Lec 13: Rewriting

IS561: Binary Code Analysis and Secure Software Systems

Sang Kil Cha

Binary Rewriting

Binary Rewriting

Binary Rewriting = Static Binary Instrumentation

Given a binary, statically instrument it in such a way that the rewritten binary will run as is while the instrumentation code is executed.

Why Binary Rewriting Matters?

Because it is extremely fast and efficient compared to other dynamic instrumentation techniques.

Why Binary Rewriting is Difficult?

```
// func1:
0x1100: push rbp
0x1103: mov rbp, rsp
0x1107: sub rsp, 0x50
...

// func2:
0x1200: push rbp
0x1203: mov rbp, rsp
...
```

← What happens when we add instrumentation code here?

Fixing Cross-References is Difficult

- Identifying ***dynamically computed references*** is difficult.
  - e.g., `call rax` and `jmp rax`.
- Sometimes addresses (or their offsets) are stored as data.
  - e.g., ***jump tables*** for switch-case statements.
- Correctly identifying code and data from a binary is difficult, which requires ***precisely recovering CFGs***.

Fixing Cross-References is Difficult

- Identifying ***dynamically computed references*** is difficult.
  - e.g., `call rax` and `jmp rax`.
- Sometimes addresses (or their offsets) are stored as data.
  - e.g., ***jump tables*** for switch-case statements.
- Correctly identifying code and data from a binary is difficult, which requires ***precisely recovering CFGs***.

Fixing all cross-references is at least as difficult as precise CFG recovery.

Existing Binary Rewriting Methods

Bypass the challenge with various ideas.

1. Compiler-assisted rewriting.
   - Rewrite binaries, but still rely on the source code.
2. Patch-based rewriting.
   - Rewrite binaries in such a way that references are not changed.
3. Table-based rewriting.
   - Rewrite binaries in such a way that references are not changed.

Compiler-Assisted Rewriters

- Assume the existence of source code or debugging symbols.
- Using ***debugging symbols*** is like a cheat key for binary analysis.
  - We know exactly the addresses of code and data.
- Tools: ATOM, Vulcan, Diablo, PEBIL, etc.

Debugging Symbols

- You can use the `-g` option to produce a binary with full symbolic information.
  - This is almost equivalent to having the source code.
- Even if you do ***not*** use the `-g` option, there still remain partial symbolic information.
- To fully strip off such symbolic information, we use the `strip` command.

Binary analysis assumes no symbolic information. When we say binary analysis, it really means binary analysis for stripped binaries.

Patch-based Rewriters

- ***Key idea***: fix the layout of the binary, so there's no need to fix the references in the binary!
- Tools: Detour[1], DynInst, E9Patch[2], etc.

But, how?

[1] Detours: Binary interception of win32 functions, ***USENIX 1999***
[2] Binary Rewriting without Control Flow Recovery, ***PLDI 2020***

Example: Fixing the Layout

```
// func1:
0x1100: push rbp
0x1103: mov rbp, rsp
0x1107: sub rsp, 0x50

...

// func2:
0x1200: push rbp
0x1203: mov rbp, rsp
...
```

Example: Fixing the Layout

// func1:
0x1100: push rbp
0x1103: mov rbp, rsp
0x1107: ~~sub rsp, 0x50~~ → jmp detour

...

// func2:
0x1200: push rbp
0x1203: mov rbp, rsp
...

detour: ←
// instrumentation routine starts here.
sub rsp, 0x50
jmp 0x110b

Example: Fixing the Layout

```
// func1:
0x1100: push rbp
0x1103: mov rbp, rsp
0x1107: sub rsp, 0x50 → jmp detour

...

// func2:
0x1200: push rbp
0x1203: mov rbp, rsp
...

detour:
// instrumentation routine starts here.
sub rsp, 0x50
jmp 0x110b
```

Several requirements:

- The detoured routines should ***not*** touch the original layout.
- Injected trampolines should ***not*** affect the fall-through code.

Challenges for Patch-based Rewriting

- Instruction-level instrumentation is not easy.
  - e.g., instrumenting one-byte instructions
- If the detour code is far away, we need to use 5-byte jumps.
  - "`e9 10 20 30 40`" means `jmp +0x40302015`.
  - Hence, instrumentation target instructions are likely to be smaller than a jump instruction.

Table-based Rewriters

- Address the applicability challenge of patch-based rewriting methods.
- Create a duplicate copy of a binary, and use an address-translation table at runtime.
  - The table maps an original address to a new address of the copy.
- Tools: PSI, Multiverse[3], etc.

[3] Superset disassembly: Statically rewriting x86 binaries without heuristics, ***NDSS 2018***.

