Control-Flow Integrity (CFI)

Principles, Implementations, and Applications (Abadi et al.)

Paper Review - Reliable System Class

What is CFI?

CFI enforces that runtime execution follows only edges in a precomputed Control-Flow Graph (CFG).

- Goal: Prevent attackers from arbitrarily steering machine-code execution.
- Claim: Ideally prevent all non logic bugs. In this paper's context: simple policy, formally analyzable guarantees, practical deployment via binary rewriting.

Threat Model

- Adversary may have **full control of program data memory (even some registers)**.
- Adversary may have full control of another thread in the same address space.
- CFI still aims to enforce valid control flow under this strong adversary.

What is a CFG in this context?

- It is similar to the function call graph, but each call/return site, is regarded as a different point.
- The CFG is computed ahead of execution.
- Every runtime control flow transfer must follow one of the precomputed CFG edges.

```
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 );
}
```

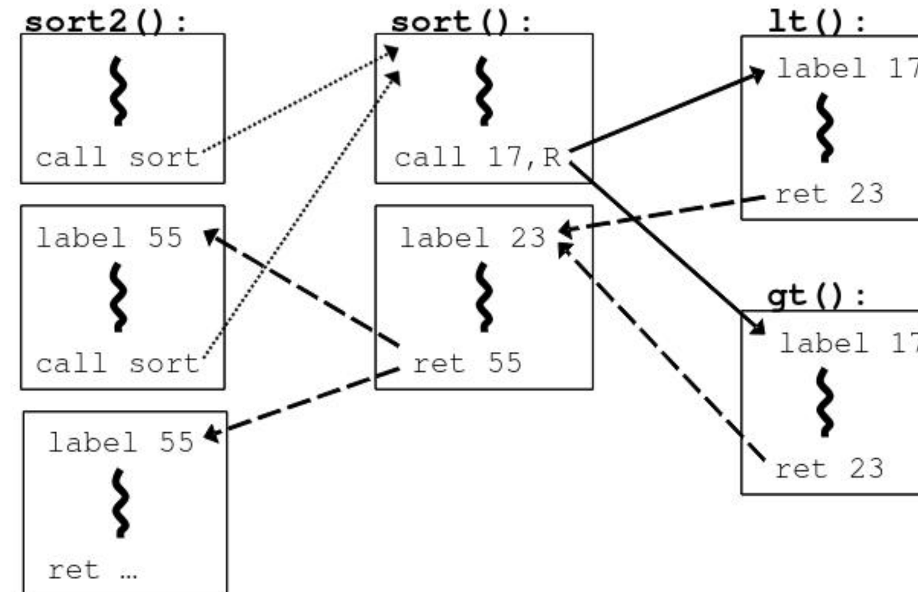


Figure 1: Example program fragment and an outline of its CFG and CFI instrumentation.

Which edges are of interest?

- **Direct control transfers** are checkable statically.
- **Computed control transfers** need dynamic enforcement:
  - Indirect calls/Function pointers
  - Returns
  - Switch statement/vtable related code

Background / Related Work (Security)

- Prior defenses: stack integrity checks (StackGuard), pointer encryption/randomization, etc.
- CFI positions itself as:
  - Stronger control-flow guarantees
  - Deployable to legacy binaries
  - Amenable to formal analysis

Similarity to Fault-Tolerance Mechanisms

Similarity:

- Both encode/check expected control flow in software.

Differences:

- Fault-tolerance mechanism: Usually probabilistic guarantees, do not expect adversary just errors.
- CFI: Adversary controlling data memory, deterministic CFG integrity guarantees.

Core Challenges

1. Infer CFG from binaries.
2. Instrument dynamic checks without breaking program semantics.
3. Keep overhead low enough for practical deployment on existing binaries.

High-Level Solution

1. Add **IDs** at valid destinations.
2. Add **ID-checks** before computed transfer sources.
3. Abort when ID-check fails.

System Overview

- System targets **Windows x86** user-level binaries.
- Built with **Vulcan** for both binary rewriting/instrumentation system and CFG extraction.
- No source code or recompilation required.

Source/Destination Classes

- Two destinations are considered equivalent if their CFG has edges to them from the same sources.
- The assumption that 2 destinations are equivalent if they share a common source can be mitigated via multiple IDs or code duplication/inlining

Imaginary Instructions for Instrumenting

Conceptual ISA introduced for clarity:

- `label ID` (effect-free marker)
- `call ID, DST` (transfer only if the destination begins with matching label)
- `ret ID` (analogous return check)

These are explanatory abstractions before mapping to concrete x86 opcodes.

Approach A (Figure 2a)

- Store ID bytes directly before destination instruction.
- Check with direct comparison against destination bytes.
- Adjust destination pointer to skip ID bytes (lea-style pattern).

Source			Destination		
Opcode bytes	Instructions		Opcode bytes	Instructions	
FF E1	jmp ecx	; computed jump	8B 44 24 04	mov eax, [esp+4]	; dst
			...		

can be instrumented as (a):

Source opcode bytes	Source instructions		Destination opcode bytes	Destination instructions	
81 39 78 56 34 12	cmp [ecx], 12345678h	; comp ID & dst	78 56 34 12	; data 12345678h	; ID
75 13	jne error_label	; if != fail	8B 44 24 04	mov eax, [esp+4]	; dst
8D 49 04	lea ecx, [ecx+4]	; skip ID at dst	...		
FF E1	jmp ecx	; jump to dst			

Tradeoff:

- Simple logic, but LABEL is part of the code now.

