Control Flow Integrity & Software Fault Isolation

David Brumley
Carnegie Mellon University
Our story so far...

未经授权的控制信息篡改

Control Flow Hijacks

- Attack
  - Buffer Overflows
  - Format String Vulnerabilities
    - More Buffer Overflows
    - Mem Read
    - Mem Write
  - DEP/NX
  - ret2libc
  - Return-Oriented Programming

- Defense
  - Computation
  - Function Pointer Subterfuge
  - Forcing
  - Randomized Code

http://propercourse.blogspot.com/2010/05/i-believe-in-duct-tape.html
Adversary Model Matters!

Cowan et al., USENIX Security 1998
StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks

“Programs compiled with StackGuard are safe from buffer overflow attack, regardless of the software engineering quality of the program.”

What if the adversary is more powerful? How powerful is powerful enough?
Reference Monitors
Subject

Op request

Op response

Object

People
Processes
Computer Operations

Files
Sockets
Computer Operations
Principles:
1. Complete Mediation: The reference monitor must always be invoked
2. Tamper-proof: The reference monitor cannot be changed by unauthorized subjects or objects
3. Verifiable: The reference monitor is small enough to thoroughly understand, test, and ultimately, verify.
Inlined Referenced Monitor

Today’s Example:
Inlining a control flow policy into a program
Control Flow Integrity

Assigned Reading:

*Control-Flow Integrity: Principles, Implementation and Applications*  
by Abadi, Budiu, Erlingsson, and Ligatti
Control Flow Integrity

• protects against powerful adversary
  – with full control over entire data memory

• widely-applicable
  – language-neutral; requires binary only

• provably-correct & trustworthy
  – formal semantics; small verifier

• efficient
  – hmm... 0-45% in experiments; average 16%
CFI Adversary Model

**CAN**
- Overwrite any data memory at any time
  - stack, heap, data segs
- Overwrite registers in current context

**CANNOT**
- Execute Data
  - NX takes care of that
- Modify Code
  - text seg usually read-only
- Write to %ip
  - true in x86
- Overwrite registers in other contexts
  - kernel will restore regs
CFI Overview

**Invariant:** Execution must follow a path in a control flow graph (CFG) created ahead of run time.

**Method:**
- build CFG statically, e.g., at compile time
- instrument (rewrite) binary, e.g., at install time
  - add IDs and ID checks; maintain ID uniqueness
- verify CFI instrumentation at load time
  - direct jump targets, presence of IDs and ID checks, ID uniqueness
- perform ID checks at run time
  - indirect jumps have matching IDs
Control Flow Graphs
**Defn Basic Block:** A consecutive sequence of instructions / control is “straight” (no jump targets except at the beginning, no jumps except at the end)

instructions in the sequence

1. \(x = y + z\)
2. \(z = t + i\)
3. \(x = y + z\)
4. \(z = t + i\)
5. `jmp 1`
6. `jmp 3`

3 static basic blocks

1. \(x = y + z\)
2. \(z = t + i\)
3. \(x = y + z\)
4. \(z = t + i\)
5. `jmp 1`

1 dynamic basic block
CFG Definition

A static *Control Flow Graph* is a graph where
– each vertex $v_i$ is a basic block, and
– there is an edge $(v_i, v_j)$ if there *may* be a transfer of control from block $v_i$ to block $v_j$.

Historically, the scope of a “CFG” is limited to a function or procedure, i.e., *intra*-procedural.
Call Graph

- Nodes are functions. There is an edge \((v_i, v_j)\) if function \(v_i\) calls function \(v_j\).

```c
void orange()
{
1. red(1);
2. red(2);
3. green();
}

void red(int x)
{
    green();
    ...
}

void green()
{
    green();
    orange();
}
```
Super Graph

- Superimpose CFGs of all procedures over the call graph

```c
void orange() {
    1. red(1);
    2. red(2);
    3. green();
}

void red(int x) {
    ..
}

void green() {
    green();
    orange();
}
```

A context sensitive super-graph for orange lines 1 and 2.
Precision: Sensitive or Insensitive

The more precise the analysis, the more accurate it reflects the “real” program behavior.

– More precise = more time to compute
– More precise = more space
– Limited by soundness/completeness tradeoff

Common Terminology in any Static Analysis:

– **Context** sensitive vs. context insensitive
– **Flow** sensitive vs. flow insensitive
– **Path** sensitive vs. path insensitive
Soundness
If analysis says X is true, then X is true.

