Reassembly Is Hard: A Reflection on Challenges and Strategies

Hyungseok Kim$^{1,2}$, Soomin Kim$^1$, Junoh Lee$^1$, Kangkook Jee$^3$, and Sang Kil Cha$^1$

$^1$ KAIST, $^2$ The Affiliated Institute of ETRI, $^3$ University of Texas at Dallas

{witbring,soomink,junoh,sangkilc}@kaist.ac.kr, kangkook.jee@utdallas.edu

USENIX Security 2023
Static Binary Rewriting Is Imperative to SW Security

Malware Analysis & Binary Testing

SW Hardening (CFI Enforcement, Code Randomization, …)

Code Repair & Binary Debloating
## Four Kinds of Static Binary Rewriting Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Applicable to COTS Binary</th>
<th>Fine-grained Instrumentation</th>
<th>Low Time &amp; Space Overhead</th>
<th>Does it Really Work?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler-assisted static rewriting</td>
<td>😞😊😊😊</td>
<td>😃😊😊😊</td>
<td>😃😊😊😊</td>
<td></td>
</tr>
<tr>
<td>Patch-based static rewriting</td>
<td>😃😊😊😊</td>
<td>😃😊😊😊</td>
<td>😃😊😊😊</td>
<td></td>
</tr>
<tr>
<td>Table-based static rewriting</td>
<td>😃😊😊😊</td>
<td>😃😊😊😊</td>
<td>😃😊😊😊</td>
<td></td>
</tr>
<tr>
<td>Reassembly-based static rewriting</td>
<td>😃😊😊😊</td>
<td>😃😊😊😊</td>
<td>😃😊😊😊</td>
<td>😃😊😊😊</td>
</tr>
</tbody>
</table>

- **Compiler-assisted static rewriting**
- **Patch-based static rewriting**
- **Table-based static rewriting**
- **Reassembly-based static rewriting**

---

*Source: KAIST*
Reassembly Is Error-Prone!

Symbolization errors produce a semantically incorrect binary

```
foo:
push ebp
lea eax, bar  

bar:  
.byte 0x20, 0x12, 0x00, 0x00  
.long foo
```

```
L1129:  push ebp  
        lea eax, 0x4010
L1220:  
L4010:  
.long L1220
.long L1129
```

Invalid reference due to missing symbolization
Data corruption due to falsely given symbol
Can We Assure the Correctness of Reassemblers?
Our Key Idea: Differential Testing

Comparing compiler-generated assembly line with reassembly-generated assembly line to identify errors

Challenges?
Challenge #1. Assembly Code Matching

Finding **the matched assembly code** is challenging due to the presence of **duplicate function bodies** and the discrepancies in opcodes sequence.

```
$ objdump -d gcc | grep ...

00000000003ca453 <analyze_function>:
  3ca453:  push    rbp
  3ca454:  mov      rbp, rdi

00000000003ce07e <analyze_function>:
  3ce07e:  push    r15
  3ce080:  push    r14

00000000003d028f <analyze_function>:
  3d028f:  push    r15
  3d0291:  push    r14
```

```
analyze_function:
    .loc 1 1946 1 is_stmt 1 view .
    .loc 1 1947 3 view .LUV1207
    .loc 1 1946 1 is_stmt 0 view .LUV1208
    push    rbp
    mov     rbp, rdi

```

**Duplicate function bodies**

**Opcode sequence mismatch**

```
804b5c7:  sub    DWORD PTR [ebp-0x2a4], 0x1
804b5ce:  jmp    804b772

804b5d3:  nop
804b5d4:  lea    esi, [esi+esi*1+0x0]
804b5d8:  sub    esp, 0x8
804b5db:  push   DWORD PTR [ebp-0x2cc]
```

---
Challenge #2. Restoring Symbolic Expressions in Data Section

Not every data value has a debugging symbol.
Challenge #3. Comparing Labels

Same labels can have different representation for each tool

**Compiler-generated assembly**

```
.L4984:
    lea  rdx,[rip + __FUNCTION__.10544 ]

.L4895:
    mov  rax, [rdx+0x8]

__FUNCTION__.10544:
    .string "reg_overlap_mentioned_p"

.L4896:
    .long  .L4895 - .L4896
    .long  .L4894 - .L4896
```

