Resolving Indirect Jumps for Dynamic Instrumentation Applications

Gabriel-Andrew Pollo-Guilbert
gabriel.pollo-guilbert@polymtl.ca

January 8, 2021

What is binary instrumentation?

Allow an application to be instrumented without recompiling from source by inserting the instrumentation before or during the execution.

- Don't rely on static instrumentation that needs recompilation
- Quick instrumentation and prototyping
- Some application don't have static tracepoint already implemented

Binary instrumentation methods

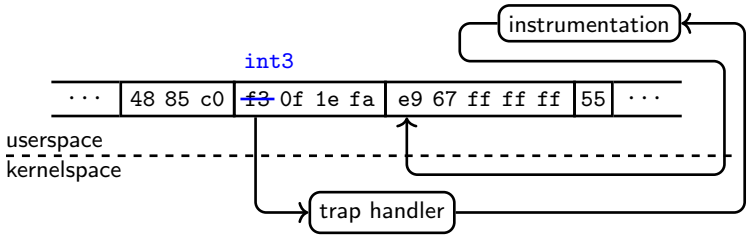


Figure: trap-based dynamic instrumentation

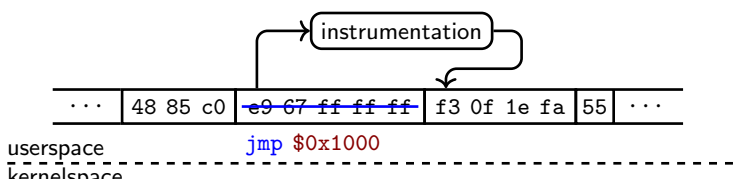
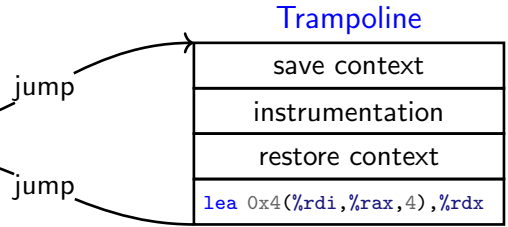


Figure: jump-based dynamic instrumentation

Jump-based binary instrumentation (example 1)

```
test   %esi,%esi
jle    0x20
lea    -0x1(%rsi),%eax
/* what is the value of %eax? */
lea    0x4(%rdi,%rax,4),%rdx
xor    %eax,%eax
xchg   %ax,%ax
add    (%rdi),%eax
add    $0x4,%rdi
cmp    %rdx,%rdi
jne    0x10
retq
nopl   0x0(%rax)
xor    %eax,%eax
retq
```

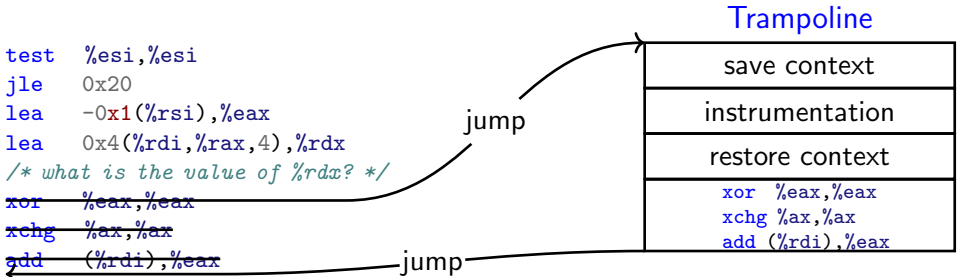



```
48 8d 54 87 04      lea 0x4(%rdi,%rax,4),%rdx
```

could be replaced by

```
e9 fb 0f 00 00      jmp $0x1000
```

Jump-based binary instrumentation (example 2)

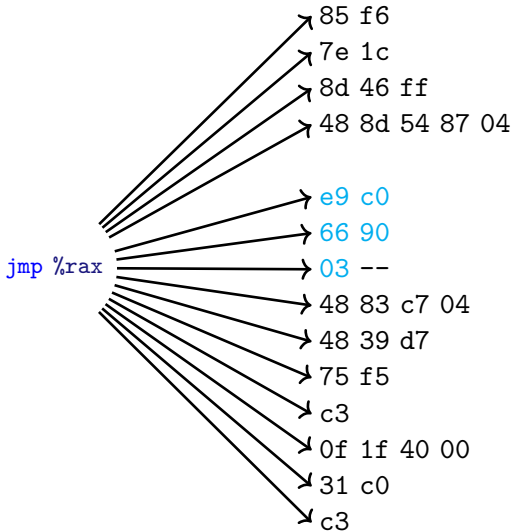



```
31 c0       xor  %eax,%eax
66 90       xchg %ax,%ax
03 07       add  (%rdi),%eax
```

