A jump-target identification method for multi-architecture static binary translation

Alessandro Di Federico
Giovanni Agosta
Politecnico di Milano

CASES 2016

October 4, 2016

Index

Introduction

Our solution

Evaluation

Static binary translation

Static binary translation requires several steps:

1. Parse an input binary
2. Identify all the code it contains
3. Translate it from the input architecture to the target one
4. Produce an output binary

Static binary translation

Static binary translation requires several steps:

1. Parse an input binary
2. Identify all the code it contains
3. Translate it from the input architecture to the target one
4. Produce an output binary

Identify the code

- The binaries are typically divide in segments
- Certain segments are marked as *executable*
- We can be sure that all the code is contained in them

Translation works at a basic block granularity

Translation works at a basic block granularity

But where does a basic block start?

Jump targets

Jump target — Any address in the executable segment where it's possible to jump to.

A jump target denotes the beginning of a new basic block.

The naïve solution

- We could create a basic block for each executable address
- However we would get:
    - very complex control flow graph
    - poor information to the user
    - increased translation time
    - increased binary output
    - increased execution time

The dispatcher

- Typically SBTs use a dispatcher to handle indirect jumps
- Maps each address to the corresponding translated code

```
dispatcher:
  switch (program_counter) {
    case 0x400000:
      goto bb_0x400000;
    case 0x400010:
      goto bb_0x400010;
    /* ... */
  }
```

A more principled approach

- Explore the code from the entry points
- Follow the control flow
- Exhaustively collect the control flow graph

This is impossible in the general case

This is impossible in the general case

```
typedef void (*fptr)(void);

int main(int argc, char *argv[]) {
  fptr function_pointer = (fptr) argv[1];
  function_pointer();
}
```

Typical challenging situations

- Indirect control-flow transfers are challenging
- We can classify them in the following categories:
    - return instructions
    - calls to function pointers
    - far jumps
    - switch statements

Far jump

```
lui     t9, 0x42
addiu   t9, t9, 0xd188
jr      t9
```

Switch statements examples (ARM)

```
cmp    r0, #240
addls  pc, pc, r0, lsl #2
b      21304
b      21320
b      21710
b      212fc
```

Switch statements examples (x86-64)

```
cmp  eax,0x21
ja   400990
mov  rbx,rdi
mov  rbp,rsi
jmp  PTR [rax*8+0x422e40]
```

How can we handle these situations?

How can we handle these situations?

Can we do it in an architecture independent way?

Index

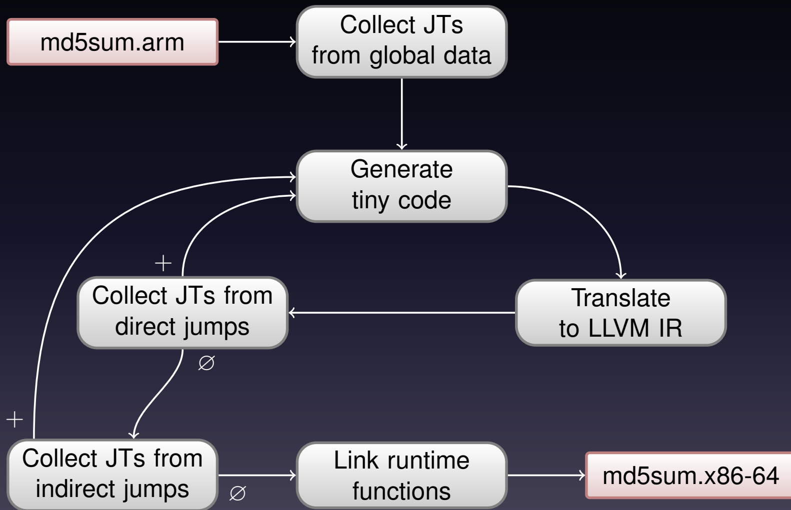
- Introduction
- Our solution
- Evaluation

The ingredients

We make heavy use of:

QEMU	Use it as a *frontend*, supports ~17 architectures. It produces an IR known as *tiny code*.
LLVM	Mature compiler framework, suitable to perform sophisticated analysis and recompile the translated code.

