rev.ng

A unified static binary analysis framework

Alessandro Di Federico

PhD student at Politecnico di Milano

ale@clearmind.me

LLVM developers meeting 2016

November 3, 2016

Index

What is `rev.ng`?

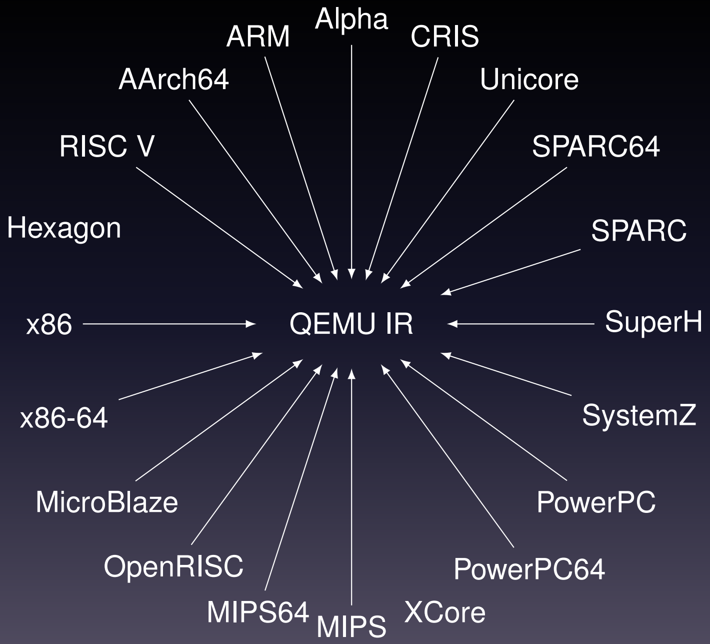
`rev.ng` is a *unified* suite of tools
for static binary analysis

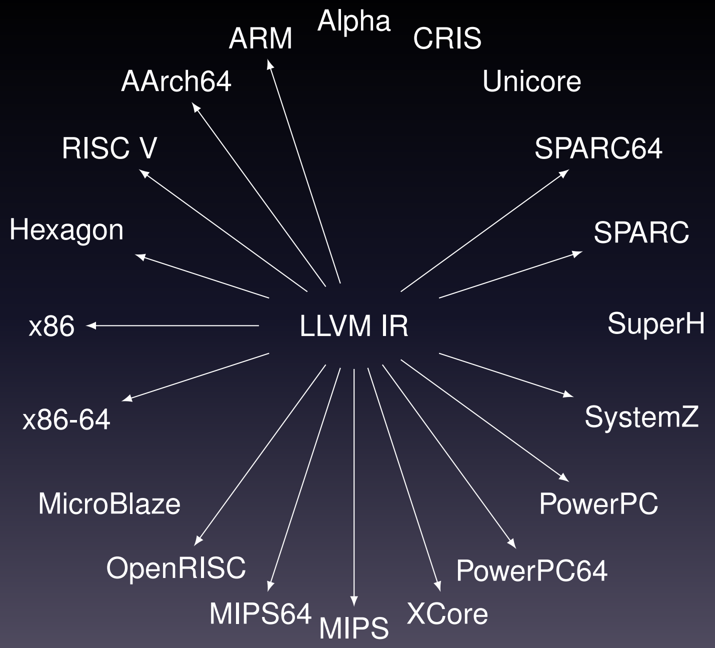
Features

- Static binary translation
- Recovery of the control-flow graph
- Recovery of function boundaries

