


Control Flow Integrity (CFI)

Anupam Datta

**CMU
Fall 2010**

Lecture Outline

- CFI: Overview and Approach 
- CFI: Implementation and Evaluation
- Building on CFI
 - IRM, SFI, SMAC, Protected Shadow Call Stack
- Formal Study

Goal

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

CFI: Broad Goal

CFI requires that, during program execution, whenever a machine-code instruction transfers control, it targets a valid destination, as determined by a CFG created ahead of time.

- Constant destination: statically checked
- Computed destination: CFI inserts checks via machine code rewriting that are checked at run time

Some connections with previous lectures

- Software analysis methods *assume CFG accurately reflects possible executions of program*
 - Software model checking (ASPIER, MOPS)
 - Static analysis (Coverity Prevent)
- Language-based methods
 - Type systems guarantee memory and control flow safety for programs written in *that* language (PCC, TAL)
 - No guarantees if data memory corrupted by another entity or flaw
- Run-time enforcement methods can be circumvented *if CFG not respected*
 - Software-based Fault Isolation (SFI)
 - Inlined Reference Monitors (IRMs)

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

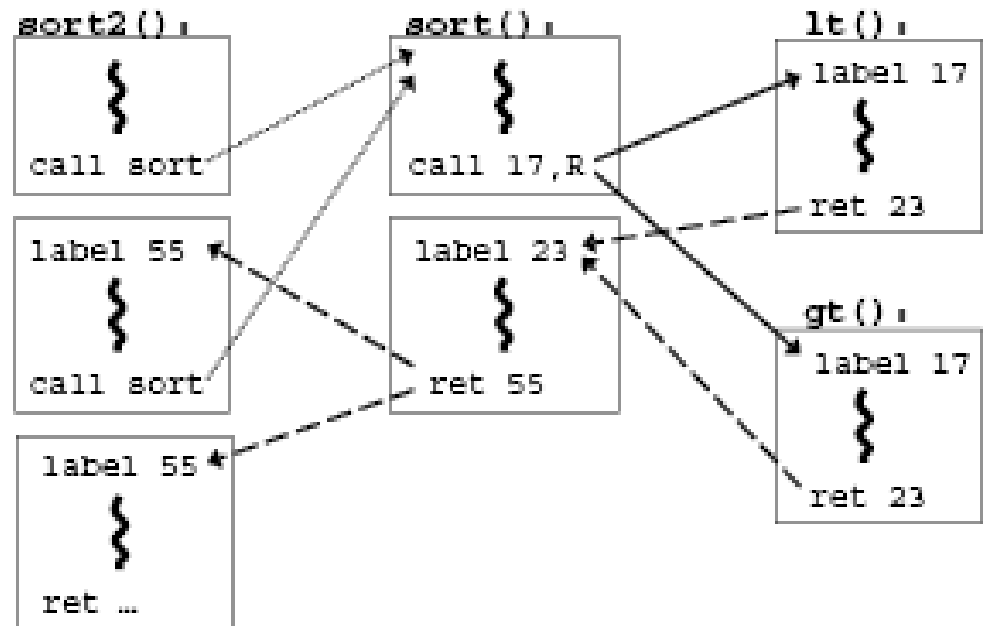
Few Assumptions:

- Data memory is Non-Executable
- Code memory is Non-Writable

Example: Code and CFI Outline

```
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 );
}
```



Note: Only indirect function calls require an ID-check

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
 - Not very straightforward for computed jumps

Example of Instrumentation

Original code

Opcode bytes	Source Instructions
FF E1	jmp ecx ; computed jump

Opcode bytes	Destination 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
```

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

