Compiler-assisted Code Randomization

Hyungjoon Koo
Vasileios P. Kemerlis
Yaohui Chen
Michalis Polychronakis
Long Lu

Stony Brook University
Northeastern University
BROWN
Introduction

❖ The need for fine-grained code randomization

• Code reuse/ROP has been the de facto exploitation technique after the introduction of W^X memory protections
• ASLR provides *insufficient* mitigation
  ✓ Defeated by information leaks
  ✓ Fixed relative distances between functions and basic blocks
• Code randomization makes gadget locations unpredictable
• The advanced JIT-ROP exploitation technique can bypass fine-grained code randomization
  ✓ Recent execute-only memory (XOM) protections prevent JIT-ROP
  ✓ XOM relies on fine-grained code randomization to be effective
Motivation

- Despite decades of research, code randomization has not seen widespread adoption
  - Diversification by *end users*
    - Source code level: recompilation
    - Binary level: static/dynamic binary rewriting
    - In both cases, the burden is placed on *end users*: responsible for carrying out a complex and cumbersome process
  - Diversification by *software vendors*
    - Appstores could deliver a randomized variant to each user
    - Increased cost for generating (compute power) and distributing (no caching/CDNs) randomized copies
Key factors for making code randomization practical

Transparency

Software distribution and installation should remain the same
Key factors for making code randomization practical

- **Transparency**: Software distribution and installation should remain the same

- **Reliability**: Binary rewriting requires ultimate precision

**Static Rewriting**
- Correctness (e.g., indirect transfers)
- Incomplete code coverage

**Dynamic Rewriting**
- Performance degradation
- Compatibility issues
Motivation: Key Factors (3/3)

Key factors for making code randomization practical

- **Transparency**: Software distribution and installation should remain the same
- **Reliability**: Binary rewriting requires ultimate precision
- **Compatibility**: Randomized binaries should remain fully functional

**Software operations**
- Crash reporting
- Code signing
- Patching
- Updating

**Code constructs**
- Shared object
- Exception handling
- Lazy binding
- Full/partial RELRO
- Compiler optimizations
- Linking time optimizations
- Position independent code
- Control flow integrity
## Prior Works (1/2)

### Comparison

<table>
<thead>
<tr>
<th>Research</th>
<th>Needed Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficient Techniques for Comprehensive Protection (USENIX ‘05)</td>
<td>Source code</td>
</tr>
<tr>
<td><em>G-Free</em> (ACSAC ‘10)</td>
<td>Source code</td>
</tr>
<tr>
<td><em>ILR</em> (Oakland ‘12)</td>
<td>Disassembly</td>
</tr>
<tr>
<td><em>Orp</em>: smashing gadgets (Oakland ‘12)</td>
<td>Disassembly</td>
</tr>
<tr>
<td>Binary Stirring (CCS ‘12)</td>
<td>Disassembly</td>
</tr>
<tr>
<td><em>XIFER</em>: gadge me (CCS ‘13)</td>
<td>Disassembly, Relocation</td>
</tr>
<tr>
<td><em>Oxymoron</em> (USENIX ‘14)</td>
<td>Disassembly</td>
</tr>
<tr>
<td><em>Readactor</em> (Oakland ‘15)</td>
<td>Source code</td>
</tr>
<tr>
<td><em>Shuffler</em> (OSDI ’16)</td>
<td>Symbol, Relocation</td>
</tr>
<tr>
<td><em>Selfrando</em> (PETS ‘16)*</td>
<td>Relocation, Function boundary</td>
</tr>
</tbody>
</table>
Prior Works (2/2)

❖ SoK: Automated software diversity (Oakland ‘14)

“Naturally, the research in software diversity can be extended; we point out several promising directions. There is currently a lack of research on hybrid approaches combining aspects of compilation and binary rewriting to address practical challenges of current techniques.”
Can we achieve the following goal?