True Things

Things I say

Trivially Sound: Say nothing

Completedness
If X is true, then analysis says X is true.

True Things

Things I say

Trivially complete: Say everything

Sound and Complete: Say exactly the set of true things!
Context Sensitive

Whether different calling contexts are distinguished

```c
void yellow()  void red(int x)  void green()
{
    {   {   
1. red(1);  ..  green();
2. red(2);   }  yellow();
3. green();   }
}
```

Context sensitive distinguishes 2 different calls to red(-)
Context Sensitive Example

```
a = id(4);
b = id(5);
```

```
void id(int z)
{ return z; }
```

Context-Sensitive (color denotes matching call/ret)

Context sensitive can tell one call returns 4, the other 5

```
a = id(4);
b = id(5);
```

Context-Insensitive (note merging)

Context insensitive will say both calls return \{4,5\}
A *flow* sensitive analysis considers the order (flow) of statements

- Flow insensitive = usually linear-type algorithm
- Flow sensitive = usually at least quadratic (dataflow)

**Examples:**
- Type checking is flow insensitive since a variable has a single type regardless of the order of statements
- Detecting uninitialized variables requires flow sensitivity

\[
x = 4;
\]

\[
\ldots\ldots
\]

\[
x = 5;
\]
Flow Sensitive Example

1. x = 4;
   ....
n. x = 5;

Flow sensitive:
- x is the constant 4 at line 1,
- x is the constant 5 at line n

Flow insensitive:
x is not a constant
Path Sensitive

A path sensitive analysis maintains branch conditions along each *execution path*

– Requires extreme care to make scalable
– Subsumes flow sensitivity
Path Sensitive Example

1. if (x >= 0)
2.   y = x;
3. else
4.   y = -x;

path sensitive:
y >= 0 at line 2,
y > 0 at line 4

path insensitive:
y is not a constant
Precision

Even path sensitive analysis approximates behavior due to:

• loops/recursion
• unrealizable paths

1. if($a^n + b^n = c^n$ && $n>2$ && $a>0$ && $b>0$ && $c>0$)
2.   $x = 7$;
3. else
4.   $x = 8$;

Unrealizable path. $x$ will always be 8
Control Flow Integrity (Analysis)
CFI Overview

**Invariant:** Execution must follow a path in a control flow graph (CFG) created ahead of run time.

**Method:**

- build CFG statically, e.g., at compile time
- instrument (rewrite) binary, e.g., at install time
  - add IDs and ID checks; maintain ID uniqueness
- verify CFI instrumentation at load time
  - direct jump targets, presence of IDs and ID checks, ID uniqueness
- perform ID checks at run time
  - indirect jumps have matching IDs
Build CFG

Two possible return sites due to context insensitivity

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);
}
Instrument Binary

- Insert a unique number at each destination
- Two destinations are equivalent if CFG contains edges to each from the same source

```c
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 );
}
```

*predicated* call 17, R: transfer control to R only when R has label 17
*predicated* ret 23: transfer control to only label 23

```c
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 );
}
```
Verify CFI Instrumentation

• **Direct jump targets** (e.g. call 0x12345678)
  – are all targets valid according to CFG?

• **IDs**
  – is there an ID right after every entry point?
  – does any ID appear in the binary by accident?

• **ID Checks**
  – is there a check before every control transfer?
  – does each check respect the CFG?

*easy to implement correctly => trustworthy*
What about indirect jumps and ret?
ID Checks

Check dest label

Fig. 4. Our CFI implementation of a call through a function pointer.

Call destination IDs, to become:

- mov eax, [ebx+8] ; load pointer into register
- cmp [eax+4], AABBCCDdh ; 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

Check dest label
**Performance**

**Size:** increase 8% avg

**Time:** increase 0-45%; 16% avg

– I/O latency helps hide overhead

---

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

**CAN**
- Overwrite any data memory at any time
  - stack, heap, data segs
- Overwrite registers in current context

**CANNOT**
- Execute Data
  - NX takes care of that
- Modify Code
  - text seg usually read-only
- Write to %ip
  - true in x86
- Overwrite registers in other contexts
  - kernel will restore regs

Assumptions are often vulnerabilities!
Let’s check our assumptions!

• Non-executable Data
  – let’s inject code with desired ID...