could be replaced by

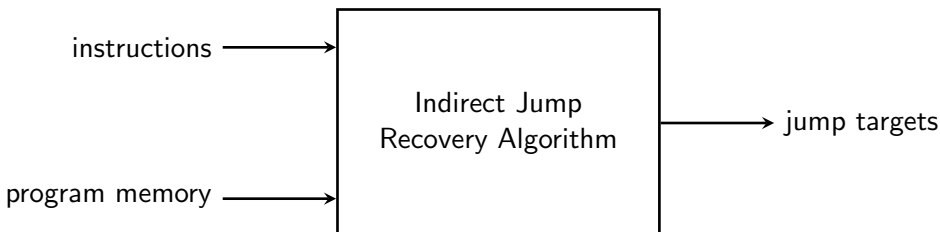
```
e9 fb       jmp $0x1000
0f 00
00 --
```

The indirect jump problem



Goal of the work

The goal of the work is to have an algorithm that can compute indirect jump target addresses.



What can generate indirect jumps?

switch cases

```
switch (n) {
case 0: m = 3; break;
case 1: m = 6; break;
case 2: m = 1; break;
case 3: m = 2; break;
case 4: m = 9; break;
case 5: m = 7; break;
case 6: m = 2; break;
default: break;
}
```

```
# %rax = n
cmp $6,%rax
ja  0x152
mov JUMP_TABLE(,%rax,8),%rbx
jmp *%rbx
```

tail-optimized indirect call

```
void call(void (*f)(void)) {
   /* ... */
   /* ... */
   /* ... */
   f();
}
```

```
# %rsi = f
jmp *%rsi
```

tail-optimized virtual call

```
class base {
public:
   virtual foo();
};

void call(base& base) {
   /* ... */
   /* ... */
   /* ... */
   base.foo();
}
```

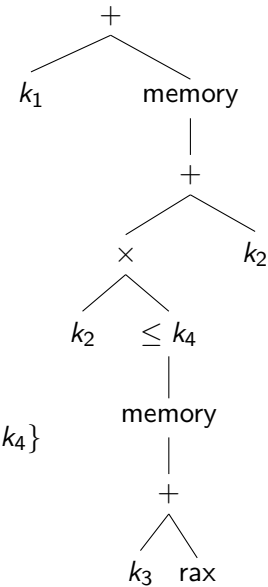
```
# %rdi = base
mov  (%rdi),%rax
jmpq *(%rax)
```

Current solutions

- Reverse engineering tools already try to recover the whole control-flow graph, thus also indirect jumps.
  - e.g. Radare2, Ghidra, IDA Pro
  - The two first failed to recovered indirect jumps in my testing.
  - Usually goes through all the instructions in a top to bottom fashion.
- Expression substitution technique similar to method 1.

Method 1 – Reconstructing the computation

```
cmpl   $4,20(%rax)
ja     1337
mov    20(%rax),%eax
lea    1000(%rip),%rbx
movslq (%rbx,%rax,4),%rax
mov    %rax,%rdx
add    %rbx,%rax
jmpq   *%rax
```



$$\text{targets} = \{k_1 + \text{memory}\,(k_2 + k_3 \times i) \mid 0 \leq i \leq k_4\}$$

Validating recovered jump results

- Manual validation:
  - Compile with `gcc -S` to get assembly.
  - Dump assembly with `objdump -D`.
  - Manually compare the results.
  - Very time consuming.
- Automatic validation:
  - Somehow modify gcc to output additional informations.
  - Have a script do the validation.
  - Very few knowledge in compiler.

Simple GCC patch

```
diff --git a/gcc/final.c b/gcc/final.c
index a3601964a8d..93f266d408b 100644
--- a/gcc/final.c
+++ b/gcc/final.c
@@ -2507,6 +2507,7 @@ final_scan_insn_1 (rtx_insn *insn, FILE *file, int optimize_p ATTRIBUTE_UNUSED,

 #ifdef ASM_OUTPUT_CASE_LABEL
           ASM_OUTPUT_CASE_LABEL (file, "L", CODE_LABEL_NUMBER (insn), table);
+          targetm.asm_out.internal_label (file, "JUMP_TABLE_START", CODE_LABEL_NUMBER(insn));
 #else
           targetm.asm_out.internal_label (file, "L", CODE_LABEL_NUMBER (insn));
 #endif
@@ -2621,6 +2622,7 @@ final_scan_insn_1 (rtx_insn *insn, FILE *file, int optimize_p ATTRIBUTE_UNUSED,
 #endif
                   }
               }
+            targetm.asm_out.internal_label (file, "JUMP_TABLE_END", CODE_LABEL_NUMBER (PREV_INSN(insn)));
 #ifdef ASM_OUTPUT_CASE_END
             ASM_OUTPUT_CASE_END (file,
                                  CODE_LABEL_NUMBER (PREV_INSN(insn)),
```