**Reassembler-generated assembly**

```
.L_2c7758:
    lea  rdx,[rip + .L_3c7750 ]

.L_2c8204:
    test  eax, eax

.L_3c7750:
    .string "reg_overlap_mentioned_p"

.L_3c75cc:
    .long  .L_2c8204 - .L_3c53e0
    .long  .L_2c7758 - .L_3c75cc
```
Solution for Assembly Code Matching (C1)

The approach for identifying the matched function

- Search for functions by comparing **opcode sequence** with **debug info**.
- Permit non-matching, specifically for no-op instructions

```
0000000000516820 <analyze_function>:
516820: push rbp
516821: mov rbp, rdi
516824: push rbx
516825: sub rsp, 0x8
516829: mov rax, QWORD PTR [rdi]
51682c: mov rdi, QWORD PTR [rax+0xb8]
```

[ Disassembled code ]

```
        .loc 1 1946 1 is_stmt 1 view -0
        .loc 1 1947 3 view .LVU921
        .loc 1 1946 1 is_stmt 0 view .LVU922
        push rbp
        mov rbp, rdi
        push rbx
        sub rsp, 0x8
        mov rax, QWORD PTR [rdi]
        mov rdi, QWORD PTR [rax+0xb8]
```

[ Compiler-generated assembly ]
Solution for Symbolic Expression
Restoring in Data Section (C2)

The method for calculating data addresses
- **Search for instructions** that references the local symbols
- Locate the corresponding instruction in binary & calculate data address

<table>
<thead>
<tr>
<th>ASM</th>
<th>Disassembled Code</th>
</tr>
</thead>
</table>

; code section
mov    eax, [eax * 4 + Lswitch.table.convert_move]
jmp   .LBB8_169

; data section
.Lswitch.table.convert_move:
.long libfunc_table
.long libfunc_table+4
.long libfunc_table+8

[ Compiler-generated Assembly ]

; code section
80d248e:  mov    eax, [ eax * 4+0x8238874 ]
80d2495:  jmp    80d255c

; data section
8238874:  .long 0x08297e08
8238878:  .long 0x08297e0c
823887c:  .long 0x08297e10

[ Disassembled Code ]
Solution for Label Comparisons (C3)

The approach for normalizing assembly lines
- Examine the definition (address) of symbolic label
- Convert labels to have **normalized names** with the corresponding address

**Compiler-generated assembly**
- Address: 0x3c75cc
- Normalized Expression: `.long L2ca3f0-L3c75cc`

**Reassembler-generated assembly**
- Address: 0x3c75cc
- Normalized Expression: `.long L2c8204-L3c53e0`
Design of REASSESSOR

1. Concatenate compiler-generated assembly files
2. Map assembly line to binary address (Solution for C1, C2)
3. Normalize assembly code (Solution for C3)
4. Identify & classify errors

Preprocessing
- Source Code
  - Compiler
    - CONCAT
      - Concatenate compiler-generated assembly files
      - Normalize assembly code
      - Map assembly line to binary address
    - STRIP
      - Remove debug symbol
  - Reassembler
    - β'
      - β
      - α_c
      - α_r

Reassessor
- Normalizer
  - NORALIZER
    - α_c'
    - α_r'
    - α_r''
  - ADDRMAPPER
    - α_c''
  - DIFFER
    - Bugs
Experimental Setup

• Dataset: **14,688 Binaries**
  - Packages: GNU Coreutils v8.30, GNU Binutils v2.31.1, SPEC CPU 2006 v1.1
  - Compilers: GCC v7.5.0, Clang v12.0
  - Linker: GNU ld v2.30, GNU gold v1.15
  - Architectures: Intel x86, x86-64
  - Optimization levels: O0, O1, O2, O3, Ofast, Os

• Reassembly Tools
  - **Ramblr** (commit 613562, Apr. 2022): *Only support non-PIE binaries*
  - **RetroWrite** (commit 613562, Apr. 2022): *Only support x86-64 PIE binaries*
  - **Ddisasm** v1.5.3 (docker a803c9, Apr. 2022): *Support all binaries*
Research Questions

• RQ1. Can the current state-of-the-art reassemblers always produce compilable assembly files?
• RQ2. How accurate is reassembler-generated code?
• RQ3. Can the current state-of-the-art reassemblers always soundly reassemble x86-64 Position Independent Executable (PIE) binaries?
RQ1. Can the Current SOTA Reassemblers Always Produce Compilable Assembly Files?

