rev.ng

An overview and an outlook to the future

Alessandro Di Federico

PhD student at Politecnico di Milano

ale@clearmind.me

LLVM Social 2017 - Italy

April 21, 2017

Index

Introduction

Back to binary

Let’s stay on the IR

Supported platforms

What is rev.ng?

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

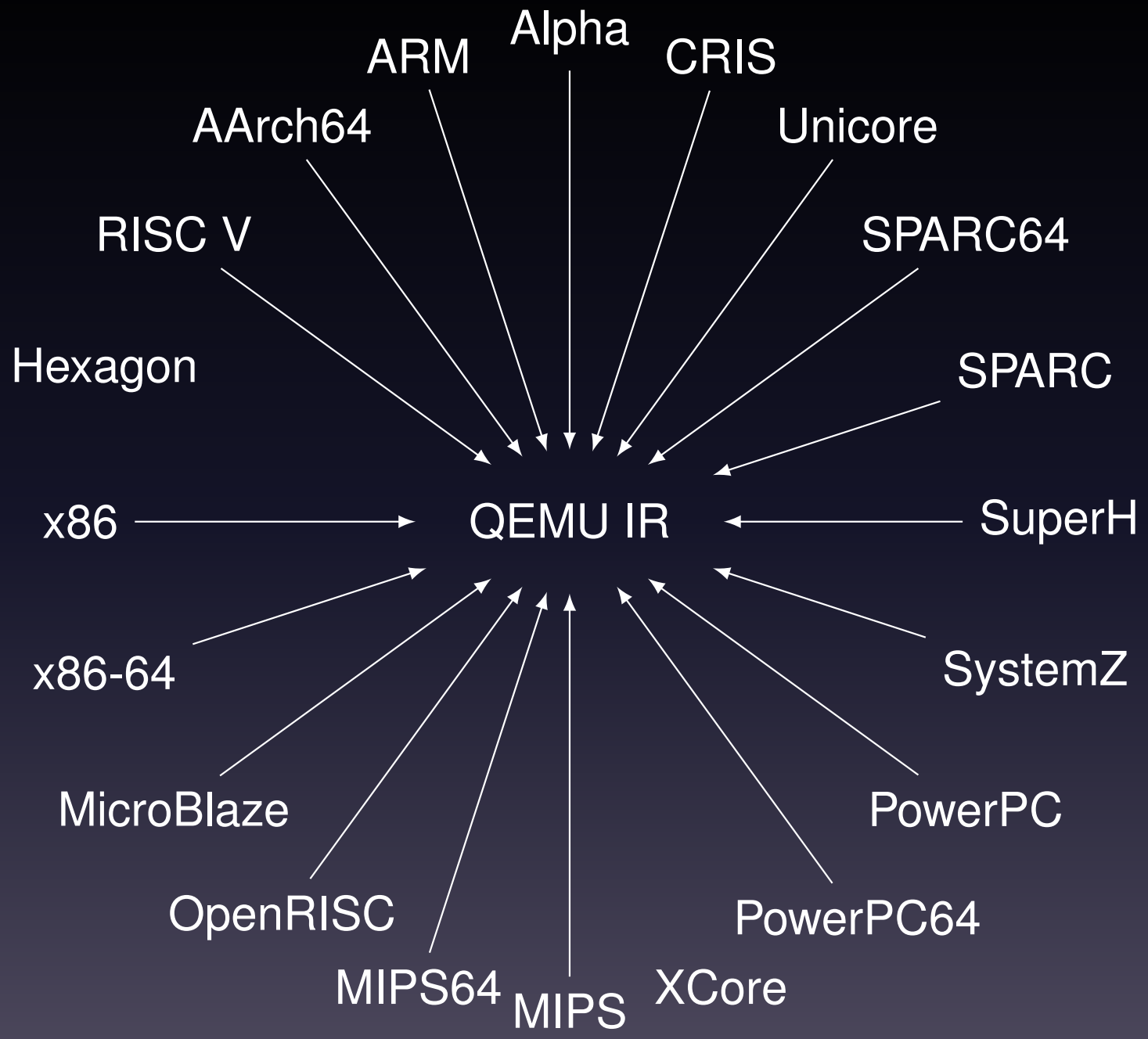
What is rev.ng?