Simple GCC patch – Results

```
# the jump computation
cmpl        $6, -4(%rbp)
ja         .L10
movl        -4(%rbp), %eax
leaq        0(,%rax,4), %rdx
leaq        .L12(%rip), %rax
movl        (%rdx,%rax), %eax
cltq
leaq        .L12(%rip), %rdx
addq        %rdx, %rax
jmp        *%rax
```

```
        # the jump table
        .section        .rodata
        .align 4
        .align 4
.L12:
.JUMP_TABLE_START12:
        .long         .L19-.L12
        .long         .L21-.L12
        .long         .L22-.L12
        .long         .L23-.L12
        .long         .L24-.L12
        .long         .L25-.L12
        .long         .L26-.L12
.JUMP_TABLE_END12:
```

Implementation

The proof of concept was done in Python with `capstone` disassembler.

To add the feature into a dynamic tracing implementation, only a single "`if`" needs to be modify with the algorithm.

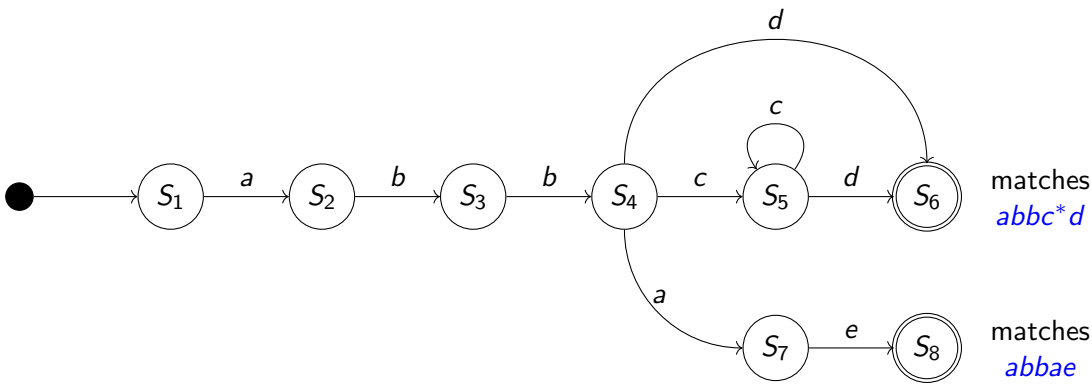
- `uftrace` – userspace function tracer
  - `check_unsupported` @ `arch/x86_64/mcount-insn.c`
  - already uses `capstone` disassembler
- `kprobe` – linux kernel
  - `can_optimize` @ `arch/x86/kernel/kprobes/opt.c`
  - more complex and time consuming to develop

Method 1 – Results

- It is complex and hard to maintain
- Current implementation fails to recover some pattern
- On `git` binary, `uftrace` now patches 94% compared to previous 90%
  - Most of the 6% isn't due to indirect jump
  - The rest is unsupported patterns

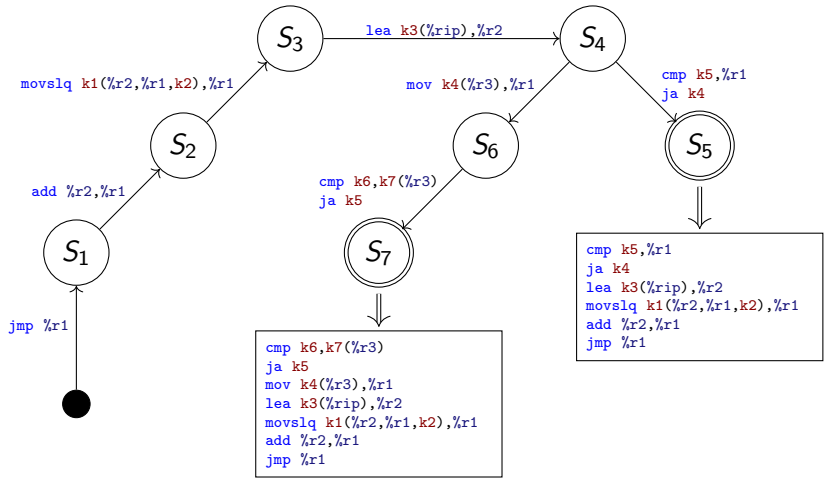
Method 2 – Pattern Matching

Regular expression are efficient thanks to the compilation process which generates a state machine that can test a string in linear time.



Method 2 – Pattern Matching

By applying this concept, we can pattern match instructions and test a given sequence of instructions in linear time.



Method 2 – Pattern Matching

This method aims to

- reduce complexity (both in performance and understanding) and
- ease the addition of new patterns.

Right now,

- the proof of concept is basically done,
- some implementation details need to be sorted and
- it is almost implemented in `uftrace`.

Questions?