Practical Control Flow Integrity & Randomization for Binary Executables

Chao Zhang  
Peking University

Tao Wei  
Peking University  
UC Berkeley

Zhaofeng Chen  
Peking University

Stephen McCamant  
University of Minnesota

Dawn Song  
UC Berkeley

Lei Duan  
Peking University

László Szekeres  
Stony Brook University

Wei Zou  
Peking University
Eternal War in Memory (Oakland’13)

Attacker

- buffer overflow ...
- code injection
- ret2libc, ROP
- code re-use
- Info Leakage
- ...

Defender

- $W \otimes X$
- ASLR
- ?

chao.100871.net
Reactive Defense is not adequate

- > 5,000 CVE vulnerabilities each year.
  - Many more are under the iceberg.
- Vulnerabilities are exploited millions of times.
Proactive Defense: Control Flow Integrity

- Control Flow Integrity, CFI [CCS’05]

- **Security policy**
  - Each *run-time* control transfer must comply with programmer’s *compile-time* intent, i.e., the control-flow graph

- A strong guarantee that can be reasoned about formally
  - ROP, JOP, UAF

- A foundation for other low-level code defenses
  - SFI, sandboxing untrusted code

- Deterministic, not probabilistic defense
Control-Flow Graph

- conditional jump (jz ...): target relative address
- direct jump/call: target relative or absolute address
- indirect jump/call: target? (read or computed from memory)
- return: target return address on the stack
# Scope of Control-Flow Integrity Protection

<table>
<thead>
<tr>
<th>Control Transfer Method</th>
<th>Transfer Targets</th>
<th>Integrity Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions</td>
<td>Exception handlers</td>
<td>SafeSEH, SEHOP</td>
</tr>
<tr>
<td>Direct CALL/JMP, Conditional jump (JZ...)</td>
<td>Hard-coded function pointers (embedded in the code)</td>
<td>W⊗X</td>
</tr>
<tr>
<td>Indirect CALL/JMP</td>
<td>In-memory function pointers</td>
<td>CFI</td>
</tr>
<tr>
<td>RET instructions</td>
<td>On-stack return addresses</td>
<td>/GS, shadow stack, CFI</td>
</tr>
</tbody>
</table>
Control Flow Integrity (CCS’05)

- build Control Flow Graph
  - source code/debug info
  - all modules

```
... call eax
next_1:

... call ecx
next_2:

foo:
  ...
  ret

bar:
  ...
  ret

fun:
  ...
  ret
```
Control Flow Integrity (CCS’05)

- build Control Flow Graph
- binary rewriting
  - instrument IDs before transfer targets
  - validate IDs before control transferring
Control Flow Integrity (CCS’05)

- build Control Flow Graph
- binary rewriting
- diversity of IDs
Control Flow Integrity (CCS’05)

- build Control Flow Graph
- binary rewriting
- diversity of IDs
- all indirect transfers
  - indirect call
  - indirect jmp
  - ret
Challenges faced by CFI

- Performance overhead: average 15%, max 46%

- ID is embedded in a slow `prefetchnta` instruction
  - executed each time the transfer target is reached

- Introduce overhead even if `foo` is called/jumped to directly
  - Most calls/jumps are direct
Challenges faced by CFI

- performance overhead
- modularity support
  - A single module cannot be hardened independently
Challenges faced by CFI

- performance overhead
- modularity support
- backward compatibility

```
... check ID_1
call eax
next_1:

foo:
  ID_1
  ...
  ret

bar:
  ID_1
  ...
  ret

test:
  ...
  ret
```
Challenges faced by CFI

- performance overhead
- modularity support
- backward compatibility
- source code/debug info dependency
  - to build Control Flow Graph
Motivation

- A light-weight CFI solution with a strong protection
  - performance overhead
  - modularity support
  - backward compatibility
  - source code independent
Assumption

- ASLR and W⊗X (e.g. DEP) are deployed

- NO self-modifying code or dynamically generated code.

