SAFER: Efficient and Error-Tolerant Binary Instrumentation†

Soumyakant Priyadarshan, Huan Nguyen, Rohit Chouhan and R. Sekar
Stony Brook University

August 11, 2023

†This work was supported by an ONR grant N00014-17-1-2891 and in part by NSF grants 1918667 and 2153056.
Why binary instrumentation?

- **Binary instrumentation** → Modify program without source code.
- Enables unique capabilities:
  - **Security** without source code:
    - Harden deployed software (almost always binary code)
    - Detect vulnerabilities (fuzzing)
    - Analyze malware
  - **Program profiling**: Identify performance bottlenecks.
  - **Debugging**: Bugs that manifest only at runtime (e.g., Valgrind)
Key challenges

- **Robustness**: Handling complex binaries
Key challenges

- **Performance**

  - **High Overhead**
    - Runtime translation
      - Pointer Map <old ptr, new ptr>
    - Runtime
      - Translate (RAX);
      - JMP *RAX;
  
  - **Error Prone**
    - Static translation
      - Change pointers
    - Runtime
      - No additional code executed
Can we combine above two?

Yes!

SAFER: Static pointer encoding + runtime translation $\approx 2\%$ overhead
SAFER’s pointer translation

- **Pre-translate** high confidence code pointers
- **Runtime AT** for others.
- *How to distinguish at run time?*
SAFER’s pointer translation

- **Pre-translate** high confidence code pointers
- **Runtime AT** for others.
- **How to distinguish at run time?**

```
1 BIT encoding

Unused
0 0 0 0 7 F 9 4 7 1 0 X X X X X

Flipped MSB
8 0 0 0 7 F 9 4 7 1 0 X X X X X

Run time checking
If encoded(target) then
target = decode(target)
Else
target = translate(target)
```
Error handling

- Data pointer misclassified as code pointer?
- Flipped MSB $\implies$ crash on read
Error handling

- Undetected pointer arithmetic?
  - Code pointer used to compute another code pointer
Error handling

- **Undetected pointer arithmetic?**
  - Code pointer used to compute another code pointer

- **New multiplicative encoding:**
  
  - \( A, B: 64 \) bit odd numbers
  - \( A \times B = 1 \)

  \[
  \text{Encode:} \quad P_{\text{enc}} = P \times A \\
  \text{Decode:} \quad P_{\text{enc}} \times B = P
  \]

  \[
  P_{\text{enc}} + X \quad \rightarrow \quad (P_{\text{enc}} + X) \times B = \text{invalid}
  \]
Jump tables

- Do programs use computed code pointers?
**Jump tables**

- Do programs use computed code pointers?
- **YES:** C/C++ switch-case → Jump tables

```
Jump_table: 100 200 500 300
```

```
target = Jump_table[idx];
Jmp *target;
```

- **Identify:** static analysis (Dyninst, Egalito, Ddisasm, etc).
Jump tables

- Do programs use computed code pointers?
- **YES**: C/C++ switch-case → Jump tables

![Jump Table]

- **Identify**: static analysis (Dyninst, Egalito, Ddisasm, etc).

**Our Contribution**: Safe jump table translation

- As opposed to best effort
Translating jump tables

Challenge:

- **Runtime translation** $\Rightarrow$ high overhead
- **In-place update**: Incorrect bound $\Rightarrow$ overwrite other data

Translating jump tables

**Challenge:**
- **Runtime translation** $\Rightarrow$ high overhead
- **In-place update:** Incorrect bound $\Rightarrow$ overwrite other data

**Solution:**
- Original data intact.
- Recreate jump table
- Change jump table access.
Translating jump tables

- **Challenge:**
  - Runtime translation $\implies$ high overhead
  - In-place update: Incorrect bound $\implies$ overwrite other data

- **Solution:**
  - Original data intact.
  - Recreate jump table
  - Change jump table access.
  - **challenge:** Other use of jump table.
  - **Example:** Jump table address used for accessing other data
Safe jump table analysis

- **Taint analysis to detect other use:**
  - Memory dereferencing
  - Move to heap
  - Call argument
  - Return value

- **83% jump tables SAFE → avoid runtime translation**
Evaluation overview

- **Experimental evaluation**
  - *fail-crash*: Can SAFER detect errors at runtime?
  - *Performance*: What is the performance cost of SAFER’s pointer translation approach?
  - *Functionality*: Can SAFER instrument real world applications?
Fail crash evaluation: Coreutils with embedded data

- Linux coreutils: `ls`, `cat`, `cp`, etc.

