Coverage-guided Fuzzing of Individual Functions Without Source Code

Alessandro Di Federico

ale@rev.ng

Politecnico di Milano

October 25, 2018

Index

Coverage-guided fuzzing

An overview of `rev.ng`

Experimental results

Fuzzing

Fuzzing

1. Generate a lot of different inputs
2. Feed them to a program
3. Wait for it to reach an invalid state
4. Collect a report for the analyst

Features

Pros:

- Easy to setup
- It can find subtle bugs

Cons:

- It might require large amount of resources
- Semi-decidable

A huge leap forward

Coverage-guided fuzzing

A huge leap forward

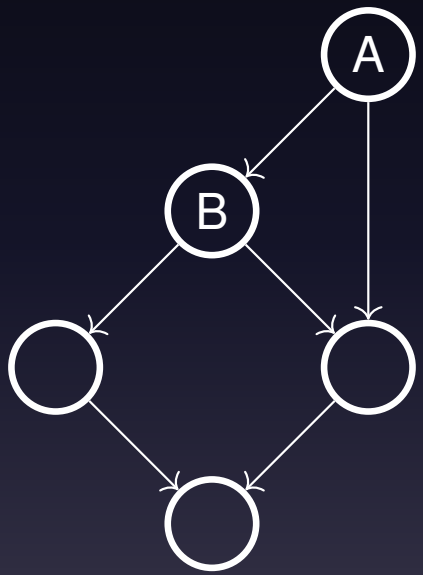
Coverage-guided fuzzing

Privilege inputs leading to cover new code paths

A huge leap forward

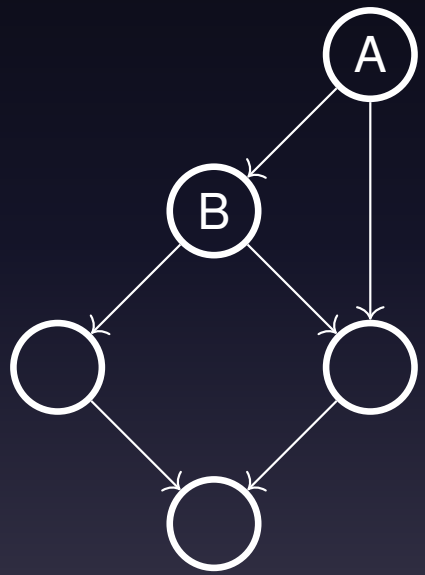
```
int main() {
  if (A && B) {
    crash();
  } else {
    all_good();
  }
}
```

