CE 874 - Secure Software Systems

Control Flow Integrity

Mehdi Kharrazi
Department of Computer Engineering
Sharif University of Technology

Acknowledgments: Some of the slides are fully or partially obtained from other sources. A reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.
Run-Time protection/enforcement

- In many instances we only have access to the binary
- How do we analyze the binary for vulnerabilities?
- How do we protect the binary from exploitation?
- This would be our topic for the next few lectures
Complete Mediation: The reference monitor must always be invoked

Tamper-proof: The reference monitor cannot be changed by unauthorized subjects or objects

Verifiable: The reference monitor is small enough to thoroughly understand, test, and ultimately, verify.
Today’s Example:
Inlining a control flow policy into a program

Subject

Reference Monitor

Policy

Op request

Op response

Object

[Brumley’15]
Control-Flow Integrity: Principles, Implementations, and Applications
Martin Abadi, Mihai Budiu, U´lfar Erlingsson, Jay Ligatti, CCS 2005
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

<table>
<thead>
<tr>
<th>Can</th>
<th>Can Not</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Overwrite any data memory at any time</td>
<td>• Execute Data</td>
</tr>
<tr>
<td>• stack, heap, data segs</td>
<td>• NX takes care of that</td>
</tr>
<tr>
<td>• Overwrite registers in current context</td>
<td>• Modify Code</td>
</tr>
<tr>
<td></td>
<td>• text seg usually read-only</td>
</tr>
<tr>
<td></td>
<td>• Write to <code>%ip</code></td>
</tr>
<tr>
<td></td>
<td>• true in x86</td>
</tr>
<tr>
<td></td>
<td>• Overwrite registers in other contexts</td>
</tr>
<tr>
<td></td>
<td>• kernel will restore regs</td>
</tr>
</tbody>
</table>

[Brumley’15]
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

“static” [Brumley’15]
Control Flow Graphs
Basic Block

control is “straight”
(no jump targets except at the beginning, no jumps except at the end)

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

[Brumley’15]
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();
}
```

[Brumley'15]
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* supergraph 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

[Brumley’15]
If analysis says X is true, then X is true.

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

True Things

Things I say

Trivially Sound: Say nothing

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() {
1. red(1);
2. red(2);
3. green();
}

void red(int x) {
..
}

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

Context sensitive distinguishes 2 different calls to red(-)

[Brumley’15]
Context Sensitive Example

\[ a = \text{id}(4); \]
\[ b = \text{id}(5); \]

\begin{center}
\begin{tikzpicture}
  \node at (0,0) {a = \text{id}(4);};
  \node at (2,0) {\text{void id(int z)};}
  \node at (4,0) {\{ \text{return z; } \};};
  \node at (6,0) {b = \text{id}(5);};

  \draw[->, green] (0,0) -- (2,0);
  \draw[->, green] (2,0) -- (4,0);
  \draw[->, green] (4,0) -- (6,0);

  \draw[->, red] (2,0) -- (4,0);
  \draw[->, red] (4,0) -- (6,0);

  \node at (8,-2) {Context-Sensitive};
  \node at (10,-2) {(color denotes matching call/ret)};
\end{tikzpicture}
\end{center}

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

Context Insensitive

\[ a = \text{id}(4); \]
\[ b = \text{id}(5); \]

\begin{center}
\begin{tikzpicture}
  \node at (0,0) {a = \text{id}(4);};
  \node at (2,0) {\text{void id(int z)};}
  \node at (4,0) {\{ \text{return z; } \};};

  \draw[->, red] (0,0) -- (2,0);
  \draw[->, red] (2,0) -- (4,0);
  \draw[->, red] (4,0) -- (6,0);

  \node at (8,-2) {Context-Insensitive};
  \node at (10,-2) {(note merging)};
\end{tikzpicture}
\end{center}

Context insensitive will say both calls return \{4,5\}
Flow Sensitive

- A flow sensitive analysis considers the order (flow) of statements.
- 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.