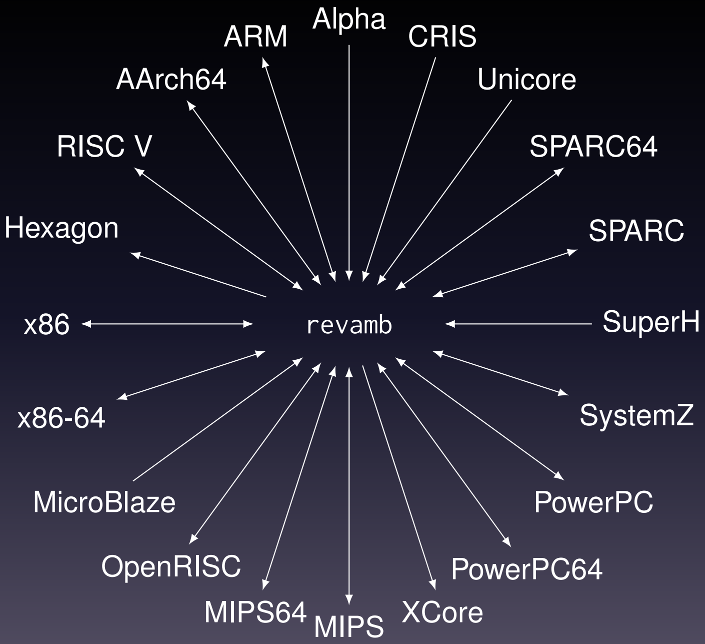
`revamb`: the static binary translator

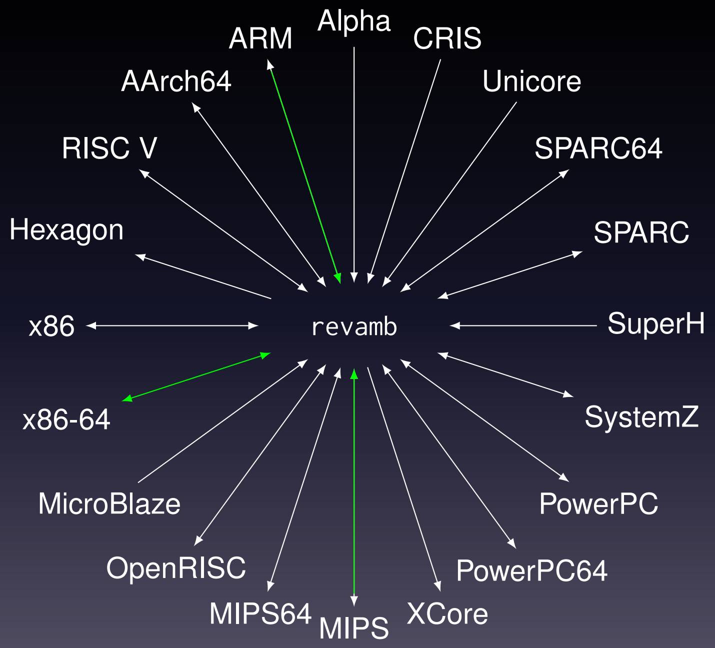
1. Parse the binary and load it in memory
2. Identify all the basic blocks in a binary
3. Lift them using QEMU's *tiny code generator*
4. Translate the output to a single LLVM IR function
5. Recompile it


Alpha
ARM
CRIS
AArch64
Unicore
RISC V
SPARC64
Hexagon
SPARC
x86
QEMU IR
SuperH
x86-64
SystemZ
MicroBlaze
PowerPC
OpenRISC
PowerPC64
MIPS64
MIPS
XCore



Alpha
ARM
CRIS
AArch64
Unicore
RISC V
SPARC64
Hexagon
SPARC
x86
LLVM IR
SuperH
x86-64
SystemZ
MicroBlaze
PowerPC
OpenRISC
PowerPC64
MIPS64
MIPS
XCore


Alpha
ARM
CRIS
AArch64
Unicore
RISC V
SPARC64
Hexagon
SPARC
x86
revamb
SuperH
x86-64
SystemZ
MicroBlaze
PowerPC
OpenRISC
PowerPC64
MIPS64
MIPS
XCore


Alpha
ARM
CRIS
AArch64
Unicore
RISC V
SPARC64
Hexagon
SPARC
x86
revamb
SuperH
x86-64
SystemZ
MicroBlaze
PowerPC
OpenRISC
PowerPC64
MIPS64
MIPS
XCore

Concept mapping

Input assembly	`revamb`
CPU register	LLVM GlobalVariable

Concept mapping

Input assembly	`revamb`
CPU register	LLVM GlobalVariable
direct branch	direct branch

Concept mapping

Input assembly	`revamb`
CPU register	LLVM GlobalVariable
direct branch	direct branch
indirect branch	jump to the dispatcher

