CS 250 Software Security

Program Hardening

Efficient Software-Based Fault Isolation

ACM SIGOPS 1993

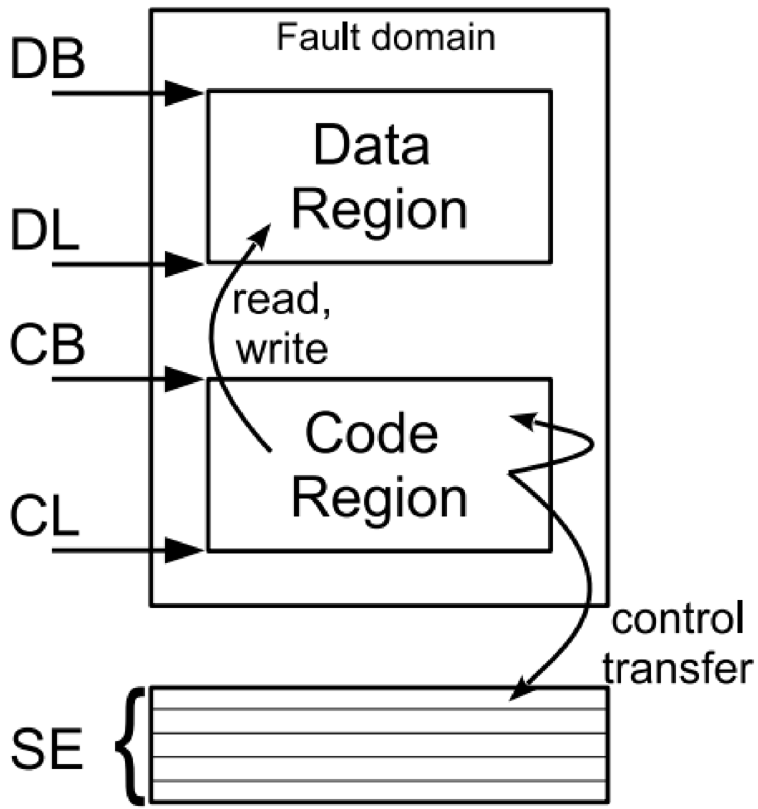
Why do we need SFI?

- Use protection domains to isolate untrusted components
  - A web browser should isolate browser plugins
  - An OS should isolate device drivers
- Designate a memory region for an untrusted component and instrument dangerous instructions in it to constrain its memory access and control transfer
- Highly desirable to isolate untrusted components in separate protection domains, grant them minimum privileges

	Context-switch overhead	Per-instruction overhead	Com-piler support	Software engineering effort
Virtual machines	very high	none	no	none
OS processes	high	none	no	none
Language-based isolation	low	medium (dynamic) or none (static)	yes	high
SFI	low	low	maybe	none or medium

Gang Tan (2017), “Principles and Implementation Techniques of Software-Based Fault Isolation”, Foundations and Trends R in Privacy and Secruity: Vol. 1, No. 3, pp 137–198. DOI: 10.1561/3300000013.