Example: Table-based Rewriting

```
// func1:
0x1100: push rbp
0x1103: mov rbp, rsp
0x1107: call rax; func2
...

// func2:
0x1200: push rbp
0x1203: mov rbp, rsp
...
```

+

```
// func1:
0x11100: push rbp
0x11103: mov rbp, rsp
; instrumentation code
...
0x11117: call table_lookup_rax
0x11119: call rax ; 0x11300
...

// func2:
0x11300: push rbp
0x11303: mov rbp, rsp
...
```

1200 $\mapsto$ 11300

Example: Table-based Rewriting

```
// func1:
0x1100: push rbp
0x1103: mov rbp, rsp
0x1107: call rax; func2
...

// func2:
0x1200: push rbp
0x1203: mov rbp, rs
...
```

+

```
// func1:
0x11100: push rbp
0x11103: mov rbp, rsp
; instrumentation code
...
0x11117: call table_lookup_rax
0x11119: call rax ; 0x11300
...

// func2:
sp
...
```

1200 ↦ 11300

Why do we keep the original copy?

Table-based Rewriting: Pros and Cons

- Instruction-level instrumentation is feasible.
- But, suffers from overhead issues:
  - Time overhead.
  - Space overhead.

All Three Approaches Are Limited

1. Compiler-assisted rewriting.
2. Patch-based rewriting.
3. Table-based rewriting.

Can we do better?

Reassembly

New Approach: Reassembler/Recompiler

Try to address the binary rewriting problem:

- no source code nor debugging symbols (vs. compiler-assisted approaches)
- full support of any instrumentation requirements (vs. patch-based approaches)
- less overhead (vs. table-based approaches).

Resassembly

Key idea: transform a binary into a ***relocatable form*** and then compile it back to another binary.

> But, this means we need to fully resolve dynamically computed references.

Assembly vs. Disassembly

Assembly

```
.LFB0:
...
        cmp DWORD PTR [rbp-0x4], 0x2a
        jle .L2
...
.L2:
        mov eax, DWORD PTR [rbp-0x4]
...
```

Disassembled binary

```
...
0x1134: cmp DWORD PTR [rbp-0x4], 0x2a
0x1138: jle +0x9

...
0x1141: mov eax, DWORD PTR [rbp-0x4]
...
```

We have to make “+0x9” relocatable.

Symbolization

Symbolization is a the process of restoring ***symbolic labels***, used to make a cross-reference in the IR, from the numeric values in the target binary.

Does Reassembly Really Work?

No. There are research attempts, but no complete solution yet.

Uroboros (2015)[4]

- Coined the term “reasembly”.
- Focused on non-PIE binaries.
  - Code section addresses are fixed.
- Assumed an ideal scenario where all numbers in the binary can be classified either as a pointer or a constant based on their values.
  - If a number falls into a section, then it is a pointer.
  - False positives? False negatives?

[4] Reassembleable Disassembling, ***USENIX Security 2015***

Reassembly Tools

Year	Tool	PIE	non-PIE	x86	x86-64
2015	Uroboros	✗	✓	✓	✓
2017	Ramblr	✗	✓	✓	✓
2020	Ddisasm	✓	✗	✓	✓
2020	RetroWrite	✓	✗	✗	✓
2020	Egalito	✓	✗	✗	✓

Tools are starting to focus more on x86-64, PIE binaries. Why?

What Makes PIE Reassembly Easy?

- PIE binaries are position independent by definition.
- Thus, they only use relative addressing.
- Whenever a number is used as an absolute address, it should be marked in the relocation table of the binary, so that the loader can correctly relocate the pointer.
- x86-64 PIE binaries are even easier because they use RIP-relative addressing.

Still Many Symbolization Errors

Source code

```
char buf[16];
int main()
{ return fprintf(stdout, "%s (%p~%p)\n",
    buf, buf, buf+sizeof(buf)); }
```

Compiler-generated assembly

```
main:
    lea r9, [rip + buf +16]
    sub rsp, 8
    mov rdi, QWORD PTR [rip + stdout]
    lea rdx, . [rip + .LC0] ; string
    mov esi, 1 ; flag for __fprintf_chk
    xor eax, eax
    lea r8, [r9 - 16] ; buf
    mov rcx, r8 ; buf
    call     __fprintf_chk@PLT
```

Still Many Symbolization Errors

Source code

```
char buf[16];
int main()
{ return fprintf(stdout, "%s (%p~%p)\n",
    buf, buf, buf+sizeof(buf)); }
```

Compiler-generated assembly

```
main:
    lea r9, [rip + buf +16]
    sub rsp, 8
    mov rdi, QWORD PTR [rip + stdout]
    lea rdx, . [rip + .LC0] ; string
    mov esi, 1 ; flag for __fprintf_chk
    xor eax, eax
    lea r8, [r9 - 16] ; buf
    mov rcx, r8 ; buf
    call    __fprintf_chk@PLT
```