Reliably randomize binaries in a transparent way, compatible with existing software.
Overview: Compiler-assisted Code Randomization

❖ Compiler-rewriter cooperation

Source Code → IR

- Compiler (LLVM)
- Assembler

Object Files

- Linker (gold)

Master Binary

Legacy Channel

Master Binary

Software Vendor

Randomized Binary

Rewriter

Metadata

Compiler-assisted Code Randomization
Transformation-assisting Metadata

❖ Precise object boundaries for transformation

- ELF Hdr
- ... 
- .plt
- .text (code)
- .data
- ...
- Sec Hdr

Executable

CRT

Obj#1

Fun#1

Fun#2

Fun#M

BBL#1

BBL#2

BBL#N
Transformation-assisting Metadata: Code Generation in LLVM Backend (1/2)

- **MC Framework** uses an internal hierarchical structure: Machine Function (MF), Machine Basic Block (MBB), Machine Instruction (MI)

Source Code
- Compiler
- Assembler

Object Files
- Linker

Diagram:
- **MF#1**
  - **MBB#1**
    - **MI#1**
    - **MI#2**
    - **MI#3**
  - **MBB#2**
    - **MI#4**
    - **MI#5**
    - **MI#6**
    - **MI#7**
  - **MBB#3**
    - **NOP**
    - **NOP**
    - **MI#8**
    - **MI#9**

Branch Instructions
- **MI#8**
- **MI#9**

 NOP Code
- **NOP**
- **NOP**

Compiler-assisted Code Randomization
MCAssembler treats code as a series of fragments:
Data Fragment (DF), Relaxable Fragment (RF), Alignment Fragment (AF)
• No high-level structure (MF or MBB)

Source Code
• Compiler
• Assembler

Object Files
• Linker

Transformation-assisting Metadata: Code Generation in LLVM Backend (2/2)

Label the parent MBB/MF per each MI
MCAssemble treats code as a series of fragments
• As layout is being determined, both MBB/MF sizes are decided.

Fragment Types
- DF: Data
- RF: Relaxable
- AF: Alignment
Transformation-assisting Metadata: Tracking Emitted Bytes in the Final Layout (2/3)

- **MCAsssembler** treats code as a series of fragments
  - Branch instructions form relaxable fragments (RF).

**Fragment Types**
- DF: Data
- RF: Relaxable
- AF: Alignment
**MCAssembler** treats code as a series of fragments
- NOP byte(s) are counted as part of MBB or MF in size.

**Fragment Types**
- DF: Data
- RF: Relaxable
- AF: Alignment

**Source Code**
- Compiler
- Assembler

**Object Files**
- Linker

**Transformation-assisting Metadata:**
- Tracking Emitted Bytes in the Final Layout (3/3)

<table>
<thead>
<tr>
<th>ELF Hdr</th>
<th>...</th>
<th>.text</th>
<th>...</th>
<th>Sec Hdr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metadata</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Transformation-assisting Metadata: Fixup Information (1/2)

- Fixup information can be resolved:
  - At compilation time → **MISSING**
  - At link time → relocations in object files
  - At load time → relocations in final executable

<table>
<thead>
<tr>
<th>Byte Code</th>
<th>Instructions</th>
<th>Byte Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 89 DF</td>
<td>mov rdi, rbx</td>
<td>48 89 DF</td>
</tr>
<tr>
<td>4C 89 F6</td>
<td>mov rsi, r14</td>
<td>4C 89 F6</td>
</tr>
<tr>
<td>E8 49 43 00 00</td>
<td>call someFunc</td>
<td>E8 8D 30 06 00</td>
</tr>
<tr>
<td>EB 0D</td>
<td>jmp short 0xD</td>
<td>EB 0D</td>
</tr>
<tr>
<td>49 39 1C 24</td>
<td>cmp [mh],ctrl</td>
<td>49 39 1C 24</td>
</tr>
<tr>
<td>48 83 C4 08</td>
<td>add rsp, 8</td>
<td>48 83 C4 08</td>
</tr>
</tbody>
</table>

Relocation Table for Object File

<table>
<thead>
<tr>
<th>TYPE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_X86_64_PC32 someFunc</td>
<td>0x4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Fixup information relationships

- Set $A = \{\text{Fixups resolved at compilation time}\}$
- Set $B = \{\text{Fixups resolved at link time}\}$
- Set $C = \{\text{Fixups resolved at load time}\}$

Offset from section base
- Dereferencing size
- Value is absolute or relative
## Metadata Summary

