Code-Pointer Integrity

Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea
R. Sekar, Dawn Song
Outline

- Problem Statement
- Existing solutions and their weaknesses
- Code-Pointer Integrity
- Implementation-dependant weakness (Related Paper)
- Discussion
Problem Statement
Problem Statement

- Attackers exploit bugs to cause memory corruption
Problem Statement

- Attackers exploit bugs to cause memory corruption
- Steal sensitive data and/or execute code on the system
Problem Statement

• Attackers exploit bugs to cause memory corruption

• Steal sensitive data and/or execute code on the system

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

• Attackers exploit bugs to cause memory corruption

• Steal sensitive data and/or execute code on the system

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

• Attackers exploit bugs to cause memory corruption

• Steal sensitive data and/or execute code on the system

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

- Adress Space Layout Randomisation (ASLR)
Existing Solutions

• Adress Space Layout Randomisation (ASLR)
  ▪ Places code and data segments at random addresses
  ▪ Complicates code-reuse (ROP)
  ▪ Defeated by pointer leaks and side channel attacks
Existing Solutions

- Adress Space Layout Randomisation (ASLR)
- Stack Cookies
Existing Solutions

• Adress Space Layout Randomisation (ASLR)

• Stack Cookies
  ▶ Protect return addresses on the stack
  ▶ Only protect against continuous buffer overflows
Existing Solutions

• Adress Space Layout Randomisation (ASLR)

• Stack Cookies

• Data Execution Prevention (DEP)
Existing Solutions

• Adress Space Layout Randomisation (ASLR)

• Stack Cookies

• Data Execution Prevention (DEP)

• Control-Flow Integrity
Existing Solutions

- Adress Space Layout Randomisation (ASLR)
- Stack Cookies
- Data Execution Prevention (DEP)
- Control-Flow Integrity
- Memory Safety
Control-Flow Integrity
Control-Flow Integrity

• Limit the set of functions that can be called at each call site
Control-Flow Integrity

- Limit the set of functions that can be called at each call site
- Coarse-grained CFI can be bypassed
Control-Flow Integrity

- Limit the set of functions that can be called at each call site
- Coarse-grained CFI can be bypassed
- Finest-grained CFI has 10-21% performance overhead
Memory Safety
Memory Safety

• Guarantees memory objects can only be accessed by pointers properly based on the specific object
Memory Safety

- Guarantees memory objects can only be accessed by pointers properly based on the specific object
  - Completely prevents control-flow hijacks
Memory Safety

- Guarantees memory objects can only be accessed by pointers properly based on the specific object
  - Completely prevents control-flow hijacks

- Requires rewriting code in memory-safe languages or retrofitting memory safety onto existing code
Memory Safety

• Guarantees memory objects can only be accessed by pointers properly based on the specific object

  ➞ Completely prevents control-flow hijacks

• Requires rewriting code in memory-safe languages or retrofitting memory safety onto existing code

• Requires runtime checks to verify correctness of pointer computations
Memory Safety

• Guarantees memory objects can only be accessed by pointers properly based on the specific object
  ➡ Completely prevents control-flow hijacks

• Requires rewriting code in memory-safe languages or retrofitting memory safety onto existing code

• Requires runtime checks to verify correctness of pointer computations
  ➡ Introduces significant performance overhead (≥2x when retrofitted)
Code-Pointer Integrity
Code-Pointer Integrity

• Goals:
  ▶ Prevent all control-flow hijack attacks
  ▶ Significantly less performance overhead than state-of-the-art
Code-Pointer Integrity

• Goals:
  ‣ Prevent all control-flow hijack attacks
  ‣ Significantly less performance overhead than state-of-the-art

• Idea:
  ‣ Use memory-safety but only protect code-pointers
Code-Pointer Separation
Code-Pointer Separation

int *q = buf + input;
*q = input2;
...
(*func_ptr)();
Code-Pointer Separation

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

• Type-based static analysis
Code-Pointer Separation

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

- Type-based static analysis
- Move only code pointers to safe memory
  - Isolate safe memory on instruction level
Code-Pointer Separation