Disassembled assembly

```
00000000000005b0 <main>:
5b0: lea    r9,[rip+0x1a79] # 2030
5b7: sub    rsp,0x8
5bb: mov    rdi,QWORD PTR [rip+0x1a6e] # 2030
5c2: lea    rdx,[rip+0x1ab]
5c9: mov    esi,0x1
5ce: xor    eax,eax
5d0: lea    r8,[r9-0x10]
5d4: mov    rcx,r8
5d7: call   5a0 <__fprintf_chk@plt>
...
2020: # buf
2030: # stdout
```

Another Example

Compiler-generated assembly

```
    .LFB0:
...
      lea rcx, [rip + time_spec + 8]
      lea rcx, [rip + time_spec_string + 16]
...
time_spec_string:
      .quad .LC19
      .quad .LC20
...
```

Disassembled assembly

```
...
0x373c: lea rcx, [rip + 0x10f05]
0x3743: lea rcx, [rip + 0x17ae6]
...
```

Another Example

Compiler-generated **assembly**

```
    .LFB0:
...
        lea rcx, [rip + time_spec + 8]
        lea rcx, [rip + time_spec_string + 16]
...
time_spec_string:
        .quad .LC19
        .quad .LC20
...
```

Disassembled **assembly**

```
...
0x373c: lea rcx, [rip + 0x10f05]
0x3743: lea rcx, [rip + 0x17ae6]
...
```

variable recovery

Research Question

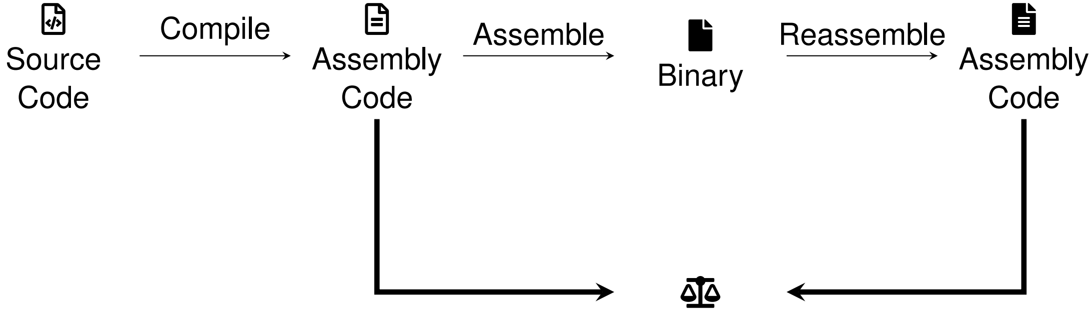
Can we test existing reassemblers and find symbolization errors?[5]

[5]Reassembly is Hard: A Reflection on Challenges and Strategies, ***USENIX Security 2023***

Symbolization Error and Runtime Behavior

Symbolization errors do not always produce crashes (or any visible evidences) when running reassembled binaries. Why?

Key Idea: Differential Testing



Challenges

- Matching two assembly files are not easy.
  - Two or more distinct functions of the same name can present in a binary.
- Not every expression has a debugging symbol in its binary.

Challenges

- Matching two assembly files are not easy.
  - Two or more distinct functions of the same name can present in a binary.
- Not every expression has a debugging symbol in its binary.

```
...
.Lswitch.table.convert_move:
    .long libfunc_table
    .long libfunc_table+4
```

Compiler-generated Assembly

```
...
.byte 0x08, 0x7e, 0x29, 0x08
.byte 0x0c, 0x7e, 0x29, 0x08
```

Disassembly

REASSESSOR Design

Employ a light-weight static analysis to overcome the challenges: normalize expressions and find matching assembly lines.

REASSESSOR Evaluation

- Ran three major reassemblers (Ddisasm, Ramblr, and RetroWrite) with 14,688 binaries compiled with various compilers and compiler options.
- Found more than a ***billion*** reassembly errors from those binaries.

The Lesson

Reassembly problem is as difficult as the variable recovery problem (or the decompilation problem), which is not solved yet.

The Lesson

Reassembly problem is as difficult as the variable recovery problem (or the decompilation problem), which is not solved yet.

What about the recent paper?

Verifiably Correct Lifting of Position-Independent x86-64 Binaries to Symbolized Assembly, ***CCS 2024***

Question?

Further Readings

- Ramblr: Making reassembly great again, ***NDSS 2017***.
- Datalog Disassembly, ***USENIX Security 2020***.
- Egalito: Layout-agnostic binary recompilation, ***ASPLOS 2020***.

Exercise

Let's write a simple program in C as below, and try to patch the binary (without modifying the source code) in such a way that it prints out the value (x) read from the user.

```
#include<stdio.h>
#include<unistd.h>
int main(void)
{
    int x;
    return read(0, &x, sizeof(x));
}
```