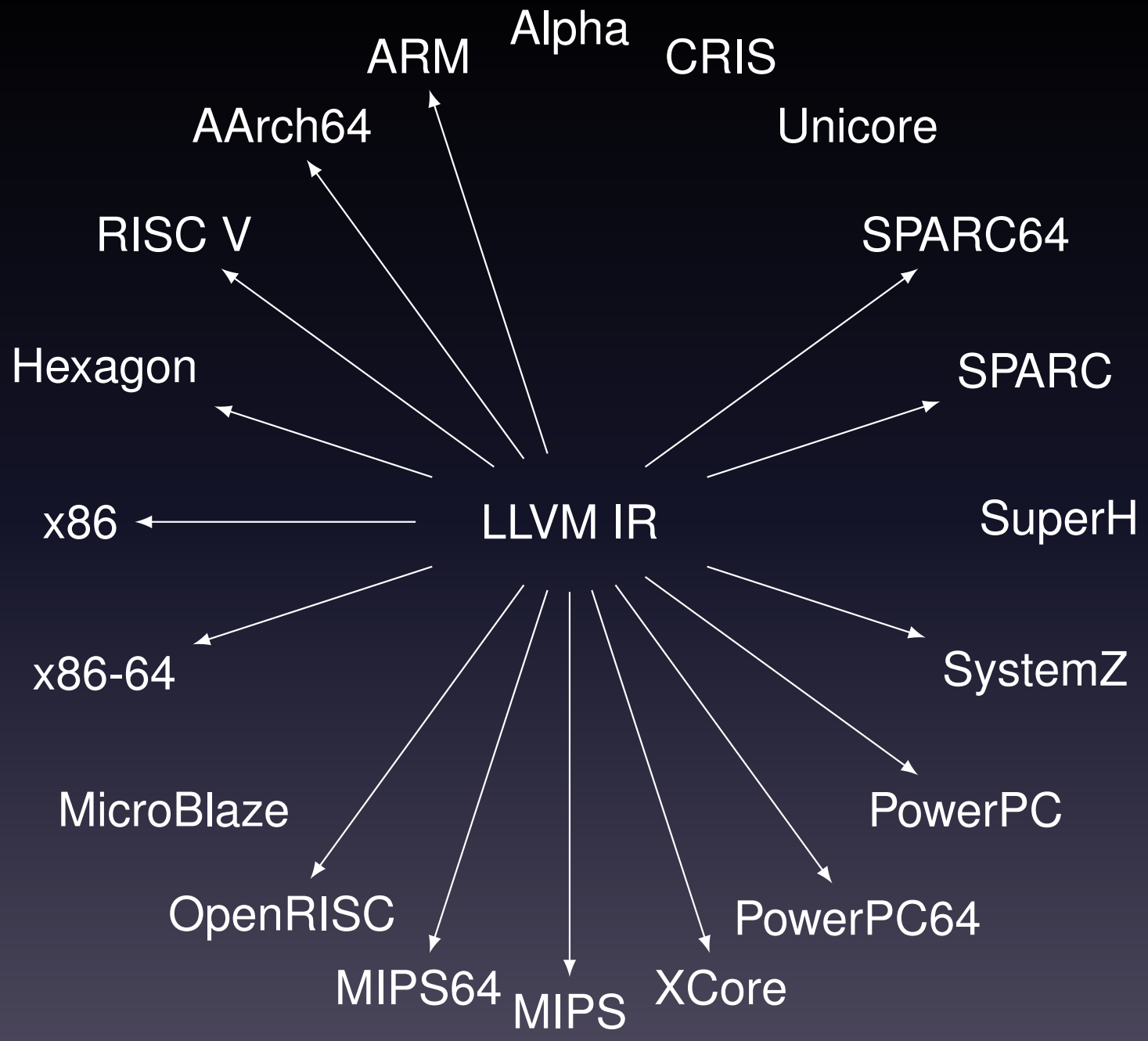
rev.ng is a *unified* suite of tools
for static binary analysis