- Limited information disclosure vulnerabilities
  - e.g., cannot read entire memory regions
Outline

- Motivation
- Intuition & Design
- Implementation
- Security Enhancement
- Evaluation
- Conclusion
Intuition

- Checking transfer targets’ kind, rather than IDs
  - indirect call/jmp can only transfer to function-pointer stubs
  - returns can only transfer to return-address stubs
Intuition: efficient check

- memory alignment
  - function pointer stubs are 8-bytes aligned
  - return address stubs are 16-bytes aligned

```
foo:
  ...test [esp],0x0f
  ret

next_1:
  test eax,7
  call eax

```

```
bar:
  ...test [esp],0x0f
  ret
```

```
fun:
  ...test [esp],0x0f
  ret

next_2:
  test ecx,7
  call ecx
```

- fast bit testing
  - performance overhead ✓

- No IDs
  - performance overhead ✓ ✓

- only check type
  - modularity support ✓ ✓ ✓
Intuition: security

- Transfer targets must be within memory region Springboard
  - 27th bit is 0

```
... test eax, M_F
call eax
next_1:

foo:
  ... test [esp], M_R
  ret

bar:
  ... test [esp], M_R
  ret

... test ecx, M_F
call ecx
next_2:

fun:
  ... test [esp], M_R
  ret
```

- Policy: all indirect transfer targets must be aligned stubs in Springboard.

\[ M_F = 0x800007 \quad M_R = 0x800000f \]
Design: Memory layout with Springboard

<table>
<thead>
<tr>
<th>Executable</th>
<th>Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>27</td>
<td>Non-executable section</td>
</tr>
<tr>
<td>No</td>
<td>26</td>
<td>Normal code section</td>
</tr>
<tr>
<td>Yes</td>
<td>3</td>
<td>Invalid entry in Springboard</td>
</tr>
<tr>
<td>Yes</td>
<td>2-0</td>
<td>fp-stub entry in Springboard</td>
</tr>
<tr>
<td>Yes</td>
<td>0</td>
<td>Sensitive ret-stub entry in Springboard</td>
</tr>
<tr>
<td>Yes</td>
<td>0</td>
<td>Normal ret-stub entry in Springboard</td>
</tr>
</tbody>
</table>

- **Springboard vs. normal code section** (27th bit)
- **Entries in the Springboard are all aligned.** (0-2 bits)
- **Three kinds of stubs in Springboard**
  - fp-stub entries vs. ret-stub entries (3rd bit)
  - sensitive vs. normal ret-stub (26th bit)
    - normal ret-stubs cannot return into the middle of sensitive functions
- **One or two fast bit testing instructions are sufficient**
Architecture of our solution: CCFIR

- Find out all indirect transfer instructions and their targets
  - **BitCover**: disassemble target binary
- Redirect transfer targets to aligned stubs in Springboard
- Validate transfer targets
  - **BitRewrite**: binary rewriting
- Verify the hardened binary
  - **BitVerify**
Outline

- Motivation
- Intuition & Design
- Implementation
- Security Enhancement
- Evaluation
- Conclusion
BitCover: disassembling

- Target: *normal* binaries generated by modern compilers
  - Support ASLR
    - On Windows, it implies the executable contains relocation information
    - We can identify all legal function pointers (indirect call/jmps’ targets)
- Automatically disassemble most of *normal* binaries
  - Based on well-defined rules
  - More accurate than IDA Pro, especially designed for binary rewriting

```
Input
PE file
Validate code entries
Identify control tables
Phase 1
code entry candidates
Other unknown data
Known control tables
Phase 2
Tag reliable code entries
Propagate valid code entries
Output
Valid code entries
Valid control tables
Suspect code entries
```
BitRewrite: binary rewriting

- Policy: all indirect transfers are enforced to aligned stubs in Springboard.

- Validate ret’s target.
  - ret-stub in Springboard
BitRewrite: binary rewriting

- Policy: all indirect transfers are enforced to aligned stubs in Springboard.

- Validate ret’s target.
  - ret-stub in Springboard

- Redirect ret’s target.
  - move call to Springboard

Original code section

Protected code section

Direct control transfer

Indirect control transfer

Springboard section
BitRewrite: binary rewriting

- Policy: all indirect transfers are enforced to aligned stubs in Springboard.