```
3E 0F 18 05    prefetchnta    ; label
78 56 34 12    [12345678h]   ; ID
8B 44 24 04    mov  eax, [esp+4] ; dst
...
```

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

Example: CFI instrumented call using a function pointer

Bytes (opcodes)	x86 assembly code	Comment
FF 53 08	call [ebx+8]	; call a function pointer
is instrumented using prefetchnta destination IDs, to become:		
8B 43 08	mov eax, [ebx+8]	; load pointer into register
3E 81 78 04 78 56 34 12	cmp [eax+4], 12345678h	; compare opcodes at destination
75 13	jne error_label	; if not ID value, then fail
FF D0	call eax	; call function pointer
3E 0F 18 05 DD CC BB AA	prefetchnta [AABBCCDDh]	; label ID, used upon the return

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

Revisiting Assumptions

- UNQ: Unique IDs
 - Required to preserve CFG semantics
- NWC: Non-Writable Code
 - Otherwise attacker can overwrite CFI dynamic check
 - Not true if code dynamically loaded or generated
- NXD: Non-Executable Data
 - Otherwise attacker could cause the execution of data labeled with expected ID

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
 - Data-only attacks
 - Incorrect arguments to system calls
 - Substitution of file names

Lecture Outline

- CFI: Overview and Approach
- CFI: Implementation and Evaluation 
- Building on CFI
 - IRM, SFI, SMAC, Protected Shadow Call Stack
- Formal Study

Implementation

- How to get CFG and how to instrument
 - Use magic of MSR Vulcan and PDB files [Srivastava et al 2001]
- How to do checks on x86
 - Compare target address of each computed control-flow transfer to a set of allowed destination addresses – **inefficient**
 - Can implement NOPs using x86 prefetching

Evaluation

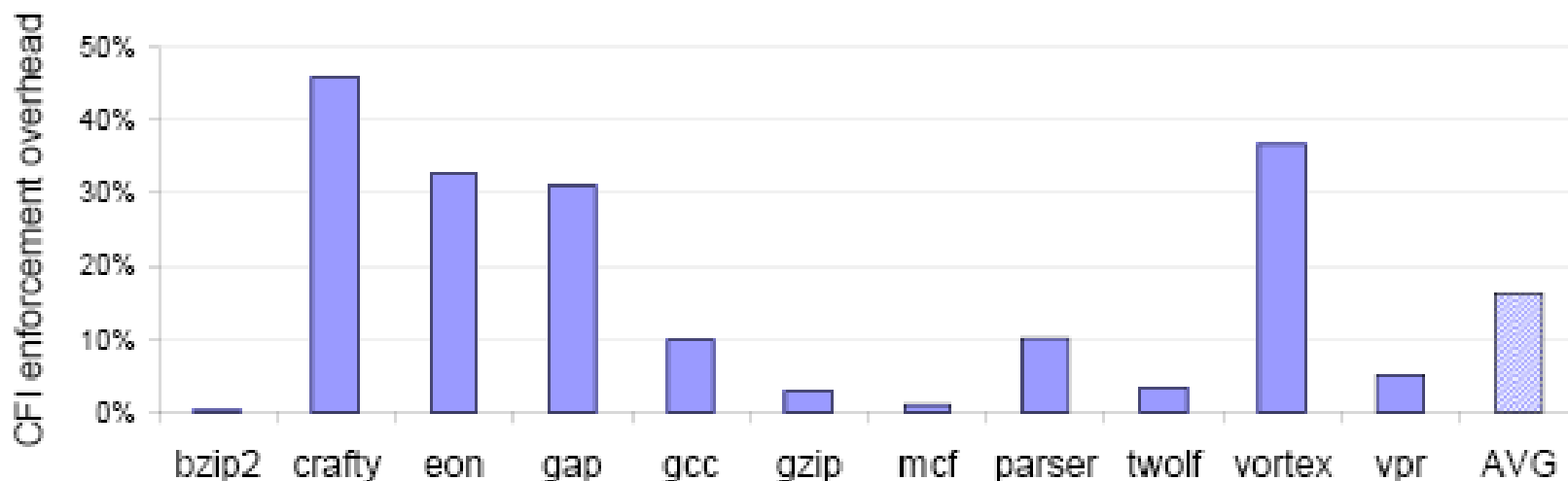



Fig. 6. Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

x86 Pentium 4, 1.8 GHz, 512MB RAM; average overhead: 16%; range: 0-45%

Evaluation (2)

- CFG construction + CFI instrumentation: ~10s
- Increase in binary size: ~8%
- Relative execution overhead:
 - crafty: CFI – 45%, program shepherding > 100%
 - gcc: CFI < 10%, program shepherding ~660%
- Security-related experiments
 - CFI protects against various specific attacks (read Section 4.3)

Lecture Outline

- CFI: Overview and Approach
- CFI: Implementation and Evaluation
- Building on CFI 
 - IRM, SFI, SMAC, Protected Shadow Call Stack
- Formal Study

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

- SMAC: Different access checks at different instructions in the program
 - Isolated data memory regions that are only accessible by specific pieces of program code => 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


CFI as a Foundation for Non-circumventable IRMs

- Inlined Reference Monitors (IRM) work correctly assuming:
 - Inserted dynamic checks cannot be circumvented by changing control flow – **enforced using CFI**
 - IRM state cannot be modified by attacker – **enforced by SMAC**

Protected Shadow Call Stack

- Suppose a call from A goes to C, and a call from B goes to either C or D
 - CFI will use the same tag for C and D, but this allows an “invalid” call from A to D
 - One solution: duplicate code (or even inline everything)