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

- Type-based static analysis
- Move only code pointers to safe memory
  - Isolate safe memory on instruction level
- Keep memory layout unchanged
Code-Pointer Separation

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

- Type-based static analysis
- Move only code pointers to safe memory
  - Isolate safe memory on instruction level
- Keep memory layout unchanged

<table>
<thead>
<tr>
<th>Safe Memory</th>
<th>Regular Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>func_ptr</strong></td>
<td><strong>buf</strong></td>
</tr>
</tbody>
</table>

2.5% memory accesses

97.5% memory accesses
Safestack
int foo() {
    char buf[16];
    int r;
    r = scanf("%s", buf);
    return r;
}
int foo() {
    char buf[16];
    int r;
    r = scanf("%s", buf);
    return r;
}

- Split into regular and safe stack

<table>
<thead>
<tr>
<th>Safe Stack</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ret address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Regular Stack</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td></td>
</tr>
<tr>
<td>buf</td>
<td></td>
</tr>
</tbody>
</table>
Safestack

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

- Split into regular and safe stack
- Statical check during compile which objects are safe
int foo() {
    char buf[16];
    int r;
    r = scanf("%s", buf);
    return r;
}

- Split into regular and safe stack
- Statical check during compile which objects are safe
int foo() {
    char buf[16];
    int r;
    r = scanf("%s", buf);
    return r;
}

- Split into regular and safe stack
- Stataical check during compile which objects are safe
- Only keep unsafe objects on the regular stack (e.g. arrays)
CPS Memory Layout
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)

Instruction-level isolation
Code-Pointer Integrity
Code-Pointer Integrity

Protecting only code pointers is not enough:
Code-Pointer Integrity

Protecting only code pointers is not enough:

```c
int *q = p + input;
*q = input2;
...

(*func_ptr)();
```
Code-Pointer Integrity

Protecting only code pointers is not enough:

```c
int *q = p + input;
*q = input2;
...
(*func_ptr)();
```
Code-Pointer Integrity

Protecting only code pointers is not enough:

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

```
Memory
  func_ptr
  struct_ptr
```
Code-Pointer Integrity

Protecting only code pointers is not enough:

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

Protecting only code pointers is not enough:

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

Protecting only code pointers is not enough:

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

Protecting only code pointers is not enough:

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

Protecting only code pointers is not enough:

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

Protecting only code pointers is not enough:

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

⇒ Indirect Pointers have to be protected as well
Code-Pointer Integrity

Protecting only code pointers is not enough:

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

➡ Indirect Pointers have to be protected as well
➡ Extend static analysis to include indirect pointers
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)

Code (Read-Only)

Instruction-level isolation
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)  ...

Code (Read-Only)

Instruction-level isolation
Summary
Summary

• CPI guarantees memory safety for all sensitive pointers (code pointers and pointers to sensitive pointers)
Summary

- CPI guarantees memory safety for all sensitive pointers (code pointers and pointers to sensitive pointers)
  
  ➡ Guaranteed protection against control-flow hijack attacks enabled by memory bugs
Summary

• CPI guarantees memory safety for all sensitive pointers (code pointers and pointers to sensitive pointers)

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

• Keeps performance overhead low by not protecting data pointers
Design
Design

• Static analysis on source code during compilation
Design

- Static analysis on source code during compilation
- Adding safe memory region while keeping the original memory layout intact
Design

• Static analysis on source code during compilation

• Adding safe memory region while keeping the original memory layout intact

• Separating the safe region from the regular region using instruction level protection:
  ▶ Hardware segment protection on x86-32
  ▶ Information hiding on x86-64 and ARM
Security analysis
Security analysis

- CPI and CPS protect against all attacks from RIPE (Runtime intrusion prevention evaluator)
Security analysis

- CPI and CPS protect against all attacks from RIPE (Runtime intrusion prevention evaluator)