<table>
<thead>
<tr>
<th>Code pointer validation method</th>
<th>Success rate</th>
<th>Safe failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (always use AT)</td>
<td>105/105</td>
<td>NA</td>
</tr>
<tr>
<td>+ Instruction boundary</td>
<td>43/105</td>
<td>62/105</td>
</tr>
<tr>
<td>+ ABI validation</td>
<td>74/105</td>
<td>31/105</td>
</tr>
<tr>
<td>+ Function prologue matching</td>
<td>105/105</td>
<td>NA</td>
</tr>
</tbody>
</table>
Safer optimizations

- **Full AT**: Fully compatible.
  - No pointers changed (including return addresses)
- **Full enc**: All pointers encoded.
- **RA opt**: Use current return addresses.
  - C++ exception incompatible
  - Update exception metadata
- **Safe jump table**: Recreate jump tables
Functionality evaluation

- 16 real world applications with 500+ shared libraries (*Size: 473MB*).
  - gimp, evince, gedit, ffmpeg, clang, Python, etc

- 6 applications use libraries with *embedded data*.
  - libgcrypt, libgnutls, libavcodec, libcrypto
Summary

- SAFER effectively combines **pointer encoding** with **runtime address translation** to get low overhead of $\approx 2\%$.

- SAFER’s novel **pointer encoding** facilitates runtime error detection (fail-crash).

- SAFER’s **safe jump table** analysis helps improve performance without compromising correctness and safety.

**Artifact URL:** http://seclab.cs.sunysb.edu/soumyakant/safer
THANK YOU!!!
Error handling

- What if we have error?
Error handling

- **What if we have error?**
- **Encoding** $\Rightarrow$ *crash* when used.
- **fail-crash** over unexpected behavior
  - Prevent data loss or security failure
  - Identify error prone module
  - **FIX:** full address translation on the module
Why multiplicative encoding?

- Why not a 15 bit checksum in leading 16 bits?
  - Time-consuming to compute
  - Requires many unused bits
  - Non-negligible rate of undetected failures
Why multiplicative encoding?

- Why not a 15 bit checksum in leading 16 bits?
  - Time-consuming to compute
  - Requires many unused bits
  - Non-negligible rate of undetected failures

- Benefits of our approach
  - Faster: Just one instruction: MULX
    - Does not affect CPU flags
  - Negligible rate of undetected arithmetic
Safe jump table analysis improvement

- No analysis...all jump tables marked safe: 1.2% overhead.
- Without function signature analysis: 55% safe (reported in paper)
  - Approx. 2% overhead
- Function signature analysis:
  - Helps improve call argument identification accuracy
  - More jump tables marked as safe: 83%
  - Approx. 1.5% performance overhead
Pointer classification

- SAFER’s default: ABI validation (2% overhead)
- Function prologue matching: \(\approx 5\%\) overhead.
Instrumentation overview

- **Constant pointers**
  - PIE
  - Is code pointer?
    - Yes: Encode
      - Data access
    - No: Non-PIE
      - Unmodified
        - Missed pointers
          - Data access
            - Yes: Validate
            - No: Old code/data replica
  - Non-PIE
    - Used as code pointer
      - Fail crash

- **Computed pointers**
  - Jump tables (Switch-case)
  - Safe to transform?
    - Yes: Recreated jump tables
    - No: Unmodified
      - Missed pointers
      - If encoded(target) then target = decode(target)
      - Else target = translate(target)

- **Valid**
  - PASS
Tolerating disassembly false positives

- **Data misinterpreted as code**
- Replication based instrumentation (PSI, BinStir, etc)

![Diagram showing original and instrumented ELF sections]

- **Original**
  - Elf headers
  - .text
  - .rodata
  - .data
  - .bss

- **Instrumented**
  - Elf headers
  - Old .text
  - .rodata
  - .data
  - .bss
  - New .text

Unchanged for data access

Instrumented code
Identifying constant pointers

- **Address taken functions**
  - **PIE:** Relocation.
  - **Non-PIE:** Scan code/data sections for 4/8 byte constants
Address translation

- Two level hashing scheme:
  - Global hash (GTT): <4K aligned Page, LTT>
    - Runtime construction
  - Local hash (LTT): Per-module translation <Old Pointer, New Pointer>

- Customized loader for above.
Exceptional cases!!

- **Return addresses used as indirect jump target**
  - Longjmp, C++ exception handling
  - **Handling**: Return addresses added to translation table

- **Supporting stack unwinding**
  - **Special metadata**: Return address dependent
  - push old RA on stack $\Rightarrow$ performance heavy
  - **Our approach**: Sync metadata with new RA
Effect of compiler optimizations

- **Average across all 6 optimizations:** 2.3%

**SPEC 2017**

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>O0</td>
<td>1.5</td>
</tr>
<tr>
<td>O1</td>
<td>3.22</td>
</tr>
<tr>
<td>O2</td>
<td>2.2</td>
</tr>
<tr>
<td>O3</td>
<td>1.85</td>
</tr>
<tr>
<td>Ofast</td>
<td>2.4</td>
</tr>
<tr>
<td>Os</td>
<td>2.6</td>
</tr>
</tbody>
</table>