• Non-writable Code
  – let’s overwrite the check instructions...
  – can be problematic for JIT compilers

• Context-Switching Preserves Registers
  – time-of-check vs. time-of-use
  – BONUS point: why don’t we use the RET instruction to return?
Time-of-Check vs. Time-of-Use

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

Fig. 4. Our CFI implementation of a call through a function pointer.

<table>
<thead>
<tr>
<th>Bytes (opcodes)</th>
<th>x86 assembly code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2 10 00</td>
<td>ret 10h</td>
<td>; return, and pop 16 extra bytes</td>
</tr>
</tbody>
</table>

is instrumented using `prefetchnta` destination IDs, to become:

8B 0C 24  mov ecx, [esp]
83 C4 14  add esp, 14h
3E 81 79 04 DD CC BB AA  cmp [ecx+4], AABBCCDDh
75 13  jne error_label
FF E1  jmp ecx

what if there is a context switch here?
Security Guarantees

Effective against attacks based on illegitimate control-flow transfer
  – buffer overflow, ret2libc, pointer subterfuge, etc.

Any check becomes non-circumventable.

Allow data-only attacks since they respect CFG!
  – incorrect usage (e.g. printf can still dump mem)
  – substitution of data (e.g. replace file names)
Software Fault Isolation

• SFI ensures that a module only accesses memory within its region by adding checks – e.g., a plugin can accesses only its own memory

\[
\text{if} (\text{module}_{\text{lower}} < x < \text{module}_{\text{upper}}) \\
\quad z = \text{load}[x];
\]

• CFI ensures inserted memory checks are executed
Inline Reference Monitors

- IRMs inline a security policy into binary to ensure security enforcement

- Any IRM can be supported by CFI + SMAC
  - **CFI:** IRM code cannot be circumvented
  - **SMAC:** IRM state cannot be tampered
Accuracy vs. Security

The accuracy of the CFG will reflect the level of enforcement of the security mechanism.

```c
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);
}
```

Indistinguishable sites, e.g., due to lack of context sensitivity will be merged.
Context Sensitivity Problems

Suppose A and B both call C.
• CFI uses same return label in A and B.

How to prevent C from returning to B when it was called from A?
• **Shadow Call Stack**
  – an protected memory region for call stack
  – each call/ret instrumented to update shadow
  – CFI ensures instrumented checks will be run
Proof of Security

Theorem (Informal):
Given state $S_0$ with
  • non-writeable, well-instrumented code mem $M_0$

Then for all runtime steps $S_i \rightarrow S_{i+1}$,
  • $S_{i+1}$ is one of the allowed successors in the CFG,
    or
  • $S_{i+1}$ is an error state

We can make these sorts of statements precise with operational semantics.
CFI Summary

Control Flow Integrity ensures that control flow follows a path in CFG

– Accuracy of CFG determines level of enforcement
– Can build other security policies on top of CFI
Software Fault Isolation

Optional Reading:
*Efficient Software-Based Fault Isolation*
by Wahbe, Lucco, Anderson, Graham
Isolation Mechanisms

• Hardware
  – Memory Protection (virtual address translation, x86 segmentation)

• Software
  – Sandboxing
  – Language-Based

• Hardware + Software
  – Virtual machines

Software Fault Isolation
≈
Memory Protection in Software
SFI Goals

• Confine faults inside distrusted extensions
  – codec shouldn’t compromise media player
  – device driver shouldn’t compromise kernel
  – plugin shouldn’t compromise web browser

• Allow for efficient cross-domain calls
  – numerous calls between media player and codec
  – numerous calls between device driver and kernel
Main Idea

Process Address Space

Module 2
*Fault Domain 2*

Module 1
*Fault Domain 1*

- Segment with id 2, e.g., with top bits 010
- Segment with id 1, e.g., with top bits 011
Scheme 1: Segment Matching

- **Check** every mem access for matching seg id
- assume dedicated registers segment register (sr) and data register (dr)
  - not available to the program (no big deal in Alpha)

![Diagram of Process Address Space with Module 1 and Module 2]

**precondition:**
- sr holds segment id 2
- dr = addr
- scratch = (dr >> 29)
- compare scratch, sr
- trap if not equal
- dst = [dr]
Safety

• Segment matching code must always be run to ensure safety.