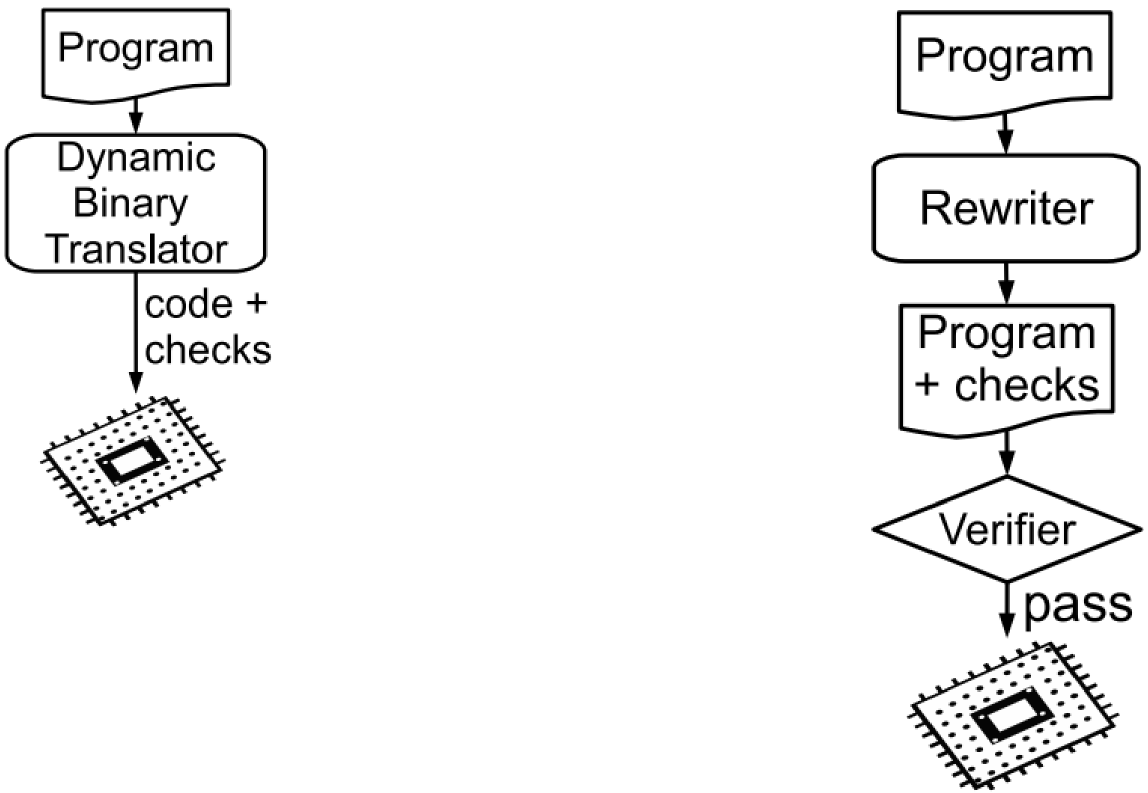
SFI Policy



- A data region holds all data needed by the code
- A code region where code is loaded into
- A set of safe external code addresses
- Three regions are mutually disjoint
- Data-access policy: All memory reads and writes by the code should be within the data region
- Control-flow policy: A control transfer by the code must stay within the code or target a safe external address

SFI Enforcement

(a) Dynamic binary translation (b) Inlined reference monitors



Implementing IRM Rewriters

- Binary-rewriting
  - Does not require source code
  - Disassembling binaries without metadata can be challenging
  - Optimizing checks in binaries is challenging

- Compiler-based instrumentation
  - Requires access to source code
  - Perform more precise static analysis for optimizing checks
  - Portable across architectures

Enforcing the Data-Access Policy

- A naïve implementation can insert checks before all memory instructions
  - Mem(r1 + 12) := r2 →

    r10 := r1 + 12
    if (r10 < DB) goto error
    if (r10 > DL) goto error
    mem(r10) := r2

  - Too much runtime overhead
  - Need optimizations

Data Region Specialization

- Make all addresses have the same upper bits.
  - These upper bits are called the data region ID.

    mem(r1 + 12) := r2 →

    r10 := r1 + 12
    r11 := r10 >> 16
    if (r11 6= 0x1234) goto error
    mem(r10) := r2

- A right-shift instruction in more efficient than a conditional jump

Integrity-only Isolation

- A typical perform performs more memory reads than writes
- To ensure the integrity of memory outside the data region, we only need to check memory writes
- So, this weakened policy leads to much lower runtime overhead

Address Masking

- Force an address to be inside the data region
  - “mem(r1 + 12) := r2” →

    r10 := r1 + 12
    r10 := r10 & 0x0000FFFF
    r10 := r10 | 0x12340000
    mem(r10) := r2
- PittSFIeld further fixes the data region ID to have only one single bit on
  - r10 := r1 + 12
    r10 := r10 & 0x2000FFFF
    mem(r10) := r2

Enforcing the Control-Flow Policy

- Control transfers by the sandboxed code must stay in the code region or target a safe external address
- One solution: A dedicated register is used to hold the jump target

- Strengthened control-flow policy
  - All control-flow transfers must target the beginning of a pseudo instruction in the code or target a safe external address.
  - Fine-grained CFI (will be discussed later)
  - Aligned-chunk enforcement (PittSFIeld)

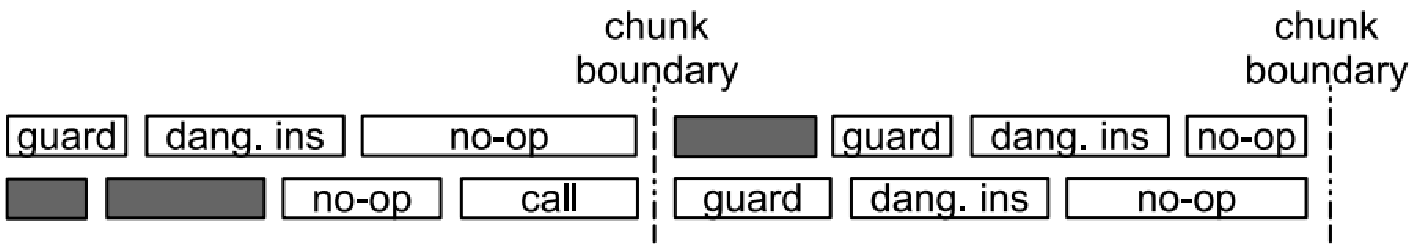


Figure 3.5: Illustration of the aligned-chunk enforcement. Black-filled rectangles represent regular (non-dangerous) instructions. Rectangles with "dang. ins" represent dangerous instructions, which are preceded by guards. For alignment, no-op instructions have to be inserted. Furthermore, call instructions are placed at the end of chunks since return addresses must be aligned.