System overview



Key characteristics

- All the code is collected in single LLVM IR function
- Each input BB is associated to a LLVM BB
- Indirect jumps go through a dispatcher
- Each part of the CPU state is mapped to a global variable

The basic block identification process

- Global data harvesting
- Simple Expression Tracker
- OSR Analysis

Global data harvesting

- Parse global data byte-by-byte
- Interpret pointer-sized integers
- Is its value a code pointer?
    - Does it point to an executable segment?
    - Does it have an appropriate alignment?

What does it catch?

- Function pointers stored in global data
- Virtual tables
- Jump tables

Simple Expression Tracker (SET)

- Consider each store to the CPU state (e.g., a register)
- Track how the stored value is computed:
    - push each operation on a helper stack
    - stop in case of more than a single non-constant operand
- Proceed until an operation with no non-constant operands
- Go through the stack applying the operations
- Obtain a possible jump target

Load instructions

1. Load from the CPU state:

   Perform a depth-first visit to all the reaching definitions

2. Load from standard memory:

   If in global data, actually read it

Example

```
    lui  $v0, 0x42
    ble  $a0, $t0, do_call
    nop
    lui  $v0, 0x88
    addi $v0, 1
do_call:
    ori  $v0, 0x1234
    jal  $v0
```

SET example

```
  store i32 0x420000, i32* @v0
  ; ...
  br i1 %3, label %call, label %ft

ft:
  store i32 0x880000, i32* @v0
  %4 = load i32, i32* @v0
  %5 = add i32 %4, 1
  store i32 %5, @v0
  br label %do_call

do_call:
  %6 = load i32, i32* @v0
  %7 = or i32 %6, 0x1234
  %8 = and i32 %7, -2
  store i32 %8, i32* @pc
  br label %dispatcher
```

SET example

```
  store i32 0x420000, i32* @v0
  ; ...
  br i1 %3, label %call, label %ft

ft:
  store i32 0x880000, i32* @v0
  %4 = load i32, i32* @v0
  %5 = add i32 %4, 1
  store i32 %5, @v0
  br label %do_call

do_call:
  %6 = load i32, i32* @v0
  %7 = or i32 %6, 0x1234
  %8 = and i32 %7, -2
  store i32 %8, i32* @pc
  br label %dispatcher
```

and -2

SET example

or 4660
and -2

```
  store i32 0x420000, i32* @v0
  ; ...
  br i1 %3, label %call, label %ft

ft:
  store i32 0x880000, i32* @v0
  %4 = load i32, i32* @v0
  %5 = add i32 %4, 1
  store i32 %5, @v0
  br label %do_call

do_call:
  %6 = load i32, i32* @v0
  %7 = or i32 %6, 0x1234
  %8 = and i32 %7, -2
  store i32 %8, i32* @pc
  br label %dispatcher
```

SET example

or 4660
and -2

```
  store i32 0x420000, i32* @v0
  ; ...
  br i1 %3, label %call, label %ft

ft:
  store i32 0x880000, i32* @v0
  %4 = load i32, i32* @v0
  %5 = add i32 %4, 1
  store i32 %5, @v0
  br label %do_call

do_call:
  %6 = load i32, i32* @v0
  %7 = or i32 %6, 0x1234
  %8 = and i32 %7, -2
  store i32 %8, i32* @pc
  br label %dispatcher
```

SET example

or 4660
and -2

```
  store i32 0x420000, i32* @v0
  ; ...
  br i1 %3, label %call, label %ft

ft:
  store i32 0x880000, i32* @v0
  %4 = load i32, i32* @v0
  %5 = add i32 %4, 1
  store i32 %5, @v0
  br label %do_call

do_call:
  %6 = load i32, i32* @v0
  %7 = or i32 %6, 0x1234
  %8 = and i32 %7, -2
  store i32 %8, i32* @pc
  br label %dispatcher
```