• Dedicated registers must not be writeable by module.
Scheme 2: Sandboxing

- **Force** top bits to match seg id and continue
- No comparison is made

**Precondition:**
- sr holds segment id 2
- \( dr = (\text{addr} \& \text{mask}) \)
- \( dr = (dr | sr) \)
- \( \text{dst} = [dr] \)

**Process Address Space**

- Module 1
- Module 2

Enforce top bits in dr are sr
Segment Matching vs. Sandboxing

Segment Matching
• more instructions
• can pinpoint exact point of fault where segment id doesn’t match

Sandboxing
• fewer instructions
• just ensures memory access stays in region (crash is ok)
Communication between domains

RPC
Native Client

Optional Reading:

Native Client: A Sandbox for Portable, Untrusted x86 Native Code
by Yee et al.
NaCL: A Modern Day Example

- Two sandboxes:
  - an inner sandbox to mediate x86-specific runtime details (using what technique?)
  - an outer sandbox mediates system calls (Using what technique?)

Browser

HTML
JavaScript

Quake

NPAPI or RPC

NaCl runtime
Security Goal

• Achieve comparable safety to accepted systems such as JavaScript.
  – Input: arbitrary code and data
    • support multi-threading, inter-module communication
  – NaCL checks that code conforms to security rules, else refuses to run.
Obligations

C1 Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.
C2 The binary is statically linked at a start address of zero, with the first byte of text at 64K.
C3 All indirect control transfers use a `nacljmp` pseudo-instruction (defined below).
C4 The binary is padded up to the nearest page with at least one `hlt` instruction (0xf4).
C5 The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.
C6 All `valid` instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.
C7 All direct control transfers target valid instructions.

What do these obligations guarantee?
Guarantees

- Data integrity: no loads or stores outside of sandbox
  - Think back to SFI paper
- Reliable disassembly
- No unsafe instructions
- Control flow integrity
NACL Module At Runtime

4 KB RW protected for NULL ptrs

60 KB for trampoline/springboard

Untrusted Code

Transfer from trusted to untrusted code, and vice-versa
Performance - Quake

<table>
<thead>
<tr>
<th>Run #</th>
<th>Native Client</th>
<th>Linux Executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>143.2</td>
<td>142.9</td>
</tr>
<tr>
<td>2</td>
<td>143.6</td>
<td>143.4</td>
</tr>
<tr>
<td>3</td>
<td>144.2</td>
<td>143.5</td>
</tr>
<tr>
<td>Average</td>
<td>143.7</td>
<td>143.3</td>
</tr>
</tbody>
</table>

Table 8: Quake performance comparison. Numbers are in frames per second.
Questions?
END
TOC/TOU

• Time of Check/Time of Use bugs are a type of race condition

```bash
$ open("myfile");
monitor does complex check

monitor OK's
OS carries out action

$ ln -s myfile /etc/passwd
monitor OK's
Action performed
```
Software Mandatory Access Control

Fine-grained SFI: SMAC can have different access checks at different instructions.

- isolated code region => no need for NX data

```assembly
call eax ; call a function pointer (destination address)

; with CFI, and SMAC discharging the NXD requirement, can become:
and eax, 40FFFFFFFh ; 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 [AABBCDDh] ; label ID, used upon the return
```
Context Sensitivity Problems

Suppose A calls C and B calls C, D.

• CFI uses same call label for C and D due to B.

How to prevent A from calling D?
• duplicate C into $C_A$ and $C_B$, or
• use more complicated labeling mechanism
Optimizations