The Control-flow Graph



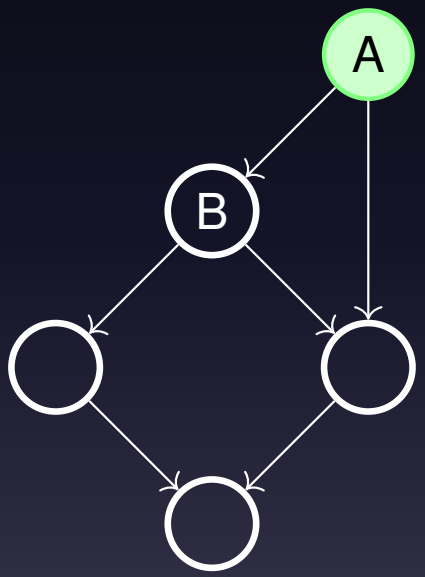
First run

Input: 0000 0000 0000 0000



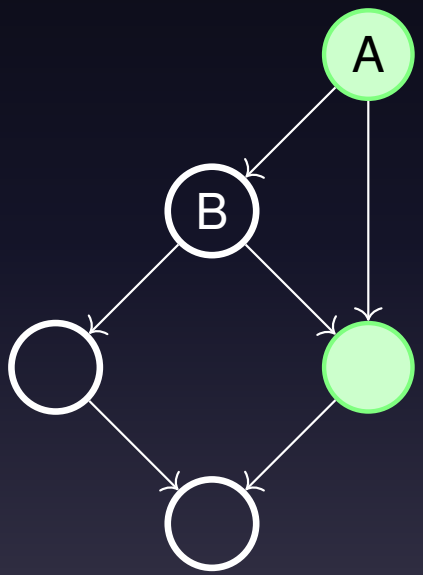
First run

Input: 0000 0000 0000 0000



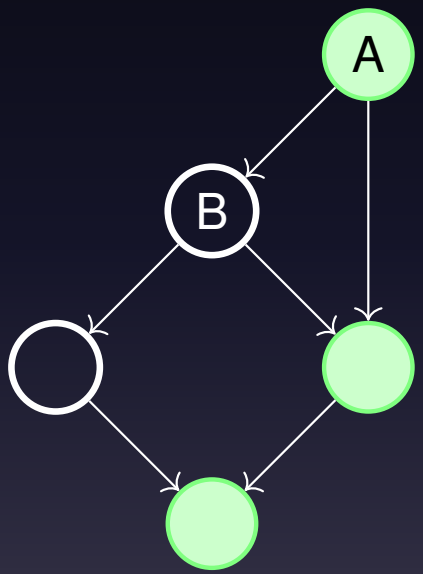
First run

Input: 0000 0000 0000 0000



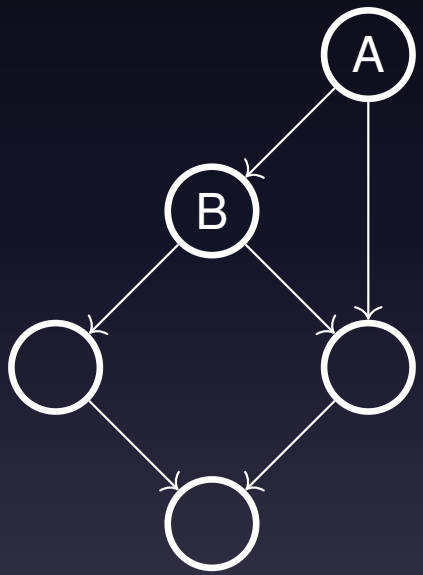
First run

Input: 0000 0000 0000 0000



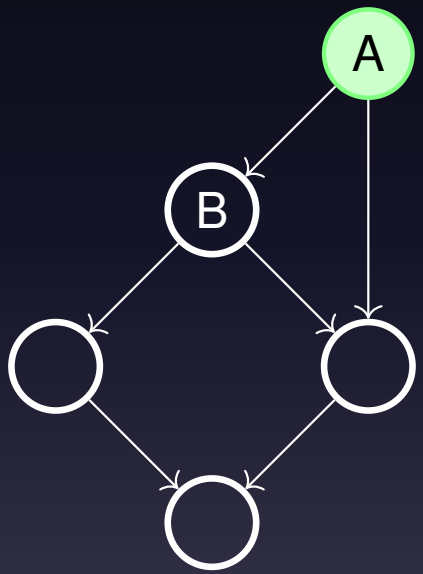
Second run

Input: 0000 0000 0000 0001



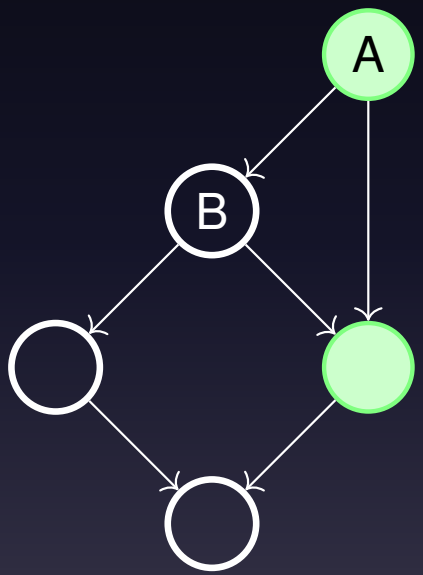
Second run

Input: 0000 0000 0000 0001



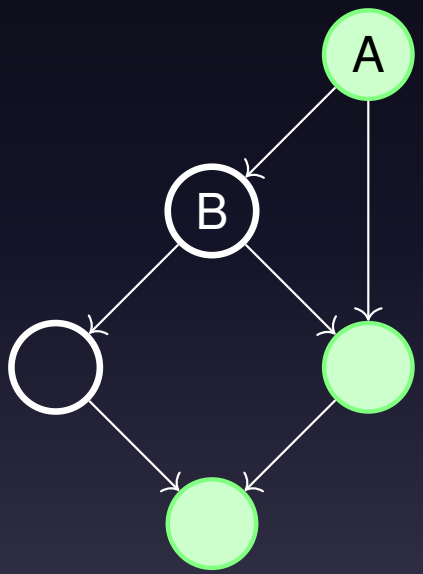
Second run

Input: 0000 0000 0000 0001



Second run

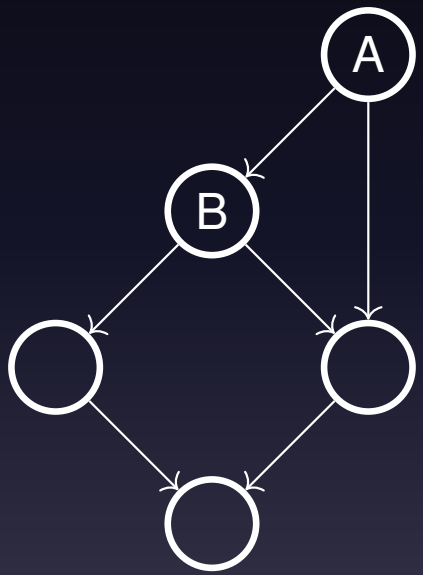
Input: 0000 0000 0000 0001



This input is not interesting!