SET example

add 1
or 4660
and -2

```
  store i32 0x420000, i32* @v0
  ; ...
  br i1 %3, label %call, label %ft

ft:
  store i32 0x880000, i32* @v0
  %4 = load i32, i32* @v0
  %5 = add i32 %4, 1
  store i32 %5, @v0
  br label %do_call

do_call:
  %6 = load i32, i32* @v0
  %7 = or i32 %6, 0x1234
  %8 = and i32 %7, -2
  store i32 %8, i32* @pc
  br label %dispatcher
```

SET example

add 1
or 4660
and -2

```
  store i32 0x420000, i32* @v0
  ; ...
  br i1 %3, label %call, label %ft

ft:
  store i32 0x880000, i32* @v0
  %4 = load i32, i32* @v0
  %5 = add i32 %4, 1
  store i32 %5, @v0
  br label %do_call

do_call:
  %6 = load i32, i32* @v0
  %7 = or i32 %6, 0x1234
  %8 = and i32 %7, -2
  store i32 %8, i32* @pc
  br label %dispatcher
```

SET example

0x880000

add 1
or 4660
and -2

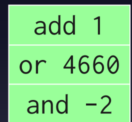
```
  store i32 0x420000, i32* @v0
  ; ...
  br i1 %3, label %call, label %ft

ft:
  store i32 0x880000, i32* @v0
  %4 = load i32, i32* @v0
  %5 = add i32 %4, 1
  store i32 %5, @v0
  br label %do_call

do_call:
  %6 = load i32, i32* @v0
  %7 = or i32 %6, 0x1234
  %8 = and i32 %7, -2
  store i32 %8, i32* @pc
  br label %dispatcher
```

SET example

0x880000



0x881234

```
  store i32 0x420000, i32* @v0
  ; ...
  br i1 %3, label %call, label %ft

ft:
  store i32 0x880000, i32* @v0
  %4 = load i32, i32* @v0
  %5 = add i32 %4, 1
  store i32 %5, @v0
  br label %do_call

do_call:
  %6 = load i32, i32* @v0
  %7 = or i32 %6, 0x1234
  %8 = and i32 %7, -2
  store i32 %8, i32* @pc
  br label %dispatcher
```

SET example

```
  store i32 0x420000, i32* @v0
  ; ...
  br i1 %3, label %call, label %ft

ft:
  store i32 0x880000, i32* @v0
  %4 = load i32, i32* @v0
  %5 = add i32 %4, 1
  store i32 %5, @v0
  br label %do_call

do_call:
  %6 = load i32, i32* @v0
  %7 = or i32 %6, 0x1234
  %8 = and i32 %7, -2
  store i32 %8, i32* @pc
  br label %dispatcher
```

or 4660
and -2

SET example

0x420000

or 4660
and -2

```
store i32 0x420000, i32* @v0
; ...
br i1 %3, label %call, label %ft

ft:
  store i32 0x880000, i32* @v0
  %4 = load i32, i32* @v0
  %5 = add i32 %4, 1
  store i32 %5, @v0
  br label %do_call

do_call:
  %6 = load i32, i32* @v0
  %7 = or i32 %6, 0x1234
  %8 = and i32 %7, -2
  store i32 %8, i32* @pc
  br label %dispatcher
```

SET example

0x420000

or 4660
and -2

0x421234

```
store i32 0x420000, i32* @v0
; ...
br i1 %3, label %call, label %ft

ft:
store i32 0x880000, i32* @v0
%4 = load i32, i32* @v0
%5 = add i32 %4, 1
store i32 %5, @v0
br label %do_call

do_call:
%6 = load i32, i32* @v0
%7 = or i32 %6, 0x1234
%8 = and i32 %7, -2
store i32 %8, i32* @pc
br label %dispatcher
```

What does it catch?