Approach B (Figure 2b)

- Use side-effect-free prefetchnta as the destination label carrier.
- Build/check ID using register arithmetic (ID-1, then increment).
- Avoids modifying the target.

or, alternatively, instrumented as (b):

```
B8 77 56 34 12     mov   eax, 12345677h     ; load ID-1        3E 0F 18 05     prefetchnta          ; label
40                 inc   eax                ; add 1 for ID     78 56 34 12        [12345678h]       ;    ID
39 41 04           cmp   [ecx+4], eax       ; compare w/dst    8B 44 24 04     mov   eax, [esp+4]   ; dst
75 13              jne   error_label        ; if != fail       ...
FF E1              jmp   ecx                ; jump to label
```

Assumptions Required for Guarantees

1. **UNQ**: ID bit patterns appear only in legitimate IDs/ID-checks.
2. **NWC**: code memory is not writable at runtime.
3. **NXD**: data memory is not executable.
4. Register/system-call constraints must preserve these assumptions.

Implementation (Instruction-Wise)

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

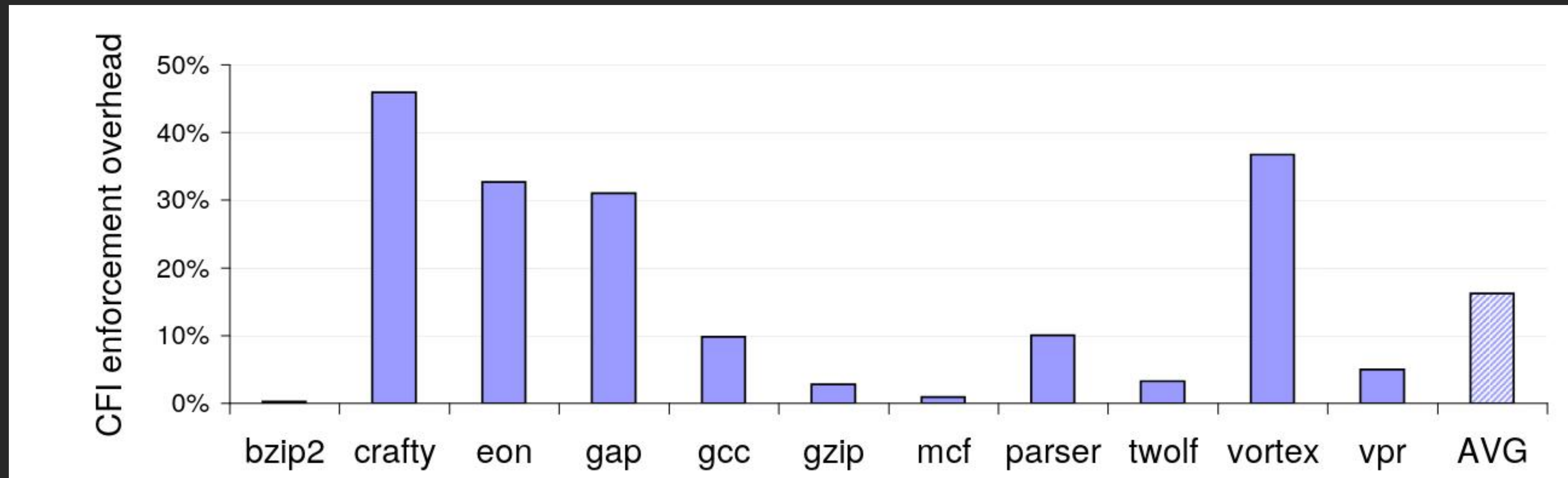
```
8B 43 08                  mov   eax, [ebx+8]          ; load fptr
3E 81 78 04 78 56 34 12   cmp   [eax+4], 12345678h    ; comp w/ID
75 13                     jne   error_label           ; if != fail
FF D0                     call eax                    ; call fptr
3E 0F 18 05 DD CC BB AA   prefetchnta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

- It is a mix of both methods.
- comparison method from a) is used, as they claim it can only allow infinite loops.
- They use prefetch to avoid altering the target.

Overhead

- CFG extraction + instrumentation time: about 10s per binary.
- Binary size increased about 8% average.
- SPEC2000 runtime overhead: 0%-45%, **16% average**.
- Competitive performance with alternatives (order of magnitude improvements in some cases).



Security Results

- It could have prevented some known vulnerabilities, while others not.
- It successfully prevented some attacks for 18 vulnerable programs.
- Attacks that remain inside the allowed CFG are out of scope.

Building on CFI: IRMs, SFI, SMAC

- **IRMs**: IRM systems, allow inserting inline code that checks whether some arbitrary policy may be violated. CFI helps ensure inserted checks cannot be bypassed.
- **Faster SFI**: SFI systems try to ensure memory safety by checking that memory accesses are within certain bounds. CFI enables stronger optimizations like avoiding multiple checks on local variables, with reported overhead reductions on benchmark examples (order of magnitude improvements).
- **SMAC**: SFI with per instruction access policies. CFI can allow for optimizations like with SFI but SMAC can relax the NWC and NXD requirements.

Protected Shadow Stack

- CFI cannot guarantee a return to the most recent callsite.
- CFI can be paired with a protected shadow call stack using x86 segments (Windows implementation).
- Authors contrast this with much higher prior reported overheads for protected shadow-stack approaches (improvements reach even 2 orders of magnitude).