Dispatcher example

```
%0 = load i32, i32* @pc
switch i32 %0, label %abort [
  i32 0x10074, label %bb.0x10074
  i32 0x10080, label %bb.0x10080
  i32 0x10084, label %bb.0x10084
  ...
]
```

Concept mapping

Input assembly	`revamb`
CPU register	LLVM GlobalVariable
direct branch	direct branch
indirect branch	jump to the dispatcher

Concept mapping

Input assembly	`revamb`
CPU register	LLVM GlobalVariable
direct branch	direct branch
indirect branch	jump to the dispatcher
complex instruction	QEMU helper function

Concept mapping

Input assembly	`revamb`
CPU register	LLVM GlobalVariable
direct branch	direct branch
indirect branch	jump to the dispatcher
complex instruction	QEMU helper function
syscalls	QEMU Linux subsystem

We statically link all the necessary QEMU helper functions

Example: original assembly

```
ldr   r3, [fp, #-8]

bl    0x1234
```

Example: QEMU's IR

```
ldr   r3, [fp, #-8]    {  mov_i32 tmp5,fp
                          movi_i32 tmp6,$0xfffffff8
                          add_i32 tmp5,tmp5,tmp6
                          qemu_ld_i32 tmp6,tmp5
                          mov_i32 r3,tmp6

bl    0x1234           {  movi_i32 tmp5,$0x10088
                          mov_i32 lr,tmp5
                          movi_i32 pc,$0x1234
                          exit_tb $0x0
```

Example: LLVM IR

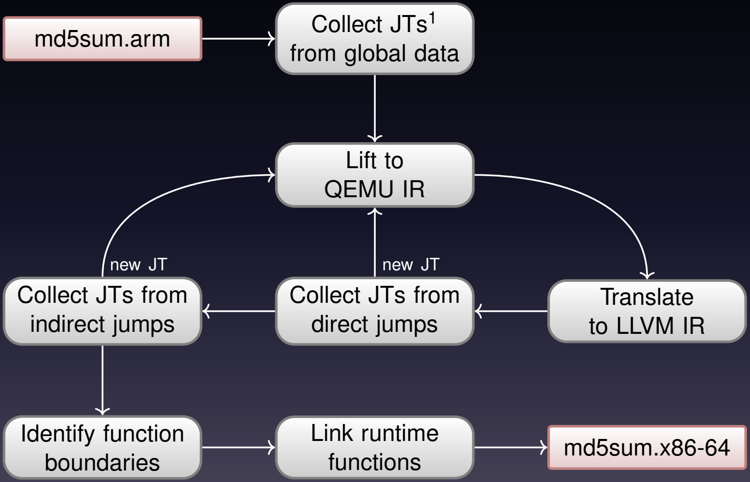
```
ldr   r3, [fp, #-8]
```

```
%1 = load i32, i32* @fp
%2 = add i32 %1, -8
%3 = inttoptr i32 %2 to i32*
%4 = load i32, i32* %3
store i32 %4, i32* @r3
```

```
bl    0x1234
```

```
store i32 0x10088, i32* @lr
store i32 0x1234, i32* @pc
br label %bb.0x1234
```

System overview



[1] JT: a *jump target*, i.e., a basic block starting address

Index

Index

Introduction

A peek inside

- Recovery of switch cases
- Function detection

Results

Conclusions

Typical lowering of a switch on ARM

```
1000: cmp r1, #5
1004: addls pc, pc, r1, lsl #2
1008: ...
100c: ...
```

OSR Analysis

- A data-flow analysis to handle `switch`
- 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)

Given two SSA values x and y:

$$y = a + b \cdot x, \text{ with } \left\{ x : \begin{array}{c} x \in [c, d] \\ x \notin [c, d] \end{array} \text{ and } x \text{ is } \begin{array}{c} \text{signed} \\ \text{unsigned} \end{array} \right\}$$

Example: the input

```
1000: cmp r1, #5
1004: addls pc, pc, r1, lsl #2
1008: ...
100c: ...
```

