Control Flow Integrity for COTS Binaries

Mingwei Zhang and R. Sekar
Stony Brook University
--
Summarized by Navid Emamdoost
University of Minnesota

Outline

- Background
  - Control Flow attacks
  - Control Flow Integrity
- Control Flow Integrity for COTS Binaries

Control Flow

- The order of instruction execution
- A subset of possible paths are intended by program
- An attacker can change this order due to
  - Programming mistakes
  - Insufficient security primitives provided by PL
  - Intrinsic complexity of architecture

Control Flow attacks

- Code injection
  - Overflow a buffer on system stack
  - Overwrite the return address
  - Divert control to injected code

Control Flow attacks

- Return to Libc
  - Overflow a buffer on system stack
  - Overwrite the return address
  - Divert control to an existing module
    - system(/bin/sh)

Control Flow attacks

- Return Oriented Programming (ROP)
  - Overflow a buffer on system stack
  - Overwrite the return address
  - Divert control to start of gadget
    - inc eax; ret;
    - pop eax; ret;

Control Flow Integrity

- Protect program's control flow integrity
  - Resist deviation from CFG
- Identify legal control transfer targets
- Prevent transfers to other targets
- Restrict program execution to the set of intended paths

Control Flow Integrity

- By Abadi et. al presented at 2005
- Computed control transfers are instrumented

Source

```
Opcode bytes          Instructions
FF E1                 jmp  ecx                ; computed jump
```

Destination

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

can be instrumented as (a):

```
81 39 78 56 34 12     cmp  [ecx], 12345678h   ; comp ID & dst
75 13                 jne  error_label        ; if != fail
8D 49 04              lea  ecx, [ecx+4]       ; skip ID at dst
FF E1                 jmp  ecx                ; jump to dst
```

```
78 56 34 12           ; data 12345678h        ; ID
8B 44 24 04           mov  eax, [esp+4]       ; dst
...
```

CFI

- Unique IDs: the bit patterns chosen as IDs must not be present anywhere in the code memory except in IDs and ID-checks
- Non-Writable Code: It must not be possible for the program to modify code memory at runtime
- Non-Executable Data: It must not be possible for the program to execute data as if it were code
- One ID value for the start of functions and another ID value for valid destinations for function returns

CFI

- Is not vulnerable to information leakage attacks, unlike
  - Stack canary
  - ASLR
- Protect against existing code reuse
  - Return-to-libc
  - ROP

Control Flow Integrity for COTS Binaries

- Goal:
  - Enforce CFI on COTS binaries
    - There is no source-code
    - No assembly-level information
    - No relocation information (unlike ASLR on windows)
    - Like shared libraries
    - Operate with less information available

Control Flow Integrity for COTS Binaries

- Steps
  - Disassemble
    - Correctly identify instructions
  - ICF analysis
    - Provide missing information (instead of using relocation info)
  - Instrument the binary
    - Enforce CFI

Disassembly

- Linear
  - Start from the first instruction of the segment
  - Assume nest instruction starts from the end of previous one
  - Problem: gaps
    - Data
    - Instruction alignment

Disassembly

- Recursive
  - Depth-first approach
  - A set of entry points
  - Add target of each direct CF transfer to the set of EP
  - Continue linearly up to an unconditional CF transfer
  - Problem: can not indentify codes reachable via ICF
    - Available from relocation infromation

COTS Disassembly

- Combination of linear and recursive
- Use static analysis of ICF to identify gaps
- Steps:
  - Linearly disassemble entire binary
  - Check for erroneous instructions
    - Invalid opcode
    - Direct CF transfer to outside of module
    - Direct CF transfer to the middle of another instruction

COTS Disassembly (cont'd)

- On an erroneous instruction
  - Move backward to reach a direct CF transfer
    - Mark as gap start
  - From ICF analysis find the first target after erroneous instruction
    - Mark as gap end
  - Repeat disassembly by avoiding gaps

ICF analysis

- Code pointer constants (CK)
  - consists of code addresses that are computed at compile-time.
- Computed code addresses (CC)
  - include code addresses that are computed at runtime.
- Exception handling addresses (EH)
  - include code addresses that are used to handle exceptions.
- Exported symbol addresses (ES)
  - include export function addresses.
- Return addresses (RA)
  - include the code addresses next of a call.