* No.

* Reassembly tools emit **compilable code only for 91.6% binaries**
  - Reassembly tools **failed** to reassemble **2.2% of the binary files**
  - **6.2% of reassembler-generated files** were **non-compilable** due to syntax errors and undefined label references

<table>
<thead>
<tr>
<th></th>
<th>Ramblr</th>
<th>RetroWrite</th>
<th>Ddisasm</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Succeed binaries</td>
<td>6,191</td>
<td>3,497</td>
<td>13,850</td>
<td>23,538</td>
</tr>
<tr>
<td>Total tried binaries</td>
<td>7,344</td>
<td>3,672</td>
<td>14,688</td>
<td>25,704</td>
</tr>
<tr>
<td>Total Succeed Rate</td>
<td>84.3%</td>
<td>95.2%</td>
<td>94.3%</td>
<td>91.6%</td>
</tr>
</tbody>
</table>
RQ2. How Accurate Is Reassembler-generated Code?

3.95% of symbolic expressions was not symbolized (FN), and 3.28% of them was symbolized w/ different expressions (FP)

- 45.11% of symbolization errors are reparable when disallowing data instrumentation
- 54.99% of the symbolization errors are problematic regardless of data instrumentation
Demo: The Impact of Symbolization Errors

```
[hskim@usec2023:~/demo]
# make recompile
gcc -ldl -pthread -m32 reassemble.s -o new_ls
[hskim@usec2023:~/demo]
# ls
Makefile bin new_ls reassemble.s
[hskim@usec2023:~/demo]
# ./new_ls
Makefile bin new_ls reassemble.s
[hskim@usec2023:~/demo]
# ./new_ls -a
  ... Makefile bin new_ls reassemble.s
[hskim@usec2023:~/demo]
# ./new_ls -R
Segmentation fault (core dumped)
```
RQ3. Can the SOTA Reassemblers Soundly Reassemble x86-64 PIE Binaries?

- No, not always.
- In x86-64 PIE binaries, 6.9% of symbolic expressions represented jump table entries, and none of reassemblers perfectly symbolized them.

```
int output=0;
const int bar[]={-0x180,-0x190,-0x1a0,-0x1b0};
void foo(unsigned int input) {
    int *p = (int*)bar - 3;
    switch(input){
        case 0: output = bar[0]; break;
        case 1: output = bar[1]; break;
        case 2: output = bar[2]; break;
        case 3: output = bar[3]; break;
        default: if(input < 7) output = p[input]; break;
    }
    printf("In:%x, Out:%x\n", input, output);
}
```

```
.section .rodata
.LJTI0_0:
.long 0xfffffe80
.long 0xfffffe70
.long 0xfffffe60
.long 0xfffffe50
bar:
.long 0xffffffff
.long 0xffffffff
.long 0xffffffff
.long 0xffffffff
```

```
[section .rodata]
0x830: 80  fe  ff  ff
0x834: 91  fe  ff  ff
0x838: a2  fe  ff  ff
0x83c: b3  fe  ff  ff
0x840: 80  fe  ff  ff
0x844: 70  fe  ff  ff
0x848: 60  fe  ff  ff
0x84c: 50  fe  ff  ff
```

[ Source code in C ] [ Compiler-generated assembly ] [ Disassembled Code ]
Contributions to Improving Reassemblers

We made PR and issues to resolve the errors we found

- Special Thanks to Fish Wang and Antonio Flores-Montoya

Hi Fish,
I am sending you an email to ask a question about Ramblr. My research team is doing some research about reassembly,
...
My student created a GitHub issue here, too: https://github.com/angr/patcherex/issues/39

Thank you in advance!
Best wishes,
Sang Kil

Hi Sang Kil,
Good to hear from you!
I have answered your students’ questions on GitHub.
...
Please send more questions my way if Reassembler fails on any binaries in your benchmark!
Also, let me know if you need more insights about reassembling in general.

Best,
Fish
Open Science

https://github.com/SoftSec-KAIST/Reassessor
Conclusion

• We propose a formal framework and present REASSESSOR, the first automated system for reassembler testing
  – We publicize REASSESSOR and our benchmark
• Through REASSESSOR, we found various reassembly errors with previously unknown patterns
  – We contributed to improving the state-of-the-art reassemblers
• Lastly, we validated strategies and challenges in reassembly
Question?