```plaintext
x = 4;
....
x = 5;
```

Flow sensitive can distinguish values of `x`, flow insensitive cannot.
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

[Brumley’15]
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 \( x \geq 0 \)
2. \( y = x; \)
3. else
4. \( y = -x; \)

path sensitive:
\( y \geq 0 \) at line 2,
\( y > 0 \) at line 4

path insensitive:
y is not a constant

[Brumley’15]
Precision

Even path sensitive analysis approximates behavior due to:

- loops/recursion
- unrealizable paths

1. if\(a^n + b^n = c^n \text{ and } n > 2 \text{ and } a > 0 \text{ and } b > 0 \text{ and } c > 0\)
2. \(x = 7;\)
3. else
4. \(x = 8;\)

Unrealizable path.
\(x\) will always be 8

[Brumley’15]
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
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);
}

Two possible return sites due to context insensitivity

[Brumley’15]
Instrument Binary

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);
}

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

\[\text{predicated} \text{ call } 17, R: \text{ transfer control to } R \text{ only when } R \text{ has label } 17\]

\[\text{predicated} \text{ ret } 23: \text{ transfer control to only label } 23\]

[Brumley’15]
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

[Brumley’15]
What about indirect jumps and ret?
ID Checks

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

\[
\text{mov eax, [ebx+8]; load pointer into register}
\]

\[
\text{cmp [eax+4], 12345678h; compare opcodes at destination}
\]

\[
\text{jne error_label; if not ID value, then fail}
\]

\[
\text{call eax; call function pointer}
\]

\[
\text{prefetchnta [AABBCDDh]; label ID, used upon the return}
\]

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</td>
</tr>
</tbody>
</table>

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

\[
\text{mov ecx, [esp]; load address into register}
\]

\[
\text{add esp, 14h; pop 20 bytes off the stack}
\]

\[
\text{cmp [ecx+4], AABBCDDh; compare opcodes at destination}
\]

\[
\text{jne error_label; if not ID value, then fail}
\]

\[
\text{jmp ecx; jump to return address}
\]
Performance

- Size: increase 8% avg
- Time: increase 0-45%; 16% avg

![Bar chart](chart.png)

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

[Brumley’15]
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)

[Brumley’15]
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

\[
\text{if}(\text{module}_{-}\text{lower} < x < \text{module}_{-}\text{upper})
\]

\[
z = \text{load}[x];
\]

• CFI ensures inserted memory checks are executed

[SFI Check] [Brumley’15]
Inline Reference Monitors

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

- Any IRM can be supported by CFI + Software Memory Access Control
  - CFI: IRM code cannot be circumvented
  - SMAC: IRM state cannot be tampered.

[Brumley’15]
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

[Brumley’15]
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
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

[Brumley’15]
Code Pointer Integrity
Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, Dawn Song, OSDI 2014
Control-Flow Hijack Attack

1. `int *q = buf + input;`
2. `*q = input2;`
3. `(*func_ptr)();`

- Attacker corruptions a data pointer
- Attacker uses it to overwrite a code pointer
- Control-flow is transferred to shell code
Memory safety prevents control-flow hijacks

• ... but memory safe programs still rely on C/C++ ...
• Sample Python program (Dropbox SDK example):

<table>
<thead>
<tr>
<th>Python program</th>
<th>3 KLOC of Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python runtime</td>
<td>500 KLOC of C</td>
</tr>
<tr>
<td>libc</td>
<td>2500 KLOC of C</td>
</tr>
</tbody>
</table>

[Kuznetsov’14]
Memory safety can be retrofitted to C/C++

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoftBound+CETS</td>
<td>116%</td>
</tr>
<tr>
<td>CCured</td>
<td>56%</td>
</tr>
<tr>
<td>(language modifications)</td>
<td></td>
</tr>
<tr>
<td>Watchdog</td>
<td>29%</td>
</tr>
<tr>
<td>(hardware modifications)</td>
<td></td>
</tr>
<tr>
<td>AddressSanitizer</td>
<td>73%</td>
</tr>
<tr>
<td>(approximate)</td>
<td></td>
</tr>
</tbody>
</table>
State of the art: Control-Flow Integrity

Static property:
limit the set of functions that can be called at each call site

Coarse-grained CFI can be bypassed [1-4]
and Finest-grained CFI has 10-21% overhead [5-6]

Programmers have to choose

Safety Security vs Flexibility Performance

[Kuznetsov’14]
Code-Pointer Integrity, provides both

Control-flow hijack protection
Practical protection
Guaranteed protection

Unmodified C/C++
0.5 - 1.9% overhead
8.4 - 10.5% overhead

Key insight: memory safety for code pointers only.

Tested on:

FreeBSD hardened
Python
SQLite
LAME
OpenSSL
GraphicsMagick
PostgreSQL
Apache

Spring 1398
Ce 874 - Control Flow Integrity
[Kuznetsov’14]
Threat Model

• Attacker can read/write data, read code
• Attacker cannot
  • Modify program code
  • Influence program loading

[Kuznetsov'14]
Memory Safety: program instrumentation