- Return addresses
- Far jumps
- Function pointers embedded in the code

OSR Analysis

- Its main objective is to handle switch statements
- It considers each SSA value
- Tracks of it can be expressed w.r.t. x:
  - plus an offset a
  - and a factor b
- For each basic block it tracks:
  - the boundaries of x
  - the *signedness* of x

An Offset Shifted Range (OSR)

$$a + b \cdot x, \text{ with } \left\{ x : \begin{array}{c} c \leq x \leq d \\ x < c, x > d \end{array} \text{ and } x \text{ is } \begin{array}{c} \text{signed} \\ \text{unsigned} \end{array} \right\}$$

Example: the input

```
cmp r1, #5
addls pc, pc, r1, lsl #2
```

```
BB1:
  %1 = load i32, i32* @r1
  %2 = sub i32 %1, 4
  %3 = icmp uge i32 %1, 4
  br i1 %3, label %BB2, label %BB3

BB2:
  %4 = icmp ne i32 %2, 0
  br i1 %4, label %exit, label %BB3

BB3:

  %5 = shl i32 %1, 2
  %6 = add i32 113372, %5
  store i32 %6, i32* @pc
```

```
BB1:
  %1 = load i32, i32* @r1
  %2 = sub i32 %1, 4 ; [x - 4]
  %3 = icmp uge i32 %1, 4
  br i1 %3, label %BB2, label %BB3

BB2:
  %4 = icmp ne i32 %2, 0
  br i1 %4, label %exit, label %BB3

BB3:

  %5 = shl i32 %1, 2
  %6 = add i32 113372, %5
  store i32 %6, i32* @pc
```

```
BB1:
  %1 = load i32, i32* @r1
  %2 = sub i32 %1, 4 ; [x - 4]
  %3 = icmp uge i32 %1, 4 ; (x >= 4, unsigned)
  br i1 %3, label %BB2, label %BB3

BB2:
  %4 = icmp ne i32 %2, 0
  br i1 %4, label %exit, label %BB3

BB3:

  %5 = shl i32 %1, 2
  %6 = add i32 113372, %5
  store i32 %6, i32* @pc
```

```
BB1:
  %1 = load i32, i32* @r1
  %2 = sub i32 %1, 4 ; [x - 4]
  %3 = icmp uge i32 %1, 4 ; (x >= 4, unsigned)
  br i1 %3, label %BB2, label %BB3

BB2: ; (x >= 4, unsigned)
  %4 = icmp ne i32 %2, 0
  br i1 %4, label %exit, label %BB3

BB3:

  %5 = shl i32 %1, 2
  %6 = add i32 113372, %5
  store i32 %6, i32* @pc
```

```
BB1:
  %1 = load i32, i32* @r1
  %2 = sub i32 %1, 4 ; [x - 4]
  %3 = icmp uge i32 %1, 4 ; (x >= 4, unsigned)
  br i1 %3, label %BB2, label %BB3

BB2: ; (x >= 4, unsigned)
  %4 = icmp ne i32 %2, 0
  br i1 %4, label %exit, label %BB3

BB3: ;                    <BB1, (x < 4, unsigned)>

  %5 = shl i32 %1, 2
  %6 = add i32 113372, %5
  store i32 %6, i32* @pc
```

```
BB1:
  %1 = load i32, i32* @r1
  %2 = sub i32 %1, 4 ; [x - 4]
  %3 = icmp uge i32 %1, 4 ; (x >= 4, unsigned)
  br i1 %3, label %BB2, label %BB3

BB2: ; (x >= 4, unsigned)
  %4 = icmp ne i32 %2, 0 ; (x > 4, unsigned)
  br i1 %4, label %exit, label %BB3

BB3: ;                          <BB1, (x < 4, unsigned)>

  %5 = shl i32 %1, 2
  %6 = add i32 113372, %5
  store i32 %6, i32* @pc
```

```
BB1:
