# Control-Flow Integrity

## Principles, Implementations, and Applications

Κωνσταντίνα Μανώλη 2634
Κωνσταντινος Δημάκης 2528
Κωνσταντίνος Αθανασίου 2690

# CFI: Goal

Provably correct mechanisms that prevent powerful attackers from succeeding by protecting against all Unauthorized Control Information Tampering (UCIT) attacks

# CFI: Idea

During program execution, whenever a machine-code instruction transfers control, it targets a valid destination, as determined by a Control Flow Graph (CFG) created ahead of time.

# Attack Model

Powerful Attacker: Can at any time arbitrarily overwrite any data memory and (most) registers

- Attacker cannot directly modify the PC
- Attacker cannot modify reserved registers

Assumptions:

Data memory is Non-Executable

Code memory is Non-Writable

# Control-Flow Integrity

Main idea: pre-determine control flow graph (CFG) of an application

- Static analysis of source code
- Static binary analysis ← CFI

Execution must follow the pre-determined control flow graph

# CFI: Control Flow Enforcement

- For each control transfer, determine statically its possible destination(s)
- Insert a unique bit pattern at every destination
  - Two destinations are equivalent if CFG contains edges to each from the same source
    - This is imprecise (later)
  - Use same bit pattern for equivalent destinations
- Insert binary code that at runtime will check whether the bit pattern of the target instruction matches the pattern of possible destinations

# CFI: Binary Instrumentation

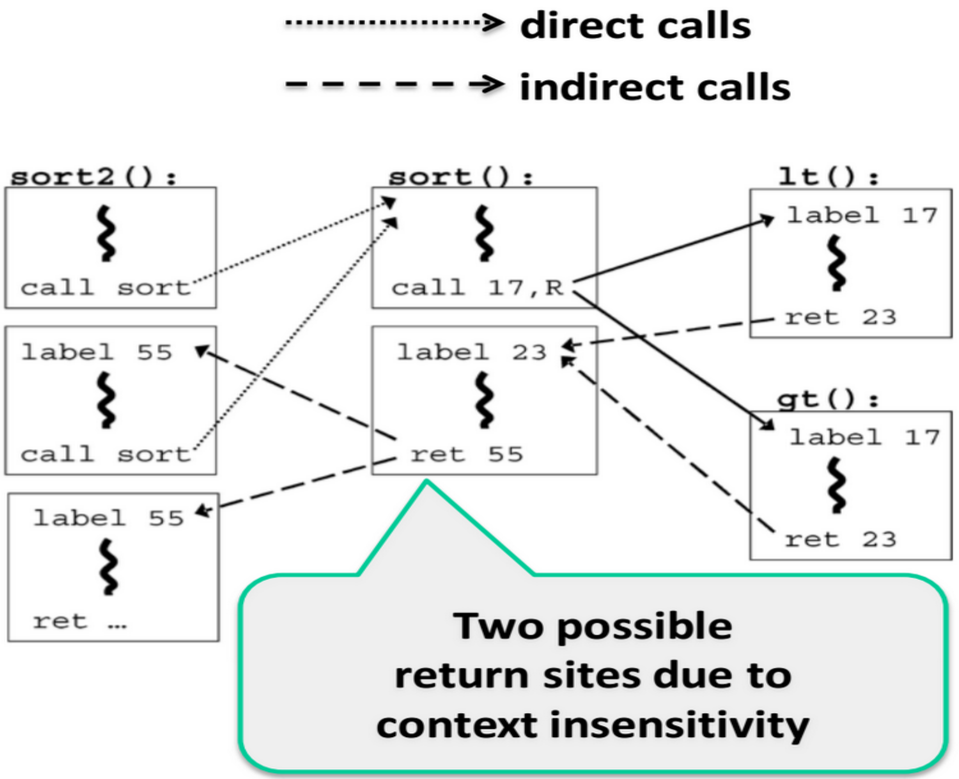
- Use binary rewriting to instrument code with runtime checks (similar to SFI)
- Inserted checks ensure that the execution always stays within the statically determined CFG
  - Whenever an instruction transfers control, destination must be valid according to the CFG
- Goal: prevent injection of arbitrary code and invalid control transfers (e.g., return-to-libc)
  - Secure even if the attacker has complete control over the thread's address space

# Phases of Inlined CFI enforcement

- Build CFG statically, e.g., at compile time
- Instrument (rewrite) binary, e.g., at install time
  - Add IDs and ID checks; maintain ID uniqueness(later)
- Verify CFI instrumentation at load time
  - Direct jump targets, presence of IDs and ID checks, ID uniqueness
- Perform ID checks at run time
  - Indirect jumps have matching IDs