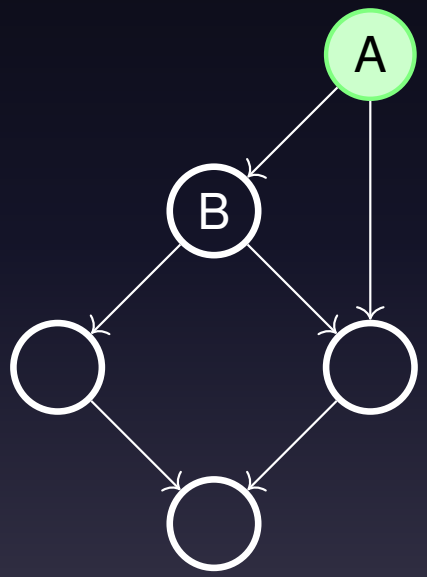
Third run

Input: 0001 0000 0000 0000



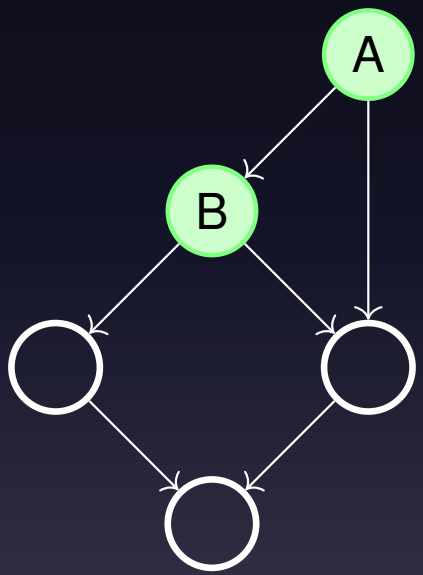
Third run

Input: 0001 0000 0000 0000



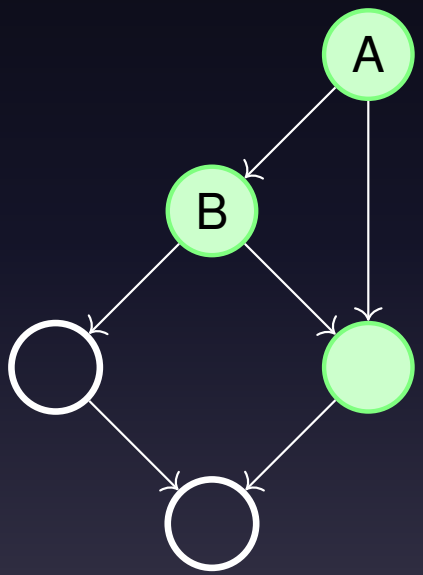
Third run

Input: 0001 0000 0000 0000



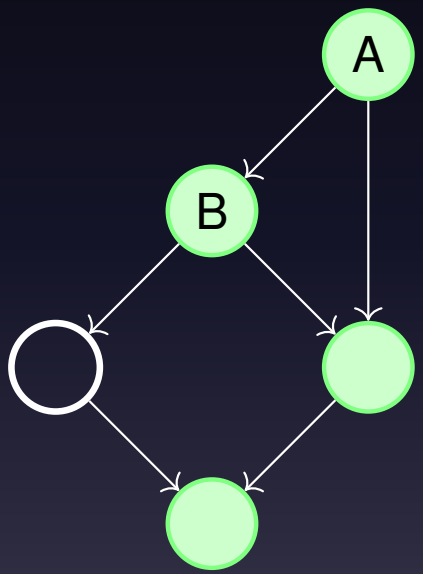
Third run

Input: 0001 0000 0000 0000



Third run

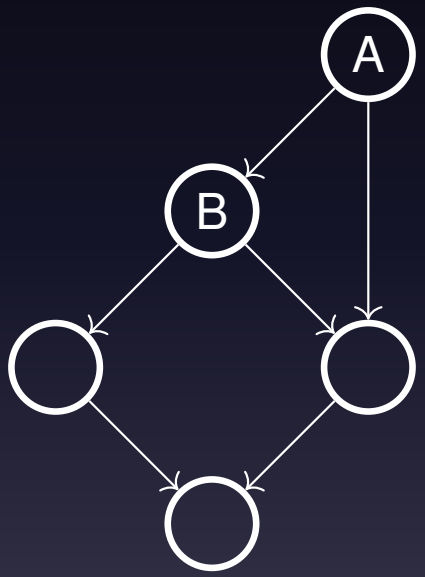
Input: 0001 0000 0000 0000



This input is interesting!
It led us to discover a new basic block

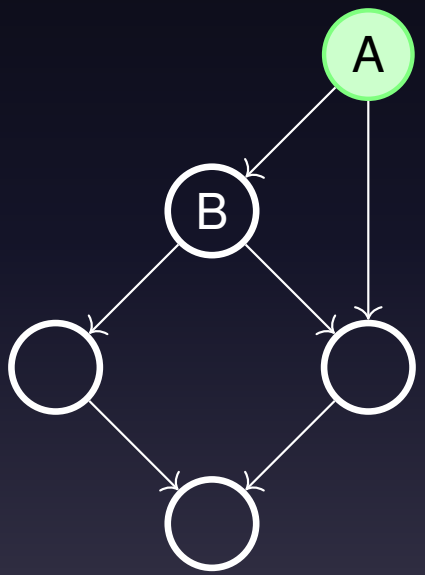
Fourth run

Input: 0011 0000 0000 0000



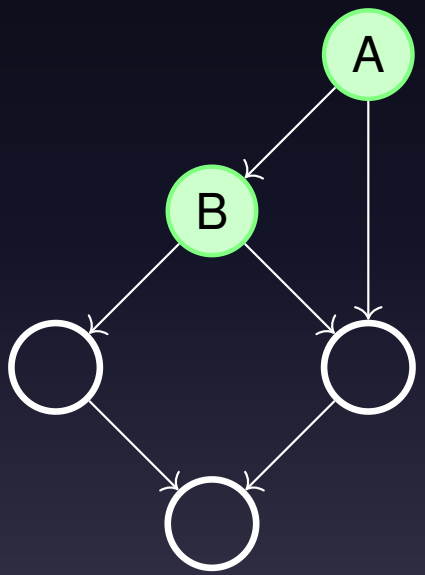
Fourth run

Input: 0011 0000 0000 0000



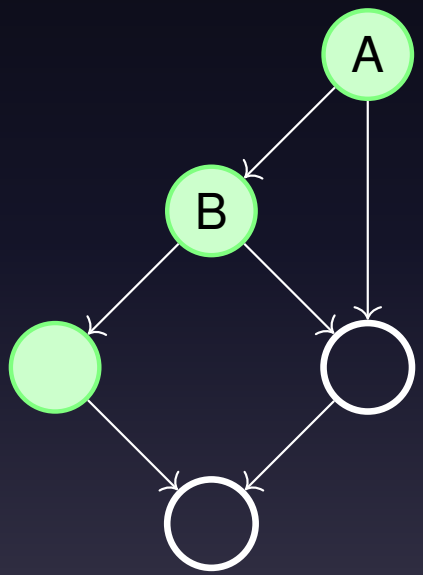
Fourth run

Input: 0011 0000 0000 0000



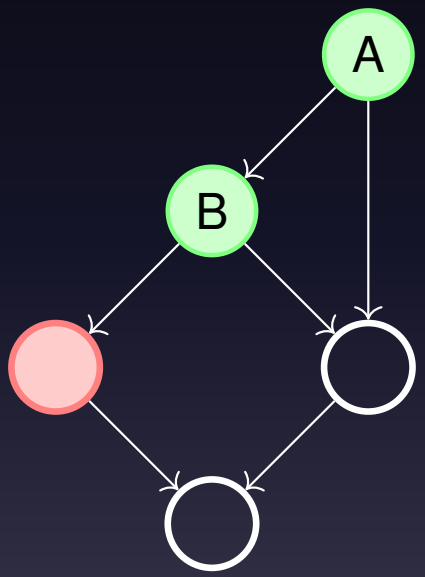
Fourth run

Input: 0011 0000 0000 0000



Fourth run

Input: 0011 0000 0000 0000



american fuzzy lop

- It made coverage-guided fuzzing popular
- Developed by lcamtuf
- Performs instrumentation to detect executed basic blocks
- Two key modes of operation:
  - Source mode
  - Binary mode

Source mode

Instrumentation is performed at compiler-level

Source mode

Instrumentation is performed at compiler-level

```
int main() {
  record(1);
  if (A && B) {
    record(2);
    crash();
  } else {
    record(3);
    all_good();
  }
  record(4);
}
```

Binary mode

An emulator is employed to detect executed basic blocks

Binary mode

An emulator is employed to detect executed basic blocks

- QEMU is the chosen emulator
- It incurs in a sensible slowdown

libfuzzer

- Alternative to afl
- It requires the source code to be available
- Based on LLVM

What's LLVM?

LLVM is a compiler framework

Famous for its C/C++ frontend (clang)
and its intermediate representation (the LLVM IR)

`libfuzzer` can be a lot faster

It doesn't fork

```
int main() {
  while (true) {
    char *new_input = random_input();
    target(new_input);
  }
}
```

Index

What is rev.ng?