- CPI correctness proof in paper guarantees security against future attacks
Security analysis

• CPI and CPS protect against all attacks from RIPE (Runtime intrusion prevention evaluator)

• CPI correctness proof in paper guarantees security against future attacks

• Does not protect against data-only attacks
Performance Benchmark

Performance in SPEC CPU2006:

400_perlbench (C)
401_bzip2 (C)
403_gcc (C)
429_mcf (C)
433_milc (C)
444_namd (C++)
445_gobmk (C)
447_dealII (C++)
450_soplex (C++)
453_povray (C++)
456_hmmer (C)
458_sjeng (C)
462_libquantum (C)
464_h264ref (C)
470_lbm (C)
471.omnetpp (C++)
473.astar (C++)
482_sphinx3 (C)
483_xalanbmk (C++)
## Performance summary

<table>
<thead>
<tr>
<th></th>
<th>Safe Stack</th>
<th>CPS</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average (C/C++)</td>
<td>0.0%</td>
<td>1.9%</td>
<td>8.4%</td>
</tr>
<tr>
<td>Median (C/C++)</td>
<td>0.0%</td>
<td>0.4%</td>
<td>0.4%</td>
</tr>
<tr>
<td>Maximum (C/C++)</td>
<td>4.1%</td>
<td>17.2%</td>
<td>44.2%</td>
</tr>
<tr>
<td>Average (C only)</td>
<td>-0.4%</td>
<td>1.2%</td>
<td>2.9%</td>
</tr>
<tr>
<td>Median (C only)</td>
<td>-0.3%</td>
<td>0.5%</td>
<td>0.7%</td>
</tr>
<tr>
<td>Maximum (C only)</td>
<td>4.1%</td>
<td>13.3%</td>
<td>16.3%</td>
</tr>
</tbody>
</table>

Performance numbers from SPEC CPU2006 Benchmark
Security Weakness on x64 and ARM
Security Weakness on x64 and ARM

• Original Paper:

  ➡ Information hiding is secure because no pointer to the safe region exists in unsafe memory
Security Weakness on x64 and ARM

- Original Paper:
  - Information hiding is secure because no pointer to the safe region exists in unsafe memory

- Paper by Evans et. al.:
  - Shows there is a way to find the safe area using side channel attack
Information Hiding
Implementation
1) Randomly choose an address to serve as base address for safe memory region
Information Hiding Implementation

1) Randomly choose an address to serve as base address for safe memory region

2) Store address in one of the segment registers provided by x64
Information Hiding Implementation

1) Randomly choose an address to serve as base address for safe memory region

2) Store address in of the segment registers provided by x64

⇒ No pointer to the safe region exists in regular memory
Information Hiding Implementation

1) Randomly choose an address to serve as base address for safe memory region

2) Store address in of the segment registers provided by x64

➡ No pointer to the safe region exists in regular memory

➡ 48 bit address space in x64 CPU makes guessing impractical, most guesses would cause crashing
Attack Description
Attack Description

1) Timing Side-channel Attack
Attack Description

1) Timing Side-channel Attack

2) Data Collection
Attack Description

1) Timing Side-channel Attack

2) Data Collection

3) Locate Safe Region
Attack Description

1) Timing Side-channel Attack
2) Data Collection
3) Locate Safe Region
4) Attack Safe Region
Mitigation of the Weakness

• Implement Hardware Segmentation in x86-64

• Switch to software fault isolation

  ➡ Introduces additional performance overhead of ~5%

• Reduce feasibility of side channel attack by changing implementation of information hiding

  ➡ Replace linear table with hash table or two-level lookup table
Discussion

Questions?

References:

• Code-Pointer Integrity - Kuznetsov et. al. (2014)
• Presentation: Code-Pointer Integrity - Kuznetsov (OSDI 2014)
• Missing the Point(er) - Evans et. al. (2015)
• Getting the Point(er) - Kuznetsov et. al. (2015)