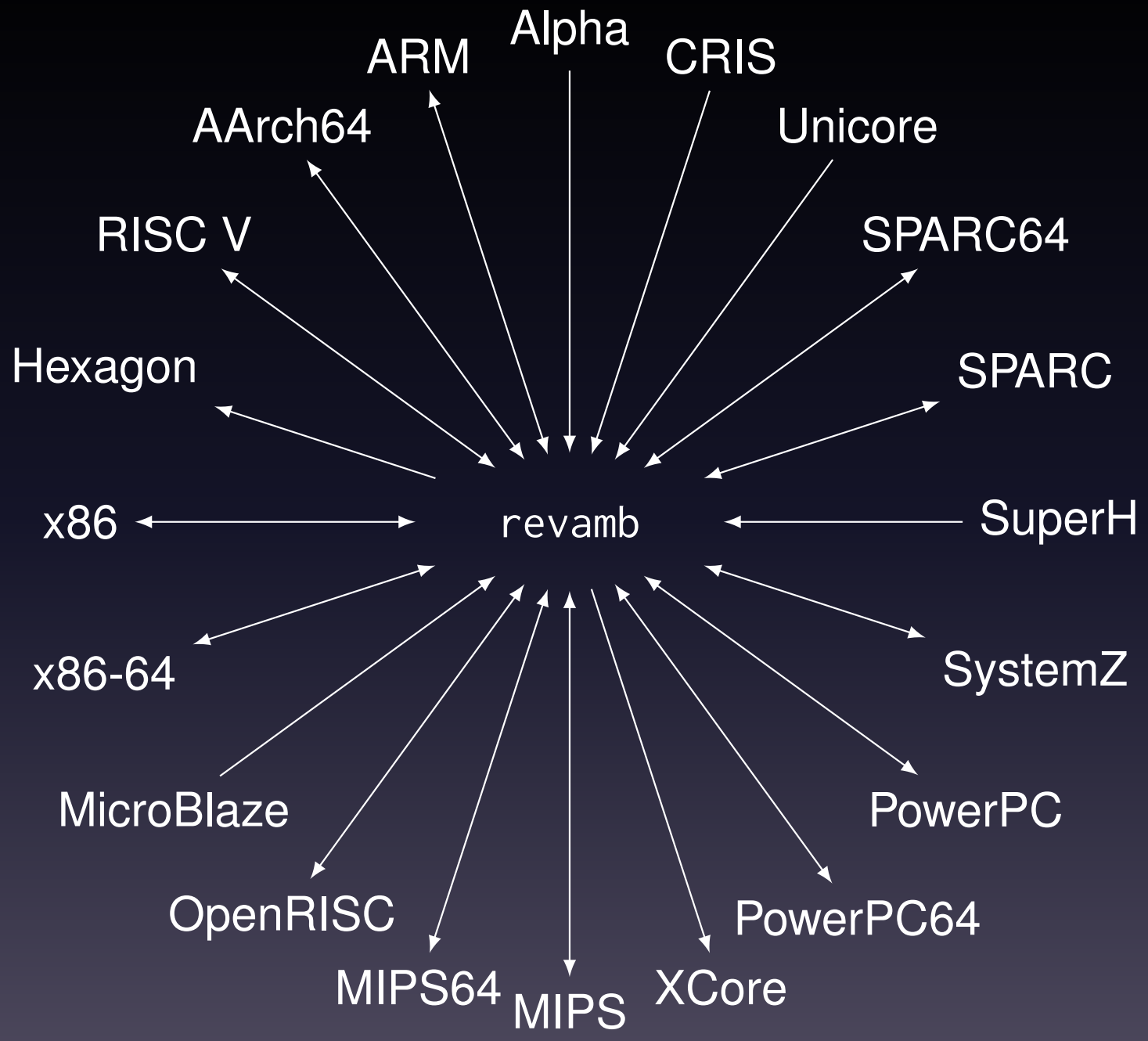
Everything you'll see is architecture agnostic