rev.ng is a *unified* framework for binary analysis based on QEMU and LLVM

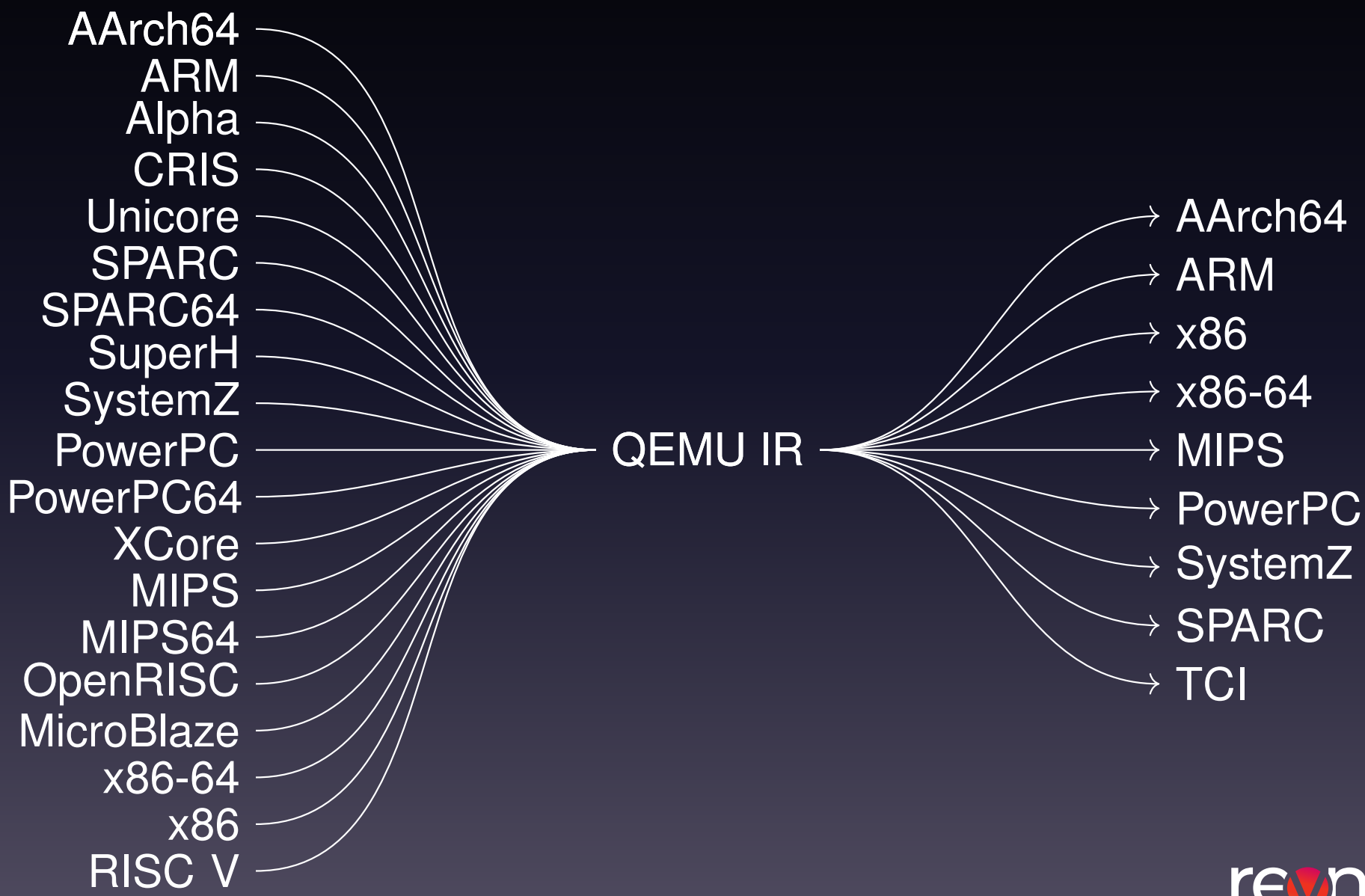
What is rev.ng?

rev.ng is a *unified* framework for binary analysis based on QEMU and LLVM

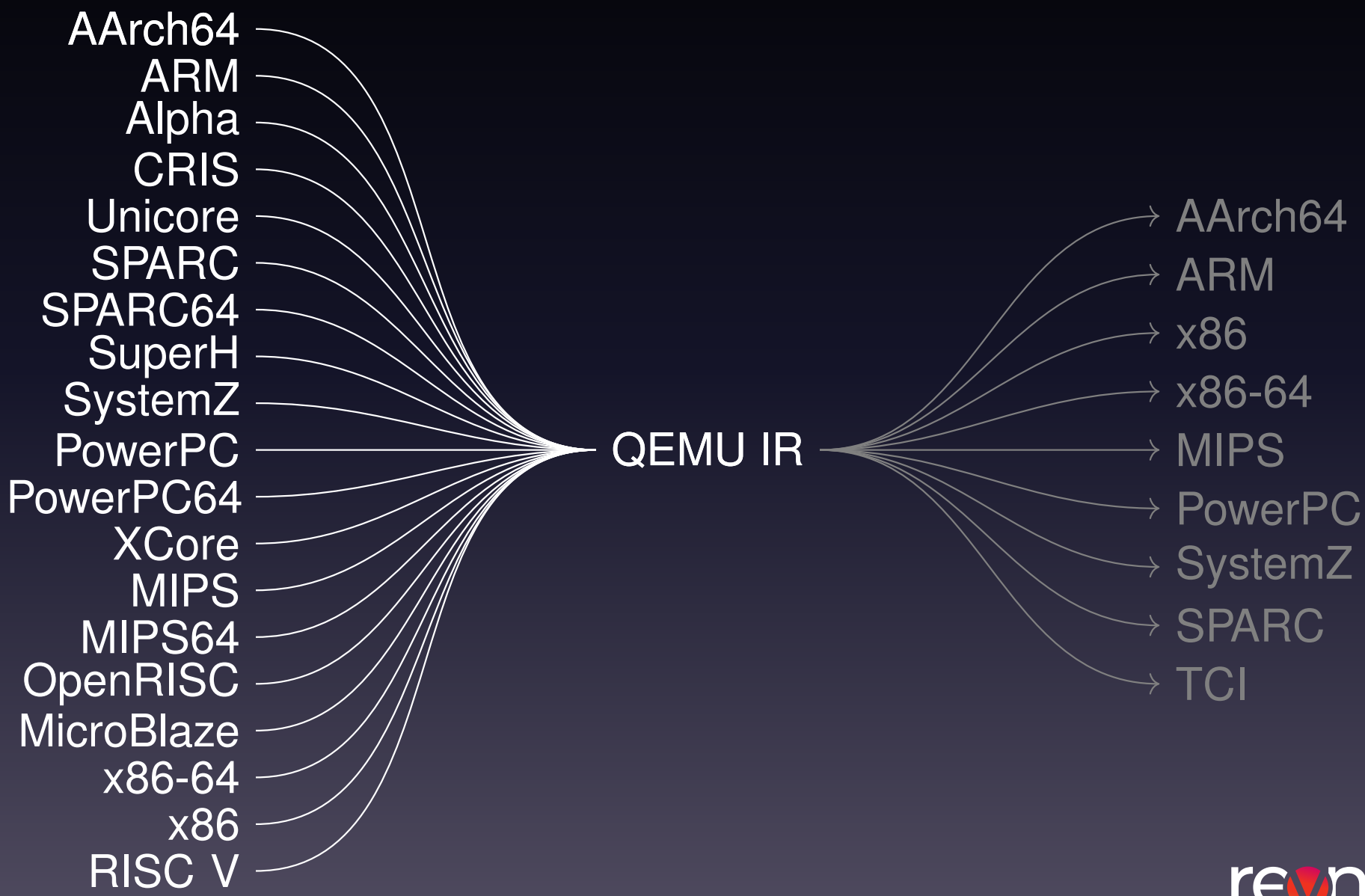
Everything you'll see here is architecture-agnostic

How does QEMU work?

