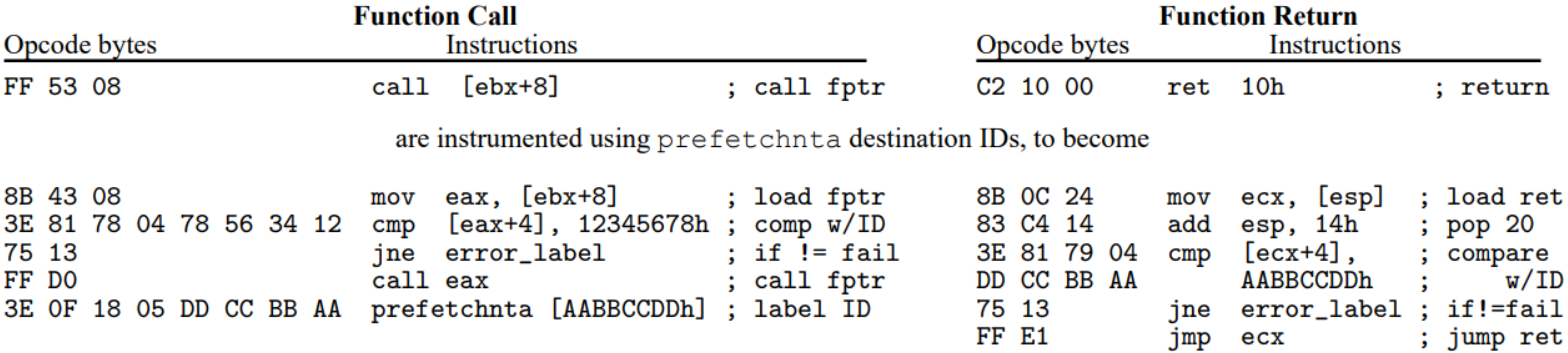
- At each destination, instrumentation inserts a bit pattern, or ID
  - Use same bit pattern for equivalent destination

- Also Insert check routine to ensure the runtime destination has the ID or bit pattern.

Assumptions

- Unique IDs
  - After CFI instrumentation, the bit pattern must not be present anywhere in the code memory except in ID's and ID-checks
- Non-Writable Code
  - Modifying code memory at runtime is not allowed
- Non-Executable Data
  - It's not possible to execute data as if it were code.

CFI Instrumentation of ‘call’ and ‘ret’



Function Call

```
Opcode bytes                  Instructions
FF 53 08                      call   [ebx+8]              ; call fptr
```

Function Return

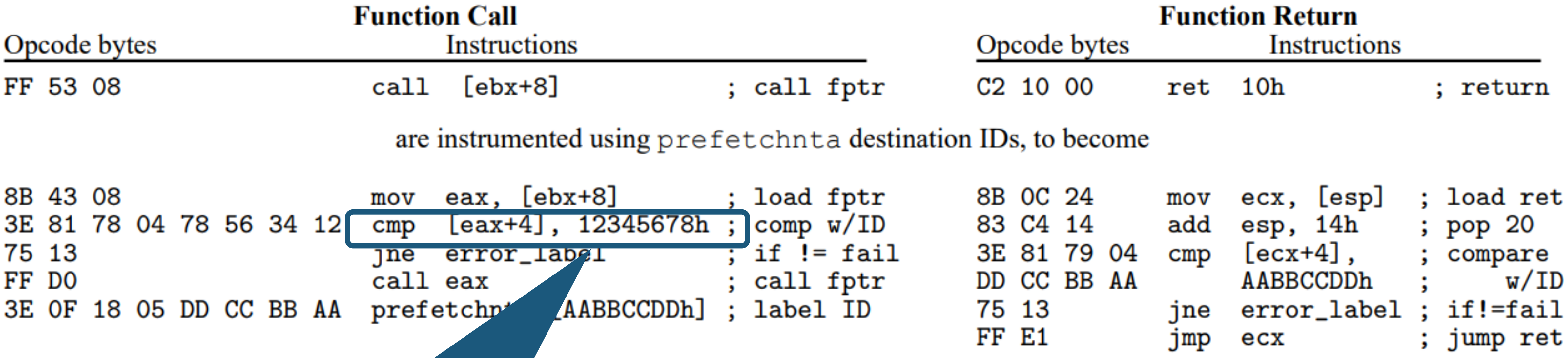
```
Opcode bytes          Instructions
C2 10 00        ret   10h                ; return
```

