Control Flow Integrity

Outline

- CFI – Control Flow Integrity at Source Code Level
- BinCFI – CFI for Binary Executables
- BinCC – Binary Code Continent
- vfGuard – CFI Policy for Virtual Function Calls

M. Abadi, M. Budiu, U. Erlingsson, J. Ligatti

Control-Flow Integrity:
Principles, Implementations, and Applications

(CCS 2005)

CFI: Control-Flow Integrity

[Abadi et al.]

- 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

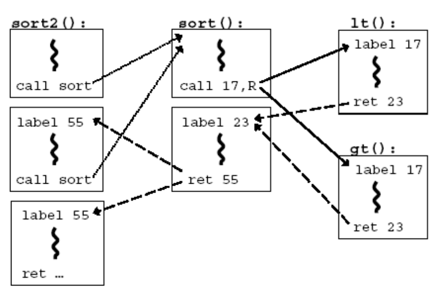
slide 5

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 );
}
```



slide 6

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

slide 7

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

slide 8

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

slide 9

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

slide 10

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

slide 11

Performance Overhead

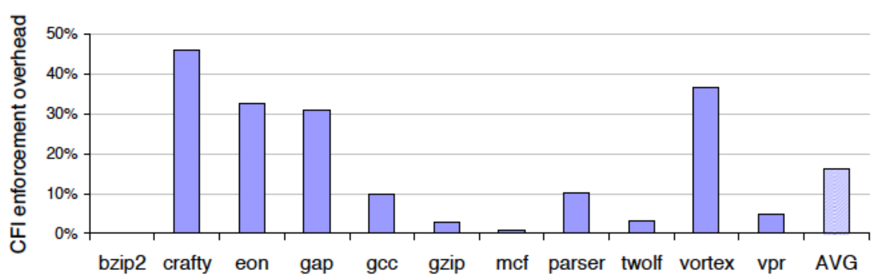


Figure 4: Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

Performance Overhead (2)

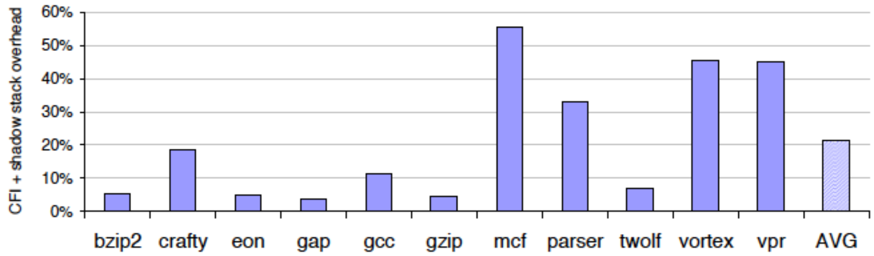


Figure 8: Enforcement overhead for CFI with a protected shadow call stack on SPEC2000 benchmarks.

Control-Flow Integrity For COTS Binaries

Mingwei Zhang and R. Sekar
Stony Brook University
USENIX Security 2013

Work supported in part by grants from AFOSR, NSF and ONR

Motivation for this work

- Many previous works closely related to CFI
  - CFI [Abadi et al 05, Abadi et al 2009, Zhang et al 2013]
  - Instruction bundling [MaCamant et al 2008, Yee et al 2009]
  - Indexed Hooks [2011], Control-flow locking [Bletsch et al 2011]
  - MoCFI [Davi et al 2012], Reins [Wartell et al 2012]…
- Require compiler support, or binaries that contain relocation, symbol, or debug info
- Do not provide complete protection
  - Binary code, libraries, loader.

Key Challenges

- Disassembly and Static analysis of COTS binaries
- Robust static binary instrumentation
  - Without breaking low-level code
  - Transparency for position-independent code, C++ exceptions, etc.
- Modular instrumentation
  - Applied to executables and libraries
  - Enables sharing libraries across multiple processes
- Assess compatibility/strength tradeoff

Disassembly Errors

- Disassembly of non-code
  - Tolerate these errors by leaving original code in place
- Incorrect disassembly of legitimate code