Code pointer constants (CK)

- In general, there is no way to distinguish a code pointer from other types of constants in code
- Every constant having properties
  - Be within the rage of code addresses
    - For shared libraries consider it as offset
    - Because there is no knowledge about base address at compile time
  - Is consistent with instruction boundaries

Computed code addresses (CC)

- Any arithmetic computation on pointers are possible in binary
- But they observed pointer arithmetic occurs just in jump tables
  - Switch case
- Properties of jump tables
  - Intra-function
  - Simple form: *(CE1+ *Ind*)+CE2
  - Within fixed sized window of instructions
    - 50 instructions

Computed code addresses (CC)

- Determine function boundaries
  - Exported functions
- Identify indirect jump and move backward to find the expression
  - CE1 and CE2 are constants
- Enumerate possible values of *Ind*
  - for every possible value if the result falls within the current region

Other code addresses

- Exception handling addresses (EH)
  - From ELF headers
- Exported symbol addresses (ES)
  - From ELF headers
- Return addresses (RA)
  - The address of instruction after the call
    - Computable after disassembly

CFI classes

- reloc-CFI
  - Types of ICF
    - Indirect Call
    - Indirect Jump
    - Return Address
- strict-CFI
  - Same as reloc-CFI
  - But uses static analysis instead of relocation info
  - Extensions for EH and Context switch
- bin-CFI
  - Has a new type of ICF: Program Linkage Table

bin-CFI

	Returns (RET), Indirect Jumps (IJ)	PLT targets, Indirect Calls (IC)
Return addresses (RA)	Y	
Exception handling addresses (EH)	Y	
Exported symbol addresses (ES)		Y
Code pointer constants (CK)	Y	Y
Computed code addresses (CC)	Y	Y

Figure 2: Bin-CFI Property Definition

CFI Instrumentation

- After instrumenting the binary, new object file is generated
- The new object file is injected into ELF file
- Prepare new segment for execution
- Update Entry point
- Mark original code segments as un-executable

CFI Instrumentation

- New code is in different segment
  - Function pointers are invalid
- Keep a table for address translation

  <original address, new address>
- For each valid ICF target
- addr_trans: a trampoline code performing translation by a hash table
- If target is within current module
  - lookup the hash
  - If no entry found, an error is sent
- If not, use a global translation table loaded by ld.so

CFI Instrumentation

- Signals
  - Intercept *signal* and *sigaction* system calls
  - Store the handlers address
  - Update system calls arguments to point to a wrapper function
  - The wrapper performes redirection to instrumented code

Evaluation

- Disassembely

Module	Package	Size	# of Ins-tructions	# of 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
libmozsqlite3.so	firefox-5.0	1.8M	128K	0
libfreebl3.so	firefox-5.0	876K	66K	0
libsoftokn3.so	firefox-5.0	756K	50K	0
libnspr4.so	firefox-5.0	776K	41K	0
libssl3.so	firefox-5.0	864K	40K	0
libm.so	glibc-2.13	620K	35K	0
libnssdbm3.so	firefox-5.0	570K	34K	0
libsmime3.so	firefox-5.0	746K	30K	0
ld.so	glibc-2.13	694K	28K	0
gimpressionist	gimp-2.6.5	403K	21K	0
script-fu	gimp-2.6.5	410K	21K	0
libnssckbi.so	firefox-5.0	733K	19K	0
libtestcrasher.so	firefox-5.0	676K	17K	0
gfig	gimp-2.6.5	442K	17K	0
libpthread.so	glibc-2.13	666K	15K	0
libnsl.so	glibc-2.13	448K	15K	0
map-object	gimp-2.6.5	257K	15K	0
libresolv.so	glibc-2.13	275K	13K	0
libnssutil3.so	firefox-5.0	311K	13K	0
Total		58M	5.84M	0

Figure 6: Disassembly Correctness

Evaluation

- CFI effectiveness:
  - Average Indirect target Reduction (AIR)
  - For n ICF transfers, and S initial targets for them

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

Evaluation

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%
mcf	99.47%	99.37%	**99.25%**	95.91%	79.35%
gobmk	99.40%	99.40%	**99.20%**	97.75%	89.08%
hmmer	98.90%	98.87%	**98.61%**	95.85%	79.01%
sjeng	99.32%	99.30%	**99.10%**	96.22%	83.18%
libquantum	99.14%	99.07%	**98.89%**	95.96%	76.53%
h264ref	99.64%	99.60%	**99.52%**	96.25%	80.71%
omnetpp	98.26%	98.08%	**97.68%**	95.72%	82.03%
astar	99.18%	99.13%	**98.95%**	96.02%	78.00%
milc	98.89%	98.86%	**98.65%**	96.03%	79.74%
namd	99.65%	99.64%	**99.59%**	95.81%	76.37%
soplex	99.19%	99.10%	**98.86%**	95.50%	77.37%
povray	99.01%	98.99%	**98.67%**	95.87%	78.03%
lbm	99.60%	99.50%	**99.46%**	96.79%	80.92%
sphinx3	98.83%	98.80%	**98.64%**	96.06%	80.75%
average	*99.13%*	*99.08%*	**98.86%**	*96.04%*	*79.27%*

Figure 8: AIR metrics for SPEC CPU 2006.