are instrumented using `prefetchnta` destination IDs, to become

```
8B 43 08                  mov   eax, [ebx+8]          ; load fptr
3E 81 78 04 78 56 34 12   cmp   [eax+4], 12345678h  ; comp w/ID
75 13                     jne   error_label           ; if != fail
FF D0                     call eax                    ; call fptr
3E 0F 18 05 DD CC BB AA   prefetchnta [AABBCCDDh]     ; label ID
```

```
8B 0C 24        mov   ecx, [esp]      ; load ret
83 C4 14        add   esp, 14h        ; pop 20
3E 81 79 04     cmp   [ecx+4],        ; compare
DD CC BB AA           AABBCCDDh       ;      w/ID
75 13           jne   error_label     ; if!=fail
FF E1           jmp   ecx             ; jump ret
```

CFI Instrumentation of ‘call’ and ‘ret’



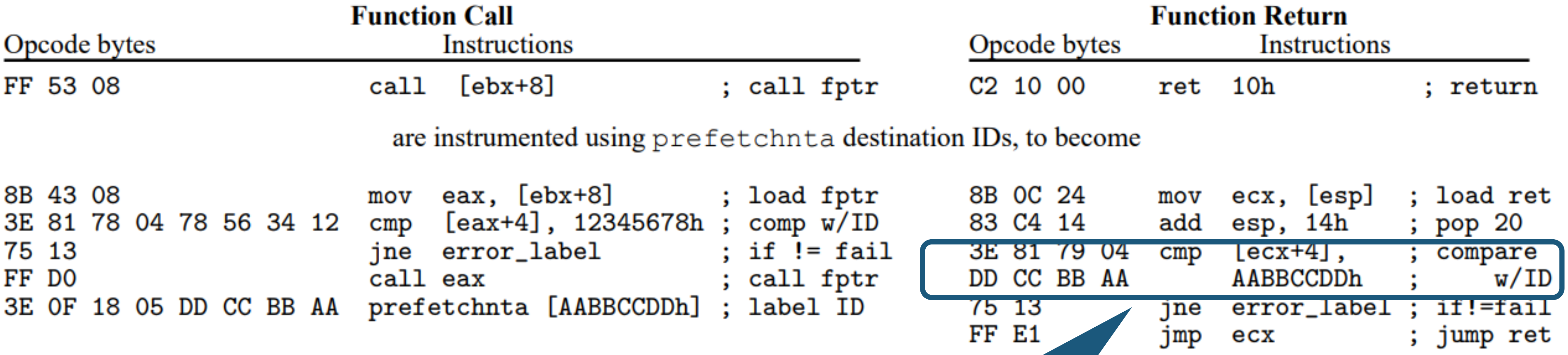
```
                    Function Call                                          Function Return
Opcode bytes                  Instructions                          Opcode bytes        Instructions
FF 53 08                 call   [ebx+8]             ; call fptr     C2 10 00      ret   10h            ; return

                 are instrumented using prefetchnta destination IDs, to become

8B 43 08                 mov   eax, [ebx+8]         ; load fptr     8B 0C 24      mov   ecx, [esp]     ; load ret
3E 81 78 04 78 56 34 12  cmp   [eax+4], 12345678h ; comp w/ID       83 C4 14      add   esp, 14h       ; pop 20
75 13                    jne   error_label          ; if != fail    3E 81 79 04   cmp   [ecx+4],       ; compare
FF D0                    call eax                   ; call fptr     DD CC BB AA         AABBCCDDh      ;     w/ID
3E 0F 18 05 DD CC BB AA  prefetchnta [AABBCCDDh] ; label ID         75 13         jne   error_label ; if!=fail
                                                                    FF E1         jmp   ecx            ; jump ret
```

Load Function Pointer and Compare with the ID

CFI Instrumentation of ‘call’ and ‘ret’