 - Can also use multiple ID 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: use a **shadow call stack** protected using (a) SMAC or (b) isolated x86 memory segment protected by CFI

Lecture Outline

- CFI: Overview and Approach
- CFI: Implementation and Evaluation
- Building on CFI
 - IRM, SFI, SMAC, Protected Shadow Call Stack
- Formal Study 

Security Proof Outline

- Define machine code semantics
- Model a powerful attacker
- Define instrumentation algorithm
- Prove security theorem

Formal study uses a simple programming language, not the x86 ISA

Machine Model

Execution State:

- Mc (code memory): maps addresses to words
- Md (data memory): maps addresses to words
- R (registers): maps register nos. to words
- pc (program counter): a word

Operational Semantics

- For each instruction, operational semantics defines how the instruction affects state (memory, pc, registers)
- Example: Semantics of *add rd, rs, rt*

If $M_c(pc)$ contains the encoding of *add rd, rs, rt*, and the current state has code memory M_c , data memory M_d , program counter value pc , and register values R , and if $pc + 1$ is within the domain of M_c , then in the next state the code memory and data memory are still M_c and M_d , respectively, pc is incremented, and R is updated so that it maps rd to $R(rs) + R(rt)$.

Semantics with SMAC: $pc + 1$ is not assumed to be in the domain of M_c

Operational Semantics (normal)

□ Semantics of *add rd, rs, rt*

$$(M_c | M_d, R, pc) \rightarrow_n (M_c | M_d, R\{r_d \mapsto R(r_s) + R(r_t)\}, pc + 1)$$

when $M_c(pc)$ holds *add rd, rs, rt* and $pc + 1$ is in the domain of M_c

\rightarrow_n : Binary relation on states that expresses normal execution steps

Operational Semantics (attacker)

- Idea: Attacker may arbitrarily modify data memory and most registers at any time
- Formally, attacker transition captured by binary relation on states \rightarrow_a

$$(M_c | M_d, R, pc) \rightarrow_a (M_c | M_d', R, pc)$$

Transitions \rightarrow are either normal transitions \rightarrow_n or attacker transitions \rightarrow_a

Instrumentation Algorithm

- $I(M_c)$: Code memory M_c is well-instrumented according to the CFI-criteria
- Example:
 - Every computed jump instruction is preceded by a particular sequence of instructions, which depends on a given CFG

Precise definition of CFG and instrumentation algorithm in paper

CFI Security Theorem

Let S_0 be a state with code memory M_c such that $I(M_c)$ and $pc = 0$, and let S_1, \dots, S_n be states such that $S_0 \rightarrow S_1 \rightarrow \dots \rightarrow S_n$. Then, for all $i \in 0..(n-1)$, either $S_i \rightarrow_a S_{i+1}$ or the pc at S_{i+1} is one of the allowed successors for the pc at S_i according to the given CFG.

- Requires *definition* of transition relation \rightarrow , instrumentation algorithm $I(M_c)$, and CFG.
- Property holds in the presence of *attacker steps*
- Proof is by *induction* on execution sequences

Sources

- Abadi et al., Control-Flow Integrity: Principles, Implementations, and Applications, TISSEC (To appear).
- Jay Ligatti provided a number of slides used in this lecture

Questions?