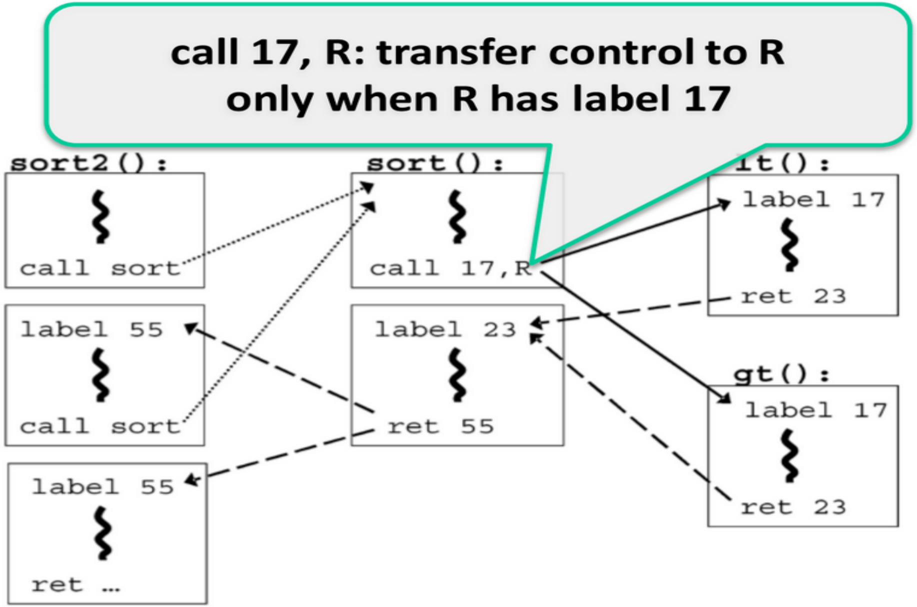
# Example CFG

```
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 );
}
```



# Instrument Binary

```
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 );
}
```




- Insert a unique number at each destination
- Two destinations are equivalent if CFG contains edges to each from the same source

# CFI: Example of Instrumentation

## Original code

**Source**

| Opcode bytes | Instructions | |
|---|---|---|
| FF E1 | jmp ecx | ; computed jump |

**Destination**

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

## Instrumented code

```
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
...
```

**Jump to the destination only if the tag is equal to "12345678"**

**Abuse an x86 assembly instruction to insert "12345678" tag into the binary**

# Verify CFI Instrumentation

- **Direct jump targets (e.g., call 0x12345678)**

– Are all targets valid according to CFG?

- **IDs**

– Is there an ID right after every entry point?

– Does any ID appear in the binary by accident?

- **ID checks**

– Is there a check before every control transfer?

– Does each check respect the CFG?

# CFI: Assumptions

- Unique IDs- Don’t conflict with opcodes. Done by making ID 32 bit number.
- Non-Writable Code .Code segment must be write protected.
- Non-Executable Data .Data segment is not executable.
- The assumptions can be somewhat problematic in the presence of self-modifying code, runtime code generation, and the unanticipated dynamic loading of code.
- Fortunately, most software is rather static - either statically linked or with a statically declared set of dynamic libraries.

# Improving CFI Precision

Function F is called first from A, then from B; what's a valid destination for its return?

- CFI will use the same tag for both call sites, but this allows F to return to B after being called from A
- Solution: shadow call stack (later)