Function Call

```
Opcode bytes                    Instructions
FF 53 08                        call  [ebx+8]              ; call fptr
```

Function Return

```
Opcode bytes         Instructions
C2 10 00       ret  10h             ; return
```

are instrumented using `prefetchnta` destination IDs, to become

```
8B 43 08                  mov   eax, [ebx+8]          ; load fptr
3E 81 78 04 78 56 34 12   cmp   [eax+4], 12345678h ; comp w/ID
75 13                     jne   error_label           ; if != fail
FF D0                     call eax                    ; call fptr
3E 0F 18 05 DD CC BB AA   prefetchnta [AABBCCDDh] ; label ID
```

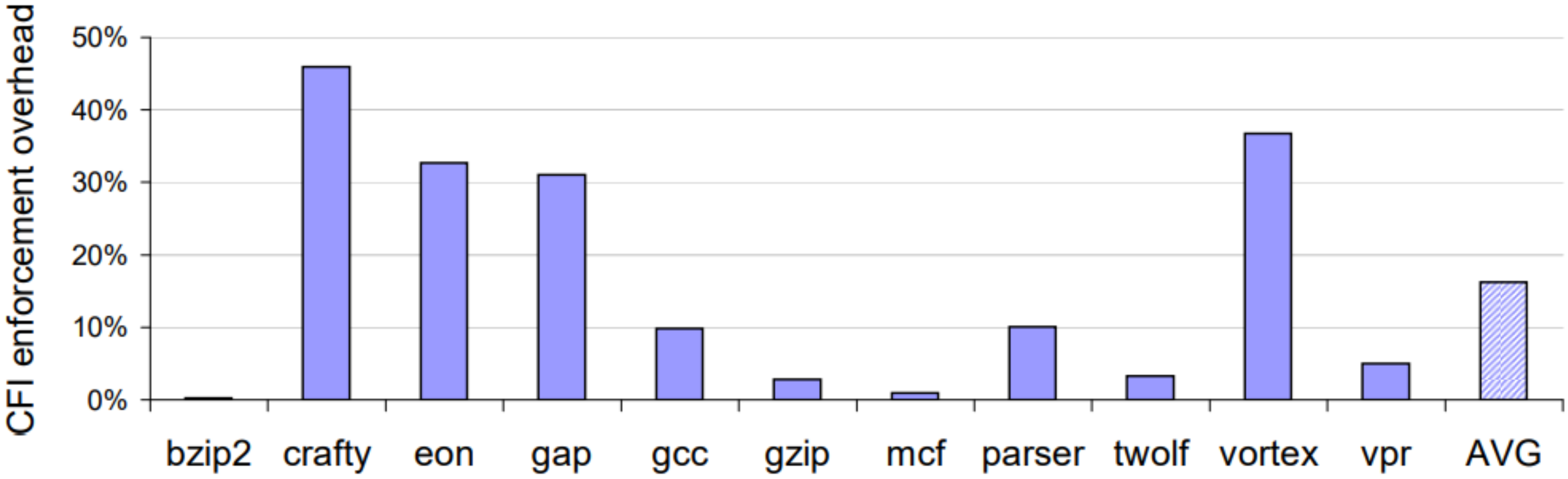
```
8B 0C 24       mov  ecx, [esp]     ; load ret
83 C4 14       add  esp, 14h       ; pop 20
3E 81 79 04    cmp  [ecx+4],       ; compare
DD CC BB AA         AABBCCDDh      ;      w/ID
75 13          jne  error_label    ; if!=fail
FF E1          jmp  ecx            ; jump ret
```

Load Return Pointer and Compare with the ID

Evaluation Setup

- Windows XP SP2 in “Safe Mode”
  - Most daemons and kernel modules are disabled
- Pentium 4 x86 processor with 512 MB RAM
- Target binaries were compiled with MS Visual C++ 7.1 using full optimizations

Execution overhead of inlined CFI



Measurements

- CFG construction + CFI instrumentation = 10 sec
- Binary increasing = 8%
- Overhead took 0~45%

- This is competitive with the cost of most comparable technique.
  - CRED: up to 130%
  - PointGuard: up to 20%
  - etc.