  %1 = load i32, i32* @r1
  %2 = sub i32 %1, 4 ; [x - 4]
  %3 = icmp uge i32 %1, 4 ; (x >= 4, unsigned)
  br i1 %3, label %BB2, label %BB3

BB2: ; (x >= 4, unsigned)
  %4 = icmp ne i32 %2, 0 ; (x > 4, unsigned)
  br i1 %4, label %exit, label %BB3

BB3: ;                       <BB1, (x < 4, unsigned)>
     ;                       <BB2, (x == 4, unsigned)>
  %5 = shl i32 %1, 2
  %6 = add i32 113372, %5
  store i32 %6, i32* @pc
```

```
BB1:
  %1 = load i32, i32* @r1
  %2 = sub i32 %1, 4 ; [x - 4]
  %3 = icmp uge i32 %1, 4 ; (x >= 4, unsigned)
  br i1 %3, label %BB2, label %BB3

BB2: ; (x >= 4, unsigned)
  %4 = icmp ne i32 %2, 0 ; (x > 4, unsigned)
  br i1 %4, label %exit, label %BB3

BB3: ; (x <= 4, unsigned) = <BB1, (x < 4, unsigned)>
     ;                    || <BB2, (x == 4, unsigned)>
  %5 = shl i32 %1, 2
  %6 = add i32 113372, %5
  store i32 %6, i32* @pc
```

```
BB1:
  %1 = load i32, i32* @r1
  %2 = sub i32 %1, 4 ; [x - 4]
  %3 = icmp uge i32 %1, 4 ; (x >= 4, unsigned)
  br i1 %3, label %BB2, label %BB3

BB2: ; (x >= 4, unsigned)
  %4 = icmp ne i32 %2, 0 ; (x > 4, unsigned)
  br i1 %4, label %exit, label %BB3

BB3: ; (x <= 4, unsigned) = <BB1, (x < 4, unsigned)>
     ;                     || <BB2, (x == 4, unsigned)>
  %5 = shl i32 %1, 2 ; [4 * x]
  %6 = add i32 113372, %5
  store i32 %6, i32* @pc
```

```
BB1:
  %1 = load i32, i32* @r1
  %2 = sub i32 %1, 4 ; [x - 4]
  %3 = icmp uge i32 %1, 4 ; (x >= 4, unsigned)
  br i1 %3, label %BB2, label %BB3

BB2: ; (x >= 4, unsigned)
  %4 = icmp ne i32 %2, 0 ; (x > 4, unsigned)
  br i1 %4, label %exit, label %BB3

BB3: ; (x <= 4, unsigned) = <BB1, (x < 4, unsigned)>
     ;                    || <BB2, (x == 4, unsigned)>
  %5 = shl i32 %1, 2 ; [4 * x]
  %6 = add i32 113372, %5 ; [113372 + 4 * x]
  store i32 %6, i32* @pc
```

Index

Introduction

Our solution

Evaluation

Implementation overview

- 7000 C++ SLOCs
- Using LLVM 3.8 and QEMU 2.5.0
- We support:
  - ARM: GCC 5.3.0 + uClibc
  - MIPS: GCC 5.3.0 + musl
  - x86-64: GCC 4.9.2 + musl
- Only static binaries are supported

Functional testing

- We considered coreutils

  `md5sum`, `ls`, `base64`...

- Translate them and run its testsuite

Coverage and basic block size

	Coverage					
	Covered	Unused	NOPs	Other	Extra	IPB
MIPS	95.37%	4.61%	0.00%	0.02%	12.51%	5.17
ARM	89.56%	8.91%	0.13%	1.40%	14.16%	3.98
x86-64	94.84%	4.70%	0.46%	0.00%	12.87%	4.22

Test suite results

	Tests			
	Skip	Pass	Fail	No JT
MIPS	128	409	43	3
ARM	132	361	87	0
x86-64	127	419	34	0

rev.ng

Thanks for your attention

License

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.