Guard Zones
• unmapped pages around segment to avoid checking offsets

Lazier SP Check
• check SP only before jumps

Figure 3: A segment with guard zones. The size of the guard zones covers the range of possible immediate offsets in register-plus-offset addressing modes.
## Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DEC-MIPS</th>
<th>DEC-ALPHA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fault Isolation Overhead</td>
<td>Protection Overhead</td>
</tr>
<tr>
<td>052.alvinn</td>
<td>FP 1.4%</td>
<td>FP 33.4%</td>
</tr>
<tr>
<td>bps</td>
<td>FP 5.6%</td>
<td>FP 5.6%</td>
</tr>
<tr>
<td>cholesky</td>
<td>FP 0.0%</td>
<td>FP 5.6%</td>
</tr>
<tr>
<td>026.compress</td>
<td>INT 3.3%</td>
<td>INT 3.3%</td>
</tr>
<tr>
<td>056.ear</td>
<td>FP -1.2%</td>
<td>FP 19.1%</td>
</tr>
<tr>
<td>023.eqntott</td>
<td>INT 2.9%</td>
<td>INT 34.4%</td>
</tr>
<tr>
<td>008.espresso</td>
<td>INT 12.4%</td>
<td>INT 27.0%</td>
</tr>
<tr>
<td>001.gcc1.35</td>
<td>INT 3.1%</td>
<td>INT 18.7%</td>
</tr>
<tr>
<td>022.li</td>
<td>INT 5.1%</td>
<td>INT 23.4%</td>
</tr>
<tr>
<td>locus</td>
<td>INT 8.7%</td>
<td>INT 30.4%</td>
</tr>
<tr>
<td>mp3d</td>
<td>FP 10.7%</td>
<td>FP 19.5%</td>
</tr>
<tr>
<td>psgrind</td>
<td>INT 10.4%</td>
<td>INT 27.0%</td>
</tr>
<tr>
<td>qcd</td>
<td>FP 0.5%</td>
<td>FP 11.2%</td>
</tr>
<tr>
<td>072.sc</td>
<td>INT 5.6%</td>
<td>INT 10.5%</td>
</tr>
<tr>
<td>tracker</td>
<td>INT -0.8%</td>
<td>INT 7.4%</td>
</tr>
<tr>
<td>water</td>
<td>FP 0.7%</td>
<td>FP 0.7%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>4.3%</strong></td>
<td><strong>21.8%</strong></td>
</tr>
</tbody>
</table>

- store and jump checked
- load, store and jump checked
Is it counter-intuitive?

• Slow down “common” case of intra-domain control transfer in order to speed up inter-domain transfer
  – Check every load, store, jump within a domain

• Faster in practice than hardware when inter-domain calls are frequent
  – Context switches are expensive
  – Each cross-module call requires a context switch
Differences between NaCL SFI and Wahbe SFI

• NaCL uses segments for data to ensure loads/stores are within a module
  – Do not need sandboxing overhead for these instructions

• Others?

• After reading Wahbe et al, how would you implement inter-module communication efficiently?
# Performance – Micro Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>static</th>
<th>aligned</th>
<th>NaCl</th>
<th>increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>200</td>
<td>203</td>
<td>203</td>
<td>1.5%</td>
</tr>
<tr>
<td>art</td>
<td>46.3</td>
<td>48.7</td>
<td>47.2</td>
<td>1.9%</td>
</tr>
<tr>
<td>bzip2</td>
<td>103</td>
<td>104</td>
<td>104</td>
<td>1.9%</td>
</tr>
<tr>
<td>crafty</td>
<td>113</td>
<td>124</td>
<td>127</td>
<td>12%</td>
</tr>
<tr>
<td>eon</td>
<td>79.2</td>
<td>76.9</td>
<td>82.6</td>
<td>4.3%</td>
</tr>
<tr>
<td>equake</td>
<td>62.3</td>
<td>62.9</td>
<td>62.5</td>
<td>0.3%</td>
</tr>
<tr>
<td>gap</td>
<td>63.9</td>
<td>64.0</td>
<td>65.4</td>
<td>2.4%</td>
</tr>
<tr>
<td>gcc</td>
<td>52.3</td>
<td>54.7</td>
<td>57.0</td>
<td>9.0%</td>
</tr>
<tr>
<td>gzip</td>
<td>149</td>
<td>149</td>
<td>148</td>
<td>-0.7%</td>
</tr>
<tr>
<td>mcf</td>
<td>65.7</td>
<td>65.7</td>
<td>66.2</td>
<td>0.8%</td>
</tr>
<tr>
<td>mesa</td>
<td>87.4</td>
<td>89.8</td>
<td>92.5</td>
<td>5.8%</td>
</tr>
<tr>
<td>parser</td>
<td>126</td>
<td>128</td>
<td>128</td>
<td>1.6%</td>
</tr>
<tr>
<td>perlbmk</td>
<td>94.0</td>
<td>99.3</td>
<td>106</td>
<td>13%</td>
</tr>
<tr>
<td>twolf</td>
<td>154</td>
<td>163</td>
<td>165</td>
<td>7.1%</td>
</tr>
<tr>
<td>vortex</td>
<td>112</td>
<td>116</td>
<td>124</td>
<td>11%</td>
</tr>
<tr>
<td>vpr</td>
<td>90.7</td>
<td>88.4</td>
<td>89.6</td>
<td>-1.2%</td>
</tr>
</tbody>
</table>

Table 4: SPEC2000 performance. Execution time is in seconds. All binaries are statically linked.