Interaction with the Outside World

- A list of predefined APIs that sandboxed code can call
- A deep copy of arguments (also called marshalling)

Control-Flow Integrity

ACM CCS 2005

CFI: Control-Flow Integrity

- Main idea: pre-determine control flow graph (CFG) of an application
  - Static analysis of source code
  - Static binary analysis ← CFI
  - Execution profiling
  - Explicit specification of security policy
- Execution must follow the pre-determined control flow graph

CFI: Binary Instrumentation

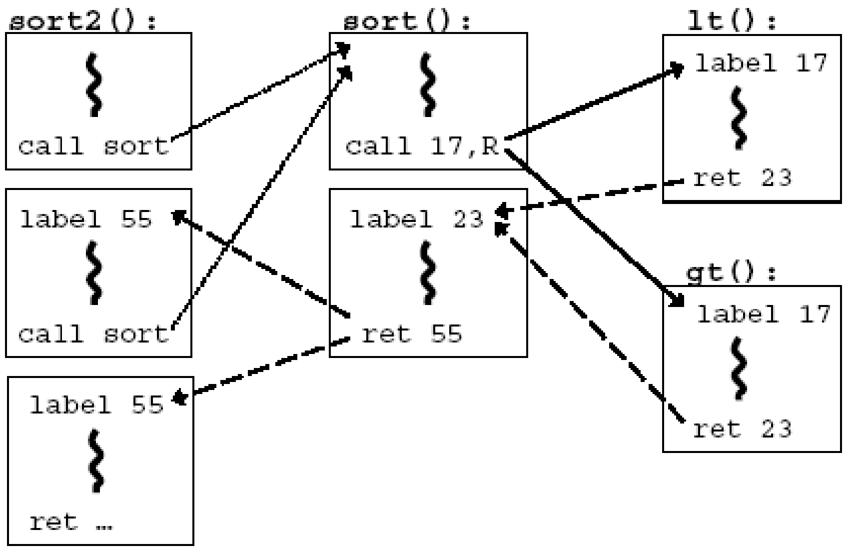
- Use binary rewriting to instrument code with runtime checks (similar to SFI)
- Inserted checks ensure that the execution always stays within the statically determined CFG
  - Whenever an instruction transfers control, destination must be valid according to the CFG
- Goal: prevent injection of arbitrary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

CFG Example

```
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 );
}
```



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 (why?)
  - 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: Example of Instrumentation

Original code

Source			Destination		
Opcode bytes	Instructions		Opcode bytes	Instructions	
FF E1	jmp ecx	; computed jump	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

CFI: Preventing Circumvention

- Unique IDs
  - Bit patterns chosen as destination IDs must not appear anywhere else in the code memory except ID checks
- Non-writable code
  - Program should not modify code memory at runtime
    - What about run-time code generation and self-modification?
- Non-executable data
  - Program should not execute data as if it were code
- Enforcement: hardware support + prohibit system calls that change protection state + verification at load-time

Improving CFI Precision

- Suppose a call from A goes to C, and a call from B goes to either C, or D (when can this happen?)
  - CFI will use the same tag for C and D, but this allows an “invalid” call from A to D
  - Possible solution: duplicate code or inline
  - Possible solution: multiple tags
- 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

CFI: Security Guarantees

- Effective against attacks based on illegitimate control-flow transfer
  - Stack-based buffer overflow, return-to-libc exploits, pointer subterfuge
- Does not 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

