Correct Relocation: Do You Trust a Mutated Binary?

Drew Bernat
bernat@cs.wisc.edu
Binary Manipulation

• We want to:
  - Insert new code
  - Modify or delete code
  - These operations move program code

• Binaries are brittle
  - Code movement may affect program semantics

• We want to move code without breaking the program
Relocation

- Relocation moves code while maintaining its original execution semantics
  - May radically transform the code
- Does not rely on external information
- Binary tools use relocation extensively
  - Execute original + relocated code (Dyninst)
  - Always execute relocated code (PIN, Valgrind, DynamoRIO, VMWare, DELI)

Relocation is critical for binary manipulation
Relocation Examples

foo:
0x1000: push ebp
0x1001: mov esp, ebp
0x1003: mov 0x8(ebp), eax
0x1006: cmp 0x5, eax
0x1009: ja 0x30
0x100b: call ebx_thunk
0x1011: add ebx, eax

ebx_thunk:
0x2000: mov (esp), ebx
0x2003: ret

0x4000: ja -0x2ff7
...

0x5000: push $0x1011
0x5005: jmp ebx_thunk'

0x5000: push $0x1011
0x5005: jmp ebx_thunk'

ebx_thunk':
0x6000: mov (esp), ebx
0x6003: call map_return
0x6008: ret
Current Approaches

• Strict Relocation
  - Maintains the semantics of each individual instruction
  - Safe in nearly all cases
  - Can impose severe slowdown
  - Trades speed for strictness

• Ad-Hoc Relocation
  - Emit more efficient code by partially emulating the original code
  - Pattern matching may fail and generate incorrect code
  - Trades strictness for speed
## Benefits and Drawbacks

<table>
<thead>
<tr>
<th></th>
<th>Safe</th>
<th>Fast</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strict Relocation</strong></td>
<td><strong>Good</strong></td>
<td><strong>Poor</strong></td>
</tr>
<tr>
<td><strong>Ad-Hoc Relocation</strong></td>
<td><strong>Poor</strong></td>
<td><strong>Good</strong></td>
</tr>
<tr>
<td><strong>Partial Relocation</strong></td>
<td><strong>Good</strong></td>
<td><strong>Good</strong></td>
</tr>
</tbody>
</table>
Our Approach

• Develop a formal model of relocation
  - Reason about the relationship of the moved code to:
    • Its new location
    • Surrounding code
  - Based on semantics of code instead of pattern-matching against syntax

• Strictness of emulation based on demands of the moved code (and surrounding code)
Effects of Code Movement

- **Moving** certain instructions will change their semantics
  - Relative branches, loads, stores
  - We call these *PC referencing* instructions

- **Patching tools** *overwrite* program code
  - Other code that references this code will be affected

- Relocation may affect non-relocated code!
foo:
0x1000: push ebp
0x1001: mov esp, ebp
0x1003: mov 0x8(ebp), eax
0x1004: cmp 0x5, eax
0x1006: ja 0x30
0x1008: call ebx_thunk
0x100d: add ebx, eax
0x100f: mov (eax), edx
0x1011: jmp edx

- No change
- Relative branch
- Relative load
- Branch to result of relative load
Effects of Overwriting Code

main:
  ...
0x0050: call foo
  ...

foo:
0x1002: jmp 0xf000
  ...

bar:
  ...
0x2010: mov (0x1000), eax
0x2015: add (0x1004), eax
Approach

• Model
  - Relocated code, surrounding code
  - Properties of code affected by relocation

• Analysis
  - Deriving these properties from the binary

• Transformations
  - How do we modify code to run correctly and efficiently?
Model

• Define properties of code that relocation affects
  - PC referencing
  - Dependence on moved or overwritten code
• A single instruction may have multiple properties
• These combinations of properties determine how to relocate the instruction
  - Or compensate non-relocated instructions
Program Regions

- \( R = \{i_i, \ldots, i_j\} \)
  - Instructions to relocate
- \( A = \{i_k, \ldots, i_l\} \)
  - Analyzed region
  - Surrounds \( R \)
- \( U = \{i_0, \ldots, i_n\} - R - A \)
  - Unanalyzed region
  - Models limits of analysis
- \( R' = \{i_p, \ldots, i_q\} \)
  - Relocated instructions
Properties of Moved Code

- **Direct (REF)**
  - Control (REF$_C$)
  - Data (REF$_D$)
  - Predicate (REF$_P$)