Function Pointer Overwrite

```
int median( int* data, int len, void* cmp )
{
        // must have 0 < len <= MAX_LEN
        int tmp[MAX_LEN];
        memcpy( tmp, data, len*sizeof(int) );
        qsort( tmp, len, sizeof(int), cmp );
        return tmp[len/2];
}
```

Function Pointer Overwrite

```
int median( int* data, int len, void* cmp )
{
        // must have 0 < len <= MAX_LEN
        int tmp[MAX_LEN];
        memcpy( tmp, data, len*sizeof(int) );
        qsort( tmp, len, sizeof(int), cmp );
        return tmp[len/2];
}
```

Input Data passed

Function Pointer Overwrite

```
int median( int* data, int len, void* cmp )
{
        // must have 0 < len <= MAX_LEN
        int tmp[MAX_LEN];
        memcpy( tmp, data, len*sizeof(int) );
        qsort( tmp, len, sizeof(int), cmp );
        return tmp[len/2];
}
```

Stack-based Buffer Overflow

Function Pointer Overwrite

```
int median( int* data, int len, void* cmp )
{
        // must have 0 < len <= MAX_LEN
        int tmp[MAX_LEN];
        memcpy( tmp, data, len*sizeof(int) );
        qsort( tmp, len, sizeof(int), cmp );
        return tmp[len/2];
}
```

Function Call ‘cmp’ Overwritten!

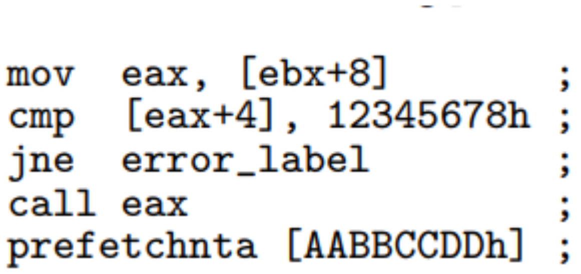
Function Pointer Overwrite

```
int median( int* data, int len, void* cmp )
{
        // must have 0 < len <= MAX_LEN
        int tmp[MAX_LEN];
        memcpy( tmp, data, len*sizeof(int) );
        qsort( tmp, len, sizeof(int), cmp );
        return tmp[len/2];
```

Vtable Overwrite :D

Function Pointer Overwrite

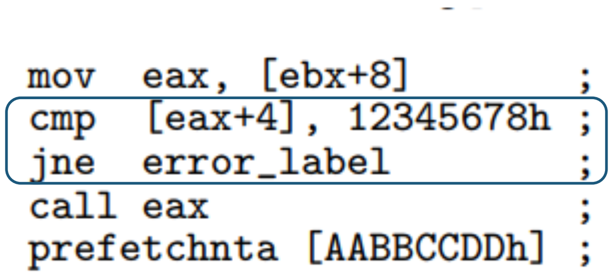
```
int median( int* data, int len, void* cmp )
{
        // must have 0 < len <= MAX_LEN
        int tmp[MAX_LEN];
        memcpy( tmp, data, len*sizeof(int) );
        qsort( tmp, len, sizeof(int), cmp )
        return tmp[len/2];
}
```



```
mov   eax, [ebx+8]          ;
cmp   [eax+4], 12345678h    ;
jne   error_label           ;
call  eax                   ;
prefetchnta [AABBCCDDh]     ;
```

Function Pointer Overwrite

```
int median( int* data, int len, void* cmp )
{
        // must have 0 < len <= MAX_LEN
        int tmp[MAX_LEN];
        memcpy( tmp, data, len*sizeof(int) );
        qsort( tmp, len, sizeof(int), cmp )
        return tmp[len/2];
```



```
mov   eax, [ebx+8]          ;
cmp   [eax+4], 12345678h    ;
jne   error_label           ;
call  eax                   ;
prefetchnta [AABBCCDDh]     ;
```

Fails :p

Security-Related Experiments

- Prevented
  - Jump to libc
  - Virtual Table Overwrite
  - etc.

- Not prevented
  - Incorrect parsing of input strings

Critique

+ Simple yet effective mitigation for many real world programs
+ Low overhead, Practical solution

- We can still use valid CFG as exploit method in some cases.

Questions? ☺