```c
char *buf = malloc(10);
buf_lower = p; buf_upper = p+10;
...
char *q = buf + input;
q_lower = buf_lower; q_upper = buf_upper;
if (q < q_lower || q >= q_upper-size)
    abort();
*q = input2;
...
(*func_ptr)();
```

116% average performance overhead (Nagarakatte et al., PLDI’09 and ISMM’10)

All-or-nothing protection

[Kuznetsov’14]
Memory Safety

116% average performance overhead

Can memory safety be enforced for code pointers only?

Control-flow hijack protection
1.9% or 8.4% average performance overhead

[Kuznetsov’14]
Practical Protection (CPS): Heap

```
int *q = buf + input;
*q = input2;
...
(*func_ptr)();
```

Instructions that access code pointers are identified using type-based static analysis.

Separation is enforced using hardware-enforced instruction-level isolation.

---

**Code pointers only**

- **Safe Memory**
  - `func_ptr`

  - 2.5% memory accesses (on SPEC2006 CPU)

- **Program memory is separated**

- **Regular Memory**
  - `buf`

  - 97.5% memory accesses (on SPEC2006 CPU)

---

All non-code-pointer data

Memory layout unchanged

---

[Kuznetsov’14]
Practical Protection (CPS): Stack

```c
int foo() {
    char buf[16];
    int r;
    r = scanf("%s", buf);
    return r;
}
```

Safe stack adds <0.1% performance overhead!
Practical Protection (CPS): Memory Layout

Safe memory
(code pointers)

Safe Heap

Safe Stack
(thread1)

Safe Stack
(thread2)

…

Regular memory
(non-code-pointer data)

Regular Heap

Regular Stack
(thread1)

Regular Stack
(thread2)

…

Code (Read-Only)

Only instructions that operate on code pointers can access the safe memory

Hardware-based instruction-level isolation
The CPS Promise

Under CPS, an attacker cannot forge a code pointer
Under CPS, an attacker cannot forge a code pointer

Contrived example of an attack on a CPS-protected program

1. `int *q = p + input;`
2. `*q = input2;`
3. `func_ptr = struct_ptr->f;`
4. `(*func_ptr)();`

① Attacker corrupts a data pointer  
② Attacker uses it to corrupt a struct pointer  
③ Program loads a function pointer from wrong location in the safe memory  
④ Control-flow is transferred to different function whose address was previously stored in the safe memory
Under CPS, an attacker cannot forge a code pointer

Contrived example of an attack on a CPS-protected program