- Validate ret’s target.
  - ret-stub in Springboard

- Redirect ret’s target.
  - move call to Springboard

- Validate call’s target.
  - fp-stub in Springboard
BitRewrite: binary rewriting

- Policy: all indirect transfers are enforced to aligned stubs in Springboard.

Validate ret’s target.
  - ret-stub in Springboard

Redirect ret’s target.
  - move call to Springboard

Validate call’s target.
  - fp-stub in Springboard

Redirect call’s target.
  - rewrite function pointers
BitRewrite: In a nutshell

- redirect function pointers and return addresses
  - rewrite function pointers in memory
  - move call instructions to springboard
- validate transfer targets with bit testing instructions

```
mov eax, foo
...|
|call eax
|next:
|   ret

foo:
...|
|test eax,8
|jz error
|test eax, M_F
|jnz error
|jmp next_sb-2

next:
...|
|mov eax, foo_sb
|...|
|call eax
|next_sb:
|   jmp back

foo_sb:
   jmp foo

M_F = 0x800007f
M_R = 0x800000f
or
M_R = 0xC00000f

Direct control transfer

Indirect control transfer
```
BitRewrite: Optimizations

- With the support of $W \otimes X$, we can skip redirecting or validating
  - directly used function pointers
    - call foo
  - directly used imported functions
    - call [imp_foo]
  - switch statements and jump table
    - jmp jump_table[eax*4]
BitRewrite:
backward compatibility

- With a full deployment (i.e., all modules hardened),
  - backward compatibility issues no longer existed.

- With incremental deployment, such issues occur when
  - CALL/JMP to functions in unhardened modules
  - RET to return addresses in unhardened modules
**case 1: Imported functions**

**Challenge**
- Import Address Table (IAT)
- load-time resolving
  - the loader will update IAT
  - we cannot statically rewrite IAT entries (function pointers)

**Solution**
- Replace IAT with a wrapper table

```plaintext
mov ecx, [foo_slot]
call ecx

foo_slot:   foo
```

```plaintext
mov ecx, [foo_slot_wrap]
call ecx

foo_slot:   foo
```

```plaintext
Springboard
foo_sb:
jmp [foo_slot]
```

```plaintext
IAT_wrapper
foo_slot_wrap: foo_sb
```

```plaintext
.iat:
...
foo_slot: foo
```

```plaintext
.iat:
...
foo_slot: foo
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```