A dynamic binary translator



The frontend is a *lifter*

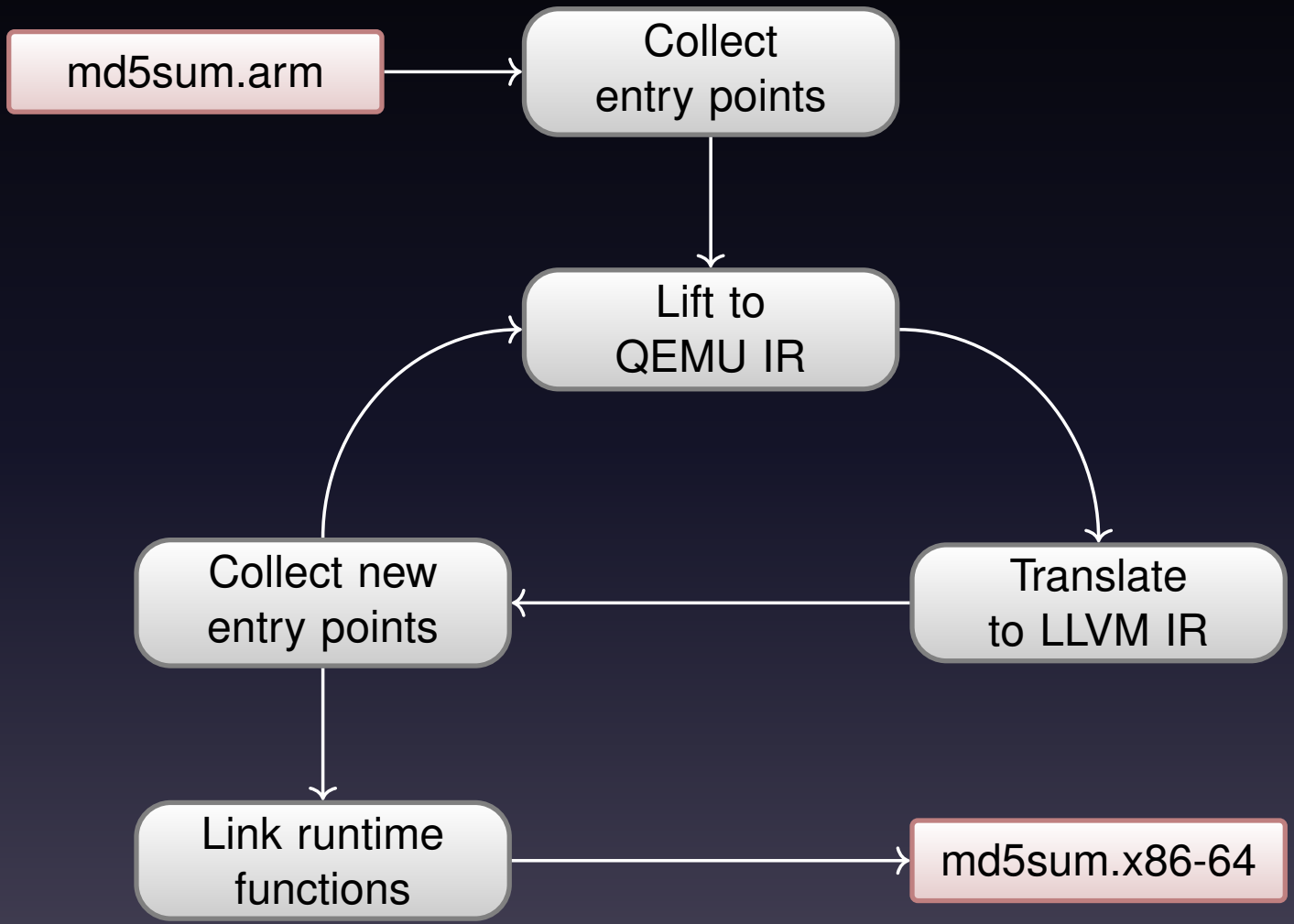


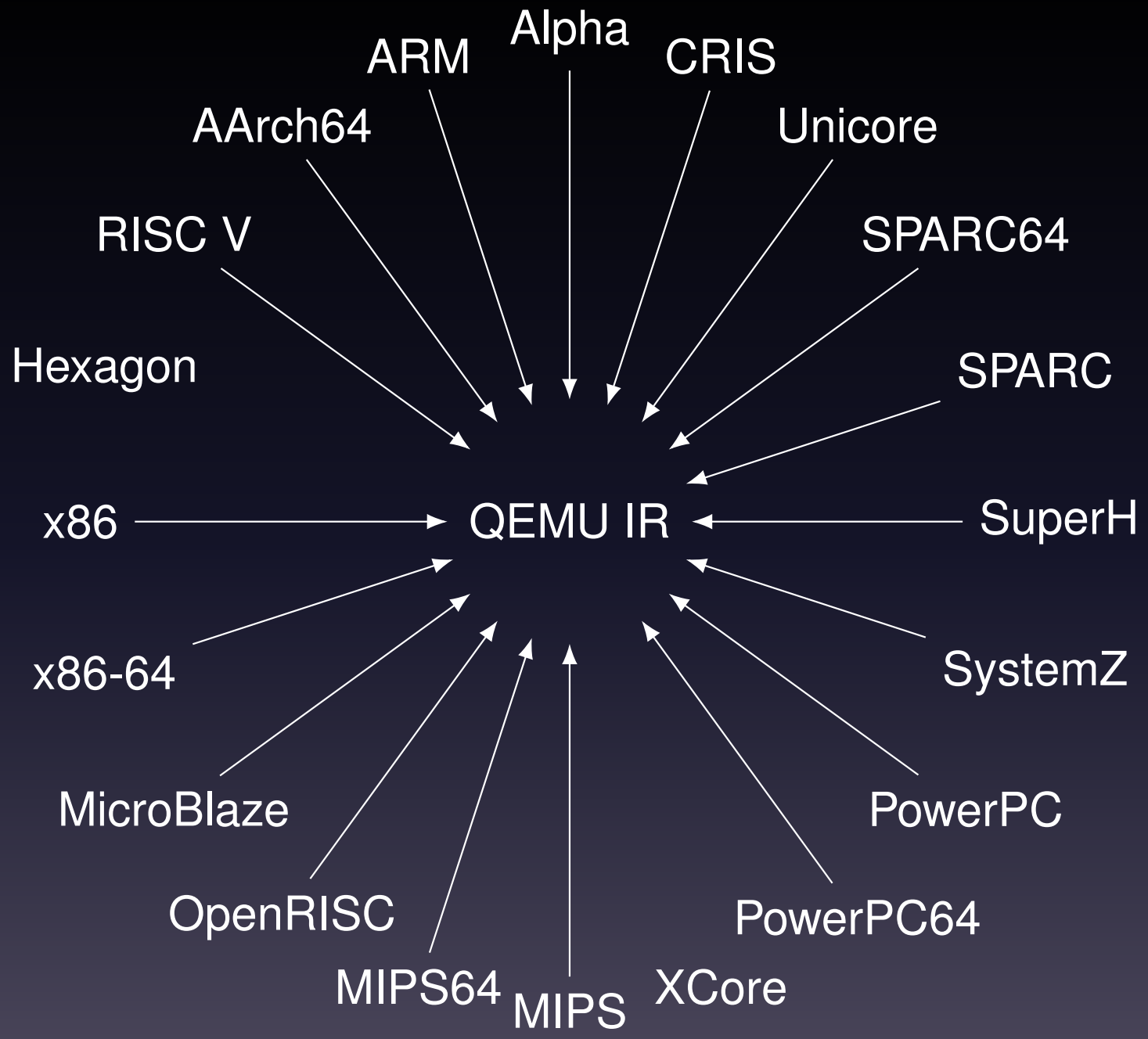
QEMU translates at run-time

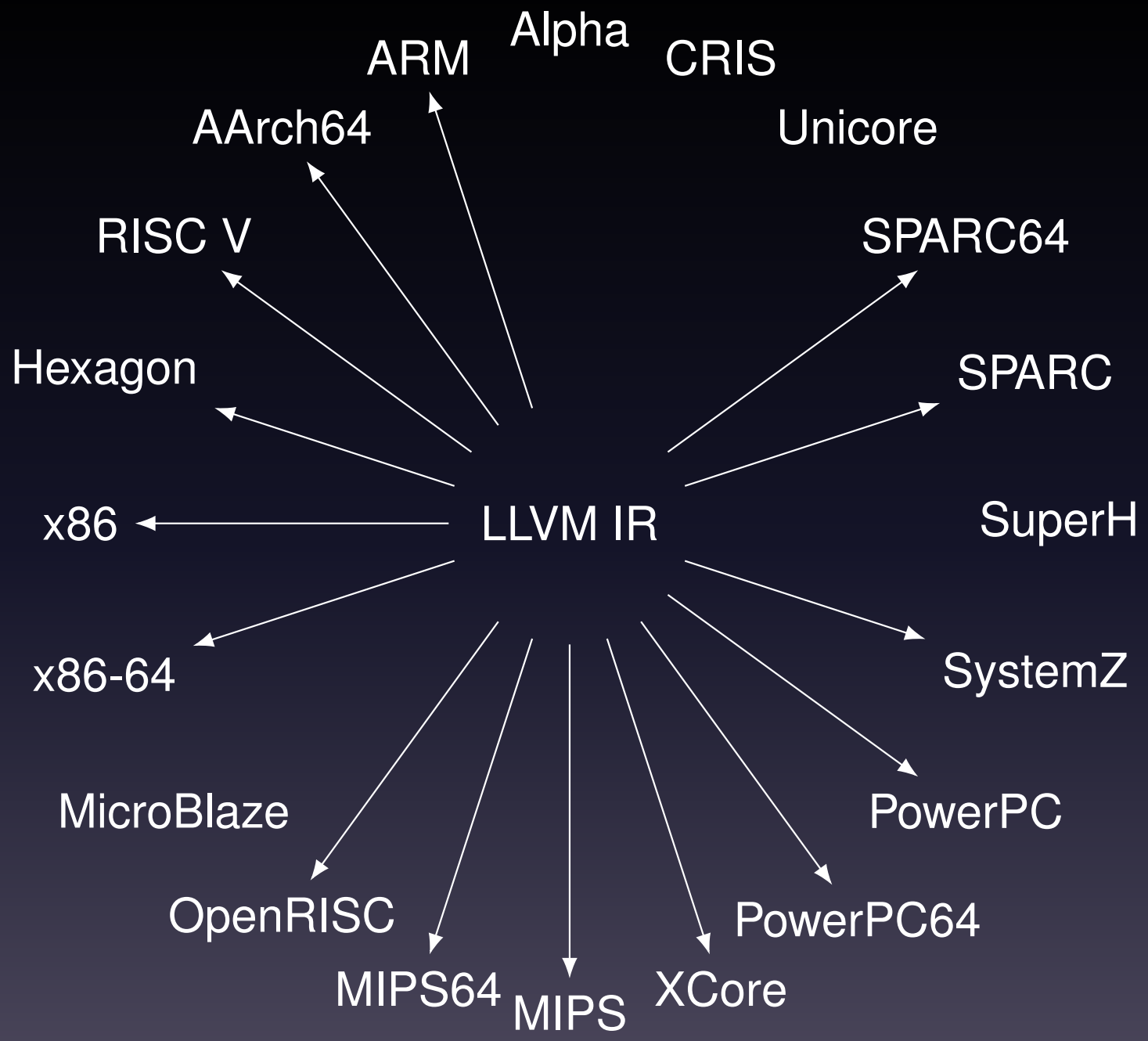
QEMU translates at run-time

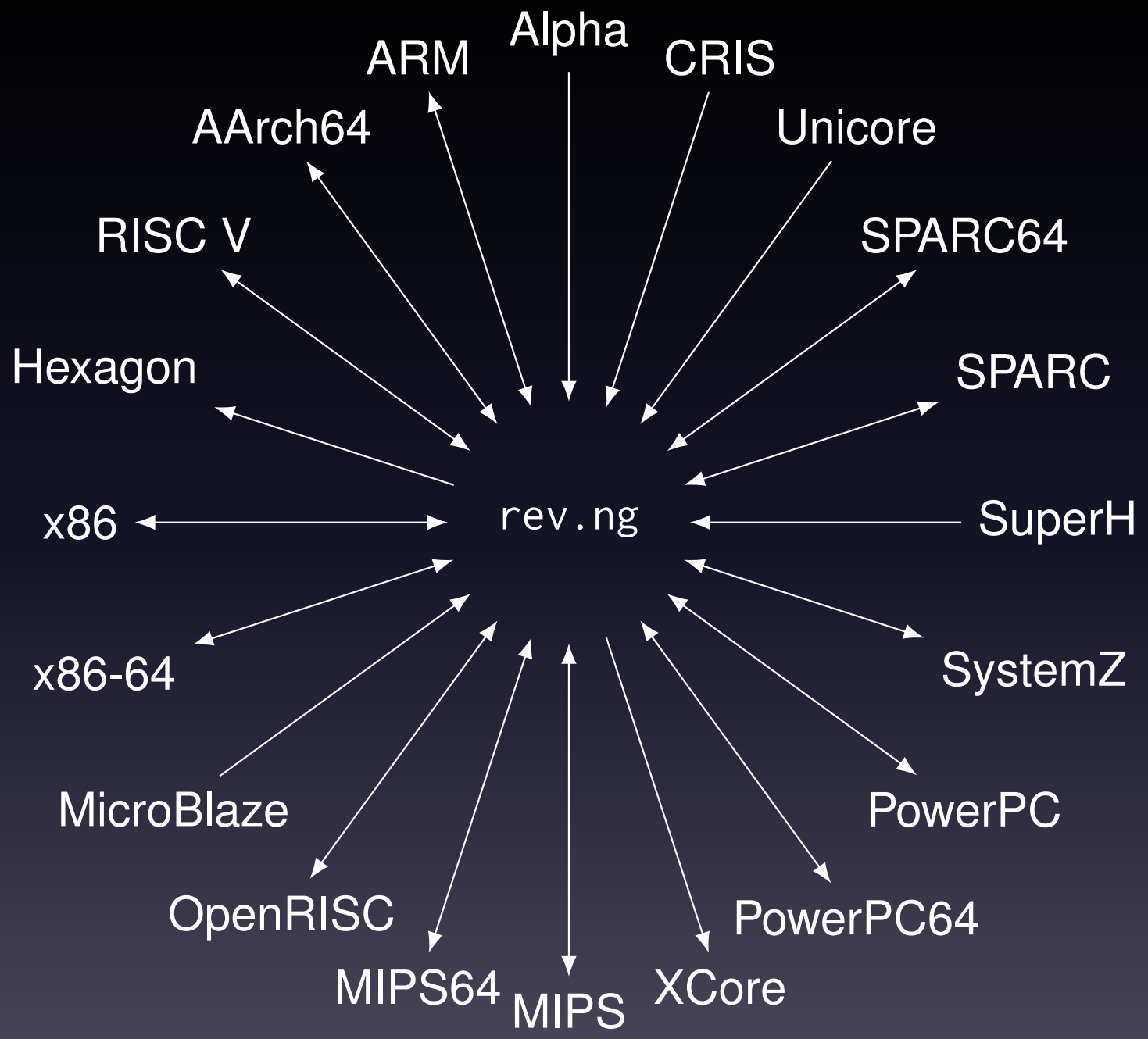
`rev.ng` translates offline

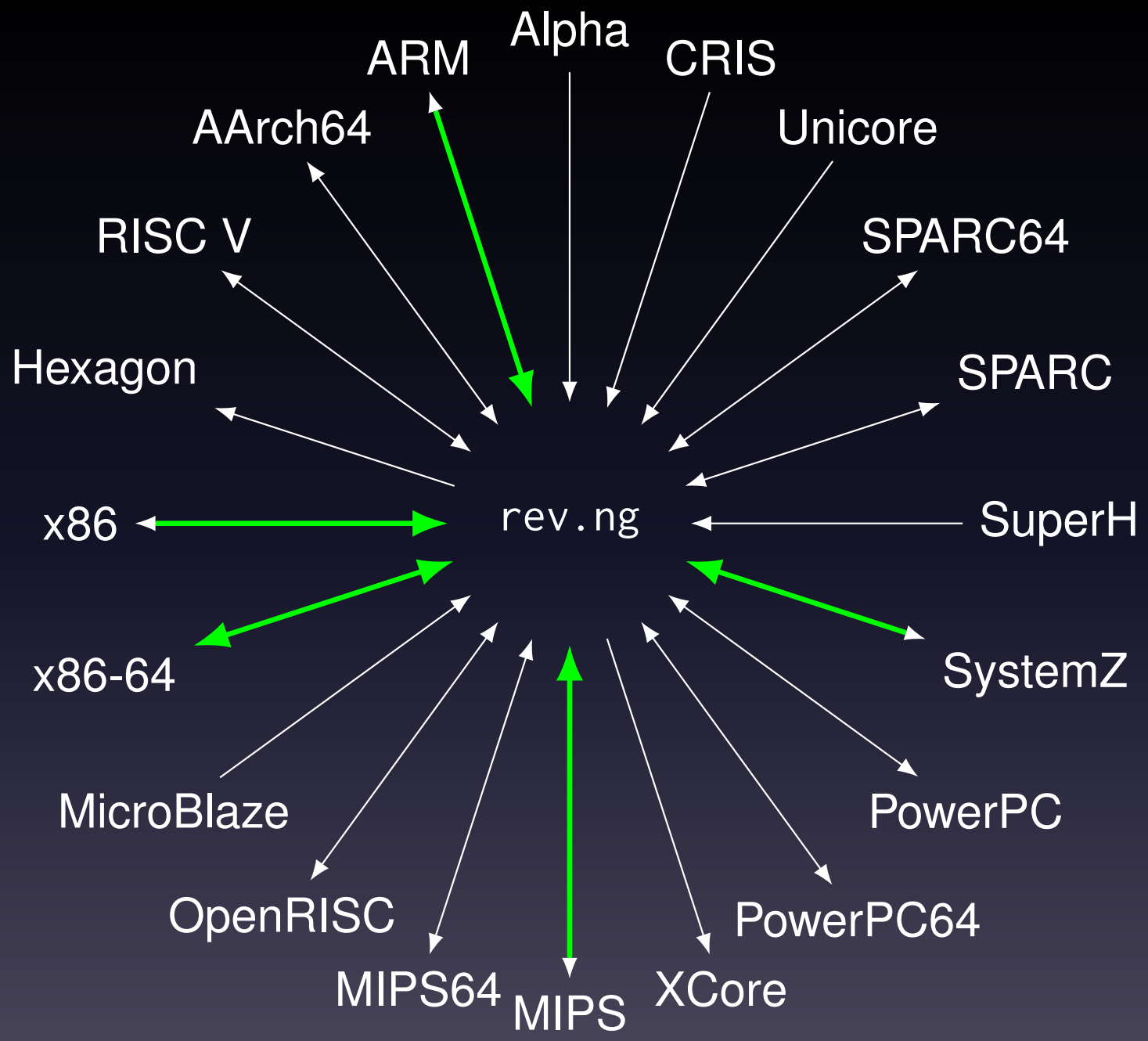
rev.ng: a static binary translator




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
rev.ng
x86
SuperH
x86-64
SystemZ
MicroBlaze
PowerPC
OpenRISC
PowerPC64
MIPS64
MIPS
XCore


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

We produce LLVM IR

We produce LLVM IR

We can employ `libfuzzer` directly

Steps

1. Lift the program to LLVM IR
2. Identify all the functions
3. Identify a function to fuzz
4. Create the fuzzing function
5. Compile fuzzing function
6. Instrument using `libfuzzer`
7. Launch the fuzzer

Steps

1. Lift the program to LLVM IR
2. Identify all the functions
3. Identify a function to fuzz MANUAL
4. Create the fuzzing function MANUAL
5. Compile fuzzing function
6. Instrument using `libfuzzer`
7. Launch the fuzzer