```c
int *q = p + input;
*q = input2;
...
func_ptr = struct_ptr->f;
(*func_ptr)();
```

Precise solution: protect all sensitive pointers

\(^1\)Sensitive pointers = code pointers and pointers used to access sensitive pointers

\(^1\)Kuznetsov’14
Code-Pointer Separation

• Identify Code-Pointer accesses using static type-based analysis
• Separate using instruction-level isolation (e.g., segmentation)

• CPS security guarantees
  • An attacker cannot forge new code pointers
  • Code-Pointer is either immediate or assigned from code pointer
  • An attacker can only replace existing functions through indirection: e.g., foo->bar->func() vs. foo->baz->func2()
Code-Pointer Integrity (CPI)

- Sensitive Pointers = code pointers and pointers used to access sensitive pointers

- CPI identifies all sensitive pointers using an over-approximate type-based static analysis:
  \[ \text{is_sensitive}(v) = \text{is_sensitive_type}(\text{type of } v) \]

- Over-approximation only affects performance
  - On SPEC2006 <= 6.5% accesses are sensitive
Guaranteed Protection (CPI): Memory Layout

**Safe memory** (sensitive pointers and metadata)
- Safe Heap
- Safe Stack (thread1)
- Safe Stack (thread2)

**Regular memory** (non-sensitive data)
- Regular Heap
- Regular Stack (thread1)
- Regular Stack (thread2)

---

Accesses are checked for memory safety
Accesses are fast

Only instructions that operate on sensitive pointers can access the safe memory

Hardware-based instruction-level isolation

[Kuznetsov’14]
Guaranteed Protection (CPI)

• Guaranteed memory safety for all sensitive pointers
  • Sensitive Pointers = code pointers and **pointers used to access sensitive pointers**

• \(\Rightarrow\) Guaranteed protection against control-flow hijack attacks enabled by memory bugs
Code-Pointer Integrity vs. Separation

- Separate sensitive pointers from regular data
  - Type-based static analysis
  - Sensitive pointers = code pointers + pointers to sensitive pointers

- Accessing sensitive pointers is safe
  - Separation + runtime (bounds) checks

- Accessing regular data is fast
  - Instruction-level safe region isolation
Security Guarantees

- Code-Pointer Integrity: formally guaranteed protection
  - 8.4% to 10.5% overhead (~6.5% of memory accesses)
- Code-Pointer Separation: strong protection in practice
  - 0.5% to 1.9% overhead (~2.5% of memory accesses)
- Safe Stack: full ROP protection
  - Negligible overhead
<table>
<thead>
<tr>
<th>Protects Against</th>
<th>Technique</th>
<th>Security Guarantees</th>
<th>Average Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory corruption vulnerabilities</td>
<td>Memory Safety</td>
<td>Precise</td>
<td>116%</td>
</tr>
<tr>
<td></td>
<td><strong>CPI</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>(Guaranteed protection)</em></td>
<td>Precise</td>
<td>8.4-10.5%</td>
</tr>
<tr>
<td>Control-flow hijack vulnerabilities</td>
<td><strong>CPS</strong></td>
<td>Strong</td>
<td>0.5-1.9%</td>
</tr>
<tr>
<td></td>
<td><em>(Practical protection)</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Finest-grained CFI</td>
<td>Medium <em>(attacks may exist)</em></td>
<td>10-21%</td>
</tr>
<tr>
<td></td>
<td><em>Göktaş et al., IEEE S&amp;P 2014</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Coarse-grained CFI</td>
<td>Weak <em>(known attacks)</em></td>
<td>4.2-16%</td>
</tr>
<tr>
<td></td>
<td>ASLR DEP Stack cookies</td>
<td>Weakest <em>(bypassable + widespread attacks)</em></td>
<td>~0%</td>
</tr>
</tbody>
</table>
Implementation

• LLVM-based prototype
  • Front end (clang): collect type information
  • Back-end (llvm): CPI/CPS/SafeStack instrumentation pass
  • Runtime support: safe heap and stack management
  • Supported ISA's: x64 and x86 (partial)
  • Supported systems: Mac OSX, FreeBSD, Linux
Current status

- Great support for CPI on Mac OSX and FreeBSD on x64
- Upstreaming in progress
  - Safe Stack coming to LLVM soon
  - Fork it on GitHub now: https://github.com/cpi-llvm
- Code-review of CPS/CPI in process
  - Play with the prototype: http://levee.epfl.ch/levee-early-preview-0.2.tgz
  - Will release more packages soon
- Some changes to super complex build systems needed
  - Adapt Makefiles for FreeBSD
Conclusion

- CPI/CPS offers strong control-flow hijack protection
  - Key insight: memory safety for code pointers only
- Working prototype
  - Supports unmodified C/C++, low overhead in practice
  - Upstreaming patches in progress, SafeStack available soon!
  - Homepage: http://levee.epfl.ch
  - GitHub: https://github.com/cpi-llvm
Acknowledgments/References