  - Instruction decoding errors (not a real challenge)
  - *Instruction boundary errors*
  - Failure to disassemble (we avoid this)

Disassembly Algorithm

1 Linear disassembly

2 Error detection

- invalid opcode
- direct jump/call outside module address
- direct control into insn

3 Error correction

- Identify “gap:” data/padding disassembled as code
  - Scan backward to preceding unconditional jump
  - Scan forward to next direct or indirect target
    - *Indirect targets obtained from static analysis*

4 Mark “gap,” repeat until no more errors

Static Analysis

Code pointers are needed:

- to correct disassembly errors
- to constrain indirect control flow (ICF) targets

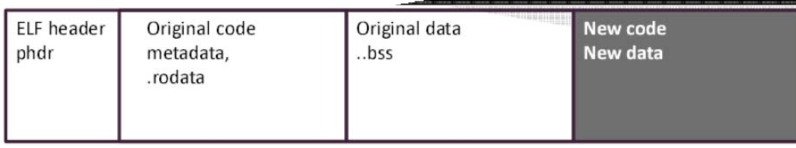
We classify code pointers into categories:

- Code Pointer Constants (CK)
- Computed Code Pointers (CC)
- Exception handlers (EH)
- Exported symbols (ES)
- Return addresses (RA)

Static Analysis

- Code pointer constaints
  - Scan for constants:
    - At any byte offset within code and data segments
    - Fall within the current module
    - Point to a valid instruction boundary
- Computed code pointers
  - Does not support arbitrary arithmetic, but targets jump tables
  - Use static analysis of code within a fixed-size window proceeding indirect jump

Instrumented Module



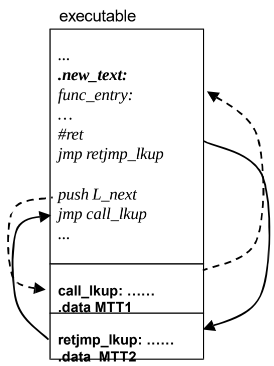
- Translating function pointers
  - Appear as constants in code, but can't statically translate
  - Solution: Runtime address translation
- Full transparency: all code pointers, incl. dynamically generated ones, target original code
  - Important for supporting unusual uses of code pointers
    - To compute data addresses (PIC-code, data embedded in code)
    - C++ exception handling

Static Instrumentation for CFI

- Goal: constrain branch targets to those determined by static analysis
  - Direct branches: nothing to be done
  - Indirect branches: check against a table of (statically computed) valid targets
- Key observation
  - CFI enforcement can be combined with address translation

Modularity

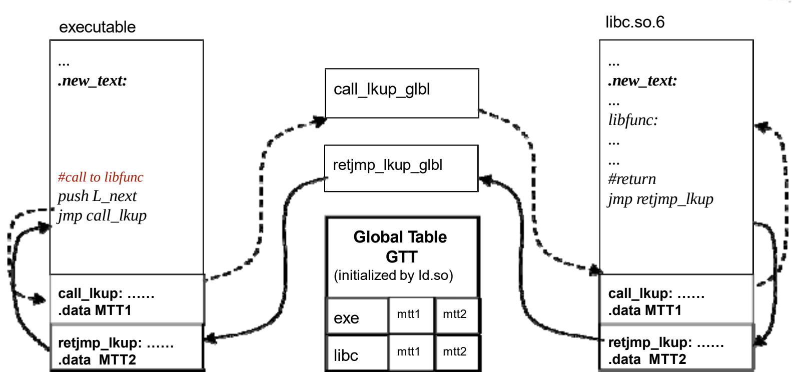
Intra-module control transfer: MTT



What if the target is out side of the module ?

Modularity

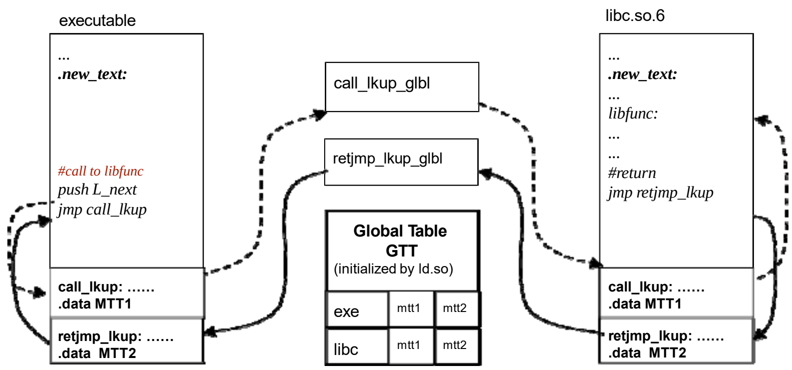
Inter-module control transfer: GTT



update of GTT is done in ld.so

Modularity

Code injection: null GTT entry



GTT only maps code !

Basic version of CFI