Pseudo C	LLVM IR	OSRA
	`BB1:`	
`a = r1`	`%1 = load i32, i32* @r1`	
`b = a - 4`	`%2 = sub i32 %1, 4`	
`c = (b >= 4)`	`%3 = icmp uge i32 %1, 4`	
`if (c)`	`br i1 %3, %BB2, %BB3`	
`{`	`BB2:`	
`d = (b == 0)`	`%4 = icmp eq i32 %2, 0`	
`if (!d)`	`br i1 %4, %BB3, %exit`	
`return`		
`}`	`BB3:`	
`e = a << 2`	`%5 = shl i32 %1, 2`	
`f = e + 0x100c`	`%6 = add i32 0x100c, %5`	
`pc = f`	`store i32 %6, i32* @pc`	

Pseudo C	LLVM IR	OSRA
	`BB1:`	
`a = r1`	`%1 = load i32, i32* @r1`	`; [x]`
`b = a - 4`	`%2 = sub i32 %1, 4`	
`c = (b >= 4)`	`%3 = icmp uge i32 %1, 4`	
`if (c)`	`br i1 %3, %BB2, %BB3`	
`{`	`BB2:`	
`d = (b == 0)`	`%4 = icmp eq i32 %2, 0`	
`if (!d)`	`br i1 %4, %BB3, %exit`	
`return`		
`}`	`BB3:`	
`e = a << 2`	`%5 = shl i32 %1, 2`	
`f = e + 0x100c`	`%6 = add i32 0x100c, %5`	
`pc = f`	`store i32 %6, i32* @pc`	

Pseudo C	LLVM IR	OSRA
	BB1:	
a = r1	%1 = load i32, i32* @r1	; [x]
b = a - 4	%2 = sub i32 %1, 4	; [x - 4]
c = (b >= 4)	%3 = icmp uge i32 %1, 4	
if (c)	br i1 %3, %BB2, %BB3	
{	BB2:	
d = (b == 0)	%4 = icmp eq i32 %2, 0	
if (!d)	br i1 %4, %BB3, %exit	
return		
}	BB3:	
e = a << 2	%5 = shl i32 %1, 2	
f = e + 0x100c	%6 = add i32 0x100c, %5	
pc = f	store i32 %6, i32* @pc	

Pseudo C	LLVM IR	OSRA
	`BB1:`	
`a = r1`	`%1 = load i32, i32* @r1`	`; [x]`
`b = a - 4`	`%2 = sub i32 %1, 4`	`; [x - 4]`
`c = (b >= 4)`	`%3 = icmp uge i32 %1, 4`	`; (x >= 4, u)`
`if (c)`	`br i1 %3, %BB2, %BB3`	
`{`	`BB2:`	
`  d = (b == 0)`	`%4 = icmp eq i32 %2, 0`	
`  if (!d)`	`br i1 %4, %BB3, %exit`	
`    return`		
`}`	`BB3:`	
`e = a << 2`	`%5 = shl i32 %1, 2`	
`f = e + 0x100c`	`%6 = add i32 0x100c, %5`	
`pc = f`	`store i32 %6, i32* @pc`	

Pseudo C	LLVM IR	OSRA
	`BB1:`	
`a = r1`	`%1 = load i32, i32* @r1`	`; [x]`
`b = a - 4`	`%2 = sub i32 %1, 4`	`; [x - 4]`
`c = (b >= 4)`	`%3 = icmp uge i32 %1, 4`	`; (x >= 4, u)`
`if (c)`	`br i1 %3, %BB2, %BB3`	
`{`	`BB2:`	`; (x >= 4, u)`
`d = (b == 0)`	`%4 = icmp eq i32 %2, 0`	
`if (!d)`	`br i1 %4, %BB3, %exit`	
`return`		
`}`	`BB3:`	
`e = a << 2`	`%5 = shl i32 %1, 2`	
`f = e + 0x100c`	`%6 = add i32 0x100c, %5`	
`pc = f`	`store i32 %6, i32* @pc`	

