Code-Pointer Integrity

Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candeа, R. Sekar, Dawn Song
Control-Flow Hijack Attack

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

① Attacker corrupts a data pointer
② Attacker uses it to overwrite a code pointer
③ Control-flow is transferred to shell code
Control-flow hijacks are still abundant today!
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>
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
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
GraphMagick
PostgreSQL
Apache
Overview

Does it solve a real problem?

How does it work?
  Threat model & background
  Practical protection: CPS
  Guaranteed protection: CPI

How secure is it?

How practical is it?
Threat Model

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

Memory Safety
program instrumentation
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
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

Program memory is separated

2.5% memory accesses
(on SPEC2006 CPU)

All non-code-pointer data

Memory layout unchanged

97.5% memory accesses
(on SPEC2006 CPU)
int foo() {
    char buf[16];
    int r;
    r = scanf("%s", buf);
    return r;
}
Practical Protection (CPS): Memory Layout

Safe memory (code pointers)
- Safe Heap
- Safe Stack (thread1)
- Safe Stack (thread2)
- \(\cdots\)

Regular memory (non-code-pointer data)
- Regular Heap
- Regular Stack (thread1)
- Regular Stack (thread2)
- \(\cdots\)
- 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

Is this enough?
In practice, yes!
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)();
```

Is this enough?
In practice, yes!

Precise solution: protect all sensitive\(^1\) pointers

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

With CPI: `struct_ptr` is sensitive and cannot be corrupted
Guaranteed Protection (CPI)

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

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

- Over-approximation doesn’t hurt security, it only affects performance:
  On SPEC2006 ≤6.5% memory 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
Guaranteed Protection (CPI)

Guaranteed memory safety for all sensitive¹ pointers

↓

Guaranteed protection against control-flow hijack attacks enabled by memory bugs

¹Sensitive pointers = code pointers and pointers used to access sensitive pointers
Instruction-Level Isolation

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

Dedicated segment register, used only to access the safe memory

```
movl input2, q
```

Perfect hiding: regular memory contains no pointers to safe memory

Alternative: Software Fault Isolation

**x86-32**
- gs.base → Safe Memory
- gs.limit → Regular Memory
- ds.base → Regular Memory
- ds.limit →

**x86-64**
- Regular Memory
- Safe Memory
- fs.base (randomized)

Alternative: Software Fault Isolation
<table>
<thead>
<tr>
<th>CPS</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Separate sensitive pointers and regular data</td>
<td>Sensitive pointers = code pointers + indirect pointers to sensitive pointers</td>
</tr>
<tr>
<td>Sensitive pointers = code pointers</td>
<td>Sensitive pointers = code pointers + indirect pointers to sensitive pointers</td>
</tr>
<tr>
<td>• Accessing sensitive pointers is safe</td>
<td>Separation + runtime checks</td>
</tr>
<tr>
<td>Separation</td>
<td></td>
</tr>
<tr>
<td>• Accessing regular data is fast</td>
<td>Instruction-level safe region isolation</td>
</tr>
<tr>
<td>Instruction-level safe region isolation</td>
<td></td>
</tr>
</tbody>
</table>
Overview

Does it solve a real problem?

How does it work?

Threat model & background

Practical protection: CPS

Guaranteed protection: CPI

How secure is it?

How practical is it?
Overview

Does it solve a real problem?
How does it work?
How secure is it?
How practical is it?
How secure is it?

- RIPE\textsuperscript{1} runtime intrusion prevention evaluator:
  - Both CPS and CPI prevent all attacks from RIPE
- Future attacks:
  - CPI correctness proof in the paper

\textsuperscript{1}Wilander at al., ACSAC 2011
<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</td>
<td>Memory Safety</td>
<td>Precise</td>
<td>116%</td>
</tr>
<tr>
<td>vulnerabilitys</td>
<td>Memory Safety</td>
<td>Precise</td>
<td>8.4-10.5%</td>
</tr>
<tr>
<td>Control-flow</td>
<td>CPI</td>
<td>Precise</td>
<td>0.5-1.9%</td>
</tr>
<tr>
<td>hijack</td>
<td>CPS</td>
<td>Strong</td>
<td>10-21%</td>
</tr>
<tr>
<td>vulnerabilitys</td>
<td>Finest-grained</td>
<td>Medium (attacks may exist) Göktaş el., IEEE S&amp;P 2014</td>
<td>4.2-16%</td>
</tr>
<tr>
<td></td>
<td>ASLR DEP</td>
<td>Weakest (bypassable + widespread attacks)</td>
<td></td>
</tr>
</tbody>
</table>
Overview

Does it solve a real problem?
How does it work?
How secure is it?
How practical is it?
Overview

Does it solve a real problem?
How does it work?
How secure is it?
How practical is it?
Implementation
Is it practical?
Is it fast enough?
Implementation

cc -fcpi foo.c

• LLVM-based prototype at http://levee.epfl.ch

• Plan to integrate upstream into LLVM
Implementation

• LLVM-based prototype at http://levee.epfl.ch
  
  • Front-end (clang):
    Collect type information
  
  • Back-end (LLVM):
    CPI/CPS and SafeStack instrumentation passes
  
  • Runtime support (compiler-rt or libc):
    Safe heap and stacks management
Full OS Distribution
with CPS/CPI protection

- Recompiled the entire FreeBSD userspace…
- … and more than 100 packages

Python
SQLite
PostgreSQL
OpenSSL
Apache
lame
GraphicsMagick
Performance overhead on Phoronix

- pgbench
- openssl
- encode-mp3
- graphics-magick 1
- graphics-magick 2
- graphics-magick 3
- graphics-magick 4
- graphics-magick 5
- hmmer
- postmark
- sqlite
- pybench
- dcraw
- crafty
- compress-lzma
- compress-pbzip2
- c-ray

Safe stack only: 0.01%
CPS: 0.5%
CPI: 10.5%
Performance overhead on SPEC2006 CPU

- Safe stack only
- CPS (practical protection)
- CPI (guaranteed protection)

Average: -5%, 0%, 5%, 10%, 15%, 20%, 25%, 30%, 35%, 40%, 45%

Safe stack: 0.03%
CPS: 1.9%
CPI: 8.4%
Overview

Does it solve a real problem?
How does it work?
How secure is it?
How practical is it?

Implementation
Is it fast enough?
Is it practical?
Code-Pointer Integrity

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

http://levee.epfl.ch