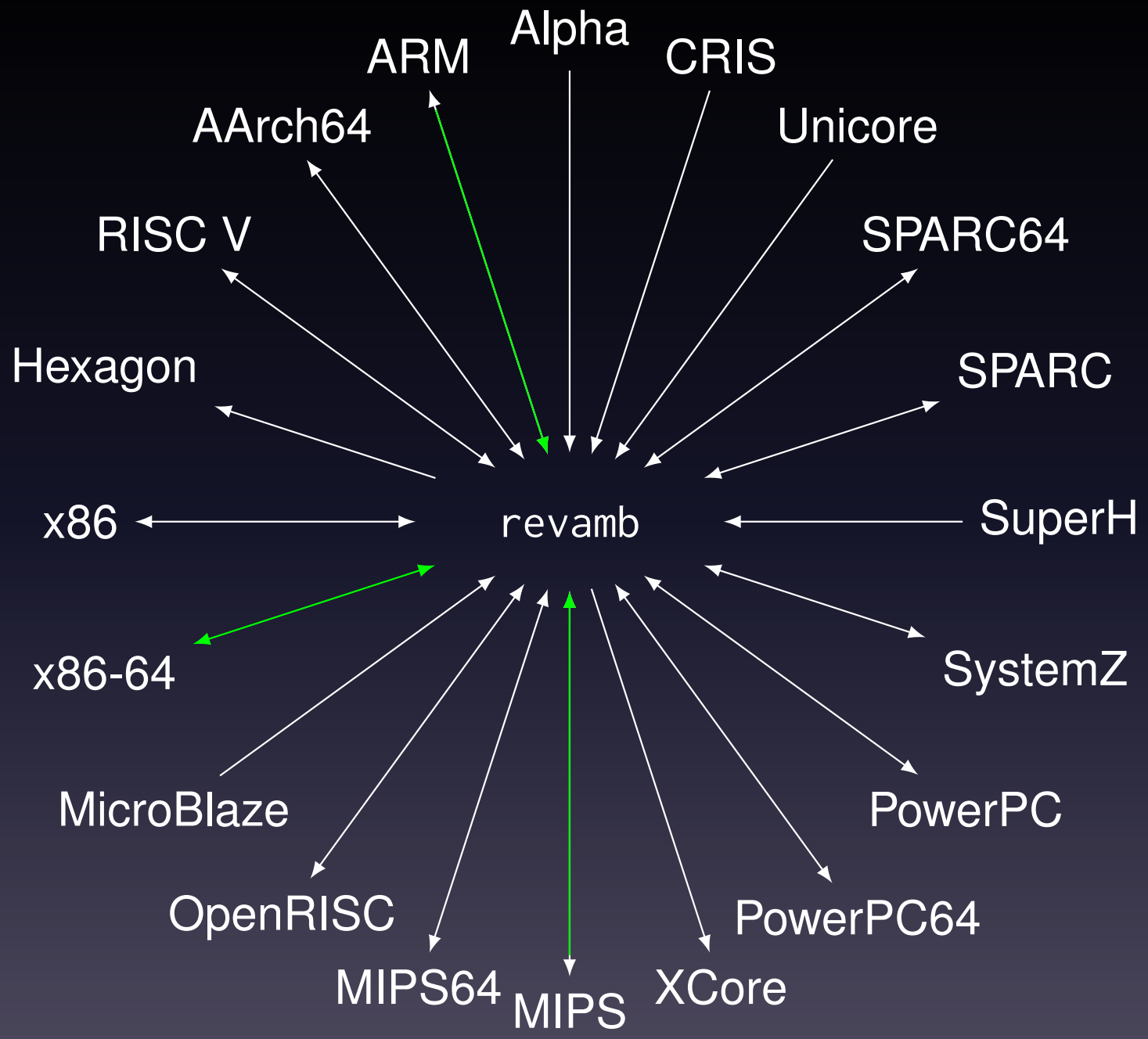
The core: the static binary translator

1. Parse the input binary and load it in memory
2. Identify all the basic blocks
3. Lift their code to *tiny code instructions* using QEMU
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`
basic block	set of `llvm::BasicBlock`

Concept mapping

Input assembly	**`revamb`**
CPU register	`llvm::GlobalVariable`
basic block	set of `llvm::BasicBlock`
direct branch	direct branch

Concept mapping

Input assembly	**`revamb`**
CPU register	`llvm::GlobalVariable`
basic block	set of `llvm::BasicBlock`
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
basic block	set of `llvm::BasicBlock`
direct branch	direct branch
indirect branch	jump to the dispatcher

Concept mapping

Input assembly	**`revamb`**
CPU register	LLVM GlobalVariable
basic block	set of `llvm::BasicBlock`
direct branch	direct branch
indirect branch	jump to the dispatcher
complex instruction	QEMU helper function