Pseudo C	LLVM IR	OSRA
	`BB1:`	
`a = r1`	`%1 = load i32, i32* @r1`	`; [x]`
`b = a - 4`	`%2 = sub i32 %1, 4`	`; [x - 4]`
`c = (b >= 4)`	`%3 = icmp uge i32 %1, 4`	`; (x >= 4, u)`
`if (c)`	`br i1 %3, %BB2, %BB3`	
`{`	`BB2:`	`; (x >= 4, u)`
`  d = (b == 0)`	`%4 = icmp eq i32 %2, 0`	
`  if (!d)`	`br i1 %4, %BB3, %exit`	
`    return`		
`}`	`BB3:`	`; (x < 4, u)`
`e = a << 2`	`%5 = shl i32 %1, 2`	
`f = e + 0x100c`	`%6 = add i32 0x100c, %5`	
`pc = f`	`store i32 %6, i32* @pc`	

Pseudo C	LLVM IR	OSRA
	`BB1:`	
`a = r1`	`%1 = load i32, i32* @r1`	`; [x]`
`b = a - 4`	`%2 = sub i32 %1, 4`	`; [x - 4]`
`c = (b >= 4)`	`%3 = icmp uge i32 %1, 4`	`; (x >= 4, u)`
`if (c)`	`br i1 %3, %BB2, %BB3`	
`{`	`BB2:`	`; (x >= 4, u)`
`  d = (b == 0)`	`%4 = icmp eq i32 %2, 0`	`; (x - 4 == 0, u)`
`  if (!d)`	`br i1 %4, %BB3, %exit`	
`    return`		
`}`	`BB3:`	`; (x < 4, u)`
`e = a << 2`	`%5 = shl i32 %1, 2`	
`f = e + 0x100c`	`%6 = add i32 0x100c, %5`	
`pc = f`	`store i32 %6, i32* @pc`	

Pseudo C	LLVM IR	OSRA
	BB1:	
a = r1	%1 = load i32, i32* @r1	; [x]
b = a - 4	%2 = sub i32 %1, 4	; [x - 4]
c = (b >= 4)	%3 = icmp uge i32 %1, 4	; (x >= 4, u)
if (c)	br i1 %3, %BB2, %BB3	
{	BB2:	; (x >= 4, u)
d = (b == 0)	%4 = icmp eq i32 %2, 0	; (x == 4, u)
if (!d)	br i1 %4, %BB3, %exit	
return		
}	BB3:	; (x < 4, u)
e = a << 2	%5 = shl i32 %1, 2	
f = e + 0x100c	%6 = add i32 0x100c, %5	
pc = f	store i32 %6, i32* @pc	

Pseudo C	LLVM IR	OSRA
	BB1:	
a = r1	%1 = load i32, i32* @r1	; [x]
b = a - 4	%2 = sub i32 %1, 4	; [x - 4]
c = (b >= 4)	%3 = icmp uge i32 %1, 4	; (x >= 4, u)
if (c)	br i1 %3, %BB2, %BB3	
{	BB2:	; (x >= 4, u)
d = (b == 0)	%4 = icmp eq i32 %2, 0	; (x == 4, u)
if (!d)	br i1 %4, %BB3, %exit	
return		
}	BB3:	; (x < 4, u)
		; (x == 4, u)
e = a << 2	%5 = shl i32 %1, 2	
f = e + 0x100c	%6 = add i32 0x100c, %5	
pc = f	store i32 %6, i32* @pc	

Pseudo C	LLVM IR	OSRA
	`BB1:`	
`a = r1`	`%1 = load i32, i32* @r1`	`; [x]`
`b = a - 4`	`%2 = sub i32 %1, 4`	`; [x - 4]`
`c = (b >= 4)`	`%3 = icmp uge i32 %1, 4`	`; (x >= 4, u)`
`if (c)`	`br i1 %3, %BB2, %BB3`	
`{`	`BB2:`	`; (x >= 4, u)`
`  d = (b == 0)`	`%4 = icmp eq i32 %2, 0`	`; (x == 4, u)`
`  if (!d)`	`br i1 %4, %BB3, %exit`	
`    return`		
`}`	`BB3:`	`; (x <= 4, u)`
`e = a << 2`	`%5 = shl i32 %1, 2`	
`f = e + 0x100c`	`%6 = add i32 0x100c, %5`	
`pc = f`	`store i32 %6, i32* @pc`	