Evaluation

- Gadget elimination

Name	Reloc CFI	Strict CFI	Bin CFI	Instr CFI
perlbench	96.62%	96.24%	93.23%	58.65%
bzip2	97.78%	95.56%	93.33%	44.44%
gcc	97.69%	97.69%	91.42%	66.67%
mcf	95.45%	90.91%	90.91%	36.36%
gobmk	98.84%	98.27%	97.69%	70.52%
hmmer	97.00%	96.00%	96.00%	58.00%
sjeng	92.75%	92.75%	91.30%	47.83%
libquantum	93.18%	90.91%	86.36%	40.91%
h264ref	98.26%	97.39%	96.52%	60.87%
omnetpp	97.12%	97.12%	93.42%	74.07%
astar	95.35%	93.02%	93.02%	46.51%
milc	95.77%	94.37%	90.14%	57.75%
namd	94.87%	92.31%	92.31%	53.85%
soplex	94.64%	93.75%	93.75%	54.46%
povray	96.75%	96.75%	95.45%	61.69%
lbm	94.12%	88.24%	88.24%	23.53%
sphinx3	95.00%	93.75%	92.50%	52.50%
average	95.95%	94.41%	92.68%	53.45%

Figure 10: Gadget elimination in different CFI implementation

Evaluation

- Performance overhead

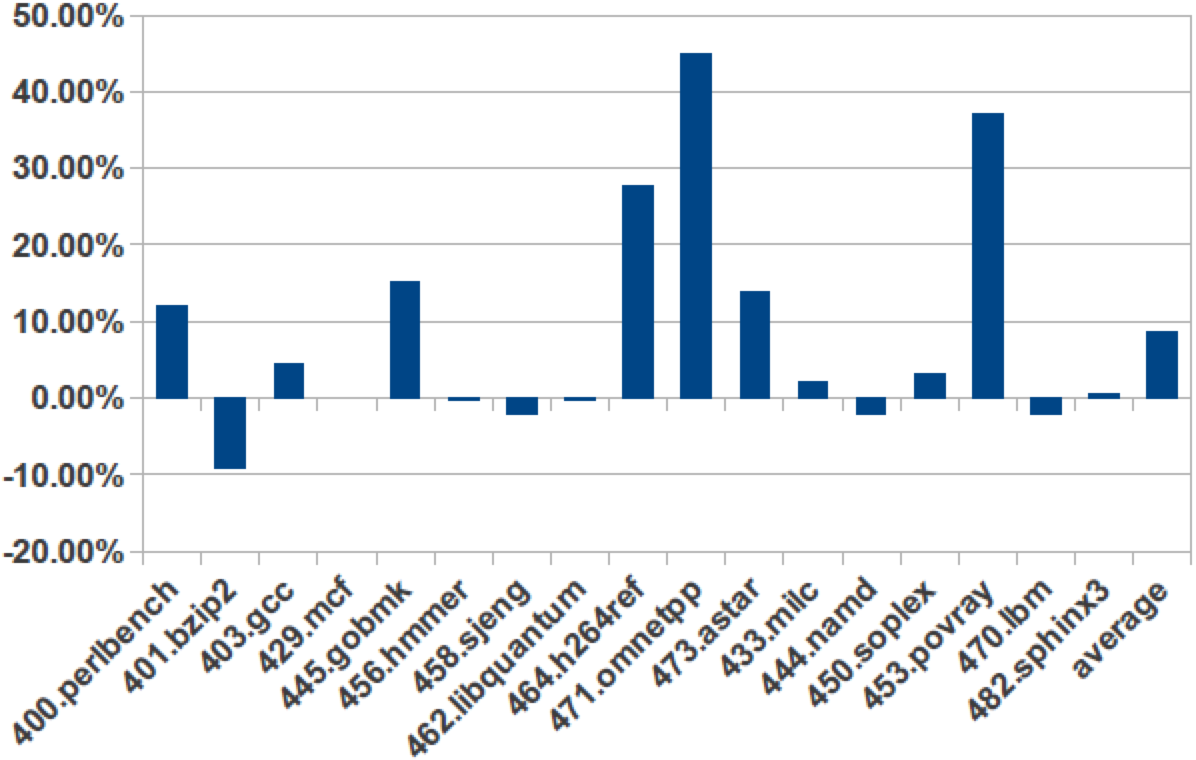

Figure 11: SPEC CPU2006 Benchmark Performance

Evaluation

- Space overhead:
  - 139% increase in file size
  - 2.2% for resident memory use

Thank You