- **return: target next of call**
- **call/jmp: target any function whose address is taken**
  - Obtainable from relocation info ("reloc-CFI")
  - matches implementation described in [Abadi et al 2005]

- How to cope with missing relocation info?
  - *Use static analysis to over-approximate function addresses taken*
- "Strict-CFI"

CFI Real-World Exceptions

- special returns
  - a. *as indirect jumps (lazy binding in ld.so)*
  - b. *going to function entries (setcontext(2))*
  - c. *not going just after call (C++ exception)*
- calls used to get PC address
- jump as a replacement of return

Measuring "Protection Strength"

- Average Indirect target Reduction (AIR)
  - a. *T : number of possible targets of jth ICF branch*
  - b. *S: all possible target addresses (size of binary)*

$$\frac{1}{n}\sum_{j=1}^{n}\left(1-\frac{|T_j|}{S}\right)$$

- AIR is a general metric that can be applied to other control-flow containment approaches

Coarser versions of CFI

bundle-CFI:

- all ICF targets aligned on 2 -byte boundary, n = 4 (PittSFIeld) or 5 (Native Client)

instr-CFI: the most basic CFI

- **all ICFTs target instruction boundaries**

AIR metric (single module)

Name	Reloc CFI	Strict CFI	**Bin CFI**	Bundle CFI	Instr CFI
perlbench	98.49%	98.44%	**97.89%**	95.41%	67.33%
bzip2	99.55%	99.49%	**99.37%**	95.65%	78.59%
gcc	98.73%	98.71%	**98.34%**	95.86%	80.63%
gobmk	99.40%	99.40%	**99.20%**	97.75%	89.08%
……	……	……	**……**	……	……
average	**99.13%**	**99.08%**	**98.86%**	**96.04%**	**79.27%**

- **Loss due to use of static analysis is negligible**
- **Loss due to binCFI relaxation is very small**

Evaluation

Disassembly testing
Real world program testing
Gadget elimination

Disassembly Testing

Module	Package	Size	Instruction#	errors
libxul.so	firefox-5.0	26M	4.3M	0
gimp-console-2.6	gimp-2.6.5	7.7M	385K	0
libc.so	glibc-2.13	8.1M	301K	0
libnss3.so	firefox-5.0	4.1M	235K	0
……	……	……	……	……
Total		58M	5.84M	0

"diff" compiler generated assembly and our disassembly

Real world program testing

Application Name	Experiment
firefox 5 (no JIT)	open web pages
acroread9	open 20 pdf files; scroll;print;zoom in/out
gimp-2.6	load jpg picture, crop, blur, sharpen, etc.
Wireshark v1.6.2	capture packets on LAN for 20 minutes
lyx v2.0.0	open a large report; edit; convert to pdf/dvi/ps
mplayer 4.6.1	play an mp3 file
…...	…..........
Total:	12 real world programs

Gadget Elimination