Pseudo C	LLVM IR	OSRA
	`BB1:`	
`a = r1`	`%1 = load i32, i32* @r1`	`; [x]`
`b = a - 4`	`%2 = sub i32 %1, 4`	`; [x - 4]`
`c = (b >= 4)`	`%3 = icmp uge i32 %1, 4`	`; (x >= 4, u)`
`if (c)`	`br i1 %3, %BB2, %BB3`	
`{`	`BB2:`	`; (x >= 4, u)`
`d = (b == 0)`	`%4 = icmp eq i32 %2, 0`	`; (x == 4, u)`
`if (!d)`	`br i1 %4, %BB3, %exit`	
`return`		
`}`	`BB3:`	`; (x <= 4, u)`
`e = a << 2`	`%5 = shl i32 %1, 2`	`; [4 * x]`
`f = e + 0x100c`	`%6 = add i32 0x100c, %5`	
`pc = f`	`store i32 %6, i32* @pc`	

Pseudo C	LLVM IR	OSRA
	BB1:	
a = r1	%1 = load i32, i32* @r1	; [x]
b = a - 4	%2 = sub i32 %1, 4	; [x - 4]
c = (b >= 4)	%3 = icmp uge i32 %1, 4	; (x >= 4, u)
if (c)	br i1 %3, %BB2, %BB3	
{	BB2:	; (x >= 4, u)
d = (b == 0)	%4 = icmp eq i32 %2, 0	; (x == 4, u)
if (!d)	br i1 %4, %BB3, %exit	
return		
}	BB3:	; (x <= 4, u)
e = a << 2	%5 = shl i32 %1, 2	; [4 * x]
f = e + 0x100c	%6 = add i32 0x100c, %5	; [0x100c + 4 * x]
pc = f	store i32 %6, i32* @pc	

Possible jump targets

`[0x100c + 4 * x]` with `(x <= 4, u)`:

```
0x100c + 4 * 0 = 0x100c
0x100c + 4 * 1 = 0x1010
0x100c + 4 * 2 = 0x1014
0x100c + 4 * 3 = 0x1018
0x100c + 4 * 4 = 0x101c
```

Index

Generality of function detection

We don’t use any *architecture-specific* heuristic

The function detection process

1. Identify function calls and return instructions
2. Create a set of *candidate function entry points* (CFEP):
   1. called basic blocks
   2. *unused* code pointers in global data (e.g., not jump tables)
   3. code pointers embedded in the code
3. Compute the basic blocks reachable from each CFEP
4. Keep a CFEP only if:
   1. it’s a called basic block, or
   2. it’s reached by a *skipping* jump instruction

`noreturn` functions

`abort`, `exit`	We identify syscalls killing the process and trivial infinite loops
`longjmp`	Any instruction overwriting the stack pointer with a value different from sp + value or loaded from such an address.

`noreturn` functions

`abort`, `exit` We identify syscalls killing the process and trivial infinite loops

`longjmp` Any instruction overwriting the stack pointer with a value different from sp + value or loaded from such an address.

1. Mark all these basic blocks as *killer basic blocks*
2. Set their successor to a common basic block, the *sink*
3. Compute the set of basic blocks it post-dominates
4. Mark as `noreturn` CFEPs in this set

Index

Coreutils test suite results

	rev.ng		QEMU
	Passed	Failed due to missing code	Passed
MIPS	90.5%	0.7%	92.0%
ARM	80.6%	0.0%	92.7%
x86-64	92.5%	0.0%	94.6%

Function detection

	Matched functions (%)			Jaccard index		
	ARM	MIPS	x86-64	ARM	MIPS	x86-64
IDA	85.31	93.38	94.47	97.75	93.64	99.69
`rev.ng`	87.91	95.08	95.66	97.08	92.89	95.72
BAP	80.26	N/A	83.51	75.37	N/A	69.91
angr	97.54	92.56	93.75	51.15	63.71	83.86

Index

Introduction

A peek inside

Recovery of switch cases

Function detection

Results

Conclusions

Current status

Tested on:

- statically linked ELF binaries
- ARM, MIPS, x86-64
- uClibc and musl

Future works

- Calling convention detection and stack analysis
- Multithreading
- Try to upstream our changes to QEMU
- Measure our performance vs QEMU vs native
- Experiment with instrumentation (fuzzing?)

Thanks for your attention!

https://rev.ng

License