Performance Overhead

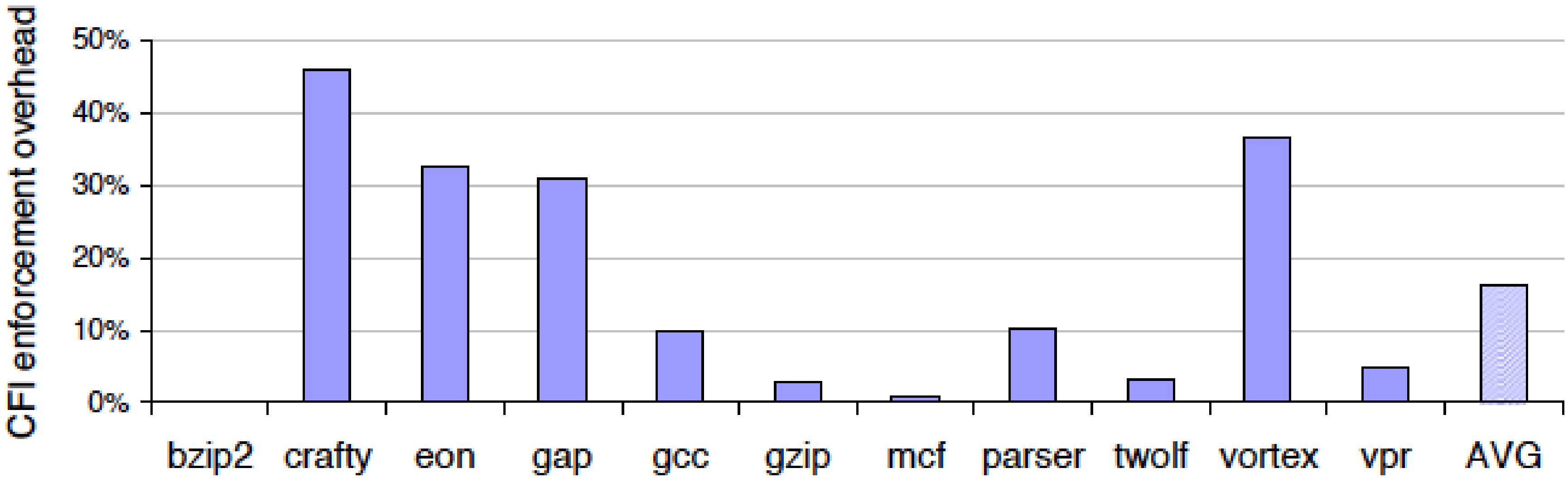


Figure 4: Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

Performance Overhead (2)

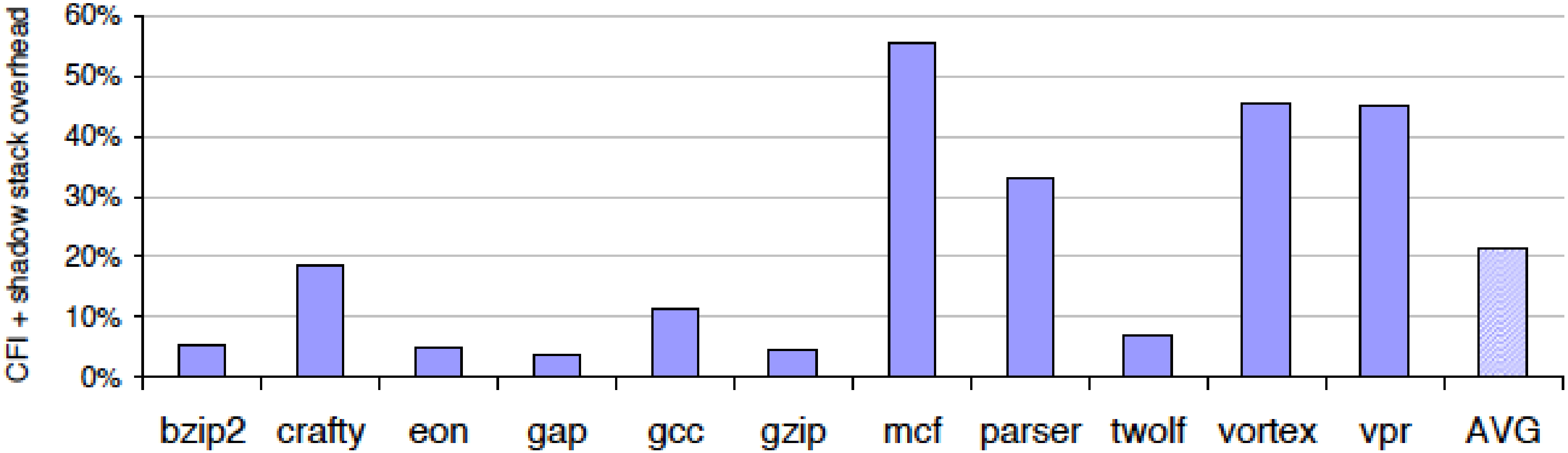


Figure 8: Enforcement overhead for CFI with a protected shadow call stack on SPEC2000 benchmarks.

What's more?

- Memory Safety
  - CCured (POPL 2002): combines type inference and runtime checking
  - SoftBound (PLDI 2009): enforce spatial memory safety
  - CETS (ISMM 2010): enforce temporal memory safety
- Data-Flow Integrity (DFI): OSDI 2006
  - DFI inserts checks before memory reads and writes to enforce the runtime data flow is compliant with the Data Flow Graph (DFG)
- Write-Integrity Testing (WIT): S&P 2008
  - For each memory write, pointer analysis is employed to compute the approximate set of objects that can be written by the memory write. At runtime, write sets are remembered by a color table and dynamic checks are used to prevent a memory write to change objects outside its write set.
- Code Pointer Integrity (CPI): OSDI 2014
  - CPI guarantees the integrity of all code pointers in a program