<table>
<thead>
<tr>
<th>Metadata</th>
<th>Collected Information</th>
<th>Collection time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Layout</strong></td>
<td>Section offset to first object</td>
<td>Linking</td>
</tr>
<tr>
<td></td>
<td>Section offset to <code>main()</code></td>
<td>Linking</td>
</tr>
<tr>
<td></td>
<td>Total code size for randomization</td>
<td>Linking</td>
</tr>
<tr>
<td><strong>Basic Block (BBL)</strong></td>
<td>BBL size (in bytes)</td>
<td>Linking</td>
</tr>
<tr>
<td></td>
<td>BBL boundary type (BBL, FUN, OBJ)</td>
<td>Compilation</td>
</tr>
<tr>
<td></td>
<td>Fall-through or not</td>
<td>Compilation</td>
</tr>
<tr>
<td></td>
<td>Section name that BBL belongs to</td>
<td>Compilation</td>
</tr>
<tr>
<td><strong>Fixup</strong></td>
<td>Offset from section base</td>
<td>Linking</td>
</tr>
<tr>
<td></td>
<td>Dereference size</td>
<td>Compilation</td>
</tr>
<tr>
<td></td>
<td>Absolute or relative</td>
<td>Compilation</td>
</tr>
<tr>
<td></td>
<td>Type (c2c, c2d, d2c, d2d)</td>
<td>Linking</td>
</tr>
<tr>
<td></td>
<td>Section name that fixup belongs to</td>
<td>Compilation</td>
</tr>
<tr>
<td><strong>Jump Table</strong></td>
<td>Size of each jump table entry</td>
<td>Compilation</td>
</tr>
<tr>
<td></td>
<td>Number of jump table entries</td>
<td>Compilation</td>
</tr>
</tbody>
</table>
Metadata Consolidation at Link Time

- Linker consolidates per-object metadata
  - Constructing the final layout
  - Resolving symbols
  - Updating relocation information
  - *Merging/adjusting collected metadata from each object file*

![Diagram of Metadata Consolidation](image)

- Object Files
  - Linker (gold)

- Binary Executable

- Metadata

- ✔ Layout
- ✔ BBL size
- ✔ Fixup Offset
Client-side Randomization (1/2)

❖ Binary rewriting at installation time

(a) Parse raw data
(b) Build layout

Master Binary → Rewriter → Variant

Integrated Metadata

Binary

OBJ_0 OBJ_1 OBJ_i

FUN_0 FUN_1 FUN_2 FUN_j

BBL_0 BBL_1 BBL_2 BBL_k

Fixup_0 Fixup_1 Fixup_2 Fixup_x

Compiler-assisted Code Randomization
Client-side Randomization (2/2)

- **Binary rewriting at installation time**

  ![Diagram]

  - **Master Binary**
  - **Rewriter**
  - **Variant**

  - **Fixup Info Rewriting**
  - **Symbol Info Rewriting**
  - **Exception Handling Info Rewriting**

  - **.text**
  - **.data**
  - **.rodata**
  - **.data.rel.ro**
  - **.init_array**
  - **.rela.dyn**
  - **.dynsym (.symtab)**
  - **.eh_frame**
  - **.eh_frame_hdr**

  ![Flowchart]

  1. **(c) Perform rand.**
  2. **(d) Rewrite binary**
  3. **Output**
Evaluation (SPEC2006)

- 0.28% runtime overhead on avg., 11.5% inc. in file size
What we have not talked about

- Challenges for enabling robust/practical transformation
  - How to handle jump table entries
  - Support for various software constructs
    - Exception handling
    - Inline assembly
    - LTO (Linking time optimization)
    - CFI (Control flow integrity)
  - Randomization constraints
  - Optimized metadata serialization
  - Implementation pitfalls and current limitations of CCR

Please read our paper!
Wrap-up

- Compiler-assisted Code Randomization
  - Function and *basic block* level permutation
  - Facilitated by *transformation-assisting metadata* stored within augmented executables
  - Transparency, reliability, and compatibility
  - Integration with Apt package manager

Open-source prototype:
https://github.com/kevinkoo001/CCR
Backup: Randomization Constraints
Backup: Jump Table Entry and Metadata

- Size of each entry and the # of entries in jump table

<table>
<thead>
<tr>
<th>Section Name</th>
<th>Compiled without PIC/PIE</th>
<th>Compiled with PIC/PIE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Byte Code</td>
<td>Disassembly</td>
</tr>
<tr>
<td>.text</td>
<td>FF 24 D5 A0 39 4A 00</td>
<td>jmp qword [rdx*8+0x4A39A0]</td>
</tr>
<tr>
<td>.rodata</td>
<td>D2 C0 40 00 00 00 00</td>
<td></td>
</tr>
</tbody>
</table>

Compiler-assisted Code Randomization
Backup: Exception Handling

Diagram showing CIE (0) and FDE (0) structures within an EH frame, with fields such as length, CIE_id, version, augmentation, address_size, segment_size, initial_instr, and padding. The Rand. Area contains Func(0) to Func(n) blocks. The .text section is also shown.