```plaintext
.
```
case 2: Functions resolved by GetProcAddress

- **Challenge**
  - GetProcAddress() returns a non-redirected function pointer.
  - The instrumented security check will fail

- **Solution**
  - wrapper for GetProcAddress
  - redirect return value at runtime
BitVerify: Offline Verification

- whether a given binary conforms to security rules?
  - 27\textsuperscript{th} bit of executable section except Springboard should be 1
  - Code stubs in Springboard are all aligned
  - Function pointers are redirected
  - Call instruction has been redirected
  - Check code has been correctly inserted
  - ...

Outline

- Motivation
- Intuition & Design
- Implementation
- Security Enhancement
- Evaluation
- Conclusion
Policy 1: Restrain sensitive functions

- Sensitive functions
  - system(), execv(), VirtualProtect(), VirtualAlloc(), ...

- Policy:
  - Sensitive functions should be used via direct calls.
    - i.e., their addresses are never taken except hard-coded in instructions

- Benefit:
  - NO fp-stubs for sensitive functions in the Springboard
  - Attackers cannot indirectly call/jump to these functions.
Policy 2: Load-time Randomization

**Policy:**
- The order of all stubs in the Springboard are randomized at load-time.

**Benefit:**
- cannot guess one stub’s address even if another address is leaked in the Springboard.
- A stronger randomization than ASLR.

- It provides an extra layer of protection from the previous CFI-style checking.
Brief Security Analysis

- **return-to-libc**
  - NO ret-stub for `system()`

- **jump-to-libc**
  - similar to return-to-libc
  - gadgets must start from fp-stubs in the Springboard

- **ROP**
  - Valid gadgets must start from ret-stubs in the Springboard
  - Few valid ROP gadgets
    - much longer, hard to chain.
Outline

- Motivation
- Intuition & Design
- Implementation
- Security Enhancement
- Evaluation
- Conclusion
Runtime Overhead

- SPECint2000, average 3.6%, max 8.6%
- SPECfp2000, average 0.59%, max 3.98%
Static Analysis Time

- SPECint2000 & SPECfp2000, 10s seconds
ROP Elimination

- Mona is used to count ROP gadgets
- Valid gadgets must start from ret-stubs in Springboard

<table>
<thead>
<tr>
<th>SPECfp2000 App</th>
<th>original #gadgets</th>
<th>new #gadgets</th>
<th>valid #gadgets</th>
</tr>
</thead>
<tbody>
<tr>
<td>wupwise</td>
<td>255</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>swim</td>
<td>116</td>
<td>134</td>
<td>0</td>
</tr>
<tr>
<td>mgrid</td>
<td>161</td>
<td>166</td>
<td>0</td>
</tr>
<tr>
<td>applu</td>
<td>182</td>
<td>172</td>
<td>0</td>
</tr>
<tr>
<td>mesa</td>
<td>21696</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>galgel</td>
<td>1515</td>
<td>952</td>
<td>0</td>
</tr>
<tr>
<td>art</td>
<td>1874</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>equake</td>
<td>1710</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>facerec</td>
<td>826</td>
<td>775</td>
<td>0</td>
</tr>
</tbody>
</table>

ROP gadgets count (part)
# Real World Exploits Protection

<table>
<thead>
<tr>
<th>ID</th>
<th>App</th>
<th>Vul Type</th>
<th>Vul Module</th>
<th>Protected</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2011-0065</td>
<td>FF 3</td>
<td>Use After Free</td>
<td>xul.dll</td>
<td>yes</td>
</tr>
<tr>
<td>CVE-2010-0249</td>
<td>IE 6</td>
<td>Use After Free</td>
<td>mshtml1.dll</td>
<td>yes</td>
</tr>
<tr>
<td>CVE-2010-3962</td>
<td>IE 6</td>
<td>Use After Free</td>
<td>mshtml1.dll</td>
<td>yes</td>
</tr>
<tr>
<td>CVE-2011-1260</td>
<td>IE 6</td>
<td>Mem. Corrupt</td>
<td>mshtml1.dll</td>
<td>yes</td>
</tr>
<tr>
<td>CVE-2005-1790</td>
<td>IE 6</td>
<td>Mem. Corrupt</td>
<td>mshtml1.dll</td>
<td>yes</td>
</tr>
<tr>
<td>CVE-2008-0348</td>
<td>coolplayer</td>
<td>Stack Overflow</td>
<td>core exe</td>
<td>yes</td>
</tr>
<tr>
<td>CVE-2010-5081</td>
<td>RM-MP3</td>
<td>Stack Overflow</td>
<td>core exe</td>
<td>yes</td>
</tr>
<tr>
<td>OSVDB-83362</td>
<td>urlhunter</td>
<td>Stack Overflow</td>
<td>core exe</td>
<td>yes</td>
</tr>
<tr>
<td>CVE-2007-1195</td>
<td>XM ftp</td>
<td>Format String</td>
<td>core exe</td>
<td>yes</td>
</tr>
<tr>
<td>OSVDB-82798</td>
<td>ComSndFTP</td>
<td>Format String</td>
<td>core exe</td>
<td>yes</td>
</tr>
</tbody>
</table>
Outline

- Motivation
- Intuition & Design
- Implementation
- Security Enhancement
- Evaluation
- Conclusion
Conclusion

- Performance and compatibility issues limit the adoption of Control Flow Integrity.

- CCFIR is a lightweight CFI solution providing a strong guarantee, can block control-flow hijacks like ROP and ret2libc.

- CCFIR overcomes the performance and compatibility issues, and make it a practical solution.
Thanks!