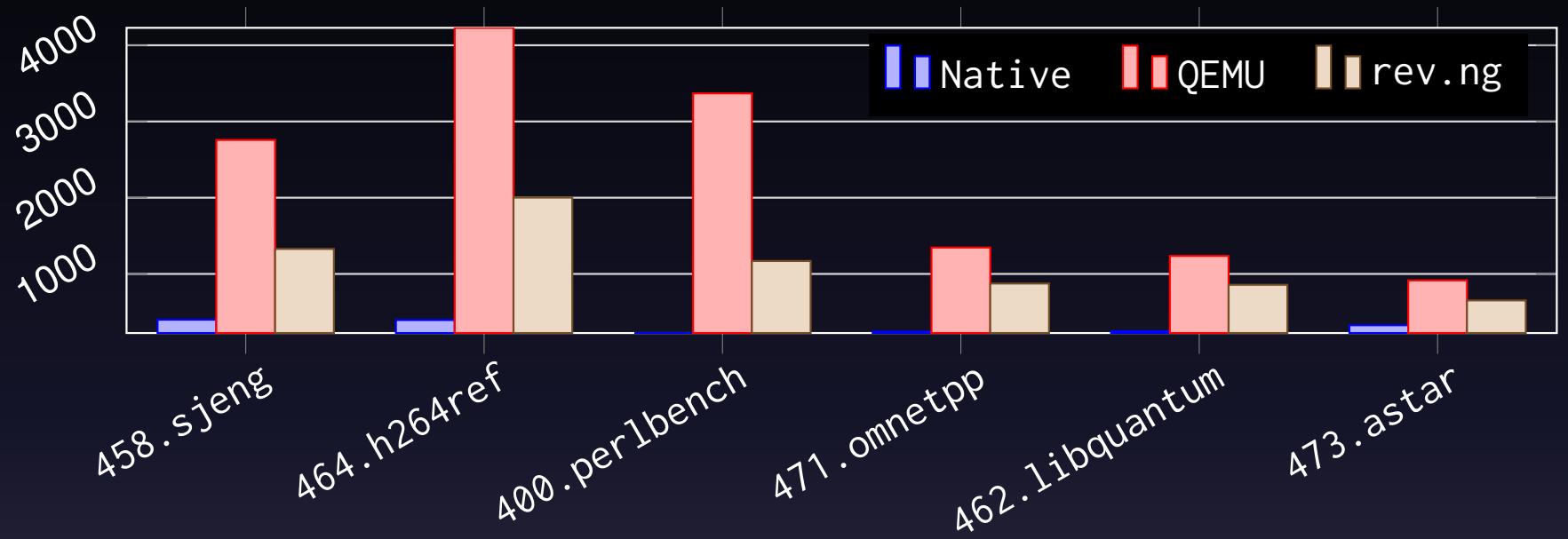
Index

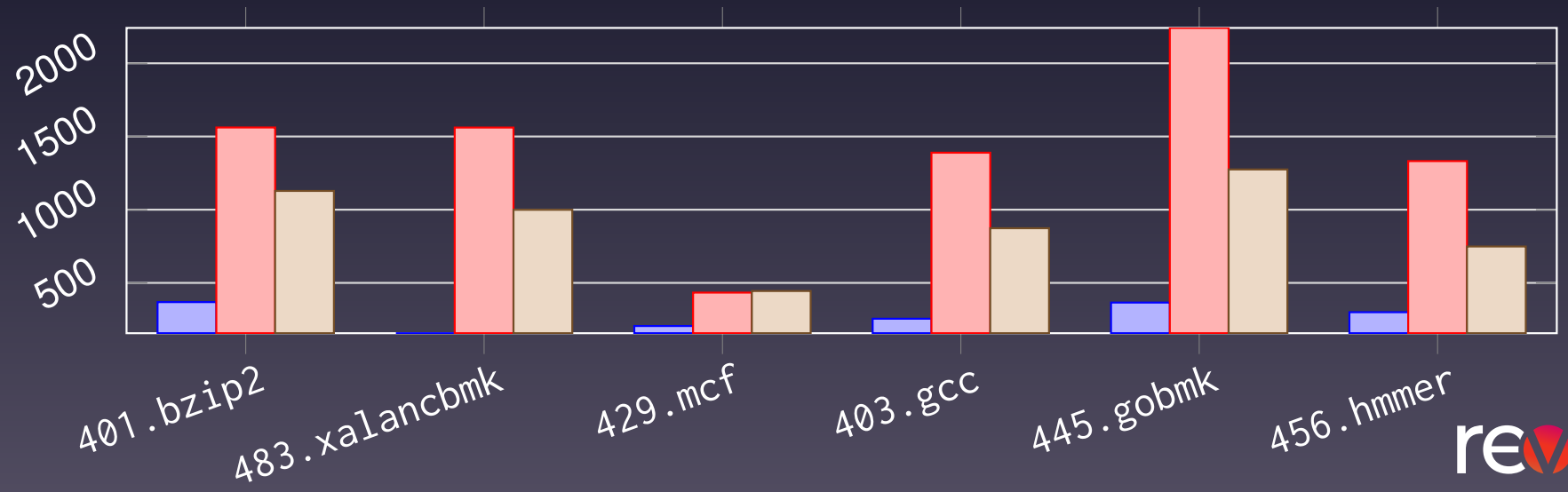
We are sensibly faster than QEMU

We are sensibly faster than QEMU

1. The LLVM optimizer has a wider view on the code
2. The translation is performed offline

Runtime (seconds)





On average, 68% faster than QEMU

A practical case study

We want to fuzz the PCRE library

A practical case study

We want to fuzz the PCRE library

Not directly, but embedded in another program (`less`)

Steps (again)

1. Lift the program to LLVM IR
2. Identify all the functions
3. Identify a function to fuzz
4. Create the fuzzing function
5. Compile fuzzing function
6. Instrument using `libfuzzer`
7. Launch the fuzzer

Steps (again)

1. Lift the program to LLVM IR
2. Identify all the functions
3. Identify a function to fuzz
4. Create the fuzzing function
5. Compile fuzzing function
6. Instrument using `libfuzzer`
7. Launch the fuzzer

Fuzzing function (simplified)

```
int LLVMFuzzerTestOneInput(uint8_t *data,
                           size_t size) {
  char input_string[] = "Test␣string!";
  void *compiled_re;
  compiled_re = pcre_compile(data);

  pcre_exec(compiled_re,
            input_string,
            strlen(input_string));

  pcre_free(compiled_re);
  return 0;
}
```

We were able to find a known vulnerability in PCRE

Comparing with afl

Are we faster than afl?

- afl fuzzing worked directly on PCRE (without `less`)
- Used black-box mode

Performances

	Execs per second			Total execs
	1 min	10 min	60 min	60 min
afl	3 582	3 495	3 682	13 187 295
rev.ng	150 617	79 701	78 306	271 217 728

Summary

- We do not require the source code
- We can fuzz any entry point
- We are sensibly faster than existing techniques

Future works

- Improve performances
- Perform symbolic execution (through KLEE)

Future works

Backup slides

Very effective!

The bug-o-rama trophy case

Yeah, it finds bugs. I am focusing chiefly on development and have not been running the fuzzer at a scale, but here are some of the notable vulnerabilities and other uniquely interesting bugs that are attributable to AFL (in large part thanks to the work done by other users):

IJG jpeg [1]	libjpeg-turbo [1] [2]	libpng [1]
libtiff [1] [2] [3] [4] [5]	mozjpeg [1]	PHP [1] [2] [3] [4] [5] [6] [7] [8]
Mozilla Firefox [1] [2] [3] [4]	Internet Explorer [1] [2] [3] [4]	Apple Safari [1]
Adobe Flash / PCRE [1] [2] [3] [4] [5] [6] [7]	sqlite [1] [2] [3] [4...]	OpenSSL [1] [2] [3] [4] [5] [6] [7]
LibreOffice [1] [2] [3] [4]	poppler [1] [2...]	freetype [1] [2]
GnuTLS [1]	GnuPG [1] [2] [3] [4]	OpenSSH [1] [2] [3] [4] [5]
PuTTY [1] [2]	ntpd [1] [2]	nginx [1] [2] [3]
bash (post-Shellshock) [1] [2]	tcpdump [1] [2] [3] [4] [5] [6] [7] [8] [9]	JavaScriptCore [1] [2] [3] [4]
pdfium [1] [2]	ffmpeg [1] [2] [3] [4] [5]	libmatroska [1]

License