Concept mapping

Input assembly	**`revamb`**
CPU register	LLVM GlobalVariable
basic block	set of `llvm::BasicBlock`
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
```

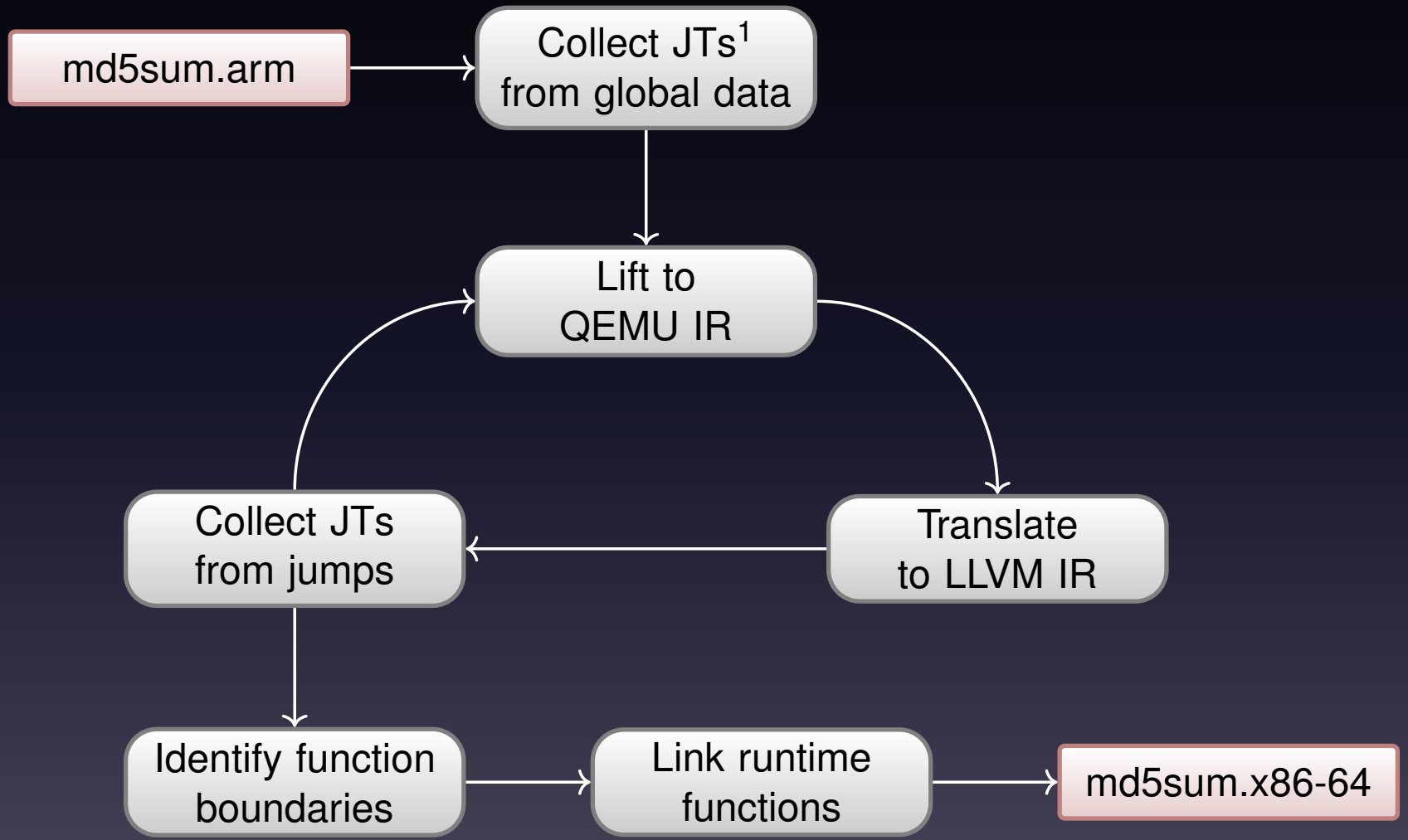
```
bl  0x1234
```

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

Overall example

```
define void @root() {
dispatcher:
  ; ...

bb.main:
  ; ...
  br label %bb.main.0x11

bb.main.0x11:
  ; ...
  br label %bb.myfunction

bb.myfunction:
  ; ...

}
```

System overview



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

Index

Introduction

Back to binary

Let’s stay on the IR

Supported platforms

Cross-architecture recompilation

What do you do when you have some LLVM IR?

Cross-architecture recompilation

What do you do when you have some LLVM IR?

You compile it!

Let’s see some use cases

Legacy programs

Have some old MIPS binaries you want to run?

Instrumentation

You can also change the IR before re-compiling it!

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
from llvmcpy import llvm

root_function = module.get_named_function("root")
r7 = module.get_named_global("r7")

dprintf = module.get_named_function("dprintf")
stderr = context.int32_type().const_int(2, True)
message_ptr = create_global_string(module, "%d\n")

builder = context.create_builder()
for basic_block in root_function.iter_basic_blocks():
  for instruction in basic_block.iter_instructions():
    if instruction.instruction_opcode == llvm.Call:
      callee = get_callee(instruction)
      if callee.name == "helper_syscall":
        builder.position_builder_before(instruction)
        load_r7 = builder.build_load(r7, "")
        arguments = [stderr, message_ptr, load_r7]
        builder.build_call(dprintf, arguments, "")
```

```
 @state_0x8a20 = global i64 0
+@message = global [4 x i8] c"%d\0A\00"

 declare void @abort()

@@ -4535,11 +4536,13 @@ bb.syscall:
    store i32 112504, i32* @pc
    %332 = load i64, i64* @env
    %333 = load i32, i32* @r7
+   %334 = load i32, i32* @r7
+   %335 = call i32 @dprintf(i32 2,
+                            i8* @message,
+                            i32 %334)
    call void @helper_syscall(...)
```