# Evaluations

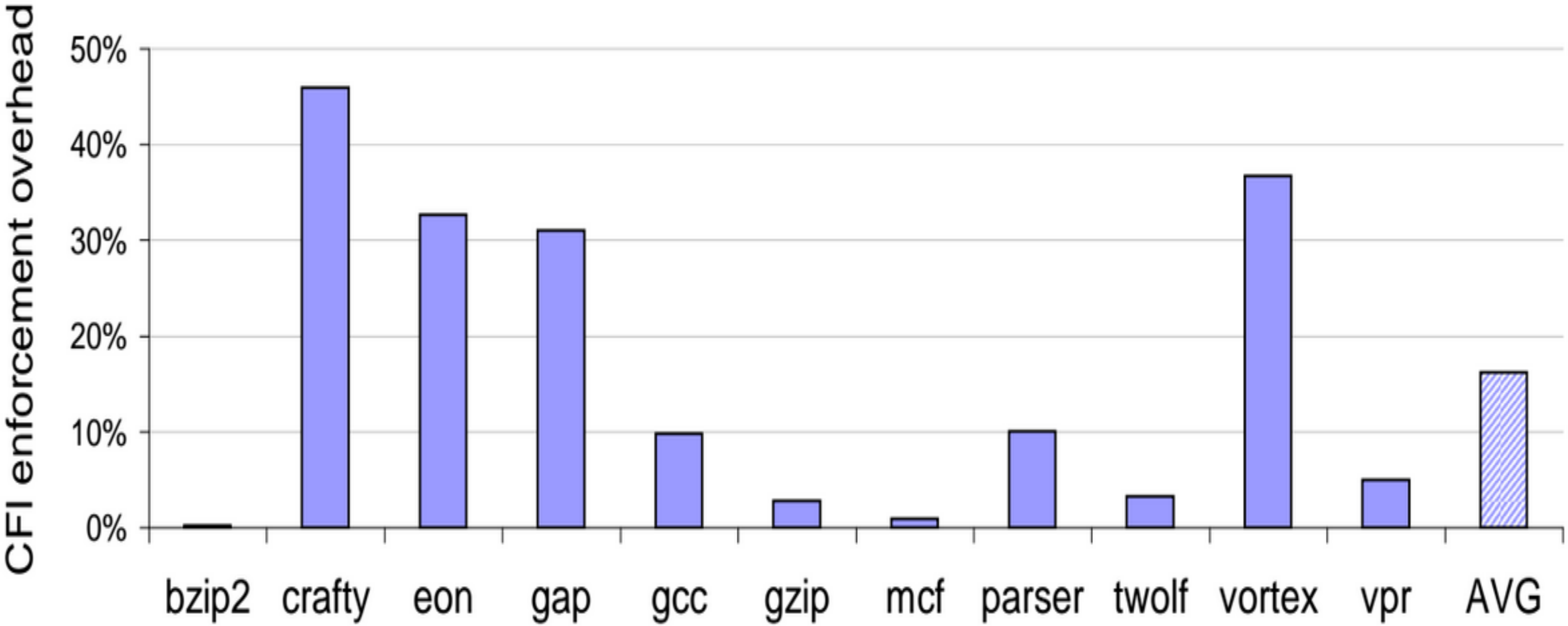



**Figure 4: Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.**

# Evaluations

CFG construction + CFI instrumentation: ~10s

Increase in binary size: ~8%

Relative execution overhead:

– crafty: CFI – 45%

– gcc: CFI < 10%

**Security-related experiments**

– CFI protects against various specific attacks

# CFI: Security Guarantees

- Effective against attacks based on illegitimate control-flow transfer
  - Stack-based buffer overflow, return-to-libc exploits, pointer subterfuge
- Does <u>not</u> protect against attacks that do not violate the program's original CFG
  - Incorrect arguments to system calls
  - Substitution of file names
  - Other data-only attacks

# Software Fault Isolation (SFI)

- Processes live in the same hardware address space; software reference monitor isolates them
  - Each process is assigned a logical "fault domain"
  - Check all memory references and jumps to ensure they don't leave process's domain
- Tradeoff: checking vs. communication
  - Pay the cost of executing checks for each memory write and control transfer to save the cost of context switching when trapping into the kernel

# Simple SFI Example

- Fault domain = from 0x1200 to 0x12FF
- Original code: `write x`
- Naïve SFI:
```
x := x & 00FF
x := x | 1200
write x
```
- Better SFI:
```
tmp := x & 00FF
tmp := tmp | 1200
write tmp
```

# Inline Reference Monitor

- Generalize SFI to more general safety policies than just memory safety
  - Policy specified in some formal language
  - Policy deals with application-level concepts: access to system resources, network events, etc.
    - "No process should send to the network after reading a file", "No process should open more than 3 windows", …
- Policy checks are integrated into the binary code
  - Via binary rewriting or when compiling
- Inserted checks should be uncircumventable
  - Rely on SFI for basic memory safety

# SFI

- CFI implies non-circumventable sandboxing (i.e.,safety checks inserted by instrumentation before instruction X will always be executed before reaching X)
- SFI: Dynamic checks to ensure that target memory accesses lie within a certain range
  – CFI makes these checks non-circumventable

# SMAC: Generalized SFI

SMAC: Different access checks at different instructions in the program

– Isolated data memory regions that are only accessible by specific pieces of program code (e.g., library function)

– SMAC can remove NX data and NW code assumptions of CFI

– CFI makes these checks non-circumventable

# Example: CFI + SMAC

```
call  eax                ; call a function pointer (destination address)
```

with CFI, and SMAC discharging the NXD requirement, can become:

```
and  eax, 40FFFFFFh      ; mask to ensure address is in code memory
cmp  [eax+4], 12345678h  ; compare opcodes at destination
jne  error_label         ; if not ID value, then fail
call eax                 ; call function pointer
prefetchnta [AABBCCDDh]  ; label ID, used upon the return
```

- Non-executable data assumption no longer needed since SMAC ensures target address is pointing to code

# Shadow Call Stack

- place stack in SMAC-protected memory region
- only SMAC instrumentation code at call and return sites

    modify stack by pushing and popping values
- Statically verify that instrumentation code is correct

# Conclusion (1)

- Use of high level programming language implies that only certain control flow has to be executed during software execution.

- The absence of runtime control-flow guarantees has a pervasive impact on all software analysis, processing, and optimization and it also enables many of today's exploits.

# Conclusion (2)

- CFI instrumentation ams to change that by embedding runtime checks within software executable to prevent from many exploits.
- Inlined CFI enforcement is practical on modern processors, is compatible with most existing software, and has little performance overhead.
- CFI is simple, verifiable, and amenable to formal analysis,yielding strong guarantees even in the presence of a powerful adversary.