- **Indirect (REF*)**
  - Control (REF$_C^*$)
  - Data (REF$_D^*$)
  - Predicate (REF$_P^*$)

```plaintext
foo:
0x1000: push ebp
0x1001: mov esp, ebp
0x1003: mov 0x8(ebp), eax
0x1004: cmp 0x5, eax
0x1006: ja 0x30
0x1008: call ebx_thunk
0x100d: add ebx, eax
0x100f: mov (eax), edx
0x1011: jmp edx
```
bool dl_open_check(char *name, void *calladdr)
{
    // Check if the caller is
    // from libdl or libc
    bool safe_open = false;
    if (IN(libc_obj, calladdr)
        || IN(libdl_obj, calladdr)
        safe_open = true;
    if (!safe_open) return false;

    // Perform further checks
    ...
}

• Safety check in library load
  - Address of caller passed in
  - Checked against legal callers

• Predicate expressions
Properties of Overwritten Code

- **Control (CF)**
  - Instructions with successors in R
  - \( \{0x0050, 0x1004\}_{CF} \)

- **Data (DF)**
  - Loads from R
  - Stores to R
  - \( \{0x2010, 0x2015\}_{DF} \)

main:

... 
0x0050: call foo
... 
foo:

... 
0x1004: cmp 0x5, eax
0x1006: ja 0x30
... 
bar:

... 
0x2010: mov (0x1000), eax
0x2015: add (0x1004), eax
Properties Summary

![Diagram showing the relationship between REF, REF*, C, P, D, CF, and DF.]

Correct Relocation
Analysis Overview

1. Choose R and A
   - R: instruction, basic block, function, ...
   - A: how much do we analyze?

2. Identify sources of REF and REF* in R
   - Follow data dependence chains into A and U

3. Determine \{\ldots\}_\text{CF} and \{\ldots\}_\text{DF}
   - Begin with interprocedural CFG and points-to analysis
   - Be conservative and assume incomplete information
REF/REF* Analysis

- Create the Program Dependence Graph
  - Covering R + A

- Identify source instructions

- Follow data dependence edges
  - Into A (or U)

```c
foo:
  ...
  0x1004: cmp 0x5, eax
  0x1006: ja 0x30
  0x1008: call ebx_thunk
  0x100d: add ebx, eax
  0x100f: mov (eax) edx
  0x1011: jmp edx
```
Transformation Goals

• We want to emulate the smallest set of original code semantics

• Transformations must maintain the properties determined by analysis
  - But any others are not required

• Our approach: define transformations for each combination of properties
Granularity of Relocation

• Current methods relocate by instruction
  - Maintain equivalence at the instruction boundary

• “Unobserved” results

• Relocate instructions as a group
  - Maintain boundary semantics of the code
  - Reduce complexity and improve efficiency
Partial Relocation Example

```
0x5000: push $0x1011  
0x5005: jmp ebx_thunk'  
0x500a: add ebx, eax  
|                     
|                       
| REF_C                
|                     
| ebx, eax             
| REF_D               

ebx_thunk':
0x6000: mov (esp), ebx
0x6003: call map_return
0x6008: ret
```

Correct Relocation
Research Plan

• This work is preliminary
  - Properties are defined
  - Analysis requirements are defined

• Still a lot to do
  - Determine transformations
  - Implementation in Dyninst
  - Performance analysis
Questions?
Relocating a Jump Table

foo2:
0x1008: jmp <0xf008>
0x1012: <jump table data>
0x1040: jmp <0xf040>
0x1060: jmp <0xf060>
0x1080: jmp <0xf080>
0x10a0: jmp <0xf0a0>

foo3:
0xf008: call ebx_thunk
0xf00d: add ebx, eax
0xf00f: mov (eax, 4), ebx
0xf011: jmp ebx
0xf012: <relocated jump table data>
0xf040: <reloc case 1>
0xf060: <reloc case 2>
0xf080: <reloc case 3>
0xf0a0: <reloc case 4>
Complex Instructions

• Instructions may have multiple properties
  - Example: a relative branch in R may be both CF and $\text{REF}_C$
• Some overlap is due to implicit control flow
  - Instructions in R may be tagged as $\text{REF}_C$ due to fallthrough to next instruction
• We can model instructions as combinations of independent operations if necessary
  - Separate out the “next PC” calculation