QEMU -strace

```
$ qemu-arm -strace hello
brk(NULL) = 0x00039000
brk(0x000394b0) = 0x000394b0
open("/dev/urandom",O_RDONLY) = 3
read(3,0xf6ffde84,4) = 4
close(3) = 0
ioctl(0,21505,-151003688,0,221184,0) = 0
ioctl(1,21505,-151003688,1,221184,0) = 0
write(1,0x372a8,13) = 13
Hello world!
exit_group(0)
```

Instrumented output

```
$ ./hello.instrumented.translated
45
45
5
3
6
54
54
4
Hello world!
248
```

Tracing

- Each input instruction generates some LLVM instructions
- They are delimited by a call to a function, `newpc`
- By default empty, but can be customized in C
- It has access to the whole CPU state
- You can easily trace the PC (or any other register)

Tracing implementation

The call to newpc:

```
bb.main:
  ; 0x00000000004000ee:  sub    rsp,0x10
  call void @newpc(i64 0x4000ee, i64 4)
  %4 = load i64, i64* @rsp
  ; ...
```

Possible tracing implementation:

```
const size_t trace_pc_buffer_size = 1024 * 1024;
uint64_t trace_pc_buffer[trace_pc_buffer_size];

void newpc(uint64_t pc, uint64_t instruction_size) {
  trace_pc_buffer[trace_pc_buffer_index++] = pc;

  if (trace_pc_buffer_index >= trace_pc_buffer_size)
    flush_trace_pc();
}
```

In-place patching [WIP]

- rev.ng can detect functions in the original binary
- We can:
  - isolate their basic block in a standalone function
  - let the user modify it
  - recompile it
  - patch it back in the binary

Retrofitting security features [IDEA]

We can do even more invasive changes

- Detect function calls and returns
- Introduce a shadow stack
- We could even enable faster black box fuzzing (AFL)

The big question

OK, but what's the overhead?

The big question

OK, but what's the overhead?

We're working on getting SPEC to work :)

The general idea

- We will be sensibly slower than native code
- But sensibly faster than qemu, valgrind and PIN!

Comparison with DBT

Dynamic binary translators view a single basic block
We can see the whole code

What does affect performance?

1. Presence of tracing

What does affect performance?

1. Presence of tracing
2. Quality of the recovered CFG

What does affect performance?

1. Presence of tracing
2. Quality of the recovered CFG
3. Quality of the recovered CFG

What does affect performance?

1. Presence of tracing
2. Quality of the recovered CFG
3. Quality of the recovered CFG
4. Quality of the recovered CFG

Goals

Ideal goal original function $\Rightarrow$ LLVM function

Realistic goal minimize the use of the dispatcher

Index

Introduction

Back to binary

Let’s stay on the IR

Supported platforms

If you want to run the translated program
correctness is mandatory

What if we just want to analyze it?

We can sacrifice some correctness to provide useful and understandable information in most cases

Example compromise

```
myfunction:
  push rax
  ; ...
  mov [rsp+???], 42
  ; ...
  pop rax
```

Decompilation

Goals:

- provide useful information to the analyst
- recover high-level abstractions
- be correct as much as possible

In practice

- Recover function boundaries
- Get high-level control-flow constructs (`if-else`, `while`...)
- Detect ABI and function signatures
- Analyze stack usage (local variables)

Index

Introduction

Back to binary

Let’s stay on the IR

Supported platforms

Current scope

We currently handle statically linked ELF binaries for Linux
on MIPS, ARM and x86-64

Dynamic binaries

- Basic support is easy: load libraries and translate them all
- Challenge: mix translated and native code
- On different architectures?

Additional ISAs

- Supporting those handled by QEMU is trivial
- We want to add support for alternative front-ends (CGEN)

Support other operating systems

- Analysis is not a problem, running them is
- BSD is doable
- Windows might be an interesting challenge
- Kernel-side code?

Other binary formats

They’re basically just containers
Supporting PE/COFF and Mach-O should be trivial

Final goal

Produce intelligible C representation of the program

Thanks for your attention!

https://rev.ng

License